Chapter 1

Induction and Co-Induction in Sparkle

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

Abstract: Sparkle is a proof assistant designed for the lazy functional programming language Clean. It is designed to facilitate proofs of first order logical predicates on existing and newly written programs. In order to achieve this it is of the utmost importance that the proof system is easy to use and integrated into the IDE. The implementation of Sparkle only has a basic induction principle so far. For many programs and complex proofs this will not suffice. To address this issue we have expanded Sparkle with several induction and co-induction proof techniques for mutually recursive functions and data types. By providing examples of proofs on programs, taken from various research papers, we show how the programmer can benefit from these added tactics. With our extensions Sparkle can be used to prove properties of a wide class of programs.

1.1 INTRODUCTION

Sparkle is the integrated proof assistant available with the lazy functional programming language Clean. 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 thirdly, the theorem prover is integrated into the develop-

[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`

ment environment of Sparkle. All these features are intended to encourage even 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 [MdM01].

Many different proof techniques have been incorporated into Sparkle, but so far only a basic form of induction was available to the user. To show what the basic induction of Sparkle is capable of, we replicate a case study in program verification in section 1.2. In order to improve the usefulness of the theorem prover we have implemented multi-predicate induction over mutually recursive algebraic data types. A description of the way algebraic data types are defined in Sparkle as well as the method used to implement multi-predicate induction is given in section 1.3. Another new feature is the possibility to have the 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. A detailed explanation of the implementation as well as the ideas behind finding the right scheme can be found in section 1.4.

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 1.5.1 we will demonstrate how a bisimulation tactic is implemented and how it can be used both for proofs on both algebraic data types as well as function calls.

Although all proofs and tactics are done within the Clean environment they are not limited to it. 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].

1.2 CHASING BOTTOMS REVISITED

In this section we will illustrate the capabilities of the basic induction scheme already present in Sparkle, by taking a recent case study. In their study of infinite

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

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

values in proofs of functional programs [ND04], Danielsson and Jansson make a strong case as to why we should bother with verifying functions for infinite and partial input. They use a simple parse and pretty printer program and try to prove that those functions are essentially eachother's inverse.

The parsed structure is made to resemble a binary tree to make the program not too trivial.

```
::T = L | B T T
```

The pretty printer does a pre-order traversal of the tree.

```
pretty :: T -> [Char]
pretty L        = ['L']
pretty (B l r)  = ['B'] ++ (pretty l) ++ (pretty r)
```

While the parser happily parses away until it either finds a leaf or continues parsing subtrees, first left, then right, when it finds a branch symbol.

```
parse :: [Char] -> (T,[Char])
parse ['L',xs]  = (L,xs)
parse ['B',xs]
   #  (l,xs) = parse xs
   #  (r,xs) = parse xs
   = (B l r,xs)
```

In their article Danielsson and Jansson first refine their properties to take into account that the objects that are parsed may be infinite. They then proceed to construct elaborate proofs using the approximation lemma and fixed point induction.

In Sparkle life is much easier when it comes to bottoms. They are a built in feature of Sparkle and even regular induction takes into account that the function might not be total and that the expression used to call the function may be infinite.

To show how Sparkle handles such cases we will prove one of two properties proven by Danielsson and Jansson. Our definition is slightly different, but equivalent to the one they use.

Proposition 1.1. $\forall x.\texttt{pretty}\ (\texttt{parse}\ x) = x \lor \texttt{pretty}\ (\texttt{parse}\ x) = \bot$

We can use the regular induction tactic in Sparkle. It will split the proof in several parts. First one has to prove the case where there is an infinite string as input.

Proposition 1.2. $\forall x.\texttt{pretty}\ (\texttt{parse}\bot) = \bot \lor \texttt{pretty}(\texttt{parse}\bot) = \bot$

A simple reduction will make Sparkle recognize that the function is strict and it will proceed to prove it. An induction scheme for finite and partial input is then generated. The proof of that induction scheme is basically the same as described in the article.

Induction proofs on lazy types are not always allowed. Predicates must be deemed admissible, which in this case they are. The way admissibility is determined within Sparkle is based on Paulson's criterium as stated in his Logic and Computation book [Pau87] In the previous example the predicate was admissible and Sparkle did not need the more elaborate proof techniques Danielsson Jandsson used.

1.3 INDUCTION ON MUTUALLY RECURSIVE DATA TYPES

As seen in the previous section, induction within Sparkle already handles the possibility of infinite values, however the implemented induction tactic only derived induction schemes for algebraic data types that are directly recursive. Mutually recursive induction schemes were not available.

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 [MJCG93], uses multi-predicate induction. In this section we will briefly describe the basics of this method and our implementation of it.

