Induction and Co-Induction in Sparkle
Thesis number 550

Author: Leonard Lensink
Supervisor: Marko van Eekelen
Second reader: Sjaak Smetsers

August 1, 2006

Abstract

Developing correct software remains one of the most important subjects in computer science. Bugs can be costly and annoying. One of the most secure ways to eliminate errors in computer programs is by proving them to be correct in a mathematical context. Sparkle is a proof assistant that helps programmers with constructing mathematical proofs about algorithms written in Clean or any other functional language. Proofs on programs that use one of the most important programming techniques, called recursion, usually need proof principles called inductive and co-inductive reasoning.

Support for these mathematical proof steps were limited within Sparkle. In order to support for reasoning about a larger class of programs we have extended the proof techniques in Sparkle.

A method that supports mutually reasoning on mutually recursive types has been added. A new method has been devised that allows for the derivation of an induction scheme from function definitions. For co-inductive reasoning, a tactic was added based on bisimulation relationships. Another speculative tactic was devised that derives a co-inductive proof principle from a function definition.

In this thesis these four methods and their implementation are presented.

Contents

1 Introduction **9**
- 1.1 Introduction . . . 9
- 1.2 Research question . . . 9
- 1.3 Reading the thesis . . . 10

2 Formal reasoning & recursive programs **11**
- 2.1 Formal reasoning . . . 11
  - 2.1.1 Formal reasoning introduced . . . 11
  - 2.1.2 Functional languages and formal reasoning . . . 11
  - 2.1.3 Sparkle introduced . . . 12
  - 2.1.4 Formal reasoning and Sparkle . . . 12
- 2.2 Recursion . . . 12
  - 2.2.1 Recursion introduced . . . 12
  - 2.2.2 Recursion and Sparkle . . . 13

3 Sparkle **15**
- 3.1 Introduction . . . 15
- 3.2 Sparkle in general . . . 15
- 3.3 A sample proof in Sparkle . . . 16
- 3.4 Previous state of affairs Sparkle . . . 17
- 3.5 Summary . . . 21

4 Mathematical background **23**
- 4.1 Introduction . . . 23
- 4.2 Inductive proof principle . . . 23
  - 4.2.1 Mathematical induction . . . 25
  - 4.2.2 Structural induction . . . 25
  - 4.2.3 Course of values and other data types . . . 25
- 4.3 Co-inductive proof principle . . . 27
  - 4.3.1 Category theory . . . 27
  - 4.3.2 A categorical approach to induction . . . 29
  - 4.3.3 A categorical approach to co-induction . . . 34
  - 4.3.4 Bi-simulation and co-induction . . . 36

4.4 Summary 37

5 Extending Sparkle **39**
5.1 Introduction 39
5.2 Induction on mutually recursive algebraic data types 40
5.2.1 Algebraic data types 40
5.2.2 The tactic 42
5.2.3 An expression language example 46
5.3 Induction schemes for mutually recursive functions 48
5.3.1 The size change principle 49
5.3.2 Derivation of induction schemes 52
5.3.3 A novel tactic for deriving induction schemes for mutually recursive functions 56
5.3.4 Example of tactic applied to gcd 61
5.4 Simple Co-Induction 62
5.4.1 Bisimulation with respect to algebraic data types . . . 63
5.4.2 Bisimulation tactic 63
5.4.3 Example of bisimulation tactic applied to power series 65
5.4.4 Guarded co-induction 66
5.4.5 Tactic for guarded co-induction 67
5.4.6 Example of guarded co-induction on the Thue Morse sequence 67
5.5 Summary 68

6 Conclusion **71**
6.1 Summary 71
6.2 Future work 72

A Induction and Co-Induction in Sparkle **75**
A.1 Introduction 76
A.1.1 A brief introduction to Sparkle 76
A.1.2 Using Sparkle for Haskell programs 77
A.1.3 Induction and co-induction in Sparkle 77
A.2 Standard techniques for recursive functions 78
A.2.1 Induction on mutually recursive data types 78
A.2.2 Co-Induction 80
A.3 Induction on mutually recursive functions 81
A.3.1 Tactic for deriving induction schemes for mutually recursive functions 82
A.3.2 Example: tactic applied to gcd 85
A.3.3 Related work 86
A.3.4 Our method compared to other methods 86
A.4 Acknowledgements 90
A.5 Future work 90

A.6 Conclusion 90

Chapter 1

Introduction

> "For, usually and fitly, the presence of an introduction is held to imply that there is something of consequence and importance to be introduced. – Arthur Machen"

1.1 Introduction

Imagine yourself sitting in an airplane, traveling at almost six hundred miles an hour at an altitude of thirty thousand feet. It is a wonderful experience until you realize what happens if you would hit the ground at that velocity. Modern aircraft rely on software to stay in the air. Failure of the software can unexpectedly and painfully cut your holiday short.

Reliability of software is very important. This thesis wants to make a contribution towards the correctness of computer programs.

In the first chapter of this thesis will be explained why formal reasoning about computer programs is important. Next, the reader will be introduced to Sparkle as a tool for formal reasoning and to recursion. Then, we will describe the research question we are trying to answer. Finally, we make clear to the reader what to expect in following chapters of this thesis.

1.2 Research question

We will establish in the next chapter that formal reasoning is a powerful tool to help us guarantee the correctness of (functional) programs. Sparkle is a good tool for formal reasoning. Recursion is one of the core methods used in functional programming. We want to find out how we can improve support for formal reasoning on a larger class of recursive programs.

The research question of this thesis reads: *How can Sparkle be extended to enable Sparkle to provide proofs on a larger class of recursive programs?*

1.3 Reading the thesis

Our approach to answering the research question will be four pronged. In the next chapter we will introduce formal reasoning and recursion. In chapter three the reader will be introduced to Sparkle. This chapter will provide more insight into how proof assistants work and what needs to be done to extend support for reasoning about recursive programs. In the introduction to Sparkle, we will also show how induction proofs were handled by Sparkle and how Sparkle was not sufficient to handle some programs.

The chapter on Sparkle is followed by a chapter on the mathematical foundations of induction and co-induction. The proof methods that will have to be added to Sparkle, have to be mathematically sound. Chapter four will provide the necessary background to understand why the methods we are going to introduce in chapter five are mathematically sound.

In chapter five, four methods will be introduced which enable Sparkle to provide proofs on a larger class of recursive programs than it did in its previous version. The four methods have been implemented in Sparkle [1]. For each method it is described what it contributes to Sparkle, on which theory the method is based on and how the method is implemented in Sparkle. Also for each method an example is given.

In our final chapter we will summarize how we have improved support for formal reasoning about recursive functions within Sparkle.

In appendix A, a contribution to the conference "Trends in Functional Programming", held in Munich in november 2004, we show how one of the tactics introduced in Sparkle is new compared to other methods.

[1] The source code is available at http://www.razorbv.nl/sparkle/Sparkle.zip

Chapter 2

Formal reasoning & recursive programs

"To err is human, but to really foul things up requires a computer" –Paul Ehrlich

2.1 Formal reasoning

2.1.1 Formal reasoning introduced

Formal reasoning about (mathematical) properties of programs is very useful. It can give a rigorous proof of the correct behavior of said program. If the proof fails, it can help pinpoint potential flaws in the program. Especially for programs where failure is not an option or extremely costly, formal reasoning can supply the needed security.

Unfortunately formal reasoning about computer programs tends to be even more complex as writing the programs themselves. It is prone to human mistakes and can be very tedious. For this reason programs have been developed to assist people in their formal reasoning process. These programs act as frameworks that check the correctness of formal reasoning steps and sometimes help by resolving parts of the proof automatically.

2.1.2 Functional languages and formal reasoning

Computer programs come in all kinds of languages. For formal reasoning purposes it is essential that the semantics of the language is defined within a mathematical context. This makes functional programming languages especially suited for that task. Their roots are within the mathematical theory of recursive functions and their semantics are easily described using lambda calculus or term or graph rewrite systems.

Although functional languages have their roots within the academic world and their acceptance outside this environment is limited, they are

suited for more than scholarly exercise [7]. The kinds of programs that can be written using a functional language are of the same level as "regular" programming languages.

There many flavors of functional languages. Among the best known are Erlang, Haskell and Clean. The latter two are quite similar. The differences are small enough to make automatic translation from one language into another possible. [2]

2.1.3 Sparkle introduced

Sparke is a dedicated theorem prover for Clean. Unlike other theorem provers, programs written in Clean have no need for a translation into some kind of specification language. This makes reasoning on the source code level possible for the programmer. Sparkle will be addressed extensively in chapter three.

2.1.4 Formal reasoning and Sparkle

Formal reasoning is a powerful tool to help us guarantee the correctness of (partial) programs and we need co-inductive and inductive reasoning to make formal proofs of most functional programs possible.

Clean is such a functional programming language and Sparkle is a theorem prover written for Clean. It allows the programmer to specify properties, using first order logic and clean expressions. It also allows a programmer to use mathematical reasoning to prove those properties.

2.2 Recursion

2.2.1 Recursion introduced

Recursion is an often used method to achieve iteration in a programming language. Furthermore, every computable function can be expressed using recursive functions. Support for mathematical techniques that are able to prove properties of recursive programs and structures are warranted. This support should be broad, intuitive and preferably automated to some extent.

Recursive programs generally come in three flavors; Those that can be proven to terminate; those that can not be proven to terminate and those that can be proven to not terminate. Properties of the first kind of programs can be proven using the mathematical technique of induction. The second kind and third kind of programs usually require another related technique, called co-inductive reasoning.

[2] For instance, Hacle is an automated tool to translate Haskell programs to Clean. http://www-users.cs.york.ac.uk/~mfn/hacle/

2.2.2 Recursion and Sparkle

Support for reasoning on recursive programs in Sparkle was limited to induction on natural numbers as well as structural induction on directly recursive algebraic data types. This allows for properties for many kinds of recursive programs. However, there were still a lot of programs for which no proofs could be constructed due to the limited nature of the induction tactic in Sparkle.

For instance, when one tries to construct a proof of the property of natural numbers that the square root of a natural number is a rational number, one quickly runs into the problem that the induction schemes generated by Sparkle to prove the afore mentioned property can not be used to prove this property.

Furthermore, there was no support for reasoning on recursive programs that do not terminate. Since induction implies termination, co-inductive proof techniques are needed to prove properties of non-terminating recursive programs.

Chapter 3

Sparkle

"The only effective way to raise the confidence level of a program significantly is to give proof for its correctness" – Edsger Dijkstra

3.1 Introduction

In this chapter a short description will be given of what Sparkle is and how it is used to prove properties of functional programs. We will show how Sparkle handles proofs by giving a small example proof for a boolean operator. In section 3.4, we will address the previous version of Sparkle, the version before the techniques addressed in this thesis were implemented. Because we are talking about recursive programs, of main importance is to describe how Sparkle used to handle proofs by induction. Subsequently, in this section we will show by means of an example in what way the induction technique, as it was implemented in the previous version of Sparkle, was insufficient. The chapter will be concluded with a short summary.

3.2 Sparkle in general

Sparkle is a dedicated theorem prover for Clean originally devised by Maarten de Mol[11, 12][3]. Unlike other theorem provers, programs written in Clean have no need for a translation into some kind of specification language. This makes reasoning on the source code level possible for the programmer.

Sparkle contains a specification language, called Core Clean. This is a subset of full Clean, yet has the same descriptive power. Clean can easily be translated into Core Clean.

Properties of programs are expressed using standard first order logic. Logical connectives like $\neg, \rightarrow, \wedge, \vee$ and $\iff$ are available, as well as existential and universal quantors: $\exists$ and $\forall$. Equality of expressions and quan-

[3]Sparkle is available at http://www.cs.ru.nl/Sparkle/

tification of expression variables connect the logic with the semantics of the programming language.

Reasoning about these properties is simple. The property one tries to prove is called a *goal*. There are tactics which correspond with formal mathematical reasoning steps that manipulate that goal, introduce hypotheses or variables into the context of that goal or create subgoals. A *tactic* is a function from a single goal to a list of goals. All tactics are sound with respect to the semantics of Clean. This means that the validity of the new goal(s) implies the validity of the original goal. When all (sub)goals are proven, a property holds.

Building a proof within Sparkle consists of repeatedly applying tactics until no goals are left to prove. The key to success is finding the right tactic to apply. In order to help the programmer with this selection, a hint mechanism has been implemented. This means that, by means of a heuristic mechanism, the proof assistant assigns scores to several tactics. These scores reflect the likelihood Sparkle thinks applying this tactic will help the proof progress towards a solution. It is possible to automatically apply the highest scoring tactic, making fully automated theorem proving possible.

3.3 A sample proof in Sparkle

In this section, we will show how Sparkle handles proofs by giving a small example proof for a boolean operator.

We look at the negation function. This function operates on boolean values and it is equivalent to the not operation from classical logic. Although in Clean the function is directly translated into machine code, a description of its functionality in Clean is:

Definition 3.3.1

```
not :: !Bool -> Bool
not True -> False
not False -> True
```

To express that value of the negation of a negation, again is the original property, one can specify in Sparkle that:

Property 3.3.2 $\forall\ a.$`not`(`not` a) $= a$.

To prove the earlier specified property for the boolean operator 'not', a proof window is opened for this property. At first, the proof window contains only one goal: the original property. In order to progress within the proof, the tactic *introduce* is chosen. This tactic introduces the variable b into the environment of the goal. The goal itself is transformed into:

Goal 3.3.3 `not(not a)` $=$ `a`, *the environment now contains an assumed variable b.*

It looks like nothing has changed. However, this manipulation of the goal made the application of other tactics possible. In this case, the *cases* tactic. This tactic is a formalization of the notion that each expression of the boolean type, can either be `True`, `False` or undefined, designated by a special symbol, called bottom: $\perp$. Applying this tactic will remove the assumed variable b from the environment.

Application of these tactics creates three subgoals to prove:

1. `not`(`not`$\perp$)) = $\perp$.
   It is proven using the *definedness* tactic. Basically this boils down to the fact that Sparkle has concluded from the definition of the not function that it can apply `not`$\perp = \perp$ twice.
2. `not`(`not True`)) = `True`.
   This goal is proven by applying the *reduce* tactic. This tactic will use the function definitions of the functions used in the expressions in the property to rewrite the property as far as it is able to. Since `not True` reduces to `False` and `not False` reduces to `True` this leaves us with `True` = `True` to prove. This rather obvious true statement is proved using the *reflexive* tactic.
3. `not`(`not False`)) = `False`.
   Again this goal is proven by using the *reduce* tactic. Rewriting `not False` to `True` and `not True` to `False`. The resulting property `False` = `False` is proven true by the same *reflexive* tactic again.

Sparkle will then notify the user that the property has been proven.

In case of the above mentioned example, automatic application of the best scoring tactic will yield a complete, even if not exactly the same, proof of the asserted property.

3.4 Previous state of affairs Sparkle

In this section, we will address the previous version of Sparkle by describing the two methods previously used within sparkle to cater for induction proofs. We will show by means of an example in what way the induction technique, as it was implemented in the previous version of Sparkle, was insufficient.

The previous version of Sparkle allowed for two methods of induction.

- First of all induction on integers, which corresponds with the well-known principle of mathematical induction.
- The other method used within Sparkle is structural induction on an inductively defined data type.

Suppose you have a data type which is defined as follows:

Definition 3.4.1

```
::Nat = Zero | Suc Nat
```

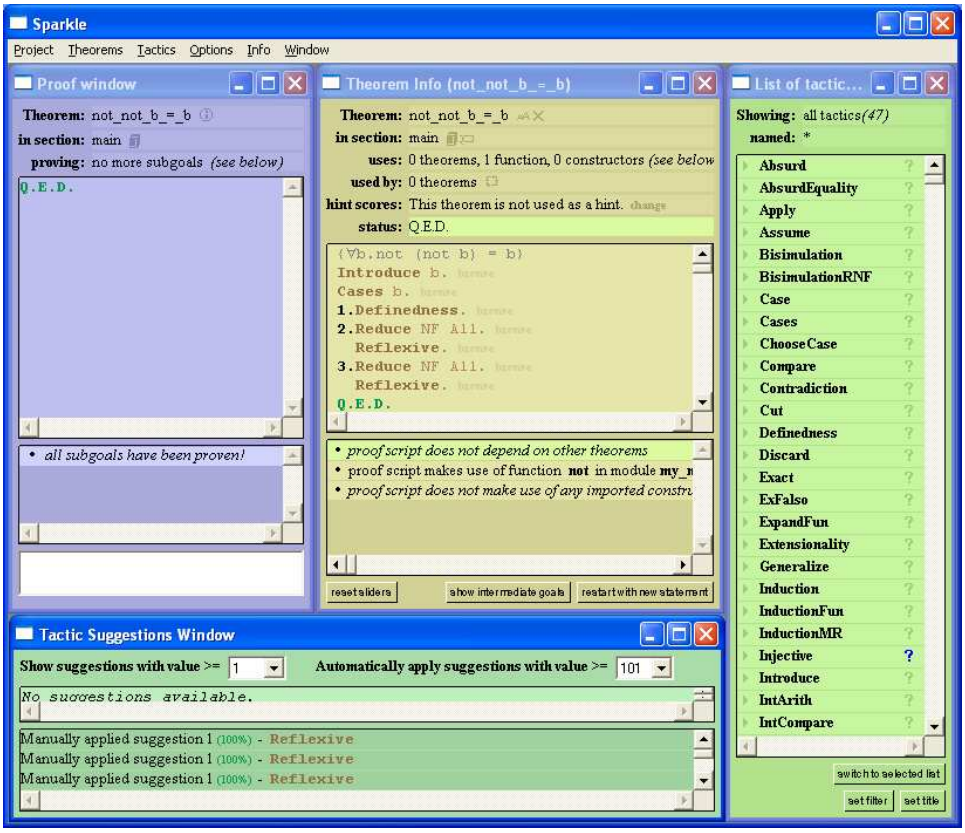


Figure 3.1: Proof of property 1.1

To construct a proof on functions using this data type, one would first need to prove $P(\texttt{Zero})$ and then prove, under the assumption that $P(x)$ holds, that $P(Suc\ x)$ is true. One must also prove that $P(\bot)$ is true.

Not all predicates are allowed within an induction proof. Predicates must be *admissible* as defined by Paulson in his book on Logic and Computation [16]. Admissible means that the predicate must be chain-complete: every chain in a partial ordered set must have a least upper bound that is in that partial ordered set. For instance, if we have a predicate that says *f is not a total function*, then that predicate is not admissible. One can construct partial functions, for which the predicate holds, that have a total function as least upper bound.

Although successful in many applications, there are limitations to what can easily be proven by using either of the above methods.

For instance, take the function that defines the greatest common devisor:

Example 3.4.2

```
gcd::Nat Nat -> Nat
```

```
gcd Zero y          = y
gcd (Suc x) Zero    = Suc x
gcd (Suc x) (Suc y)
    | y <= x     = gcd (x - y,Suc y)
    | otherwise = gcd (Suc x,y - x)
```

Where the type Nat stands for natural numbers greater than or equal to zero.

Let us try to find a proof for the next proposition,

Proposition 3.4.1 $\forall a, b \exists s, t$: $\texttt{gcd}(a, b) = a.s + b.t$, *where* a *and* b *are natural numbers, except zero and* s *and* t *are natural numbers.*

We will not describe into detail which tactics need to be used for each step within the proof, because sometimes several rather obvious steps are skipped. It should be obvious from the description of the proof steps which tactics can be used.

