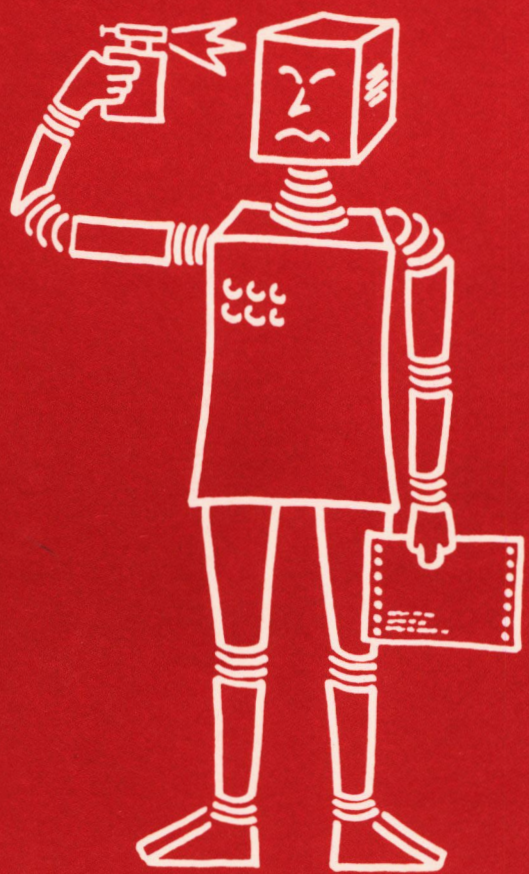
SEKI-PROJEKT

SEKI MEMO

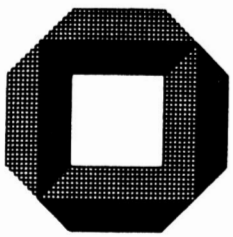
Universität Kaiserslautern
Fachbereich Informatik
Postfach 3049
D-6750 Kaiserslautern
West Germany

Universität Karlsruhe
Institut für Informatik I
Postfach 6380
D-7500 Karlsruhe
West Germany



Existence Proofs by Induction Using Methods of Program Synthesis

Susanne Biundo
Frantiŝek Zboray



UNIVERSITÄT KARLSRUHE
FAKULTÄT FÜR INFORMATIK
Postfach 63 80, D 7500 Karlsruhe 1

Bericht Nr. 16/84

Susanne Biundo
Institut für Informatik I
Universität Karlsruhe
Postfach 6380
D-7500 Karlsruhe 1

Frantiŝek Zboray*
Institut für Angewandte Informatik
und Formale Beschreibungsverfahren
Universität Karlsruhe
Postfach 6380
D-7500 Karlsruhe 1

* on leave from

Katedra pocitacov (EV SVST)
Universität Bratislava
Vazovova 5
81218 Bratislava (Tschechoslowakei)

EXISTENCE PROOFS BY INDUCTION USING METHODS OF PROGRAM SYNTHESIS

Susanne Biundo
Frantiŝek Zboray

1. Introduction

1.1 Automatic Induction

Induction is the basic technique used to prove properties of functions and predicates defined on well-founded sets. As a consequence proofs by induction play a central role in several subfields of mathematics such as arithmetic, formal logic, formal languages, algebra etc.

In computer science proofs by induction are important to verify programs, i.e. to prove properties of loops and recursive procedures.

Proving theorems by induction using an automated theorem prover presupposes a formalism which makes it possible to

- define well-founded sets,
- define functions and predicates which operate on these well-founded sets and
- formulate properties of these functions and predicates.

In Boyer and Moore's theorem proving system [BM79] for instance so-called shells are used to represent well-founded sets. The system offers a LISP-like definition principle to define functions using the techniques of definition by cases, by recursion and by functional composition. Terms and formulas are represented like S-expressions in LISP [WH81]

The language of the Boyer/Moore system can also be viewed as a programming language based on LISP:

The shell principle makes it possible to define abstract data-types [GTW77] and by the "definition principle" [BM79] functions operating on them can be introduced. This fact is not accidental:

Since the system was designed to mechanize theorem proving by induction, i.e. to prove theorems in constructive theories, the objects under consideration must be defined in a constructive way.

Therefore the induction system presently being developed as a part of the

MARKGRAF KARL REFUTATION PROCEDURE,

(an automated theorem prover developed at the University of Karlsruhe [BES81, DMW82, Ohl82]) also provides some kind of programming language to make possible the definition of well-founded sets as well as definitions of functions and predicates *over* these well-founded sets by means of so-called structure, function, and predicate expressions.

For example the concept of natural numbers (*nat*) together with the *predecessor* (*pred*) and *difference* (*diff*) functions and the predicate *even* may be defined by the following expressions:

```
STRUCTURE    0    s(nat) : nat
```

0 is a base constant, *s* is the constructor function which applied to an element of *nat* always yields another element of *nat*.

```
FUNCTION   pred(x:nat) : nat =
             IF  x = 0  THEN  0
             IF  ∃u : nat   x = s(u)  THEN  u
FUNCTION   diff(x,y:nat) : nat =
             IF  y = 0  THEN  x
             IF  ∃v : nat   y = s(v)  THEN pred(diff(xv))

PREDICATE  even(x:nat) =
             IF  x = 0  THEN  true
             IF  ∃u : nat   x = s(u)  THEN not even(u)
```

For each function or predicate expression the induction system will carry out a so-called consistency check.
This check consists in testing whether

(i) the function (predicate) symbol is new in the sense that no definition for it already exists

(ii) the definition is complete, i.e. the function (predicate) is totally defined

(iii) the definition is unique

(iv) the function (predicate) terminates, i.e. in every recursive call of the function (predicate) occurring in the then-part of the definition the parameters decrease according to a certain ordering.