Take for instance a simple expression language consisting of arithmetic and boolean expressions. This example can be found on the HOL web-site[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. For brevity we will omit the function bodies from our paper.

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

Both the evala, evalb pair and substa, substb pair of functions will have mutually recursive function calls to each other because of the nature of the data type. When trying to prove properties of these functions, it will undoubtedly be necessary to build an induction scheme where we can assume that a certain property for evala holds while trying to prove evalb and the other way round.

1.3.1 A tactic for induction on mutually recursive types

Induction on algebraic data types, be they mutually recursive or not is based on the implicit well founded ordering that is present in the definition of the data type. A specific instance of a data type is bigger than another if it can be constructed from

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

it using the rules in the algebraic data type definition. The basis of an induction scheme will be the wellfoundedness theorem.

Theorem 1.1. $Wellfounded(<) \rightarrow \forall P.(\exists x.P(x) \rightarrow \forall y.(x < y \rightarrow P(x) \rightarrow P(y)) \rightarrow \forall x.P(x))$

This theorem basically states that if we have a well founded ordering and we want to prove a certain predicate, we may assume in our proof that the property holds for all elements we can prove to be smaller than a certain element. Translated to data types this means that we may assume the predicates over the smaller data types to be true when constructing a bigger data type out of them. The only caveat is that there must be a smallest element, otherwise the ordering will not be well founded. For instance, the data type definition `::List a = Element a (List a)` will not yield a valid induction scheme.

Algebraic data types in Clean are built in the following way:

```
ADT [TypeVariable] = [TypeConstructor]
TypeConstructor    = Constructor [Type]
Type               = BasicType
                   | ADT [Type]
                   | TypeVariable
                   | ArrowType
ArrowType          = Type -> ArrowType
BasicType          = Int
                   | Real
                   | Bool
                   | Char
                   | Array
```

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 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.

A simple test checking whether one of the constituent algebraic data type definitions contains a non-recursive option will be enough to satisfy the criterion of having a smallest element.

The procedure that creates the induction scheme is as follows:

```
FOR EACH ADT that is mutually recursive
  get the ADT definition;
  FOR EACH TypeConstructor in the list
  of the ADT definition
    FOR EACH Type in the list that follows
    the Constructor
      instantiate a variable;
      IF Type is an ADT that is in the list of
      mutually recursive ADT's THEN
```

```
        construct an antecedent with
        the predicate of that ADT;
      FI
    CHAE ROF
    Construct a proposition from the
    constructor and the variables and
    assume the antecedent(s);
    Have the user prove the proposition;
  CHAE ROF
CHAE ROF
```

Consider the following data type definition:

```
::MRTree a = Leaf a
         | TreeNode a (RMList a) (RMList a)
::RMList a = EmptyRMList
         | ListNode a (MRTree a)
```

Suppose we have a predicate Q::(MRTree a) and predicate P::(RMList a). When the above mentioned procedure is applied to this example, we will find that both the Leaf and the EpmtyRMList constructor contain no reference to one of the recursive data types. This means that they have to be proven independently. However, for the TreeNode and ListNode constructors we may assume antecedents. Combined, this yields the following induction scheme:

Proposition 1.3. $\forall PQ.P(\texttt{EmptyRMList}) \wedge$
$(\forall x :: \texttt{a}, y :: (\texttt{MRTree a}).Q(y) \rightarrow P(\texttt{ListNode}\ x\ y)) \wedge$
$(\forall x :: \texttt{a}.Q(\texttt{Leaf a})) \wedge$
$(\forall x :: \texttt{a}, y :: (\texttt{RMList a}), z :: (\texttt{RMList a}).P(x) \wedge P(y) \rightarrow Q(\texttt{TreeNode}\ x\ y\ z))$
$\rightarrow \forall x :: (\texttt{RMList a}), y :: (\texttt{MRTree a}).P(x) \wedge P(y)$

1.3.2 Mutually recursive tactic applied to a simple language

To look at how this tactic benefits the programmer in conducting proofs of properties of his programmer we return to the previously mentioned example. 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 1.1. $\forall \texttt{env}\ s\ a.\texttt{evala env}\ (\texttt{substa}\ s\ a) = \texttt{evala}(\lambda x.\texttt{evala env}\ (s\ x))\ a \wedge$
$\forall \texttt{env}\ s\ b.\texttt{evalb env}\ (\texttt{substb}\ s\ b) = evalb(\lambda x.\texttt{evala env}\ (s\ x))\ b$

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

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

1.4 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. 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 [D.K96]. Boulton and Slind use a multi predicate induction scheme and a proof procedure to find the proper induction hypotheses to use [Bou00].