A proof attempt in sparkle quickly runs into the problem that one has to do induction on the first and the second argument separately and neither of them decrease constantly.

A work-around could be to rewrite the current function into an equivalent one that has an auxiliary argument that does decrease constantly and on which it is possible to do induction.

Example 3.4.3

```
gcd'::Nat Nat Nat-> Nat
gcd' Zero y z       = y
gcd' (Suc x) Zero z  = Suc x
gcd' (Suc x) (Suc y) (Succ z)
    | y <= x     = gcd' (x - y,Suc y,z)
    | otherwise = gcd' (Suc x,y - x,z)
```

Where the type Nat stands for natural numbers greater than or equal to zero.

This will give us a slightly different proposition:

Proposition 3.4.2 $\forall a, b, c. a + b < c \rightarrow \exists s, t : \texttt{gcd}'(a, b, c) = a.s + b.t$, *where* a *and* b *are natural numbers, except zero and* s *and* t *are natural numbers.*

Proof 3.4.4 *Induction on* c*:*

1. *Base case* $c = \texttt{Zero}$ *gives us the equation* $a+b < \texttt{Zero}$*, which is evidently false and leaves the proposition true.*
2. *Induction hypothesis:* $\forall a, b a + b < c \rightarrow \texttt{gcd}'(a, b, c) = a.s + b.t$*. Now we have to prove that* $\forall a, b a + b < \texttt{Suc } c \rightarrow \texttt{gcd}'(a, b, \texttt{Suc } c) = a.s + b.t$*. In order to prove that we do a case analysis on all the possible reductions of* `gcd`*.*
3. • *Case* $\texttt{gcd}'(\texttt{Zero}, b)$*: Choose s=0,t=1.*

- *Case* $\texttt{gcd}'(\texttt{Suc}\ x, \texttt{Zero})$*: Choose s=1,t=0.*
- *Case* $\texttt{gcd}'(\texttt{Suc}\ x, \texttt{Suc}\ y)$*: In order to be complete we should split this in two cases, but we will show for one case that the equation holds and by symmetry the other one holds as well. We apply the induction hypothesis to* $\texttt{gcd}'(x - y, \texttt{Suc}\ y, c)$*, which gives:* $x - y + \texttt{Suc}\ y < c \rightarrow \texttt{gcd}'(x - y, \texttt{Suc}\ y, c) = (x - y).s' + (\texttt{Suc}\ y).t'$*. We know that* $\texttt{Suc}\ x + \texttt{Suc}\ y < \texttt{Suc}\ c$*, which makes* $\texttt{Suc}\ x < c$ *true. So we know:* $(x - y).s' + (\texttt{Suc}\ y).t'$*. Now choose* $s' = s$ *and* $t' = t + s$*. Then we get* $(x-y).s+y.t+y.s+s+t = x.s+y.t+t+s = \texttt{Suc}\ x.s+\texttt{Suc}\ y.t$

However that leaves the problem of proving the equivalence of the two functions.

So we also have to prove that:

Proposition 3.4.3 $\forall a, b, c.a + b > c \rightarrow \texttt{gcd}'(a, b, c) = \texttt{gcd}(a, b)$.

Proof 3.4.5 *Using induction on c, and then splitting different cases we easily get the desired result.*

- *Base case:* $m + n < \texttt{Zero} \rightarrow \texttt{gcd}'\ m\ n\ \texttt{Zero} = gcd\ m\ n$ *is true because* $m + n < \texttt{Zero}$ *is false.*
- *Induction hypothesis:* $m + n < z \rightarrow \texttt{gcd}'\ m\ n\ z = \texttt{gcd}\ m\ n$ *To prove:* $m + n < \texttt{Suc}\ z \rightarrow \texttt{gcd}'\ m\ n\ (\texttt{Suc}\ z) = \texttt{gcd}\ m\ n$ *There are two cases, either* $m = \texttt{Zero}$ *or* $m = \texttt{Suc}\ x$*.*
  - *If* $m = \texttt{Zero}$ *then both* gcd *functions reduce to n.*
  - *However, if* $m = \texttt{Suc}x$*, then there are again two cases, either* $n = \texttt{Zero}$ *or* $n = \texttt{Suc}\ y$
    - *If* $n = \texttt{Zero}$*, then both functions reduce to Suc m*
    - *If* $n = \texttt{Suc}\ y$*, then you get either* $\texttt{gcd}(x-y, \texttt{Suc}\ y)$ *or* $\texttt{gcd}(\texttt{Suc x}, y-x)$*. To both one can apply the induction hypothesis, provided that the equations* $\texttt{Suc}\ x + y - x < z$ *and* $x - y + \texttt{Suc}\ y < z$ *hold. We know* $\texttt{Suc}(x + y) < z$*, therefor surely* $\texttt{Suc}\ x < z$ *and* $\texttt{Suc}\ y < z$*.*

As witnessed in this rather laborious proof of a simple property of the greatest common divisor, the induction tactic within Sparkle needs to be improved. In this case it was still possible to produce a proof, but there are cases, like mutually recursive functions, that can not be handled using this method.

Induction methods are needed that express the invariants present in the function-definition. Moreover, induction methods are needed that can handle mutually recursive functions as well as functions that do not terminate.

3.5 Summary

We have described that Sparkle is a dedicated theorem prover for Clean. Programs for which proofs are constructed need no translation into a specification language. This makes reasoning on the source code level possible for the programmer.

The property of a program one tries to prove is called a goal.

Tactics correspond with formal mathematical reasoning steps. They manipulate goals, introduce hypotheses or variables into the context of that goal, or create subgoals.

Building a proof within Sparkle consists of repeatedly applying tactics until no goals are left to prove.

We have also shown how Sparkle handles proofs by giving an example proof for a boolean operator. In this proof, a property was specified for the boolean-operator. In order to progress within the proof, a tactic is chosen which introduces a variable into the environment of the goal. This leads to a transformation of the goal into simpler subgoals. One can apply other tactics to these subgoals. Once all the subgoals are proven, the original property has been proven.

Finally we have addressed how Sparkle used to be before the new inductive and co-inductive tactics were introduced. Since this thesis is about improving the support for proofs on recursive programs, we showed how Sparkle used to handle proofs by induction.

Techniques for co-induction were not included in the previous version of Sparkle and only two methods of induction were allowed for: Induction on integers and structural induction on inductively defined data types.

By giving an example of a simple property, it was shown that the induction technique, as it was implemented in the previous version of Sparkle, was cumbersome and insufficient.

Chapter 4

Mathematical background

"Structures are the weapons of the mathematician." – Nicolas Bourbaki

4.1 Introduction

In this chapter, we will try to familiarize the reader with the mathematical notions that underpin induction an co-induction. That is, as far as is needed to understand the reasoning behind the inductive and co-inductive methods described in the following chapter.

We address Set Theory and Category Theory. The reader will need to have a notion of these theories in order to understand that induction and co-induction are valid proof methods.

Set Theory is used to define an inductive proof principle in section 4.2. We explain how Set Theory is used, induction as a mathematical theorem is proven and we show three kinds of induction to give the reader a better notion on what induction is and what kind of proof methods induction allows for.

The main importance of Category theory is, that is used to define a co-inductive proof principle in section 4.3. An inductive proof principle based on Category Theory is shown primarily to help understand the co-inductive proof principle later on. The section ends with a subsection on bi-simulation, as bi-simulation is a co-inductive technique used in the tactics we implemented in Sparkle. The chapter will be concluded with a short summary.

4.2 Inductive proof principle

Inductive reasoning as used in mathematics should not be confused with the inductive reasoning within philosophy. Although the ideas are the same: "Induce a general property from instances of that property", within mathematics inductive reasoning has a very specific meaning.

The general notion of induction is based upon the idea that if there is a set of ordered elements for which one tries to prove a certain property, it will suffice to prove it for the minimal elements of that set and then prove that the property holds for an arbitrary element under the assumption that it holds for all smaller elements.

This means that the set must have minimal elements, or in other words, the ordering relationship must be well-founded.

Note that in set theory an ordering must be reflexive. This means that xRx must be part of ordering.

Definition 4.2.1 *A relationship R is well founded if there is no infinite chain of descending elements in that relationship:*
$\neg(\forall a, b.R\ a\ b \rightarrow \exists c.R\ c\ a)$

Or alternatively:

Definition 4.2.2 *A relationship R is well founded when* $\forall P.(\exists w.P(w)) \rightarrow \exists min, P(min) \wedge \forall b.b \leq min \rightarrow \neg P(b)$

In order to show that a certain property P holds for all x, we have to show that $P(s)$ follows from the assumption that $P(t)$ holds for all $t \leq s$. Mathematically:

Theorem 4.2.3 *Let $\leq$ be a well-founded partial order, then* $\forall s[\forall t[t \leq s \rightarrow P(t)] \rightarrow P(s)] \rightarrow [\forall x P(x)]$

Proof of this theorem is not very complicated and can probably be found in many texts on set theory.

Proof 4.2.4

- *Assume $<$ is well founded.*
- *Assume* $\forall t, s.t \leq sP(t) \rightarrow P(s)$
- *Assume* $\exists x.\neg P(x)$
- *Since $\leq$ is well founded, we can conclude by means of the definition 3.1.2 that* $\exists min.\neg P(min)$
- *Therefore,* $\forall t.t \leq min \rightarrow P(t)$*, thus $P(min)$ is true.*
- *However, this is in direct contradiction with the assumption* $\exists x.\neg P(x)$*. This means that* $\forall x.P(x)$ *holds.*

This is also called strong induction. Weak induction lets one use immediate predecessors, like in mathematical induction.

4.2.1 Mathematical induction

Mathematical induction is a special case of well-founded induction on natural numbers, where the relationship used for the ordering is the successor relationship and the minimal element is 0.

It boils down to the method that if you have a proposition $P(n)$, you prove it holds for 0, then assume it holds for n and proceed to prove it holds for $n + 1$.

Definition 4.2.1 $P(0) \rightarrow (P(n) \rightarrow P(n+1)) \rightarrow \forall x.Px$

Example 4.2.5 *For instance, we want to prove that* $\forall n.0+1+2+\ldots+n = \frac{n(n+1)}{2}$*, where n is a natural number. Using induction this would reduce to proving the next two properties:*

- $P(0)$*:* $0 = \frac{0(0+1)}{2}$*, which is clearly true by calculation.*
- $P(n) \rightarrow P(n+1)$*:* $\forall n.0+1+2+\ldots+n = \frac{n(n+1)}{2} \rightarrow 1+2+\ldots+n+(n+1) = \frac{(n+1)(n+2)}{2}$*. This means that we may assume* $\forall n.0 + 1 + 2 + \ldots + n = \frac{n(n+1)}{2}$*, commonly called the induction hypothesis, to be true when we attempt to prove the property that is on the right side of the arrow. Since we can rewrite* $\frac{(n+1)(n+2)}{2}$ *to* $\frac{n(n+1)+2(n+1)}{2}$ *and subsequently to* $\frac{n(n+1)}{2} + (n + 1)$*. Using this equality, combined with substitution of the induction hypothesis into the property proves the statement.*

4.2.2 Structural induction

Structural induction is a special case of well-founded induction. Given a set of expressions S, the partial order chosen is the relation "is a subexpression of". This is well-founded because expressions cannot be circular.

Suppose we have a set of blocks B_1 and B_2. If we want to connect them we can put them next to each other. If we want to prove that a certain property holds for all of the structures we can build this way we can use structural induction.

First one has to prove the property for the individual blocks B_1 and B_2. The inductive part of the proof is in the reasoning step that follows. If we assume that the property holds for the smaller elements B_1 and B_2 and we can prove that the property still holds when we construct a bigger structure by combining two of them, we can conclude that it holds for any structure that can be built using those blocks and those construction rules.

4.2.3 Course of values and other data types

In the previous examples we have just looked at induction schemes that are derived from an ordering that is suggested by the construction of the data type. However, that might not always be enough.

Example 4.2.6 *Take for instance the well known fibonacci series. Those numbers are recursively defined as this:*

```
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

It is clear that induction proofs on this sequence probably need an induction scheme that not only depends on the $n-1$ as is the case with regular mathematical induction, but $n-2$ as well.

Mathematical induction is an incarnation of the induction theorem, with the following ordering relation defined by the predecessor function. If we define the ordering to contain not only the predecessor function, but the predecessor of the predecessor, we get the following set of equations defining that relationship:

- $n-1 < n$
- $n-2 < n$
- $\neg\exists(1 < t)$
- $\neg\exists(0 < t)$

Using these in the definition of well-founded induction gives us the next scheme.

Proposition 4.2.7 $(P(0) \wedge P(1) \wedge ((P(n-1) \wedge P(n-2)) \rightarrow P(n))) \rightarrow \forall x.P(x)$

This new relationship is well-founded. In fact, one can construct those new well-founded relationships from the ones that are already established.

For instance, one can prove that the total closure operation on a relationship R will be well-founded, provided that the relationship R already is.

Therefore, the total closure of the predecessor relationship, which gives us the familiar $<$ relationship will lead to another induction scheme. In this case scheme that is also known as "course-of-values".

Proposition 4.2.8 $((\forall t.(t < n) \rightarrow P(t)) \rightarrow P(n)) \rightarrow \forall x.P(x)$

So far we have seen how we can express an inductive proof principle as a mathematical theorem based in set theory. We have shown by means of different kinds of induction schemes that this well founded induction theorem can be instantiated with different relationships. These incarnations yield different kinds of induction schemes like mathematical induction, structural induction or course of values induction.

4.3 Co-inductive proof principle

The notion of co-induction is a bit harder to capture without delving into deep mathematics. As earlier mentioned, co-inductive reasoning is closely related to inductive reasoning and usually deals with infinite structures.

In induction, the notion of having smallest elements is of prime importance. When reasoning about co-induction these elements are not available anymore. Instead we look at the behavior of functions in general.

First of all, we have to get familiar with the rather arcane terminology of category theory.

Category theory studies the abstract properties of algebras. At the heart of understanding co-inductive proof techniques lie the notions *functor*, and *initial* and *final* objects in a category. A good introduction to this topic can be found in the introductory article on category theory by Jacobs and Rutten. [2]

Induction as a definition principle and as a proof principle is introduced and reformulated in an abstract way, using initiality. This description of induction in terms of algebras can be seen as having a dual easily and leads us to the theory of co-algebras.

The categorical approach to co-induction is described in the final section of this chapter.

4.3.1 Category theory

It is not the intention to give a full introduction to category theory. However, some terms need to be explained.

Definition 4.3.1 *A category is a combination of a set of* objects *and sets of* morphisms *between those objects.*

- *For every pair X, Y of objects, there is a set of morphisms $Hom(X, Y)$. If f is a morphism from X to Y, we write $f : X \to Y$.*
- *For every object X there is a morphism Id_X in $Hom(X, X)$, called the identity.*
- *For every (X, Y, Z) of objects, there is a map $Hom(X, Y) \times Hom(Y, Z) \to Hom(X, Z)$. This is called* composition*, also written: if $f : X \to Y, g : Y \to Z$ then $(g \circ f) : X \to Z$.*

Identity, morphisms and composition satisfy the axioms of associativity and identity:

- *If $f : X \to Y, g : Y \to Z, h : Z \to W$, then $h \circ (g \circ f) = (h \circ g) \circ f$.*
- *If $f : X \to Y$, then $Id_Y \circ f = f$ and $f \circ Id_X = f$*

The category *Set* has as objects sets and the usual functions, but it is also possible to create a deductive system as a category with formulas as objects and proofs as morphisms. Although category theory is definitely not restricted to sets, for simplicity we will restrict ourselves to the category of sets from now on.

Definition 4.3.2 *An isomorphism is a morphism for which the following holds: Let $f : X \to Y$ be a morphism, then there exists a morphism $g : Y \to X$ such that $g \circ f = Id_X$ and $f \circ g = Id_Y$*

One of the concepts that needs some explaining is the notion that many operations on sets are *functorial*. This means that they not only acts on sets, but also on functions between sets.

For example, suppose we have a set A', created by the algebraic data constructors $Lista$, which is a set of finite lists of elements of set A. For a function f from A to another set of elements B one can define a function $g : List\ A \to List\ B$ between the corresponding sets of lists. It constructs a mapping from a finite list $(a_1, \ldots, a_n)$ of elements of set A to the list $(f\ a_1, \ldots, f\ a_n)$ of elements of set B, by applying f elementwise. This function g preserves the identity and composition functions. This means that if we have the identity functions $id_A : A \to A$ for A and $id_{List\ A} : List\ A \to List\ A$, then $g(id_A) = id_{List\ A}$. An analogous property holds for the composition function $\circ$.

Now we define for two sets X, Y the Cartesian product $X \times Y$. $X \times Y = \{(x, y) | x \in X \wedge y \in Y\}$.

The product operation that creates a Cartesian product from two sets X and Y not only applies to sets, but also to functions. Suppose we have functions $f : X \to X'$ and $g : Y \to Y'$, then one can define a function $h : X \times X' \to Y \times Y'$ by defining $h(x, y) = (f(x), g(y)$. One can also write this function, using $\times$ in an overloaded fashion, as $f \times g : X \times X' \to Y \times Y'$.

One can verify that this operation $\times$ preserves both the identity and compository properties.

- $id_X \times id_y = id_{X \times Y}$
- $(f \circ h) \times (g \circ k) = (f \times g) \circ (h \times k)$

This expresses the *functorial* behavior of the $\times$ operation: It applies to functions as well as sets and it does in such a way that it preserves the identity and composition.

Many more operations are functorial: The disjoint union, or co-product, the powerset operation, identity operation.

A very useful set of functors are called *polynomial* functors. They are built up using constants, identity functors, products, co-products and powersets. They are described by giving their actions on sets, like $T(X) = X + (C \times Y)$.

By definition, 1 represents the singleton set, while 0 represents the empty set.

For every set there is exactly one function $X \to 1$. This means that 1 is *final* in the category of sets and functions. Every function $1 \to X$ represents an element of X.

For every set X there is exactly one function $0 \to X$, namely the empty function. This property is the *initiality* of 0.

Definition 4.3.3 *Let T be a functor. An* algebra *of T is a pair consisting of a set U and a function $a : T(U) \to U$.*

U is called the *carrier* of the algebra and the function a the algebra structure or operation.

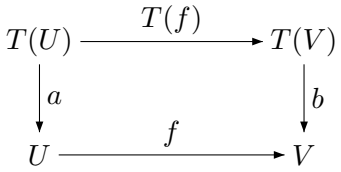
Let us take the successor function for example. It is used to model natural numbers and represented within Clean as the following algebraic data type:

Example 4.3.4 `::Nat :== Zero | Suc Nat`

These functions $0 : 1 \to \mathbb{N}, S : \mathbb{N} \to \mathbb{N}$ on the natural numbers are an algebra: $[0, S] : 1 + \mathbb{N} \to \mathbb{N}$ of the functor $T(X) = 1 + X$.

Homomorphism of an algebra are structure preserving functions between algebras. They are functions between the carrier sets of the algebras that commute with the operations.

Definition 4.3.5 *Let T be a functor with algebras $a : T(U) \to U$ and $b : T(V) \to V$. A homomorphism of algebras (or algebra map) from (U, a) to (V, b) is a function $f : U \to V$ between the carrier sets which makes the diagram commute with the operations $f \circ a = b \circ T(f)$.*



4.3.2 A categorical approach to induction

As we will see soon, induction can be explained in terms of the categorical theory of algebras. Although we already have a sound underpinning of induction within the framework of well founded orderings, understanding the categorical approach will help us better understand the mathematical definition of co-induction within category theory. It also helps us to see the similarities between induction and co-induction.

To better understand the category theoretic approach to data structures and programs, we first show how regular induction is handled within this framework.

Example 4.3.6 *Suppose we have a data structure for lists, defined as:*

```
List a :: Nil | El a (List a)
```

Consider a fixed data set of A, then the set A' is the set of all finite sequences of elements of the data set A. It is possible to inductively define a length function over that set.

```
length :: (List a) -> Int

length Nil   = 0
length El e l = 1 + (length l)
```

As seen earlier, a typical induction proof will proceed along the lines of first having to prove the property for the empty list, $P(Nil)$ and then proving $\forall a\ l.P(l) \rightarrow P(\texttt{El}\ a\ l)$. The induction proof makes use of the property that all finite lists of elements of A' can be constructed using the two operations $\texttt{Nil} \in A'$ and $\texttt{El} : A \times A' \rightarrow A'$.

In an inductive definition of a function f, one defines the value of f on all constructors.

The notion of construction is a pivotal point within the algebra of inductive structures. The dual within co-inductive structures is observation.

But first we have to define what *initiality* means.

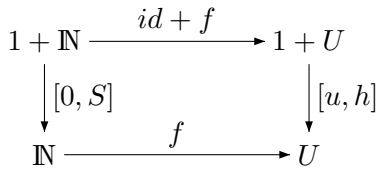
Definition 4.3.7 *An algebra $a : T(U) \rightarrow U$ is* initial *if for each algebra $b : T(V) \rightarrow V$ there is a unique homomorphism of algebras from (U, a) to (V, b).*

This uniqueness has two aspects:

- The existence of an algebra map from the initial algebra to another algebra. The existence can be used as an inductive definition principle.
- The uniqueness of this algebra. This means that any two algebras going from an initial algebra to another algebra have to be the same. This corresponds with an inductive proof principle.

For the earlier mentioned natural numbers the map $[0, S] : 1 + \mathbb{N} \rightarrow \mathbb{N}$, forming an algebra of the functor $T(X) = 1 + X$, to be the initial algebra of this functor we have to prove initiality.

We assume an arbitrary set U carrying an algebra structure $[u, h] : 1 + U \rightarrow U$. We define a homomorphism $f : \mathbb{N} \rightarrow U, where f(n) = h^{(n)}(u)$. Thus $f(0) = u$ and $f(n + 1) = h(f(n))$.



Proof 4.3.8 *To prove that this is a homomorphism we have to show that the operations make the diagram commute:* $f[0,S] = [u,h] \circ (id + f)$. *That they do is shown by considering that an arbitrary element* $x \in 1+\mathbb{N}$ *is either* $< 0, * >$ *or* $< 1, n >$, *according to the definition of the co-product. So we have to prove:*

- $f([0,S] < 0, * >) = f(0) = u = [u,h] < 0, * >= [u,h]((id + f)(< 0, * >)$
- $f([0,S] < 1, n >) = f(S(n)) = h(f(n)) = [u,h] < 1, f(n) >= [u,h](id + f) < 1, n >$

Now we have proven that f is a homomorphism, we still have to prove that this f is unique.

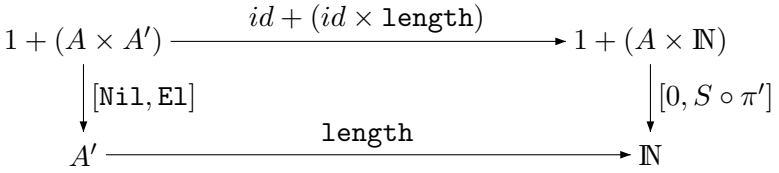
Suppose there is another $g : \mathbb{N} \to U$, *that also commutes, then applying the definition of the co-product we get that* $g(0) = u$ *and* $g(n+1) = h(g(n))$. *By induction on n we get that* $f(0) = u = g(0)$ *and if we assume* $f(n) = g(n)$, *then* $g(n+1) = h(g(n)) = h(f(n)) = f(n+1)$.

Lists as an example

In the same way as has been done for the natural numbers, the representation of lists, $a : [\texttt{Nil}, \texttt{El}] : 1+(A \times A') \to A'$, can be shown to be an initial algebra as well.

Using initiality as a defining property

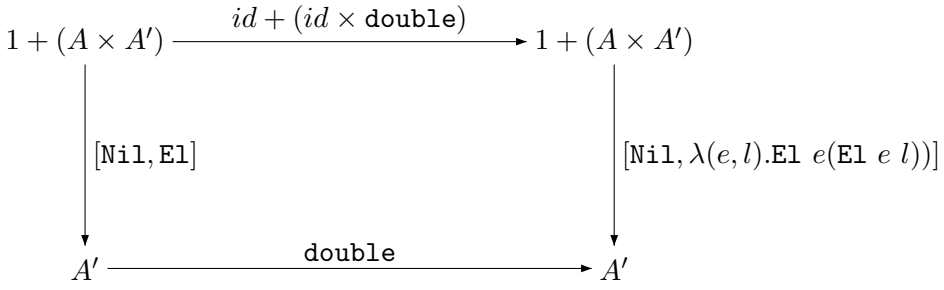
Suppose we want to define a length function for lists, then we can use the diagrams we have used earlier, but instead of the general function f, we use our length function:



The algebraic structure used on $\mathbb{N}$ can be used to derive the defining clauses for the length function:

Definition 4.3.9
`length Nil = 0`
`length (El e l)` $= S(\pi'(id \times$ `length`$)($`e`$,$ `l`$))$
$= S($`length`$(\pi'($`e`$,$ `l`$))) = S($`length l`$) =$ `1 + length l`

The same can be done for the function that creates a duplicate list by concatenating its arguments to itself:



The defining functions derived from this diagram are:

```
double EmptyList     = EmptyList
double (Element e l) = Element e (Element e (double l))
```

Using category theory for induction proofs

We have seen in the previous section that we can use the description of functions within the context of category theory automatically leads us to proper definitions of those functions. We can also use the same theory to conduct induction style proofs of properties of those programs.

Suppose we want to prove the following property:

Property 4.3.10 $\forall ls.$`length`$($`double`$\ ls) = 2 *$ `length`$($`ls`$)$

The old fashioned way
In order to compare the induction proof one would get using the initiality of an algebra, to regular induction we first provide the proof as it would be when regular well-founded induction on the structure of the list would be used.

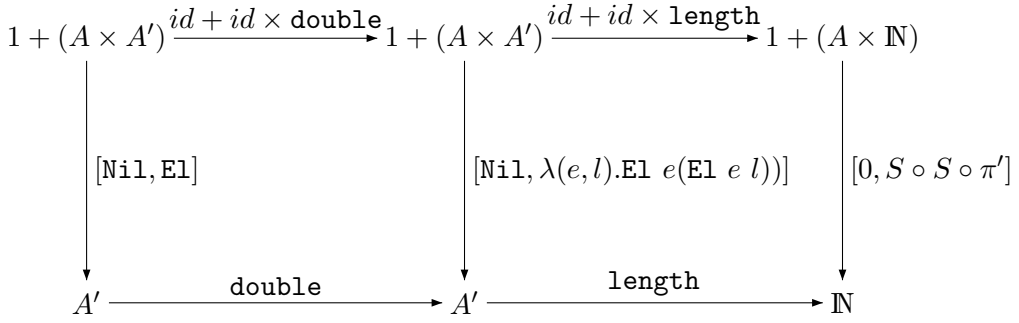
Proof 4.3.11 *A regular structural induction proof would make use of the structure of lists. They are either an empty list,*`Nil`*, or an element followed by another list,*`El e (List l)`*. This would raise two proof obligations:*

- *The proof of the property for the smallest element* `Nil` *follows immediately by reducing the* `double` *function in the property:*
  `length` (`double Nil`) = `length Nil` = `0` = `2 * 0` = `2 * length Nil`.

- *The inductive part of the proof would be proven like this:*
  `length` (`double` (`El e ls`))
  = `length` (`El e` (`El e` (`double ls`)))
  = `1` + `1` + `length ls`.
  By applying the induction hypothesis to this statement we get:
  `2` + `2` ∗ (`length ls`) = `2` ∗ (`1` + `length ls`) = `2` ∗ (`El e ls`).

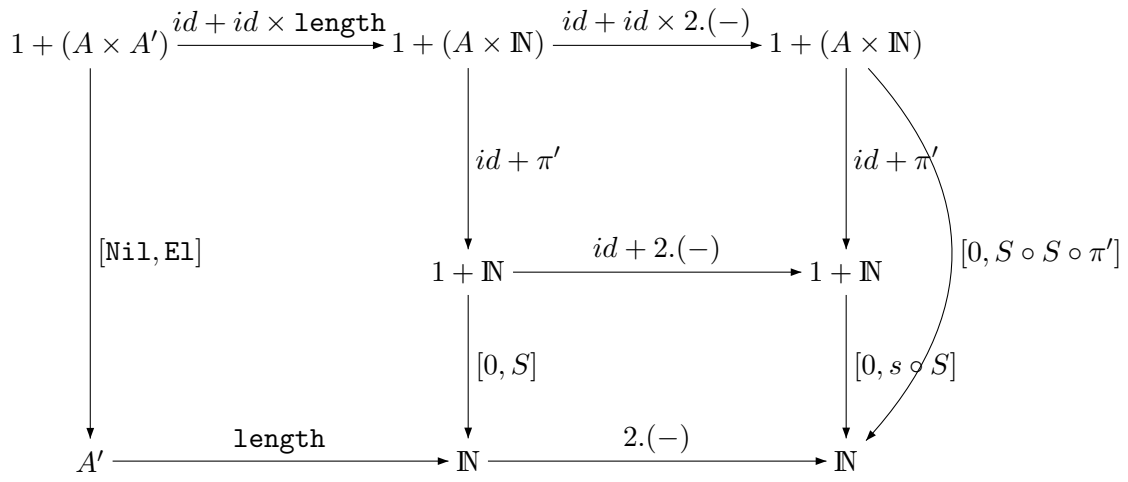
Using category theory
Proving this property by using the initiality of the algebras results in having to prove that both `length`∘`double` and $2*(-)\circ$`length` are homomorphisms from the initial algebra $(A', [\texttt{Nil}, \texttt{Element}])$ to the algebra $(\mathbb{N}, [0, S \circ S \circ \pi'])$. If they are, they must be equal by initiality.

- So first we have to check that `length` ∘ `ddouble` is a homomorphism:



The rectangle on the left commutes by the definition of double and the commutation of the rectangle on the right follows from the definition of length.

- We also have to check that $2.(-) \circ$ `length` is a homomorphism:



The square on the left again commutes by the definition of length. Computation yields that the upper right square commutes as well. The lower right square can be interpreted of a function definition of $2.(-) : \mathbb{N} \to \mathbb{N}$. The definitions derived from the algebra are:

```
(.) 2 0    = 0
(.) 2 S(n) = S (S 2.n)
```

The existence of multiplication was assumed in our earlier regular induction proof.

Using this extended proof we have seen how we can use category theory to conduct induction proofs of functional programs by making use of the initiality of an algebra. In the next section we will explain the dual of an algebra, a co-algebra and how the finality of a co-algebra can be used to conduct co-inductive proofs.

4.3.3 A categorical approach to co-induction

In this section we will explain the dual of an algebra, a co-algebra and how the finality of a co-algebra can be used to conduct co-inductive proofs.

Definition 4.3.12 *For a functor T a co-algebra is a pair (U, c) consisting of a set U and function $c : U \to T(U)$.*

Like for algebras the set U is the carrier and the function c the structure or operation of the algebra.

The difference between an algebra and a co-algebra is the difference between construction and observation. An algebra tells us how to construct

the elements in set U. A co-algebra tells us how we can observe the elements in set U. It is not known to the observer how the elements in a co-algebraic structure are made within U, the only thing the observer can tell is how it reacts to operations acting on U.

Due to its origin in the theory of processes, the set U is also called the *state space*.

Consider a functor $T(X) = A \times X$, where A is a fixed set. A co-algebra $U \to T(U)$ consists of two functions: $value : U \to A$ and $next : U \to U$. Given an element in U we can either produce an element in $A : value(a)$ or proceed to a next element in $U : next(U)$. Each possible sequence that can produce a value and for each element in U is a possible observation of U.

If two elements of U produce the same sequence of observable elements, we call those elements, even though they are not the same elements in U, bi-similar.

Definition 4.3.13 *A homomorphism of co-algebras from a co-algebra $U_1 \longrightarrow_{c1} T(U_1)$ to another co-algebra $U_2 \longrightarrow_{c2} T(U_2)$, consists of a function $f : U_1 \to U_2$, that commutes with the operations $c2 \circ f = T(f) \circ c1$.*

$$\begin{array}{ccc} U_1 & \xrightarrow{\quad f \quad} & U_2 \\ \downarrow{\scriptstyle c_1} & & \downarrow{\scriptstyle c_2} \\ T(U_1) & \xrightarrow{\quad T(f) \quad} & T(U_2) \end{array}$$

Definition 4.3.14 *A final co-algebra $d : W \to T(W)$ is a co-algebra such that for every co-algebra $c : U \to T(U)$ there is a unique map of co-algebras $(U, C) \to (W, D)$.*

The principle of finality for co-algebras, which is the dual of initiality for algebras, allows the definition of functions into the state space.

So how does this work out for the list example?

Again we take a functor $T(X) = A \times X$, where A is a fixed data set. The final co-algebra of this functor is the set $A^{\mathbb{N}}$ of infinite lists of elements from A, with the following structure: $< \mathtt{hd}, \mathtt{tl} >: A^{\mathbb{N}} \to A \times A^{\mathbb{N}}$. The functions `hd` and `tl` are defined as $\mathtt{hd}(\alpha) = \alpha(\mathtt{0})$ and $\mathtt{tl}(\alpha) = \lambda x.\alpha(x + 1)$.

To prove that this is a final co-algebra we have to show that it is a homomorphism and that it is unique.

The pair of functions $< \mathtt{hd}, \mathtt{tl} >: A^{\mathbb{N}} \to A \times A^{\mathbb{N}}$ is an isomorphism.

Suppose we have an arbitrary co-algebra $< value, next >: U \to A \times U$. Then a homomorphism given by $f : U \to A^{\mathbb{N}}$; for all $u \in U$ and $n \in \mathbb{N}$, $f(u)(n) = value(next^n(u))$.

It follows that $\texttt{hd} \circ f = value$ and $\texttt{tl} \circ f = f \circ next$. This makes it a map of co-algebras. It can easily be checked that f is unique in satisfying these two equations.

Now that it is established that we have uniqueness it is possible to construct co-inductive proofs by the finality of the structure.

4.3.4 Bi-simulation and co-induction

Instead of the rather cumbersome proofs by the finality of, there is another proof method that can be used to prove properties by co-induction. The method employs bi-simulation.

Definition 4.3.15 *Let T be a functor and $c : U \to T(U)$ a co-algebra. A bisimulation on U is a relation R on U, for which there exists a co-algebra structure $\gamma : R \to T(R)$ such that the two projection functions $\pi_1 : R \to U$ and $\pi_2 : R \to U$ are homomorphisms of co-algebras:*

$$U \xleftarrow[\pi_1]{} R \xrightarrow{\pi_2} U$$

$$T(U) \xleftarrow[T(\pi_1)]{} T(R) \xrightarrow{T(\pi_2)} T(U)$$

Theorem 4.3.16 *Let $c : Z \xrightarrow{\cong} T(Z)$ be the final co-algebra. $\forall z, z' \in P.R(z, z')$ for some bi-simulation R on $Z \to z = z'$.*

The correctness of this theorem follows from the finality of the co-algebras.

For the functor $T(X) = A \times X$, that has the final co-algebra of infinite lists of $A^{\mathbb{N}}$ with structure $< \texttt{hd}, \texttt{tl} >$, a bi-simulation is the relation R on $A^{\mathbb{N}}$ satisfying:

$$R(\alpha, \beta) \to \begin{cases} \texttt{hd}\ \alpha = \texttt{hd}\ \beta, \texttt{and} \\ R(\texttt{tl}\ \alpha, \texttt{tl}\ \beta) \end{cases}$$

We consider R to be a set of pairs and supply it with a $A \times (-)$ co-algebra structure by defining the function $\gamma : R \to A \times R$ as $(\alpha, \beta) = (\texttt{hd}\ \alpha, (\texttt{tl}\ \alpha, \texttt{tl}\ \beta))$. We have to show that the two projection functions $\pi_1 : R \to A^{\mathbb{N}}$ and $\pi_1 : R \to A^{\mathbb{N}}$ are homomorphisms. This is easily proven. We can then conclude from the uniqueness of the finality of $A^{\mathbb{N}}$ that $\pi_1 = \pi_2$ and consequently that $R(\alpha, \beta) \to \alpha = \beta$.

4.4 Summary

It is obvious that inductive and co-inductive proof principles need a sound mathematical underpinning. We have established these foundations by researching two different mathematical theories.

- We looked at set theory to provide us with the theorem of well founded induction. This theorem states that for every well founded relationship an induction scheme can be created. This induction scheme consists of two elements. First one has to prove a predicate for the minimal elements for that relationship. Then smaller elements of this relationship can act as induction hypothesis in order to prove the predicate for the bigger elements within that relationship.

  Once these proof obligations are fulfilled the predicate can be considered proven for all elements from the set for which the well founded ordering was defined.

- Category theory provides a justification for the principle of co-inductive proofs. We prove how bi-simulations, relationships between the observed behavior of functions, can be used as a proof principle for non-terminating recursive programs.

Both of these proof principles are used to extend Sparkle with tactics that will enable proofs of a wider selection of recursive programs.

Chapter 5

Extending Sparkle

"Civilization advances by extending the number of important operations which we can perform without thinking of them." –Alfred North Whitehead

5.1 Introduction

In chapter three we have shown by means of an example how laborious it can be to construct proofs using the tactics that Sparkle provides the programmer. Moreover, we have shown that proofs for some recursive programs were impossible. Specifically, programs that use mutually recursive algebraic data types and programs that do not terminate.

Since our primary goal is to facilitate reasoning on a wider range of recursive programs there is a need for new tactics that will help the programmer construct proofs for these programs.

In this chapter, we will show how we have extended Sparkle with four more tactics. These tactics make it possible to reason about mutually recursive programs as well as recursive programs that do not terminate.

- First of all, we introduce a general method of deriving induction schemes for mutually recursive algebraic data types in section. This tactic is in no way new, but it is a large improvement, because Sparkle could only derive induction schemes for directly recursive algebraic data types before.
- Secondly, we describe a new method for deriving induction schemes based on a (possibly) well founded ordering. This ordering is derived from the function definitions and closely mimics the call flow of the functions used. Before we arrive at this new method, we will first look at two existing methods in order to compare it to the technique we introduce. These methods are proving termination based on the size change principle and the derivation of induction schemes from function definitions.
- Thirdly, we introduce a co-inductive tactic that allows the programmer to construct proofs using a bisimulation on the structure of algebraic data

types. Again, this is hardly novel, but broadens the scope of potential proofs that can be made within Sparkle considerably.

- And finally we come up with a novel method to construct co-inductive proofs on recursive functions using a bisimulation on the structure of the recursive function calls.

These four methods will be addressed in section 5.2 up to and including section 5.5. In each section will be explained why, and how, the method extends Sparkle as well as the way the method functions. We will show for the both derivation of an induction scheme for mutually recursive algebraic data types, as well as the derivation of an induction scheme, using a well founded ordering suggested by the respective function definitions, that these methods are mathematically sound. The first co-inductive tactic will also be shown to be correct. The last method to construct a co-inductive proof principle is speculative and we will not provide a proper justification for it's use.

We will show by means of examples how the methods function as well as why they extend Sparkle, as well as how they are improvements upon each other.

5.2 Induction on mutually recursive algebraic data types

In this section a we will describe why we have implemented induction on mutually recursive algebraic data types. What the theory behind it is and how it is implemented. By means of an example we show how we have expanded the realm of possible proofs in Sparkle.

5.2.1 Algebraic data types

Although many functional programs are directly recursive, there are many programs that are mutually recursive. Especially within compiler construction mutual recursion is often used to express operations on different kinds of language constructs as we will see in the final example of this section.

Since Sparkle only allowed induction proofs on directly recursive algebraic data types, extending this to the generation of induction schemes for mutually recursive would mean a significant improvement.

Mutual recursion in algebraic data types require proof techniques that can handle the inter-dependency between the constituent parts. A well known method, employed for instance in the HOL theorem prover [13], uses multi-predicate induction.

Before we delve into the theoretical justification of the derivation of induction schemes for mutually recursive functions, we will show by means

of a few examples what mutually recursive algebraic data types are and what kind of induction schemes would be needed to prove properties of programs using those data types.

An algebraic data type is a type which is constructed out of other types by means of a constructor. These definitions are allowed to be circular.

Example 5.2.1 *Take for instance the following definition of a binary tree.*