Tesketh uses middle out reasoning to guide the matching of induction hypotheses [Hes91] and Slind describes a method to derive induction schemes from translating function definitions into higher order logic [Sli97].

1.4.1 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. Furthermore, we added Tarjan's algorithm for finding strongly connected components in a directed graph [Tar72] to determine the connectedness of arguments. 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.

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 where each argument of each of the mutually recursive function is a vertex and the way arguments are used is represented by directed edges.

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.

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.

```
FOR EACH function
  get function definition;
  make a vertex in the graph for each argument
  of the function;
  FOR EACH of the recursive function calls in this
     definition add a vertex in the graph for each
     argument that is used in the recursive call;
  HCAE ROF
HCAE ROF
```

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[1] and strings, size[2] 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 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 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 [Sli97]. 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
```

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

[2] 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.

```
 | 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 1.5. $\forall f :: (Int, []) \rightarrow \texttt{Bool}.(\exists w.f\,w) \rightarrow \exists m.f\ m \wedge \forall b.\texttt{isSmaller}\ b\ m \rightarrow \neg f\ b$

1.4.2 Induction on functions 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 1.6. $\forall xy.\gcd(x+x)(y+y) > 1$

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

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

1.5 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 [JG04]. 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 sections 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.

1.5.1 Bisimulation with respect to algebraic data types

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 [McI99]. 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.

Bisimulation tactic

As described in other articles, in particular [BJ96], 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 1.2. $\forall a,b.R(a,b) \rightarrow \forall a,b.a = b$

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 equality defines the 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
```

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

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)

1.5.2 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[1]. 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)
```

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.

Tactic for guarded co-induction

This method is mentioned in Gibbons' and Hutton's article [JG04] 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[Coq94].

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
```

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

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: $\texttt{odd}(\texttt{tm}\ [\texttt{True}]) = tm\ [\texttt{True}]$

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

Proposition 1.9. $\forall n.\texttt{tm}\ n = \texttt{odd}(\texttt{tm}(\texttt{cmpl}\ [\texttt{True}]\ n))$

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

Proposition 1.10. $\forall n.\texttt{odd}(\texttt{cmpl}\ [\texttt{True}]\ n) = \texttt{cmpl}\ [\texttt{True}]\ n$

Proposition 1.11. $\forall n.\texttt{odd}(\texttt{cmpl}\ x\ n) + +z) = \texttt{odd}(\texttt{cmpl}\ x\ n) + +(\texttt{odd}\ z)$.

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.

1.6 CONCLUSION AND RELATED WORK

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.

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[RvK04]. 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.

By adding bisimulation to the proof tactics we have also expanded the capabilities of Sparkle as a theorem prover for modeling techniques like process algebra. An implementation of process algebra and proofs of the equality of models within process algebra should be fairly easy now.

REFERENCES

[BJ96] J. Rutten B. Jacobs. A tutorial on (co)algebras and (co)induction. *Bulletin of the EATCS*, pages 222–259, 1996.

[Bou00] Richard Boulton. Multi-predicate induction schemes for mutual recursion. Informatics research report EDI-INF-RR-0046, University of Edinburgh, 2000.

[Coq94] 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.

[D.K96] 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.

[Hes91] J.T. Hesketh. *Using Middle-Out reasoning to Guide Inductive Theorem Proving*. PhD thesis, University of Edinburgh, 1991.

[JG04] G. Hutton J. Gibbons. Proof methods for corecursive programs. *Fundamenta Informaticae XX*, pages 1–13, 2004.

[McI99] M. Douglas McIlroy. Functional pearls: Power series, power serious. *Journal of Functional Programming*, (9):323–335, 1999.

[MdM01] R. Plasmeijer M. de Mol, M. van Eekelen. Theorem proving for functional programmers. *Selected Papers from the 13th International Workshop on Implementation of Functional Languages*, 2001.

[MJCG93] T. F. Melham M. J. C. Gordon. *A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[ND04] P. Jansson N. Danielsson. Chasing bottoms. *Mathematics of Program Construction: 7th International Conference, MPC 2004*, pages 85–109, 2004.

[Pau87] L.C. Paulson. *Logic and Computation: interactive proof with Cambridge LCF*. Cambridge University Press, 1987.

[RvK04] M. de Mol R. van Kesteren, M. van Eekelen. Proof rules for parametrized classes. In *TFP 2004: Fifth Symposium on Trends in Functional Programming*, 2004.

[Sli97] 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.

[Tar72] R. Tarjan. Depth first search and linear graph algorithms. *SIAM journal of Computing*, pages 146–160, 1972.