```
Tree a = Leaf a
       | Branch (Tree a) (Tree a)
```

This is a definition of a generic tree. The variable `a` *signifies an arbitrary type. This means this algebraic data type can be used for a binary tree of integers or any other type.*

The constructors are `Leaf` and `Branch`. The | symbol denotes a choice, which means that a `Tree` can either be a `Leaf` or a `Branch`.

One can see the recursion in the definition of a `Branch`. The constructor is followed by two references to a `Tree a`.

This kind of direct recursion is what Sparkle was able to handle all along. It could derive an induction scheme for these kind of algebraic data types.

However, if we specify another algebraic data without direct recursion, Sparkle would not be able to derive an induction scheme.

Example 5.2.2 *Take for instance the following example of a rose that is constructed out of a petal or branches of an indeterminate amount of roses.*

```
Rose a = Petal a
       | Branch a (List (Rose a))

List a = Nil
       | Node a (List a)
```

Sparkle would not be able to construct an induction scheme for this algebraic data type, because there is no direct recursion. Although there is a reference to `Rose a` *within the definition of* `Rose a`*, it is obscured by a* `List` *type.*

This example is not a mutually recursively defined type, yet Sparkle fails to provide a proper induction scheme for it. It can not look beyond the `List` type to find the recursion. This means that derivation of induction schemes for mutually recursive will not work either. In our next section we will show how we can derive an induction scheme for algebraic data types that are not directly recursive or that are mutually recursively defined.

5.2.2 The tactic

As we have seen in chapter 3, induction schemes can be derived from the well founded induction theorem.

Theorem 5.2.3 *Let $<$ be a well founded partial order, then* $\forall s[\forall t[t < s \rightarrow P(t)] \rightarrow P(s)] \rightarrow [\forall x P(x)]$

In order to derive an induction scheme for a mutually recursive data type, all one has to do is find the right well founded partial order and instantiate it within this theorem.

Examples of functioning tactic

By means of some examples we will show how we can derive an induction scheme from a well founded order. Also, it is shown how we can derive an order from the definition of the algebraic data type.

Example 5.2.4 *Let us look again at the binary tree.*

```
Tree a = Leaf a
       | Branch (Tree a) (Tree a)
```

For this example of a binary tree, the induction scheme that would be generated would be derived from the way the data type is constructed.

There are several deductions about a possible well founded ordering one can make from the definition of the `Tree a` type.

- The smallest element is `(Leaf a)`.
- The next element that can be constructed is `(Branch a (Leaf a) (Leaf a))`. That means that `(Leaf a)` $<$ `(Branch a (Leaf a) (Leaf a))`.
- From `(Branch a (Leaf a) (Leaf a))` it is possible to construct two more elements and we get the next two equations.
- `Branch a (Leaf a) (Leaf a)` $<$ `Branch a (Leaf a) (Branch a (Leaf a) (Leaf a))` and
- `Branch a (Leaf a) (Leaf a)` $<$ `Branch a (Branch a (Leaf a)(Leaf a)) (Leaf a)`

From the well-founded order specified above we can deduce that the following equalities hold:

- $\forall t1, t2.t1 <$ `Branch` $a\ t1\ t2$
- $\forall t2, t1.t2 <$ `Branch` $a\ t1\ t2$

- $\forall t.\neg(t < \texttt{Leaf}\ a)$

Since s, in the well founded induction theorem 5.2.3, can only be a `Branch` or a `Leaf` we can derive the following induction scheme:

Proposition 5.2.5 $\forall y, z : P(\texttt{Leaf}\ y) \wedge (P(t1) \wedge P(t2) \rightarrow P(\texttt{Branch}\ z\ t1\ t2)) \rightarrow \forall x.P(x)$

So far, we have covered familiar territory. Deriving an induction scheme like the one above was already a possibility within Sparkle as mentioned earlier. The second example of the previous section was not possible. Let us see if we can construct an ordering out of the data type definition of `Rose a`.

Example 5.2.6 *Take for instance the following example of a rose that is constructed out of a petal or branches of an indeterminate amount of roses.*

```
Rose a = Petal a
       | Branch a (List (Rose a))

List a = Nil
       | Node a (List a)
```

In this example the definition of the data type rose depends on the definition of a list. The smallest element is `(Petal a)`. From this smallest element we can construct the next one, which is `(Branch a Nil)`.

The next in line is `(Branch a (Node (Petal a) Nil))` and from then on the ordering starts to fan out. There are two ways to construct bigger elements now.

Either add to the list: `(Branch a (Node (Petal a) (Node (Petal a) Nil)))` or add to the rose: `(Branch a (Node (Branch a Nil) Nil))`.

This example would have a mutual induction scheme like this:

Proposition 5.2.7 $\forall P1, P2, a.(P1(\texttt{Petal}\ a) \wedge (P2(l) \rightarrow P1(\texttt{Branch}\ a\ l)) \wedge P2(Nil) \wedge (P2(l) \wedge P1(r) \rightarrow P2(\texttt{Node}\ r\ l))) \rightarrow \forall x, y.P1(x) \wedge P2(y)$

Finally, we will look at some data types that mutually depend on each other for their definition. It is only reasonable to expect that the induction schemes that will be derived will be of a form that supports mutual induction as witnessed in the earlier example.

Example 5.2.1 *Mutually dependent data types*

```
MutRec a = MrNil | MrCons a (RecMut a)
RecMut a = RmNil | RmCons a (MutRec a)
```

This scheme does not have one smallest element. There are two ways you can start constructing elements within this ordering. One can start with either `MrNil` or `RmNil`. Two strains in the ordering then develop:
`MrNil` $<$ `MrCons a` (`RmNil`) $<$ `MrCons a` (`RmCons a` (`MrNil`))
and `RmNil` $<$ `RmConsa`(`MrNil`) $<$ `RmConsa`(`MrConsa`(`RmNil`))
. From this we can deduce a set of equations.

- $\forall t1.t1 <$ `MrCons a` $(t1)$
- $\forall t2.t2 <$ `RmCons a` $(t2)$
- $\neg\exists(t <$ `MrNil`$)$
- $\neg\exists(t <$ `RmNil`$)$

Once again applying those equations to the well-founded induction scheme as defined earlier we get:

$(P1($`MrNil`$) \wedge P2($`RmNil`$)\wedge$
$(P1(t) \rightarrow P2($`RmCons` a $(t)))$
$(P2(t) \rightarrow P1($`MrCons` a $(t))))$
$\rightarrow \forall x, y.P1(x) \wedge P2(y)$

Example 5.2.2 `MutRec a = MrNil | MrCons a (RecCons a)`
`RecCons a = RecCons a (MutRec a)`

The difference with the previous example is small yet potentially important. This ordering will have only one smallest element, `MrNil` and one can construct in either one step or two steps the next element in line `MrCons a (RecCons a (MrNil))`.

The equations

- $\forall t1.t1 <$ `MrCons a` $(t1)$
- $\forall t2.t2 <$ `RecCons a` $(t2)$
- $\neg\exists(t <$ `MrNil`$)$

will yield the following induction scheme:
$(P1(MrNil) \wedge (P2(t) \rightarrow P1($`MrCons` a $(t1)))\wedge$
$(P1(t) \rightarrow P2($`RecCons a` $(t))))$
$\rightarrow \forall x, y.P1(x) \wedge P2(y)$

Generalisation into a tactic

In order to make a tactic that is usable within Sparkle the process of creating a well founded ordering out of the definition of the algebraic data types must be automated.

First of all, we must define what an algebraic data type can be in Clean.

Definition 5.2.8

```
AlgebraicType  = TypeName {TypeVariable}+ '=' TypeDefinition
TypeDefinition = Constructor {TypeOptions}"
TypeOptions    = TypeVariable | AlgebraicType | NativeType
```

One of the properties of Clean is that all the algebraic data type definitions must be guarded with a constructor and each constructor must be unique. This means that we do not have to worry about non-productive data types or ambiguous derivations.

The notation $\{x\}$ means that zero or more occurrences of x are allowed, while $\{x\}+$ means that there are one or more occurrences of x allowed.

A type can be defined by a type-name with optional variables to make the type polymorphic followed by its definition.

The definition lists the several options an algebraic data types can have. A definition option consists of a constructor followed by a list of type-variables, basic types or algebraically defined variables, including itself.

Therefore every algebraic type one can possibly conceive must be constructable and we can simply impose an order on them by the way they are constructed. Since they also have smallest elements, that order will necessarily be well founded.

[Note, it is not required for algebraic data types to have smallest elements, but we can analyze the definition to see if they have]

This order can then be used to construct an induction scheme.

The only restriction to the (mutually) recursive algebraic data type is that at least one of the recursive definitions has a non-recursive part. If not, we don't have a smallest element to build our ordering on.

Definition 5.2.9 • *Let $A = \{D_1 \dots D_n$ be the set of mutually recursive algebraic type definitions.*

- *Let $B = \{E_1 \dots E_n$ be the set algebraic type definitions that are not mutually recursive.*
- *Let $D_k \alpha_{k1} \dots \alpha_{ki} = C_k D_{k1} \dots D_{kj}$ be one of the algebraic type definitions. Then there are four possibilities:*
- $D_{kh} \in \alpha_{kl}, 1 \leq l \leq i, 1 \leq h \leq j$

- $D_{kh} \in A$
- $D_{kh} \in E$
- $D_{kh} \in NativeCleanType$

We construct a well founded relationship $<$ on $A \times A$ for all the tuples $(D_{kj}, D_{kh} \in A)$.

In order to derive an induction scheme from this relationship we have to prove that this relationship is indeed well founded.

Proof 5.2.10 *This relationship is well founded because we can make a mapping function that counts all the constructors. We can prove from the definition of (D_{kj}) that $count(D_{kj}) < count(D_{kh})$.*

Now that we have constructed a well founded relationship we can apply this to the induction theorem. A problem that arises is the fact that Clean expressions, unlike logical predicates are typed.

If we want to prove a predicate on mutually recursive algebraic data types, this means that the property that the predicate describes will be dependent on typed Clean expressions. We assume that the programmer provides different predicates for each of the defined algebraic data types.

Definition 5.2.11 *We define the general predicate $P(x)$, where x is a variable of a type from A, as $P(x) = \bigwedge typeOf(x) = D_n \rightarrow P(D_n)$, where $D_n \in A$.*

In our implementation within Sparkle we immediately introduce the the induction hypotheses whenever possible.

To see how our implementation works we provide an example of Sparkle in action with a simple expression language.

5.2.3 An expression language example

Take for instance a simple expression language consisting of arithmetic and boolean expressions. This example can be found on the HOL web-site [4]. Arithmetic expressions can contain a boolean condition, while boolean expressions may be a comparison between arithmetic expressions.

```
::Aexp a = Var a
         | Num Int
         | Sum (Aexp a) (Aexp a)
         | If (Bexp a) (Aexp a) (Aexp a)
::Bexp a = Less (Aexp a) (Aexp a)
         | And (Bexp a) (Bexp a)
```

[4] http://www.cl.cam.ac.uk/Research/HVG/Isabelle/library/HOL/Induct/ABexp.html

On this small language substitution and evaluation functions are defined. The substitution function performs a substitution, defined as a map from pointers to arithmetic expressions, on an expression. The evaluation function reduces an expression to a value within a certain environment. That environment is defined as a map from pointers to integer values.

```
evala :: (a->int) (Aexp a) -> int
evala e (If b a1 a2)
   | evalb e b = evala e a1
   | otherwise = evala e a2
evala e (Sum a1 a2) = (evala e a1) + (evala e a2)
evala e (Var v) = e v
evala e (Num n) = n

evalb :: (a->int) (Bexp a) -> bool
evalb e (Less a1 a2) = (evala e a1) < (evala e a2)
evalb e (And b1 b2)  = (evalb e b1) && (evalb e b2)

substa :: (a -> Aexp a) (Aexp a) -> Aexp a
substa s (If b a1 a2)= If (substb s b) (substa s a1) (substa s a2)
substa s (Sum a1 a2) = Sum (substa s a1) (substa s a2)
substa s (Var v)     = s v
substa s (Num n)     = Num n

substb :: (a -> Aexp a) (Bexp a) -> Bexp a
substb s (Less a1 a2) = Less (substa s a1) (substa s a2)
substb s (And b1 b2)  = And (substb s b1) (substb s b2)
```

We want to prove that it does not matter whether we evaluate all the substitutions and use that as an environment or we perform all the substitutions first and then evaluate the resulting expression. Formulated within Sparkle it gives us the following property to prove.

Lemma 5.2.12
$\forall$`env` s a.`evala env` (`substa` s a) = `evala`(λx.`evala env` (s x)) $a \wedge$
$\forall$`env` s b.`evalb env` (`substb` s b) = `evalb`(λx.`evala env` (s x)) b

Sparkle's new induction tactic will recognize that the induction will be done on the variables a and b, of a mutual recursive type and offers an induction scheme whereby for each non recursive part of the data types a proof of the property is required, while for each recursive definition an antecedent is constructed. The entire scheme and complete proof resulting from it is a bit too unwieldy to show here, but from the part that is needed to prove the property for when we encounter an `IF` the structure will be readily apparent.

Proposition 5.2.13 $\forall b1\ a1\ a2.$
$(\forall \texttt{env}\ s.\texttt{evalb env(substb}\ s\ b1) = \texttt{evalb}(\lambda x.\texttt{evala env}(s\ x))b1) \wedge$
$\forall \texttt{env}\ s.\texttt{evala env(substa}\ s\ a1) = \texttt{evala}(\lambda x.\texttt{evala env}(s\ x))a1) \wedge$
$\forall \texttt{env}s.evala\ \texttt{env(substa}\ s\ a2) = \texttt{evala}(\lambda x.\texttt{evala env}(s\ x))a2)) \rightarrow$
$\forall \texttt{env}s.\texttt{evala env(substa}\ s(\texttt{IF}\ b1\ a1\ a2)) =$
$\texttt{evala}(\lambda x.\texttt{evala env}\ (s\ x))(\texttt{IF}\ b1\ a1\ a2)$

This proposition is proven in Sparkle by assuming the induction hypotheses and then reducing the `evala` function.

5.3 Induction schemes for mutually recursive functions

The multi-predicate induction principle for algebraic data types, although useful in many instances, will not make easy proofs possible for a large group of programs that need more sophisticated induction schemes.

For instance this function to calculate the greatest common divisor is not easily within reach of conventional induction. Even if we leave out that integers are not an algebraic data type, the way the function is structured, with either the first or the second argument decreasing by an arbitrary amount, makes it hard to find a proper induction scheme.

```
gcd :: Int Int -> Int
gcd 0 y = y
gcd x 0 = x
gcd x y
  | x < 0 || y < 0 = abort ("Arguments must be positive")
  | y < x          = gcd (x-y) y
  | otherwise      = gcd  x   (y-x)
```

A different kind of induction scheme is needed for functions where the arguments are combined or switched around. Functions where the recursive structure of its definition does not match the structure of the data type of its arguments need a different kind of scheme as well. This new kind of induction scheme that matches the structure of the function definition will make easy proofs possible.

Much research has been done the past few years to find the proper induction schemes that will make proving certain properties easy. Deepak and Shakur have used the coverset induction principle [5]. Boulton and Slind use a multi predicate induction scheme and a proof procedure to find the proper induction hypotheses to use [3].

Tesketh uses middle out reasoning to guide the matching of induction hypotheses [6] and Slind describes a method to derive induction schemes from translating function definitions into higher order logic [18].

Because there is no well established method yet to derives those induction schemes, we will first look closer at two methods; The one from Slind and one from Lee et al. [9].

Slind's method will be the starting point for a novel method to generate induction schemes. We will describe this method in secrion 3 and a comparison with Lee's method can be found in the annexed article, which was presented at the Trends in Functional Programming conference in november 2004 [10].

5.3.1 The size change principle

In their article on the size change principle Lee et al. present a method to evaluate the termination of recursive programs. This method can be used to automatically prove termination for many kinds of programs. In the appendix a comparison is made between our novel method of deriving induction schemes from the definition of mutually recursive functions and Lee's method.

What Lee, Jones and Ben-Amram call the size-change termination principle is the notion that a program terminates on all inputs if every infinite call sequence (following program control flow) would cause an infinite descent in some data values.

A terminating program (as shown in chapter 3) is a sine qua non for determining an induction scheme. Usually an induction scheme can directly be constructed from a termination relationship. The authors of the article do not go that far however. They stick to termination analysis.

Termination analysis is done in two distinct phases. First a set of size-change graphs is extracted from the program. Each recursive function call in a function definition is a node in the graph. Two kinds of vertices are added. One for each measure that decreases and another kind for each measure that stays the same. The measure usually is a well founded argument, but can be a predefined measure function.

In the article programs are first order functional programs. Each program consists of a set of function definitions. A function definition $f(x_1, .., x_n) = e^f$ has an expression as function body. The parameters of this function definition are referred to as $Param(f)$.

This expression can be:

- x, a variable from $x_1, .., x_n$.
- `if` e_1 `then` e_2 `else` e_3. A case expression.
- op(e_1,...,e_n), where op is a primitive operation.
- c:f(e_1,...,e_n), where f is a function definition and c a numeric label that identifies the function call.

The programs in this functional language are evaluated using standard call by value semantics. A semantic operator ε is defined, where $\varepsilon[\![e]\!]\overrightarrow{v}$ is the value of an expression e in environment $\overrightarrow{v} = v_1, ..v_n$ - a tuple containing values of parameters.

ε is a function from expressions and finite value sequences to finite value sequences combined with $\perp$ to model non termination and *Err* to model runtime errors.

Definition 5.3.1
A program p is terminating on input $\overrightarrow{(}v)$ *if* $\varepsilon[\![e^{f_{initial}}]\!]\overrightarrow{v} \neq \perp$

Size change graphs

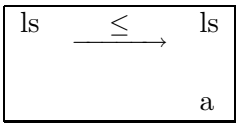
A size change graph $G : f \rightarrow g$ for function names f,g in a program is a bipartite graph from the parameters of f to the parameters of g. The edges of this directed graph are a subset of $Param(f) \times Param(g)$. Each edge is labeled with a symbol to denote whether a parameter is decreasing $<$ or may be decreasing $\leq$.

This size-change graph is used to capture information about a function call. An edge with the symbol $<$ denotes that a data value must decrease in this call with respect to the ordering on the value domain. The edges with the symbol $\leq$ indicate that a value must either decrease or stay the same. If no edge is present between a pair of parameters it means that none of these relations is asserted to be true for them.

For example:

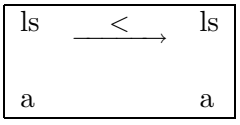
- ```
  rev ls = 1:r1 ls []
  r1 ls a = if ls = [] then a
            else 2:r1 tl ls (cons (hd ls) a)
  ```

  This function which creates the reverse of a list has the following size change graphs $G_1 : rev \rightarrow r1$:

  and

  $G_2 : r1 \rightarrow r1$:

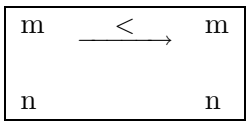
  

- This function, defining the Ackermann function has lexically ordered parameters.
  ```

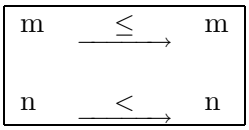
```
a m n = if m=0 then n+1
        else if n = 0 then 1:a (m-1) 1
             else 2: a (m-1) (3:a m (m-1))
```

The graphs for the first two function calls in the body of a are:

$G_1, G_2 : a \to a$:



$G_3 : a \to a$:



To approximate the behavior of the program, these function calls must be concatenated into a multi-path. This is a finite or infinite sequence of size change graphs.

Definition 5.3.2 *Suppose* $G = \{G_c | \texttt{c is a call in p}\}$ *is a set of size-change graphs for a program p.*

- *Suppose the function body of f contains a call c:* $g(e_1, \ldots, e_n)$*. Then an edge from f to g is safe if according to the defined relationship on the value domain the confirms the intended meaning of the symbol associated with that edge.*
- *A size change graph is safe for a call c if every edge in* G_c *is safe.*
- *A set of size change graphs is called a safe description of program p if* G_c *is safe for every call c.*

It is easily shown that the size change graphs of our examples is safe for their respective function calls.

Termination analysis

The analysis is done by defining two sets of infinite call sequences. Those that are possible according to the program's flow graph and those that necessarily cause an infinite descent.

Definition 5.3.3 $FLOW = \{cs = c_1 c_2 \ldots | cs \text{ is well formed and } c1 : f_{initial} \to f_1\}$

Note: a call sequence (cs) is well formed if and only if there is a sequence of functions $f_0 \underset{\to}{(} c_1) f_1 \underset{\to}{(} c_2) \ldots \underset{\to}{(} c_k) f_k$. This is a formalization that the call sequence is only well formed if it can occur according to the program flow.

Definition 5.3.4 $DESC = \{cs \in FLOW |$ *some thread has infinitely many descending edges* $\}$

Theorem 5.3.5 *If* $FLOW = DESC$ *then program p terminates.*

We will not give a formal proof of this theorem, it can be found in the article by Lee, Jones and Ben-Amram. We do give a proof sketch: suppose p is not terminating. This means that there must be a call sequence cs in FLOW, but not in DESC. This also means that there must be an infinitely long decreasing sequence on a well founded domain. This is impossible. Thus p must be terminating.

To find out whether $FLOW = DESC$ a graph manipulation algorithm will be needed.

First of all graph composition is defined in the way one would expect it to be. The composition of two call graphs is the connection of the individual edges. The resulting new edge gets its size change attribute according to the following rule. If either one of the edges has $<$ as an attribute the resulting edge has that attribute as well. The resulting edge gets a $\leq$ attribute otherwise.

By means of transitional closure of the initial graph, the $FLOW$ graph can be constructed. If the resulting graph has an edge from one of arguments of the initial function to the same argument of the initial function with attribute $<$, we know that $DESC$ is the same as $FLOW$ and we can conclude that the program terminates.

Applied to the earlier mentioned examples this will yield for example 1 that the $FLOW$ set is $\{1; 2; 2; \ldots\}$. The call sequence that $1; 2; 2; \ldots$ will give an infinitely descending parameter ls. This means $FLOW = DESC$ and therefor this program will terminate.

For the other example the $FLOW$ set is $\{(1 + 2 + 3); (1 + 2 + 3); \ldots\}$. The set of all infinite strings composed of 1, 2 and 3.

This method of deriving proving termination is simple and general. Lexical orders, indirect function calls and permuted arguments are handled automatically and without special treatment. There is no need to supply argument orders manually. The main drawback is that it does not automatically generate an induction scheme.

A method that specifically deals with generating induction schemes will be described in the next section.

5.3.2 Derivation of induction schemes

In [17] and [18] Konrad Slind describes a method to derive from a recursive function definition and a given termination relationship a valid induction-scheme.

General method

First of all a well-founded induction theorem is introduced:

Theorem 5.3.1

$$(f = WFREC\ R\ M) \rightarrow WF(R) \rightarrow \forall x.f(x) = M\ (f|R, x)x$$

This theorem is an extension of the well-foundedness principle to function calls. It means that if you can place the arguments of recursive calls into a well founded relationship, then the function will terminate. After all, there is no infinite chain of function calls with decreasing arguments.

The functions are defined in higher order logic, using ML-type pattern matching.

Suppose you have a function definition like this:

$$\begin{aligned} f(pat_1) &= rhs1[f(a_{11})..f(a_{1k_n})] \\ &\vdots \\ f(pat_n) &= rhs1[f(a_{n1})..f(a_{1k_n})] \end{aligned}$$

1. First the function description is translated into a functional M.
2. Subsequently a well-foundedness relationship R on the arguments is defined. We use the functional M and the relationship R to define the recursive function f as described in the well-founded induction theorem.
3. This will give us a higher order logical expression which is simplified and rewritten in such a way that the termination conditions are captured and brought to the top level. This process makes use of the following theorem and several rewrite rules. The theorem is a consequence of function restriction.

   Theorem 5.3.2 $R\ x\ y \rightarrow (f|R\ y)x = f\ x$

   The termination conditions are found by checking whether the defined termination relationship holds when the recursive function calls are made in the right hand side of the function definitions.

   If the termination conditions can be proven to be true, we know that the function will always terminate.

4. Suppose the following termination conditions have been extracted:

$$\begin{aligned} &\forall(\Gamma(a_{11}) \rightarrow R\ a_{11}\ pat_1), \ldots, \forall(\Gamma(a_{1k_1}) \rightarrow R\ a_{1k_1}\ pat_1), \\ &\vdots \\ &\forall(\Gamma(a_{n1}) \rightarrow R\ a_{n1}\ pat_n), \ldots, \forall(\Gamma(a_{nk_n}) \rightarrow R\ a_{nk_n}\ pat_n), \end{aligned}$$

The final induction scheme is then derived by the conjunction of all the left-hand-side patterns and proving them to be true under the assumption that property is true for the recursive function-call in the context when the function-call is made.

Then the shape of the induction scheme is like this:

$$\begin{pmatrix} \forall & \begin{pmatrix} (\forall(\Gamma(a_{11}) \rightarrow P\ a_{11})) & \wedge \\ \vdots & \wedge \\ (\forall(\Gamma(a_{1k_1}) \rightarrow\ Pa_{1k_1})) & \end{pmatrix} & \rightarrow P(pat_1) \end{pmatrix} \quad \wedge$$

$$\vdots$$

$$\begin{pmatrix} \forall & \begin{pmatrix} (\forall(\Gamma(a_{n1}) \rightarrow P\ a_{n1})) & \wedge \\ \vdots & \wedge \\ (\forall(\Gamma(a_{nk_n}) \rightarrow P\ a_{nk_n})) & \end{pmatrix} & \rightarrow P(pat_n) \end{pmatrix} \quad \rightarrow \forall\ v.P(v).$$

An example of a derivation

What does that mean for the gcd function as earlier defined? We apply the procedure as outlined in the preceding section.

1. First of all, the pattern-matching is eliminated by translating the function into one that uses case constructs, which yields the following function:

$$\begin{array}{l} \lambda gcd\ z.case_{prod} \\ \quad (\lambda v\ v_1.case_{nat} v_1(\\ \quad\quad \lambda v_2.case_{nat}(Suc\ v_2) \\ \quad\quad\quad (\lambda v_3.\text{if} v_3 \leq v_2 \\ \quad\quad\quad\quad \text{then } gcd(v_3 - v_2, Suc\ v_3) \\ \quad\quad\quad\quad \text{else } gcd(Suc\ v_2, v_2 - v_3)) \\ \quad\quad\quad v_1)\ v)\ z \end{array}$$

   Where $case_{nat}\ f_1\ f_2\ 0\ =\ f_1$ and
   $case_{nat}\ f_1\ f_2\ (Suc\ x) =\ (f_2\ x)$ and
   $case_{prod}\ f_1\ (z\ X\ y)\ =\ f_1\ z\ y$.

2. We define the well-foundedness relationship for this function as the measure of pairs of arguments (a_1, b_1) and (a_2, b_2) where R (a_1, b_1) (a_2, b_2) holds iff $a_1 + b_1 < a_2 + b_2$.

   Lemma 5.3.3 *$measure(x + y)$ is well founded*

   Proof 5.3.6 *Well founded means that there is no infinite strictly decreasing sequence of pairs that are related to each other. Suppose we have a pair*

(a, b), then by induction on a and b, we prove that under the assumption that (a, b) has no infinite strictly decreasing sequence, then $(a, Suc\ b)$ and $(Suc\ a, b)$ have none either. Suppose you have a pair $(c, d) < (a, Suc\ b)$, then either $(c, d) = (a, b)$ or $(c, d) < (a, b)$. In both cases the sequence they are on is finite. By symmetry the same goes for $(Suc\ a, b)$. The base case $(0, 0)$ is trivial, since it is the smallest pair.

3. Now we apply the recursion theorem to the defined function and we get:

$$
\begin{array}{l}
gcd\ z = case_{prod} \\
\quad (\lambda v\ v_1.case_{nat} v_1(\\
\qquad \lambda v_2.case_{nat}(Suc\ v_2) \\
\qquad\quad (\lambda v_3.\text{if} v_3 \leq v_2 \\
\qquad\qquad \text{then } (gcd|measure(Suc\ v2 + Suc\ v3))(v_3 - v_2, Suc\ v_3) \\
\qquad\qquad \text{else } (gcd|measure(Suc\ v2 + Suc\ v3))(Suc\ v_2, v_2 - v_3)) \\
\qquad\quad v_1)\ v)\ z
\end{array}
$$

The process of extracting the termination conditions proceeds by checking under what conditions the recursive function calls will be called. There are two recursive function calls: $gcd(Succ\ v2, v3{-}v2)$ and $gcd(v2{-}v3, Suc\ v3)$. Both are called when the previous arguments have matched the pattern $(Suc\ x)\ (Suc\ y)$.

This will extract the following termination conditions:

$$y \leq x \rightarrow x - y + (Suc\ y) < Suc\ x + Suc\ y$$

and

$$\neg(y \leq x) {-}{>} Suc\ x + y - x < Suc\ x + Suc\ y$$

The context of the termination condition is not relevant when trying to prove that those statements are true. By simplifying you get that $Suc\ x < Suc\ x + Suc\ y$ and , which both are true.

4. When trying to construct the induction-scheme we first determine what the values are for all the relevant arguments and patterns.

$\Gamma(a31) = y \leq x$
$\Gamma(a32) = \neg(y \leq x)$
$pat1 = (0, y)$
$pat2 = (Suc\ x, 0)$
$pat3 = (Suc\ x, Suc\ y)$

Filling this out in the earlier mentioned scheme this gives us:

$$\forall P((\forall y.P(0,y)) \wedge \forall x.P(Suc\ x,0)) \wedge$$
$$\forall x,y.y \leq x \rightarrow P(x-y,Suc\ y)) \wedge$$
$$\neg(y \leq x) \rightarrow P(Suc\ x,y-x)) \rightarrow P(Suc\ x,Suc\ y))$$
$$\rightarrow \forall v,v_1.P(v,v_1)$$

Now we return to our original proposition, which gave us some trouble when we tried to prove it in Sparkle.

Lemma 5.3.4 $\forall a,b \exists s,t : gcd(a,b) = a.s + b.t$

This time the induction runs much smoother:

First the base cases:

1. $P(0,y)$: For this we have to prove that $gcd(0,y) = y = 0.s + y.t$, which is achieved by choosing $t = 1$.
2. $P(Suc\ x,0)$: Now we have to prove that $gcd(Suc\ x,0) = Suc\ x = (Suc\ x).s + 0.t$. This time we choose $s = 1$.

Now the induction case:

- Assume $P(x-y,Suc\ y)$ and $P(Suc\ x,y-x)$.
- To prove: $P(Suc\ x,Suc\ y) = \exists s,t.gcd(Suc\ x,Suc\ y) = (Suc\ x).s' + (Suc\ y).t'$. Do this by rewriting $gcd(Suc\ x,Suc\ y)$. You then get that this equals $gcd(x-y,Succ\ y)$ or $gcd(Succ\ x,y-x)$
  - In the first case you get by invoking the induction hypothesis: $(x-y).s + (Sucy).t$. Now choose $s = s', t = s' + t'$. Leaving us with $(x-y).s' + (Suc\ y).(s' + t') = x.s' + y.t' + s' + t'$
  - In the last case you get by invoking the induction hypothesis: $(Suc\,x).s + (y-x).t$. Now choose $s = t' + s', t = t'$. Substitution gives $(Suc\ x).(t' + s') + (y-x).t'$, which equals $x'.s' + t' + s' + y.t'$.

5.3.3 A novel tactic for deriving induction schemes for mutually recursive functions

Inspired by Slind's use of combinations of well founded sets to construct well founded orderings of the arguments of functions we have extended this idea to mutually recursive function definitions.

When trying to find a well founded ordering for mutually recursive functions one quickly runs into the problem that the number and position of arguments for each function can vary across function calls.

Applications of other functions not included in the mutually recursive function set make it hard to track the arguments that should be used for the well founded ordering.

To track the way the original arguments are used in subsequent recursive function calls a graph is built where each argument of each of the mutually recursive function is a vertex and the way arguments are used is represented by directed edges.

We use Tarjan's algorithm for finding strongly connected components in a directed graph [20] to determine the connectedness of arguments.

By finding the strongly connected components in the graph we can deduce which arguments are relevant and whether they should be grouped together or not.

This will find the best possible arrangement of those arguments in a well-founded ordering. We provide the user with a flexible induction scheme, suited to the program for which she is trying to prove properties.

Only in rare cases the user will have to come up with some kind of measurement function to prove that at each successive recursive call the arguments are decreasing. Also, the use of connectedness ensures that irrelevant and combined arguments are recognized and there is no need for the user to write "the perfect" function in order for the induction scheme to work.

We will describe the algorithm in detail later on, but first we will introduce Clean's core language, which the algorithm uses to determine the proper induction scheme.

Core language programs

Although Sparkle is a proof assistant for Clean, the actual proofs happen on a subset of Clean, called Core-Clean. There are slight differences between Clean and Core-Clean. Due to an automatic translation performed by Sparkle when reading in the program file, these will be almost non existent to the user.

A program in core clean is a set of symbol definitions. Those symbols can either represent user defined functions, delta-rules or internal functions and constructor symbols. Those constructor symbols are used in abstract data type definitions.

Each function definition has a list of function variables associated with it as well as an expression that represents the function definition.

The set of expressions that form the body of a function definition is inductively defined as:

$\varepsilon = \{var\ x\} \cup \{basic\ b\} \cup \{symbols\ \varepsilon_1 \ldots \varepsilon_n\} \cup \{apply\ \varepsilon_1\ to\ \varepsilon_2\} \cup \{case\ \varepsilon\ of\ alts\} \cup \{let\ x_1 = \varepsilon_1, \ldots, x_n = \varepsilon_n\ in\ \varepsilon\} \cup \{let!\ x_1 = \varepsilon_1\ in\ \varepsilon_2\} \cup \{\bot\}$

- An expression can be a variable. In a function definition those variables must be local. They refer to either an argument of the function definition, a variable defined in a let binding, or a pattern in a surrounding case expression.

- Furthermore, an expression can be a basic value. This can be a boolean, integer, string, real or character.

- An expression can also be a symbol expression. This represents the application of a symbol on a list of arguments. These symbols are defined functions or delta rules. The latter are predefined internal functions.

- An application expression represents a general application of one expression to another. It resembles a symbol expression, yet the application node has not yet reduced to a symbol.

- A case distinction is a choice based on the structure of the input expression. All pattern matching, guards and cases in clean are translated into (nested) case expressions.

  A case distinction consists of a case expression and a list of alternatives. Each of the alternatives has a pattern and a result. The case expression is matched against the pattern and if it matches the result expression will be returned. If multiple matches occur, the first will be chosen and if there is no match, $\perp$ will be returned. Note that in the patterns new local variables may be defined.

- Lazy let definitions are representations of shared computations. It binds variables to shared expressions. Those definitions may be recursive.

- Strict let definitions have only one variable and the expression that is bound to the variable will always be evaluated before the root itself is processed.

- $\perp$ is the erroneous expression. All reductions that do not lead to a weak head normal form will be considered erroneous. A erroneous expression can also result from a non matching case expression.

Constructing a call graph

The user provides the theorem prover with a list of functions with matching predicates for which the induction scheme has to be derived. The proof assistant then analyzes all the recursive function definitions and builds a graph where all the arguments are vertices and where for each argument of the definition that is used as an argument in the recursive call, an edge is added from the vertex that represents the argument in the function definition to the argument in the recursive call.

Let us define the set of mutually recursive functions and matching predicates as (F, P). Where $F = \{f_1, \ldots, f_n\}$ and $P = \{P_1, \ldots, P_n\}$.

Each recursive function definition has arguments $a_{k1}, \ldots, a_{kl}$. Whereby the arity of function $f_k = l$.

A predicate is called a matching predicate when it starts with a for all quantifier over a variable of the same type as the corresponding function definition and the function definition occurs within the property itself. A syntactic check of the property as well as the function definition can guarantee that this is the case.

When constructing a recursive call graph one has to keep in mind that recursive function calls can happen at several places within the expression that forms the function body.

Besides that, due to the let bindings and pattern matching, variables that are used within the recursive function call may be derived from the arguments of the function definition. This means we have to keep track of how the variables that are used in the recursive function call make use of the original arguments of the function.

For each of the possible structures of an expression within a function body, we will describe how it influences the resulting call graph.

For this purpose we define a set of variable mappings M, a set containing tuples of variables, and an operator ϕ that generates that mapping. We recursively define this set in the following way.

- Occurrences of the lazy and strict let will be handled the same, since the way lets are evaluated does not change the recursive call structure of the program. It only influences the ordering in which the calls take place.

  $\phi(\texttt{Let } x_1 = e_1, \ldots, x_n = e_n \texttt{in } e) =$
  $x_1 \times \phi(e_1)$
  $\cup$
  $\ldots$
  $\cup\, x_n \times \phi(e_n$

  where times is defined as $\forall a, b, c.a \times \{(b, c)\} = \{(a, b), (a, c)\}$.

  All the variables x_1 up to x_n are added to the mapping, with on the right hand side of the tuples all the variables used in the expression the variable is bound to.

- In case expressions the variable may be mapped to other variables, because in patterns new variables may be introduced. These patterns can either be a basic pattern, or a cons-pattern. In the basic pattern no new variables are introduced, but a cons-pattern is a constructor symbol, followed by variables. These variables are matched against the matching expressions following the constructor symbol of the case expression.

  Suppose $Ce_1 \ldots e_n$ is the case expression, while $Cx_1 \ldots x_n$ is the pattern. The operator ϕ works in the following way:

  $\phi(\texttt{case } e \texttt{ of} C_1 x_{11} \ldots x_{1n} \rightarrow e_1$
  $\ldots$

$C_k x_{k1} \ldots x_{km} \rightarrow e_k$
$e_{default})$
$= x_{11} \times \phi(e_1) \cup \ldots \cup x_{1n} \times \phi(e_{1n})$
$\ldots$
$x_{k1} \times \phi(e_k) \cup \ldots \cup x_{km} \times \phi(e_k)$

- Within symbol applications, whether they be delta-rules, function calls or constructor applications there can be two cases. Either the symbol is one of the mutually recursive functions we are trying to use for our induction scheme or not.

  – If the symbol is not one of the recursively defined functions, the operator ϕ just proceeds

  – If the symbol is one of the recursively defined functions, graph nodes are returned for

This graph is then used to calculate all the strongly connected components.

Take for instance these mutually recursive functions, where arguments are combined and switched around.

```
f a b c = g (c+b) a
g x y   = f x y 1
```

The graph representing this function shows that f a b and g x y are strongly connected. The ordering we impose on this set of mutually recursive functions therefore has to take into account that for each function call f inside the function body of g, the arguments x and y combined have to be smaller than a and b combined.

In order to impose a well founded ordering on arguments that do not have one by itself, like integers [5] and strings, size [6] functions are constructed for all algebraic data types. Combinations of arguments are constructed by addition in the case of integers and size functions and concatenation in the case of strings. Both operations maintain the well-foundedness ordering of their arguments.

All arguments that have not been combined by their well-connectedness are then lexicographically ordered. This ordering is a relationship between pairs of functions and their arguments and by applying the well-foundedness theorem on this relationship we can derive an induction scheme that is tailored to the function definitions at hand. This induction scheme is available

[5] Integers are not well founded of course.
The algorithm imposes a lower bound on them to make them well founded.

[6] For nested algebraic data types no proper size function that maps onto an integer can be constructed. To prove properties of programs that use these kinds of arguments, either induction on mutually recursive data types has to be used or a well-foundedness relationship has to be provided by the user.

to the user and whenever it is applied to a recursive function call within the body of a function definition it raises for the user the obligation to prove that in this particular instance the arguments arranged according to the derived induction scheme are smaller than when entering the function definition. The program then deduces that the property holds for that recursive function call. If the user can not prove that the well founded relationship holds for these particular arguments, then the induction hypothesis can not be used.

If no proper well founded ordering can be found the algorithm permits the user to supply her own ordering. First of all, the user can supply functions that map the arguments of the functions for which an induction scheme is to be derived onto either integers or strings. From then on the algorithm uses an induction scheme either based on the ordering provided by the $<$ operator for integers and strings.

If no such function can be found, it is always possible to directly define an ordering relation by supplying functions that define an ordering relationship. Of course this means that an extra proof obligation for the well-foundedness of this relationship is added.

Since Sparkle only allows first order logic and has no concept of sets the ordering relationship has to be modeled by using ordinary Sparkle functions. These functions must return a boolean and be proven to terminate.

Take for instance this example from Slind's article [18]. Although the function is a simple mapping onto integers and it need not have been extended to a function that maps pairs of arguments onto a boolean value it is given here as an example of the procedure.

```
variant x l
| isMember x l  = variant (x+1) l
| otherwise     = x
```

The relationship supplied would be a function:

```
isSmaller (x1,l1) (x2,l2) =
  ((max l1) + 1 - x1) < (max l2) + 1 - x2
```

Since there is no recursion involved in this function it is easily proven to terminate. Proving well-foundedness would then entail that no infinite chain of arguments exist for which `isSmaller` is always true:

Proposition 5.3.7 $\forall f :: (Int, []) \rightarrow \texttt{Bool}.(\exists w.fw) \rightarrow \exists m.f\ m$
$\wedge\ \forall b.\texttt{isSmaller}\ b\ m \rightarrow \neg f\ b$

5.3.4 Example of tactic applied to gcd

For our earlier defined gcd function we want to prove that the greatest common divisor of all even numbers is bigger than 1.

Proposition 5.3.8 $\forall xy.\, \gcd(x+x)(y+y) > 1$

When we apply this induction tactic to this proposition, specifying that we want it to derive an induction scheme which follows the recursive structure of the gcd function, the algorithm will determine which arguments are strongly connected. It will find that both arguments are, so the ordering imposed on them will be $R((x1, y1), (x2, y2)) = x1 + y1 < x2 + y2$. This ordering, combined with the well-foundedness theorem will yield this induction scheme:

Proposition 5.3.9 $\forall P.(\forall x1x2y1y2.x1 + y1 < x2 + y2 \rightarrow (P(x1, y1) \rightarrow P(x2, y2))) \rightarrow \forall xy.P(x, y)$

The proof then proceeds by reducing the gcd function and splitting it into each of the different cases. The user has to prove that the property holds when one of the arguments is bottom or 0. No induction hypothesis is needed for them.

Only when the user has to prove that the property holds for either $\gcd(x - y)y$ or $\gcd x(y - x)$ an induction hypothesis is needed due to the recursive function call. Both proofs are identical, so we assume we just have to prove that $\gcd((x+x)-(y+y))(y+y) > 1$ At that point an instantiation of our induction scheme will mean that if we can prove that $x-y+y < x+y$, which should be easy, since the case $y = 0$ has already been handled, we may assume that $\gcd((x - y) + (x - y))(y + y) > 1$, which is the same as $\gcd((x + x) - (y + y))(y + y) > 1$.

5.4 Simple Co-Induction

Since Sparkle is a lazy evaluating functional language, co-recursive proof techniques can prove particularly useful. Several methods for dealing with co-recursive programs are described in [8]. Fixed point induction, fusion, bisimulation and the approximation lemma are mentioned. Although bisimulation is viewed as a last resort, because it requires a bisimulation relation to be defined, we contend that those relationships are suggested by the properties one tries to prove. It even turns out that the approximation lemma can easily be viewed as a special case of bisimulation with respect to an algebraic data type.

In our next section we will show how bisimulation with respect to expressions of a recursively defined types is defined and how one can generalize this to a bisimulation with respect to the reduction of expressions to root normal form. Both methods do not require the user to define any bisimulation relationship.

5.4.1 Bisimulation with respect to algebraic data types

Co-recursive programs are less well known than recursive programs, but the growing insight that co-recursion can be a very useful and elegant way of programming, means that proof techniques and tactics need to be made available to programmers. The power of co-recursion is shown in an McIlroy's article on power series [15]. This article shows how power series can be implemented using streams of fractions. These infinite streams of integers represent the coefficients of the power series.

For instance the power series for cos x; $1-\frac{x^2}{2!}+\frac{x^4}{4!}-\frac{x^6}{6!}+\ldots$. is represented by the infinite list of fractions: [1,0,-1/2,0,1/24,0,-1/720,...]. Using infinitely recursing functions on these streams mathematical operations like addition, subtraction and even calculus can be defined on these power series.

An example of how simple these definitions are is the implementation of the sinus and co-sinus function:

```
 sin x = integral cos x
 cos x = 1 - (integral sin x)

integral x = [0:int1 x 1] where
   int1 [x:xs] n = [x/n:int1 xs (n+1)]
```

Since these programs operate on a lazy, infinite list, they will never terminate. Regular induction proofs can be used on infinite objects, provided that the predicates are admissible, as explained in chapter 3.4. Proofs on lazy lists require $P(\bot)$ to be proven. A feat that can be very hard for many propositions. Co-inductive proof techniques are usually more suited for that task.

5.4.2 Bisimulation tactic

As described in other articles, in particular [2], a bisimulation is described as a relation on a set of (infinite) elements of a certain data type with a co-algebraic structure defined by its destructor functions. For lists this relationship would be: $R(a,b) \rightarrow \texttt{hd}(a) = \texttt{hd}(b) \wedge R(\texttt{tail}\ a, \texttt{tail}\ b)$. This relationship defines the co-inductive proof principle;

Theorem 5.4.1 $\forall a, b.R(a,b) \rightarrow \forall a, b.a = b$

As earlier stated, the main problem is finding the proper bisimulation relationship. However, usually the relationship is defined by the property one is trying to prove.

Take for instance next property of lazy lists:
$\forall a.\texttt{merge}(\texttt{odd}(a), \texttt{even}(a), a) = a$.

The tactic operates on an equality of two expressions provided by the user. At the moment this means that goal on which the tactic is applied must

be an equality of two expressions. If the type of the expressions on either side of the equality is a recursively defined data type, then the tactic uses the equality of these expressions as the the bisimulation relationship needed to prove this property. When applied to the co-inductive proof principle this simplifies to the following properties that need to be proven:

1. $\forall a.\texttt{head}(\texttt{merge}(\texttt{odd}\ a)(\texttt{even}\ a)) = \texttt{head}\ \ a$

2. $\forall a.R(a,a) \rightarrow \forall a.R(\texttt{tail}(\texttt{merge}(\texttt{odd}\ a)(\texttt{even}\ a), \texttt{tail}\ a)$

In practice this means that if we have a property which we try to prove that consists of an equality of two expressions, we look what the data type of that property is. If it is an algebraic data type, then we construct the destructor functions according to the definition of the algebraic data type and fill them out in the general scheme suggested earlier. We can easily abstract this to mutually recursive data types in the same way we have done for induction on mutually recursive functions.

```
FOR EACH ADT
  get datatype definiton;
  FOR EACH constructor in the datatype definition
    get constructor definition;
    FOR EACH type in the list following the constructor
      IF type is one of the ADT's THEN
        construct a transactional deconstructor;
      ELSE
        construct an observational deconstructor;
      FI
    HCAE ROF
  HCAE ROF
  FOR EACH observational deconstructor
    create a proof obligation where the observational
    deconstructor is applied to the corresponding
    equality expression;
  HCAE ROF
  FOR EACH transactional deconstructor
    create a proof obligation where the user has
    to prove that the transactional deconstructor applied
    to the equality relation satisfies the equality
    relation;
  HCAE ROF
HCAE ROF
```

Take for instance the following completely different implementations of a simple stack, the first one uses recursion to model a stack, the second one is parameterized. Proving these to be the same is now an easy task, since both can be programmed within clean and proven to be correct in Sparkle.

```
C = u * D * C
D = d + u * D * D

C 0 = u * (C (n+1))
C n = d * (C (n-1)) + D * (C n)
```

The bisimulation needed to prove that these implementations is the same is:

Proposition 5.4.2 $\forall n.C(n+1) = D * (Cn)$

Proof 5.4.3 $C(n+1) = d * (Cn) + u * (Cn+2)$
$= d * (Cn) + u * D * (Cn+1)$ *[using bisimulation]*
$= d * (Cn) + u * D * D * (Cn)$ *[using bisimulation]*
$= (d + u * D * D) * (Cn)$ *[using distributivity law of +]*
$= C(n+1)$ *[using definition of C n]*

5.4.3 Example of bisimulation tactic applied to power series

Since we have a proof technique now for infinitely recursing programs we can return to our initial example of operations on power series. Suppose we want to prove the mathematical property of the sinus function that $-(\sin x) = \sin(-x)$.

Using the bisimulation tactic directly on this equation relation unfortunately does not yield an easy and obvious proof of this property. However, once we realize that the bulk of the work of the sinus function is done by the integral function we can choose another property to help us:

Proposition 5.4.4 $\forall zn. - (\texttt{int1}\ z\ n) = \texttt{int1}(-z)n$

Applying the bisimulation tactic to this equality relationship we get two proof obligations:

1. First we have to prove that the heads of both infinite streams are the same.

   ```
   hd -(int1 z n) = hd (int1 (-z) n)}
   ```

   By reducing both sides of the equation we get:

   ```
   -int1 [x:xs] n = -[x/n:int1 xs (n+1)] =[-x/n:-int1 xs (n+1)]
   int1 -[x:xs] n = int1 [-x:-xs] n = [-x/n:int1 -xs (n+1)]
   ```

   That leaves us to prove that the hd of these streams is the same:

   ```
   hd [-x/n:-int1 xs (n+1)] = -x/n
   hd [-x/n:int1 -xs (n+1)] = -x/n
   ```

2. Subsequently we have to prove that the tail of both infinite streams satisfy the stated equality relationship.

```
tail [-x/n:-int1 xs (n+1)] = -int1 xs (n+1)
tail [-x/n:int1 -xs (n+1)] = int1 -xs (n+1)
```

choose z = xs and n = n+1 to see that it satisfies the required relationship.

Once we have that $\forall zn. -(\texttt{int1}\ z\ n) = \texttt{int1}(-z)n$, it is easy to prove that -(sin x) = sin (-x)

5.4.4 Guarded co-induction

Although the above mentioned bisimulation relationship will work in many cases there are some where a more flexible approach is needed. It is equivalent to the proof technique on mutual recursive data types. The question arises whether it is possible to generalize this method in the same way we have extended the induction principle to mutually recursive function calls. In fact, we are looking for a fixed point for properties of functions instead of data types.

A set of very interesting stream functions is formed by self-similar functions, like the Thue-Morse sequence [7]. These fractal-like streams are the same on any scale. One of the interesting properties this stream has is that all the odd elements taken separately and concatenated will form a Thue Morse sequence as well. The sequence is generated by a substitution map using the following rules:

1. $0 \rightarrow 01$
2. $1 \rightarrow 10$

Starting with 0 this gives the following sequence:
$0 \rightarrow 01 \rightarrow 0110 \rightarrow 01101001 \rightarrow ...$

A direct implementation of this substitution map would not be a productive function, however we construct an equivalent function in a lazy functional language to have the same effect:

```
tm :: [Bool] Int -> [Bool]
tm x n = (cmpl x (n+1)) ++ tm (cmpl x (n+1))

cmpl :: [Bool] int -> [Bool]
cmpl x 0 = x
cmpl x n = inverse x ++ cmpl (x ++ inverse x) (n-1)
```

[7] A description can be found at http://mathworld.wolfram.com/Thue-MorseSequence.html

Ordinary bisimulation on tm will not do, since the list you have to take elements from will get progressively longer at each iteration. The function definition suggests that we have to prove that for each iteration of tm a certain property holds.

5.4.5 Tactic for guarded co-induction

This method is mentioned in Gibbons' and Hutton's article [8] as guarded co-induction, but it is not clear whether they meant to apply it to more than co-induction on lazy data types. If we want to extend this guarded co-induction principle to cover the Thue-Morse function as defined earlier, we have to view the function itself as an object. For guarded co-induction to work we have to make sure that it is productive as defined in Coquand's article on infinite objects in type theory[4].

Coquand guarantees productiveness by means of a syntactic restriction on the functions on which guarded co-induction is allowed. Guarded co-induction is only allowed on those functions that have constructors in the function definition preceding the recursive function call. Although this covers many functions, many times the recursive function call is hidden behind a function that ultimately will return a constructor value.

In order to make this method more useful we have relaxed the syntactic restrictions. At the moment we have not proven that this relaxation is allowed, so our method is highly speculative and until a formal proof has been given, should only be used for testing purposes.

The productiveness of our function is enforced by demanding that it can be reduced to an expression in root normal form. If this is possible, we may safely assume that whenever we encounter a recursive function call that satisfies our equality relationship we can substitute it with its counterpart.

Suppose we have an equality relationship between two expressions, one a reference to a function definition and the other one not in root normal form. We then start to unravel the function definition and whenever we encounter a recursive function call that satisfies our equality relationship, we can assume that no further unraveling is necessary.

```
FOR EACH equality expression
  bring both sides of the expression in root normal form;
    add an assumption that if a function call satisfies
    the equality relationship we may assume they are equal;
  have the user prove that these expressions are equal;
HCAE FOR
```

5.4.6 Example of guarded co-induction on the Thue Morse sequence

The Thue-Morse sequence has many interesting properties. One of them is that taking each odd element from the thue morse sequence equals the

Thue-Morse sequence: `odd(tm [True]) = tm [True]`

We can not use this proposition as a direct bisimulation relationship, since it is not quantified over any variable. A better candidate is:

Proposition 5.4.5 $\forall n.$`tm` $n =$ `odd(tm(cmpl [True]` $n))$

We need to prove some auxiliary lemmas first using regular induction on the natural number n.

Proposition 5.4.6 $\forall n.$`odd(cmpl [True]` $n) =$ `cmpl [True]` n

Proposition 5.4.7 $\forall n.$`odd(cmpl` x $n) + +z) =$ `odd(cmpl` x $n) + +($`odd` $z)$.

Using the tactic on the suggested bisimulation relationship will bring both sides of the expression in root normal form.

```
tm [True] n = (cmpl [True] (n+1)) ++ tm (cmpl [True] (n+1))
            = [False] ++ cmpl ([True] ++ inverse [True]) n)
            = [False:cmpl ([True] ++ inverse [True]) n)
                 ++ tm (cmpl [True] (n+1))]
```

And

```
odd(tm [True] n)
            = odd [False:False:cmpl ([True] ++ inverse [True]
               ++ inverse [True] ++ [True]) n+1)]
            = [False:odd(cmpl ([True] ++ inverse [True]
               ++ inverse [True] ++ [True]) n+1)
               ++ tm (cmpl [True] (n+1))]
            =
```

Using both lemmas and the guarded induction hypothesis on both sides of the equation we can conclude that the property holds for all n.

5.5 Summary

In this chapter, we have shown how we have extended Sparkle with four more tactics, two inductive tactics and two co-inductive tactics.

These tactics make it possible to reason about mutually recursive programs as well as recursive programs that do not terminate.

Firstly, Sparkle is extended with a general method of deriving induction schemes for mutually recursive algebraic data types. Since the previous version of Sparkle only allowed induction proofs on directly recursive algebraic data types, extending this to the generation of induction schemes for mutually recursive expands the real of possible proofs in Sparkle, which is a significant improvement.

Secondly, Sparkle is extended with a new method for deriving induction schemes based on a well founded ordering. This ordering is derived from the function definitions and closely mimics the call flow of the functions used.

Thirdly, a co-inductive tactic is introduced. This tactic allows the programmer to construct proofs using a bisimulation on the structure of algebraic data types. Implementing this tactic in Sparkle broadens the scope of potential proofs that can be made within Sparkle considerably, because the proofs are not limited to terminating recursive programs anymore.

Finally, a novel method to construct co-inductive proofs on recursive functions using a bisimulation on the structure of the recursive function calls is implemented. Although this method is speculative, it allows for a wider range of possible proofs, since it is not limited to mimicking the structure of the algebraic data type.

Combined, these methods provide for proofs on a a greater class of recursive programs.

Chapter 6

Conclusion

"He did not arrive at this conclusion by the decent process of quiet, logical deduction, nor yet by the blinding flash of glorious intuition, but by the shoddy, untidy process halfway between the two by which one usually gets to know things." –Margery Allingham

6.1 Summary

Formal reasoning is a powerful tool to help us guarantee the correctness of (functional) programs. In this thesis, Sparkle is introduced as a tool for formal reasoning and recursion is introduced as one of the core methods of functional programming. Key question of this thesis is how Sparkle can be extended to be able to provide proof on a larger class of recursive programs.

The inductive and co-inductive proof principles are the corner stones on which we have built the tactics that extend Sparkle. They are mathematically sound and offer can be implemented as algorithms within Sparkle.

These proof principles lead to four methods, or tactics as they are also called, that can be used to proof properties of functional programs.

First of all, Sparkle is extended with a general method of deriving induction schemes for mutually recursive algebraic data types. Besides that, Sparkle has received a new method for deriving induction schemes based on a well founded ordering. This ordering is derived from the function definitions and closely mimics the call flow of the functions used. Co-inductive proof techniques are provided as well. One uses the structure of the algebraic data types to create a bisimulation relationship. The last method uses the structure of the function definition to provide a tactic that can be used to prove properties that can not be handled by the first co inductive proof technique. Unfortunately, this technique has not been proven to be correct.

By extending the induction and co-induction proof techniques for Sparkle we have opened the theorem prover up for a wide set of proofs on properties of programs. Furthermore, by automating the derivation of induction

schemes for mutually recursive functions, to closely match the structure of the underlying program, we have achieved that proofs of properties of those programs can proceed with ease and require little intervention of the user.

By adding bisimulation to the proof tactics we have also expanded the capabilities of Sparkle as a theorem prover for programs that manipulate infinite structures.

All in all, the new tactics have increased the usefulness of Sparkle as a proof assistant for a wider range of recursive programs.

6.2 Future work

These new tactics are by no means the be all and end all of what can be done to improve Sparkle more.

In order to further enhance the usability of Sparkle, a few additions are certainly warranted. At the time the automation level of proofs is low.

Although a plethora of techniques is available, there is no smart algorithm present to find out which proof tactic should be applied at a certain time.

At the moment, R. van Kesteren is enhancing Sparkle with proof rules for parameterized classes[21]. This will make proofs of properties of generic classes possible.

Another extension of Sparkle could be the inclusion of features of Clean like dynamics and overloading, which are currently not really supported within the proof system, as well as an extension of the collection of pre-proved basic properties of programs and data types.

A formal justification of the tactic of guarded induction is needed as well before it can officially be allowed to be used in proofs.

Bibliography

[1] Andreas Abel and Thorsten Altenkirch. A predicative analysis of structural recursion. *Journal of Functional Programming*, 12(1):1–41, January 2002.

[2] J. Rutten B. Jacobs. A tutorial on (co)algebras and (co)induction. *Bulletin of the EATCS*, pages 222–259, 1996.

[3] Richard Boulton. Multi-predicate induction schemes for mutual recursion. Informatics research report EDI-INF-RR-0046, University of Edinburgh, 2000.

[4] T. Coquand. Infinite objects in type theory. In *Proceedings of the international workshop on Types for proofs and programs*, pages 62–78. Springer Verlag, 1994.

[5] M. Subramaniam D.Kapur. Automating induction over mutually recursive functions. In *Proceedings of the 5th international conference on algebraic methodology and software technology*, volume 1101 of *Lecture notes in computer science*. Springer Verlag, 1996.

[6] J.T. Hesketh. *Using Middle-Out reasoning to Guide Inductive Theorem Proving*. PhD thesis, University of Edinburgh, 1991.

[7] J. Hughes. Why functional programming matters. *The Computer Journal*, 32(2), 1989.

[8] G. Hutton J. Gibbons. Proof methods for corecursive programs. *Fundamenta Informaticae XX*, pages 1–13, 2004.

[9] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In *ACM Symposium on Principles of Programming Languages*, volume 28, pages 81–92. ACM press, january 2001.

[10] Lensink, Leonard, van Eekelen, Marko. Induction and Co-induction in Sparkle. In Hans-Wolfgang Loidl, editor, *Fifth Symposium on Trends in Functional Programming (TFP 2004)*, pages 273–293. Ludwig-Maximilians Universität, München, November 2004.

[11] M. van Eekelen M. de Mol. A prototype dedicated theorem prover for clean. Technical report CSI-R9913, Radboud University Nijmegen, 1999.

[12] R. Plasmeijer M. de Mol, M. van Eekelen. Theorem proving for functional programmers. *LNCS:Selected Papers from the 13th International Workshop on Implementation of Functional Languages*, 2001.

[13] T. F. Melham M. J. C. Gordon. *A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[14] R. Plasmeijer M. van Eekelen. *Concurrent Clean Language Report (version 2.0)*. University of Nijmegen, 2001.

[15] M. Douglas McIlroy. Functional pearls: Power series, power serious. *Journal of Functional Programming*, 9:323–335, 1999.

[16] L.C. Paulson. *Logic and Computation: interactive proof with Cambridge LCF*. Cambridge University Press, 1987.

[17] Konrad Slind. Function definition in higher order logic. *Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics*, pages 381–397, 1996.

[18] Konrad Slind. Derivation and use of induction schemes in higher order logic. *Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics*, pages 275–290, 1997.

[19] J. Giesl T. Arts. Automatically proving termination where simplification orderings fail. In *Proceedings Colloquium on Trees in Algebra and Programming*, volume 1214, pages 261–272. Berlin: Springer-Verlag, 1997.

[20] R. Tarjan. Depth first search and linear graph algorithms. *SIAM journal of Computing*, pages 146–160, 1972.

[21] van Kesteren, Ron, van Eekelen, Marko, and de Mol, Maarten. An Effective Proof Rule for General Type Classes. In Hans-Wolfgang Loidl, editor, *Fifth Symposium on Trends in Functional Programming (TFP 2004)*, pages 149–165. Ludwig-Maximilians Universität, München, November 2004.

Appendix A

Induction and Co-Induction in Sparkle

Proceedings of Trends in Functional Programming 2004, Munich, Ludwig-Maximilians University

Leonard Lensink[1], Marko van Eekelen[2]

[1]Email:llensink@xs4all.nl
[2]Nijmegen Institute for Computing and Information Sciences,
Radboud University Nijmegen, Toernooiveld 1, Nijmegen,
6525 ED, The Netherlands; Phone +31 (0)24-3653410;
Email: M.vanEekelen@niii.ru.nl

Abstract: Sparkle is a proof assistant designed for the lazy evaluating functional programming language Clean. It is designed with the clear objective in mind that proofs of first order logical predicates on programs should be as easy as possible for programmers. Since recursion is at the heart of functional languages, support for recursive reasoning should be as exhaustive as possible. Until now the induction tactic in Sparkle was rather basic. We have extended Sparkle with two standard techniques to perform inductive and co-inductive reasoning about programs. By means of examples, we show from research papers how they benefit the programmer. More importantly, we propose a new technique to derive induction schemes for mutually recursive programs by using strongly connected components of complete call graphs to derive a well founded ordering and induction scheme. These induction schemes can be used to semi-automatically prove properties of programs out of reach of other automated proof techniques. Together this extends the realm of programs for which easy proofs in Sparkle can be constructed by the programmer.

A.1 Introduction

In this introduction we will first briefly acquaint the reader with the proof assistant Sparkle, before the subject of induction and co-induction within Sparkle is introduced.

A.1.1 A brief introduction to Sparkle

Sparkle is the integrated proof assistant available with the lazy functional programming language Clean [14]. The main purpose of the theorem prover is to help the programmer prove properties of programs. Several features are especially useful.

Firstly, the reasoning takes place in first order logic on the level of the program itself, so no translation of the program is needed. Secondly, the theorem prover is partly automated. And finally, the theorem prover is integrated into the development environment of Clean.

All these features are intended to encourage people who are not well versed in proof assistants to make use of the advantages a correctness proof can give. A brief description of Sparkle's capabilities can be found in [12].

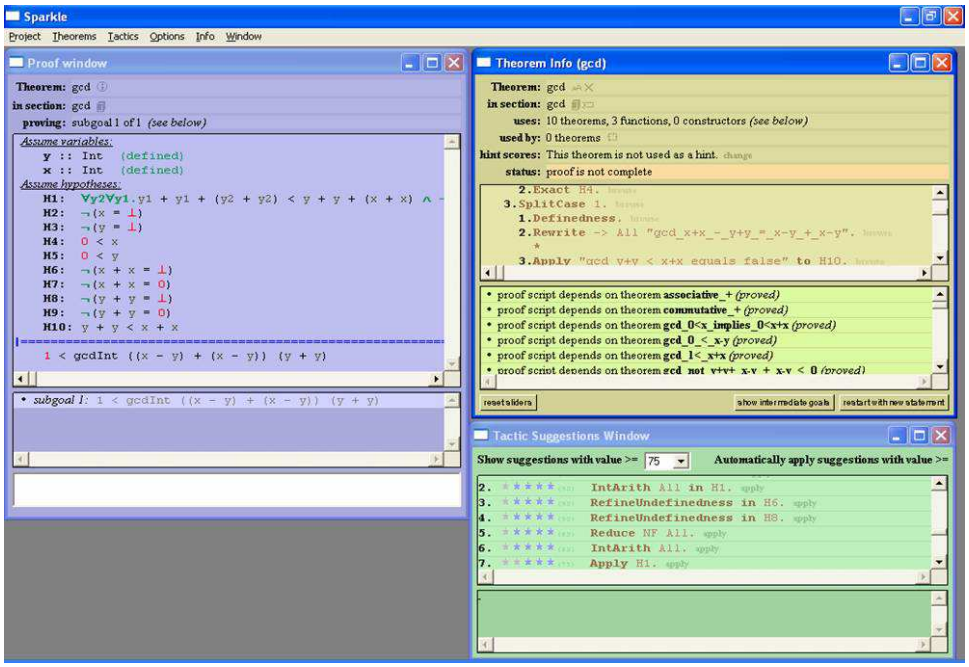


Figure A.1: Sparkle in Action

Reasoning in sparkle is similar to other theorem provers. The user starts off with a goal, a property, that needs to be proven. A goal has a goal

environment in which the local hypotheses and introduced variables are stored. Several tactics are available to the user, which can be applied to the goal. Each tactic is a function from a goal and its environment to a list of goals and environments. Whenever all the goals are proven, the property is proven correct.

In order to make life as easy as possible for the programmer, Sparkle also contains a hint mechanism. This means it will try to guess which tactic might be applicable at a certain time. These hints are assigned a score. The higher the score, the more likely the tactic will be useful to the programmer in order to complete the proof. By having Sparkle apply the highest scoring tactics automatically, propositions can be proven without any user intervention whatsoever.

A.1.2 Using Sparkle for Haskell programs

Although all proofs and tactics are done within the Clean environment they are not limited to this environment. The tactics can be implemented, and some already are, in other theorem provers. Sparkle can also be used to prove properties of Haskell programs. A fully automated Haskell to Clean converter is available, called Hacle[1]. The similarities of both functional languages make reasoning on the source code level of Haskell programs possible.

To show how all the programs and proofs are constructed in Sparkle, all the programs and sections containing the proof are available to the interested reader. The extended Sparkle version, as well as the Clean compiler needed to compile the programs are at the same site[2].

A.1.3 Induction and co-induction in Sparkle

Many different proof techniques have been incorporated into Sparkle, but so far only a basic form of induction was available to the user. This tactic generated an induction scheme for directly recursive algebraic data types and integer functions.

In order to improve the usefulness of the theorem prover we have implemented multi-predicate induction over mutually recursive algebraic data types. Since this kind of induction has been implemented a few times already, we briefly introduce this tactic by means of an example from the literature in section A.2.1.

Besides an extension of the induction proof principle, co-inductive proof techniques by means of bisimulation are introduced. Since Clean is a lazy functional language, many properties of lazily defined data types and functions can only be proven using this tactic. In section A.2.2 we will demonstrate by means of an example taken from the literature how this new bisim-

[1] Hacle can be found here: www-users.cs.york.ac.uk/ mfn/hacle/

[2] http://www.xs4all.nl/ llensink/Sparkle.zip

ulation tactic can be used for proofs of interesting properties of non terminating functional programs.

Another new feature is the possibility to have the proof assistant generate schemes for predicates involving mutually recursive function calls. This method constructs induction schemes based on the way the mutually recursive function definitions are interdependent. New in this induction method, compared to already existing methods is that the well-foundedness ordering that is the basis of the induction scheme is derived from the arguments of the recursive function calls in the function definitions. A graph is constructed and analyzed to find the best combination of arguments to build an ordering from. Although, there are more sophisticated methods of using call graphs to prove termination of recursive functions, we argue that our method, despite its simplicity yields good results and is especially suitable for Sparkle, because it generates an induction scheme that can be used in combination with other tactics to prove properties of a wide range of recursive programs. A detailed explanation of the implementation as well as a comparison with one of the other methods can be found in section A.3.

A.2 Standard techniques for recursive functions

In this section we will show by example how the implementation of two standard techniques of constructing inductive and co-inductive proofs is realized in Sparkle. The examples, taken from research papers, show how they enable the programmer to reason about her code with ease.

A.2.1 Induction on mutually recursive data types

Mutual recursion in algebraic data types require proof techniques that can handle the inter-dependency between the constituent parts. A well known method, employed for instance in the HOL theorem prover [13], uses multi-predicate induction. In this section we will briefly describe our implementation of it by means of an example.

One of the properties of Clean and Haskell is that all the algebraic data type definitions must be guarded with a constructor and each constructor must be unique. This means that we do not have to worry about non-productive data types or ambiguous derivations. The only restriction to the (mutually) recursive algebraic data type is that at least one of the recursive definitions has a non-recursive part. If not, we don't have a smallest element to build our ordering on.

Another point of interest is that an expression of a certain data type may not terminate or be undefined. In Sparkle this is represented by $\perp$. Since predicates on non terminating functions may not always be allowed they must be deemed admissible. The way admissibility is determined within

Sparkle is based on Paulson's criterium as stated in his Logic and Computation book [16]

Take for instance a simple expression language consisting of arithmetic and boolean expressions. This example can be found on the HOL website[1]. Arithmetic expressions can contain a boolean condition, while boolean expressions may be a comparison between arithmetic expressions.

```
::Aexp a = Var a
         | Num Int
         | Sum (Aexp a) (Aexp a)
         | If (Bexp a) (Aexp a) (Aexp a)
::Bexp a = Less (Aexp a) (Aexp a)
         | And (Bexp a) (Bexp a)
```

On this small language substitution and evaluation functions are defined. The substitution function performs a substitution, defined as a map from pointers to arithmetic expressions, on an expression. The evaluation function reduces an expression to a value within a certain environment. That environment is defined as a map from pointers to integer values. To conserve space we only give the function body of the evala function as an example.

```
evala :: (a->int) (Aexp a) -> int
evala e (If b a1 a2)
   | evalb e b = evala e a1
   | otherwise = evala e a2
evala e (Sum a1 a2) = (evala e a1) + (evala e a2)
evala e (Var v) = e v
evala e (Num n) = n

evalb :: (a->int) (Bexp a) -> bool
substa :: (a -> Aexp a) (Aexp a) -> Aexp a
substb :: (a -> Aexp a) (Bexp a) -> Bexp a
```

We want to prove that it does not matter whether we evaluate all the substitutions and use that as an environment or we perform all the substitutions first and then evaluate the resulting expression. Formulated within Sparkle it gives us the following property to prove.

Property A.2.1 $\forall$`env` s a.`evala env` (`substa` s a) = `evala` (λx.`evala env` (s x)) $a \wedge \forall$`env` s b.`evalb env` (`substb` s b) = `evalb` (λx.`evala env` (s x)) b

Sparkle's new induction tactic will recognize that the induction will be done on the variables a and b, of a mutual recursive type and offers an induction scheme whereby for each non recursive part of the data types a proof of the property is required, while for each recursive definition an antecedent is constructed. Sparkle will generate eight subgoals that need to be proven. Two of them are the cases when a or b are $\perp$. To show all the

[1] http://www.cl.cam.ac.uk/Research/HVG/Isabelle/library/HOL/Induct/ABexp.html

goals will take up too much space, but from goal A.2.2. the structure of the other goals will be readily apparent.

Goal A.2.2 $\forall b1\ a1\ a2.$
$(\forall \texttt{env}\ s.\texttt{evalb env}\ (\texttt{substb}\ s\ b1) = \texttt{evalb}\ (\lambda x.\texttt{evala env}\ (s\ x))\ b1) \wedge$
$\forall \texttt{env}\ s.\texttt{evala env}\ (\texttt{substa}\ s\ a1) = \texttt{evala}\ (\lambda x.\texttt{evala env}\ (s\ x))\ a1) \wedge$
$\forall \texttt{env}\ s.\texttt{evala env}\ (\texttt{substa}\ s\ a2) = \texttt{evala}\ (\lambda x.\texttt{evala env}\ (s\ x))\ a2)) \rightarrow$
$\forall \texttt{env}\ s.\texttt{evala env}\ (\texttt{substa}\ s\ (\texttt{IF}\ b1\ a1\ a2)) =$
$\texttt{evala}\ (\lambda x.\texttt{evala env}\ (s\ x))\ (\texttt{IF}\ b1\ a1\ a2)$

This proposition is proven in Sparkle by assuming the induction hypotheses and then reducing the evala function.

In this section we have shown how Sparkle derives an induction scheme for mutually recursive data types. The derived subgoals are subsequently easily proven by reducing the evaluation function. There is almost no interaction required from the programmer.

A.2.2 Co-Induction

Co-recursive programs are less well known than recursive programs, but the growing insight that they can be a very useful and elegant way of programming, means that proof techniques and tactics need to be made available to programmers.

The power of co-recursion is shown in an McIlroy's article on power series [15]. This article shows how power series can be implemented using streams of fractions. These infinite streams of integers represent the coefficients of the power series.

For instance the power series for cos x; $1-\frac{x^2}{2!}+\frac{x^4}{4!}-\frac{x^6}{6!}+\ldots$. is represented by the infinite list of fractions: [1,0,-1/2,0,1/24,0,-1/720,...]. Using infinitely recursing functions on these streams mathematical operations like addition, subtraction and even calculus can be defined on these power series.

An example of how simple these definitions are is the implementation of the sinus and co-sinus function:

```
sin x = integral cos x
cos x = 1 - (integral sin x)
integral x = [0:int1 x 1] where
  int1 [x:xs] n = [x/n:int1 xs (n+1)]
```

Since these programs operate on an infinite list, they will never terminate. This makes it impossible to use induction to conduct our proofs. A co-inductive proof tactic is needed. Several methods for dealing with co-recursive programs are described in [8]. Fixed point induction, fusion, bisimulation and the approximation lemma are mentioned. Our tactic uses the bisimulation relationship suggested by the property one is trying to prove. As described in [2], a bisimulation is a relation on a set of (infinite) elements of a certain data type with a co-algebraic structure defined

by its destructor functions. For lists this relationship would be: $R(a, b) \rightarrow$ `hd` $a =$ `hd` $b \wedge R($`ta` $a,$ `tl` $b)$. This relationship defines the co-inductive proof principle;

Theorem A.2.3 $\forall a, b.R(a, b) \rightarrow \forall a, b.a = b$

Suppose we want to prove the mathematical property of the sinus function that $-(\sin x) = \sin(-x)$, using the co-inductive proof principle. Applying the bisimulation tactic directly on this equation will not yield an easy proof. There is another recursive function within the definition of sin. As a general rule it is more productive to proof properties of that inner recursion first. So choose another property to help us:

Property A.2.4 $\forall z\ n. -($`int1` $z\ n) =$ `int1` $(-z)\ n$

Applying the bisimulation tactic to this equality relationship we get two proof obligations:

1. First we have to prove that the heads of both infinite streams are the same: `hd -(int1 z n) = hd (int1 (-z) n)`. By reducing int1 in both sides of the equation we get: `[-x/n:-int1 xs (n+1)] = [-x/n:int1 -xs (n+1)]`. Clearly the `hd` of those two lists is the same.

2. Subsequently we have to prove that the `tl` of both infinite streams satisfy the stated equality relationship. Choose $z =$ `xs` and $n = n + 1$ to see that it satisfies the required relationship.

Once we have proven property A.2.4, it is easy to prove that -(sin x) = sin (-x).

In this section we have shown how proofs on recursive functions that do not terminate can be conducted within Sparkle. As demonstrated by this example, it is fairly easy for a programmer to prove interesting properties using the bisimulation tactic.

A.3 Induction on mutually recursive functions

The multi-predicate induction principle for algebraic data types, although useful in many instances, will not make easy proofs possible for a large group of programs that need more sophisticated induction schemes.

For instance this function to calculate the greatest common divisor is not easily within reach of conventional induction. The way the function is structured, with either the first or the second argument decreasing by an arbitrary amount, makes it hard to find a proper induction scheme. For reasons of presentation we assume that integers can be represented by natural numbers, disregarding any overflow issues.

```
gcd :: Int Int -> Int
gcd 0 y = y
gcd x 0 = x
gcd x y
  | x < 0 || y < 0 = abort ("Arguments must be positive")
  | y < x          = gcd (x-y) y
  | otherwise      = gcd  x   (y-x)
```

A different kind of induction scheme is needed for functions where the arguments are combined or switched around. Functions where the recursive structure of its definition does not match the structure of the data type of its arguments need a different kind of scheme as well. This new kind of induction scheme that matches the structure of the function definition will make easy proofs possible.

We will propose a method that makes use of call graphs to determine which arguments should be used to construct a well founded ordering. This ordering will be the basis of the induction scheme.

In the next section we first describe the tactic we created to derive induction schemes for (mutually) recursive functions. Then we briefly describe what kind of related work is done on the subject of terminating recursive functions and the derivation of induction schemes. Finally we compare our method with the size change principle. One of the methods from the related work section.

A.3.1 Tactic for deriving induction schemes for mutually recursive functions

When trying to find a well founded ordering for mutually recursive functions one quickly runs into the problem that the number and position of arguments for each function can be different. Furthermore, applications of other functions not included in the mutually recursive function set make it hard to track the arguments that should be used for the well founded ordering. To track the way the original arguments are used in subsequent recursive function calls a graph is built. Each argument of the mutually recursive functions is a vertex and the way arguments are interdependent is represented by directed edges.

Creating a call graph

Suppose there is a set of mutually recursive functions $S = \{f_1, .., f_n\}$ Let f_k be one of those mutually recursive functions. Graph G is a pair (E, V) of edges E and vertices V.

For each function definition, for instance f_k the arguments $a_1, .., a_j$ of the function definition are added to the set of vertices. For all function calls $f_g \in S$ in the body of the function definition, the variables that are used in

the f_g function call are added to the set of vertices V and edges from (f_k a_i) to (f_g a_i) are added to the set of edges E.

For instance, take these function definitions:

```
f a1 a2 =
   | a1 < a2   = f (a1 + a2) a2
   | a1 == a2  = a2
   = g a1 a2
g a1 a2 = f a2 a1
```

They yield the directed graph $G = (V, E)$, where $V = \{(\texttt{f a1}), (\texttt{f a2}), (\texttt{g a1}), (\texttt{g a2})\}$ and $E = \{((\texttt{f a1}), (\texttt{f a1})), ((\texttt{f a1}), (\texttt{f a2})), ((\texttt{f a2}), (\texttt{f a2})), ((\texttt{f a1}), (\texttt{g a1})), ((\texttt{f a2}), (\texttt{g a1})), ((\texttt{g a1}), (\texttt{f a2})), ((\texttt{g a2}), (\texttt{f a1}))\}$.

Note that the first occurrence of a recursive function determines towards which recursive call the arguments are counted. A recursive call like `f a1 = g (f a1)` will yield the (sub)-graph: $V = \{(\texttt{f a1}), (\texttt{g a1})\}$, $E = \{((\texttt{f a1}), (\texttt{g a1}))\}$.

The only arguments that can possibly be of value to an induction argument are the ones that are able to create an infinite chain. Otherwise a simple exhaustion argument will do. Since there is a finite number of arguments and functions, only the arguments on a cycle in the graph are candidates for a well founded relation. Elements on a graph are on a cycle if their vertices are in the same strongly connected component. Using Tarjan's [20] algorithm we can find these components quickly.

Building a well founded relationship

Take for instance these mutually recursive functions, where arguments are combined and switched around.

```
f a b c = g (c+b) a
g x y   = f x y 1
```

The graph representing this function shows that f a b and g x y are strongly connected. The ordering we impose on this set of mutually recursive functions therefore has to take into account that for each function call f inside the function body of g, the arguments x and y combined have to be smaller than a and b together.

The key idea of this paper is how we construct a well founded relationship from these strongly connected components:

- For each possible transition between recursive functions $f_i \rightarrow f_j$, all vertices a_i and a_j that are in the same strongly connected component are added. If they are not, the lexicographical ordering of them is constructed.
- If a_i is an algebraic data type, a size function that counts the number of constructors in the definition is created.

In order to prove that the resulting function signifies a well founded relationship we have to prove that the functions we have constructed satisfy the definition of well foundedness:

Definition A.3.1 $WF(R) \equiv \forall P.(\exists w.P(w)) \rightarrow \exists min.P(min) \wedge \forall b.R\ b\ min \rightarrow \neg P(b)$

It formalizes the notion that for any non empty subset of elements of a set, there is a smallest element in that subset. This means that there can not be an infinitely descending chain of smaller elements.

- This means that for our addition and size functions we have to prove that if we have a well founded relationship $<$ for natural numbers or strings, the relationship constructed by mapping the arguments onto integers is well founded as well.

  Proposition A.3.2 $\forall f.\forall P.(\exists w.P(f\ w)) \rightarrow \exists min.P(min) \wedge$
  $\forall b.f\ b < f\ min \rightarrow \neg P(f\ b)$

  Proof A.3.3 *Suppose we have a non-empty well founded set S, a subset of the set of natural numbers or strings. We define T as* $\{x | b \in S \rightarrow x = f\ b\}$. *Since S is non empty, and f is a total function, T is non empty as well. Both addition and size functions are total. Since S is well founded, we have a smallest element* min'. *Define the smallest element* $min \in T \wedge min' = f\ min$. *min is indeed the smallest element, because if there is a smaller element b' in T,* $b' < f\ min$ *holds. This would mean, by definition of T from S that there is a* $f\ b$ *smaller than* $f\ min$. *However, this directly contradicts the well foundedness of S.*

- To show that the lexicographical ordering is well founded, we first define it as:

  Definition A.3.4 $\forall a, c \in A\ b, d \in B.R(a, b)(c, d) \equiv a < c \vee a = c \wedge b < d$

  We then have to prove that each subset has a smallest element.

  Proof A.3.5 *We can construct the smallest element directly. By taking the smallest element m from set A, and then look at all pairs (m,n) and take the smallest n from that set.*

Now that the well foundedness of the resulting function is established we can use it to derive induction schemes by means of the following theorem.

Theorem A.3.6 $WF(<) \rightarrow \forall P.(\exists x.P(x) \rightarrow \forall y.(x < y \rightarrow P(x) \rightarrow P(y)) \rightarrow \forall x.P(x))$

The instantiation of this theorem with the derived ordering is available to the user. Whenever it is applied to a recursive function call within the body of a function definition it raises for the user the obligation to prove that in this particular instance the arguments arranged according to the derived induction scheme are smaller than when entering the function definition. The program then deduces that the property holds for that recursive function call.

If the well founded ordering creates subgoals that can not be proven, the algorithm permits the user to supply measure functions for all the different possible function calls. These measure functions must map the arguments of a function call onto a natural number or string. From then on, the algorithm uses an induction scheme based on the ordering provided by the $<$ operator for integers and strings.

A.3.2 Example: tactic applied to gcd

For our earlier defined gcd function we want to prove that the greatest common divisor of all even numbers is bigger than 1.

Property A.3.7 $\forall x\ y.\, \gcd\ (x + x)\ (y + y) > 1$

When we apply this induction tactic to this proposition, specifying that we want it to derive an induction scheme which follows the recursive structure of the gcd function, the algorithm will determine which arguments are strongly connected. It will find that both arguments are, so the ordering imposed on them will be $R((x1, y1), (x2, y2)) = x1 + y1 < x2 + y2$. This ordering, combined with the well-foundedness theorem will yield this induction scheme:

Proposition A.3.8 $\forall P.(\forall x1\ x2\ y1\ y2.x1 + y1 < x2 + y2 \rightarrow (P(x1, y1) \rightarrow P(x2, y2))) \rightarrow \forall xy.P(x, y)$

The proof then proceeds by reducing the gcd function and splitting it into each of the different cases. The user has to prove that the property holds when one of the arguments is $\perp$ or 0. No induction hypothesis is needed for them.

Only when the user has to prove that the property holds for either $\gcd\ (x - y)\ y$ or $\gcd\ x\ (y - x)$ an induction hypothesis is needed due to the recursive function call. Both proofs are nearly identical, so we assume we just have to prove:

Goal A.3.9 $\gcd\ ((x + x) - (y + y))\ (y + y) > 1$

At that point an instantiation of our induction scheme will show that if we can prove that $x - y + y < x + y$, the property holds for the recursive function call.

Proof A.3.10 *This is easy, since the case* $y = 0$ *has already been handled, we may assume that* $\gcd((x - y) + (x - y))\ (y + y) > 1$, *which is the same as* $\gcd((x + x) - (y + y))\ (y + y) > 1$.

A.3.3 Related work

Much research has been done the past few years to find the proper induction schemes that will make proving properties easy. Deepak and Shakur have used the coverset induction principle [5]. Boulton and Slind use a multi predicate induction scheme and a proof procedure to find the proper induction hypotheses to use [3].

Tesketh uses middle out reasoning to guide the matching of induction hypotheses [6] and Slind describes a method to derive induction schemes from translating function definitions into higher order logic [18].

A few more authors have done closely related work by proving termination by means dependency graphs or call graphs. The earliest work was done by Arts and Giesl in, [19]. They translate functional programs into term rewrite systems and use dependency pairs to construct a well founded ordering. Abel and Altenkirch have written a system called foetus [1]. It only works for structurally smaller arguments however. Other research was done by Lee, Jones and Ben-Amram, described in [9]. They construct call graphs and prove termination by showing that if the program does not terminate it will contain an infinitely long chain of elements taken from a well founded set.

All those methods have in common that they require that the program analyzing the graph only considers arguments that are either known to get smaller or remain the same. Although this guarantees that termination is proven, sometimes it can prove to be more useful to leave the termination proof for later and to "guess" a well founded ordering.

Our tactic differs from the other methods in the way a well founded relationship is built using the information of the strongly connected components of the complete call graph.

A.3.4 Our method compared to other methods

The algorithm we have constructed creates a well founded relationship, however, it usually does not automatically prove that all recursive function calls have smaller arguments according to this relationship. Theoretically this may be considered to be a weaker result than most other methods that only look at arguments that they know are either equal or smaller. However, this weaker result is more powerful in practice as is shown below. Postponing the proof that the well founded ordering is a termination relationship has an advantage: The rest of the machinery of the theorem prover can be used to deduce that indeed the arguments are getting smaller.

- For instance in the following program on natural numbers, the information that the arguments are getting smaller is contained within the guards of the program. A program that simply looks at the arguments will not be able to deduce that the first argument is indeed getting smaller.

```
f x y z
| y < z = f (x + y - z) y z
| z < y = f (x - y + z) y z
= f (x-1) y z
```

- Another instance where programs that look at arguments will fail is if the information is hidden within the predicate that one is trying to prove. Take the following function and predicate:

```
f x y
| x = 0    = y
= f (x-y) y
```

  $\forall x\ y.y > 0 \rightarrow$ `f` $x\ y = y$.

  The only way the tactic can conclude that the argument is getting smaller each consecutive call is if it knows the precondition that can be derived from the predicate.

- A third instance where the methods from the related work section fail is in programs where one argument is increasing, but not as hard as the argument it is combined with is decreasing.

```
f x y    = g (x+1) (y-2) g
g x y    = h x+y-1
h x
 | odd x = f x/2 x/2
 = f ((x/2)-1) x/2
```

  None of the tactics that look at arguments that are getting smaller or stay the same can derive a well founded ordering for this set of functions, because the argument of h in the definition of g is not smaller than x or y.

All the methods mentioned in related work will not be able to derive an induction scheme for the above two mentioned cases. Another added advantage of guessing which variables are important and constructing a well founded relationship out of them is that unlike for instance the size change principle from Lee, Jones and Ben-Amram [9], the algorithm is not PSPACE hard. Tarjan's algorithm's complexity is in the order of the number of the edges and vertices combined.

It must also be mentioned that even if termination is proven, it is not immediately clear what kind of induction scheme can be derived from the

termination proof. After all, an induction hypothesis may only be invoked if an element of a call sequence is smaller and the algorithm of Lee, Jones and Ben-Amram only deduces it must decrease at some point, but not at which point.

Of course the downside of guessing a well founded relationship is that one can be wrong. The algorithm we construct will fail if the algorithms that need to be analyzed start with arguments that increase. A way to address this problem is by letting the programmer indicate which variables can be ignored in the derivation of the well founded ordering.

To see how our approximation algorithm works compared to the earlier mentioned procedure from Lee, Jones and Ben-Amram, we have taken the examples from their article and checked how we can prove them in Sparkle using our new tactic.

- Reverse function with accumulating parameter.

```
rev ls = r1 ls []
r1 [] a = a
r1 [l:ls] a = r1 (ls) [l:a]
```

  For this function will be deduced that the first and the second parameter of r1 are in different strongly connected components. Therefore the ordering will be the lexicographical product of both arguments. The subsequent proof obligation for the recursive function call of r1 will be that $\texttt{size ls} < \texttt{size } [\texttt{l} : \texttt{ls}] \vee (\texttt{size ls} == \texttt{size } [\texttt{l} : \texttt{ls}] \wedge \texttt{size } [\texttt{l} : \texttt{a}] < \texttt{size a})$.

  This is trivial to prove.

- Program with indirect recursion.

```
f []     x = x
f [l:ls] x = g ls x l
g a b c = f a [c:b]
```

  For this function the algorithm will detect that (g a) and (f ls) are in the same strongly connected component and (g b), (g c) and (f x) are in the same strongly connected component. That means that the proof obligation of the call to g from function definition f will be that $\texttt{size ls} < \texttt{size } [\texttt{l} : \texttt{ls}] \vee (\texttt{size x} + \texttt{size } [\texttt{l} : \texttt{ls}] < \texttt{size x})$

  While the proof obligation for the recursive function call of f from g will be $\texttt{size a} < \texttt{size a} \vee \texttt{size b} + \texttt{size c} < \texttt{size b} + \texttt{size c}$. This latter condition will never be true, so it is impossible to use the induction hypothesis.

  The solution is to reduce g in the proof of the property of f. Then one can use the f to f relationship, which states that $\texttt{size ls} < \texttt{size } [\texttt{l} : \texttt{ls}] \vee (\texttt{size ls} = \texttt{size ls} \wedge \texttt{size x} < \texttt{size x})$. The proof of this is straightforward.

- Function with lexically ordered parameters.

```
a m n
| m == 0 = n + 1
| n == 0 = a (m-1) 1
= a (m-1) (a m (n-1))
```

  For this program we will find that m and n are both in different components, so the algorithm will construct the lexicographical ordering on m and n to create an induction scheme. This means that the proof obligation for the first call of a is $m-1 < m \vee (m-1 == m \wedge n <$ `a` $m\ (n-1))$ and for the second call $m < m \vee m == m \wedge n - 1 < n$. All trivially easy to prove.

- Program with permuted parameters.

```
p m n r
| r > 0 = p m (r-1) n
| n > 0 = p r (n-1) m
= m
```

  This program has all arguments in the same strongly connected component. The proof obligation in order to use the induction hypothesis for the first call to p will therefore be $m+n+r-1 < m+n+r$, while for the second it will be $r+n-1+m < m+n+r$. Both are again easily proven.

- Program with permuted and possibly discarded parameters.

```
f x [] = x
f [] y = f y (tl y)
f x y  = f y (tl x)
```

  Again both arguments are in the same component. This means that for the first recursive call of f `size y` + `size (tl y)` < `size y` should hold. Unfortunately it does not, so the program will have guessed the wrong well founded relationship. It is up to the user to provide a measure function. For instance: `measure x y = size y`. By supplying the algorithm with this function, the user has to prove for the first function call that `size (tl y)` < `size y` and for the second function call a further reduction of f and a split case tactic on the variable x is needed to find that `size (tl y)` < `size y`, before the induction hypothesis may be used.

- Program with late-starting sequence of descending parameter values.

```
f a []      = g a []
f a [b:bs]  = f [b:a] bs
g [] d      = d
g [c:cs] d  = g cs [c:d]
```

The algorithm will conclude that all the elements are in different strongly connected components. That means that a lexical ordering is created for all the arguments. The following proposition does not hold:

$\texttt{size}\ [\texttt{b} : \texttt{a}] < \texttt{size}\ \texttt{a} \vee (\texttt{size}\ [\texttt{b} : \texttt{a}] == \texttt{size}\ \texttt{a} \wedge$
$\texttt{size}\ [\texttt{b} : \texttt{bs}] < \texttt{size}\ [\texttt{bs}])$

However, properties of a function like this can much more easily be proven by induction on the second argument of f and the first argument of g separately.

In the previous sections we have shown how a complete call graph is built from the function definitions and how a well founded relationship is constructed by means of the strongly connected components of that graph. This relationship is the basis of an induction scheme. We have shown for what kind of programs our method will yield an induction scheme where other methods fail. And by means of examples, taken from another research paper, we have shown that in practice, the examples which are intended to show the advantages of a theoretically strong method like the size change principle, can easily be proven using Sparkle extended with our new tactic.

A.4 Acknowledgements

I would like to thank all the participants of the Fifth Symposium on Trends in Functional Programming for their valuable input.

A.5 Future work

R. van Kesteren is enhancing Sparkle with proof rules for parameterized classes [21]. This will make proofs of properties of generic classes possible.

The authors are also working on a new co-inductive technique, based on guard-ed co-induction without the syntactic constraint that is required in other methods.

Furthermore, inclusion of features of Clean like dynamics and overloading, for which there is limited support now within Sparkle, deserve consideration.

A.6 Conclusion

By extending the induction and co-induction proof techniques for Sparkle, we have opened the theorem prover up for a wide set of proofs on properties of programs. We have devised a new method to derive induction schemes for mutually recursive functions, that closely match the structure of the underlying program. This method can create induction schemes to prove

properties of functional programs where other methods fail. Combined with the already easy to use interface, a programmer can now easily prove properties of recursive and co-recursive programs, bringing the integration of programming and proving closer.