Let us assume that a definition of the following form is given:

FUNCTION f(x:nat) : nat =

IF A_1 THEN t_1

.

.

.

IF A_n THEN t_n.

The A_i ($1 \leq i \leq n$) are formulas characterizing the definition cases and the t_i are terms. The uniqueness test (iii) consists in giving the definition as input to an algorithm which works as follows:

From the input definition a so-called uniqueness formula will be constructed which is the prenex normal form of

$$\bigwedge_{ij} [A_i \wedge A_j \rightarrow t_i = t_j] \qquad (1 \leq i,j \leq n;\ i \neq j) .$$

That means whenever two conditions are true the corresponding terms have to be equal.

Now certain (logical) simplification rules (for instance, tautology-elimination and subsumption) are applied to the uniqueness formula and the simplification algorithm returns TRUE, FALSE or a simplified formula which is equivalent to the uniqueness formula.

If the result is a simplified formula this formula will be given to the theorem proving system.

The tests (ii) and (iv) are carried out similarly; test (i) simply consists in looking up a table of all defined symbols.

In case of a successful consistency check (i.e. test (i) succeeds and the formulas of tests (ii) - (iv) given back by the simplification algorithm are TRUE or are proved by the system) the definition is accepted and will then be translated into a set of first-order-formulas, the so-called definition formulas, which provide a semantic for the structure-, function- and predicate expressions.

Thus, because they satisfy the consistency check the definitions of *nat*, *pred*, *diff* and *even* above lead to the following formulas:

(AX1) $\forall x : nat\ x = 0 \vee \exists y : nat\ x = s(y)$

every element of *nat* is equal to the base constant or equal to a constructor-term

(AX2) $\forall x : nat\ s(x) \neq 0$

every constructor-term is unequal to the base constant

(AX3) $\forall x,y : nat\ s(x) = s(y) \leftrightarrow x = y$

the constructor function is injective

(AX4) $\forall x : nat \quad x = 0 \quad \rightarrow \quad pred(x) = 0$ — pred

(AX5) $\forall x,u : nat\ x = s(u) \rightarrow pred(x) = u$

(AX6) $\forall x,u : nat\ y = 0 \quad \rightarrow \quad diff(xy) = x$ — diff

(AX7) $\forall x,y,v : nat\ y = s(v) \rightarrow diff(xy) = pred(diff(xy))$

(AX8) $\forall x : nat \quad x = 0 \quad \rightarrow \quad [even(x) \leftrightarrow \underline{true}]$ — even

(AX9) $\forall x,u : nat\ x = s(u) \rightarrow [even(x) \leftrightarrow \underline{not}\ even(u)]$

As can be seen from the form of our axioms we are dealing with a many-sorted first-order calculus. From now on we shall omit the sortsymbols except when treating concrete formulas like the above axioms.

It can be proved that because of the consistency check each set of first-order formulas which consists only of definition formulas (like the AXi above) is consistent and therefore possesses a model. These formulas AX together with an infinite set of induction axioms IND^* constitute the axiom set of the induction theorem proving system.

If the system is given a formula Φ to be proved the system selects a finite subset IND of IND^* which seems to be adequate to prove Φ, and then attempts to infer Φ from AX $\cup$ IND.

For instance if Φ is given as $\forall x,y$: nat $\psi(xy)$ where $\psi(xy)$:= pred(diff(xy)) = diff(pred(x)y) the system selects the induction axiom $\psi_1 \wedge \psi_2 \rightarrow \Phi$

where ψ_1 := $\forall x$: nat $\psi(xO)$

and ψ_2 := $\forall y$: nat [$\forall x$: nat $\psi(xy)$ $\rightarrow$ $\forall z$: nat $\psi(zs(y))$]

and attempts to find a proof for ψ_1 and ψ_2:

ψ_1: $\forall x$: nat pred(diff(xO)) = diff(pred(x)O)

$\downarrow$ by AX6

$\forall x$: nat pred(x) = pred(x)

$\downarrow$ by reflexivity of the equality

TRUE

ψ_2: $\forall y$: nat [$\forall x$: nat $\psi(xy)$ $\rightarrow$ $\forall z$: nat pred(diff(zs(y))) = diff(pred(z)s(y))]

$\downarrow$ by AX7

$\forall y$: nat [$\forall x$: nat pred(diff(xy)) = diff(pred(x)y)
$\rightarrow$ $\forall z$: nat pred(pred(diff(zy))) = pred(diff(pred(z)y))]

$\downarrow$ by an instance of the induction hypothesis $\psi(xy)$

$\forall y$: nat [$\forall z$: nat pred(diff(pred(z)y)) = pred(diff(pred(z)y))

$\downarrow$ by reflexivity of the equality

TRUE

1.2 Existential Quantifiers and Proofs by Induction

However difficulties arise if the task is to prove a formula involving an existential quantifier, for instance to prove the formula

(0) $\forall x,y : nat\ \exists z : nat\ diff(zy) = x.$

In the context of induction proofs where the task is to prove formulas in constructive theories it is sometimes difficult to prove such formulas because the existentially quantified variable causes a search for a solution.

In a given formula

(1) $\forall x_1,..,x_n\ \exists y\ \psi[x_1...x_n y]$

it is therefore a useful prooftechnique to replace the existential quantified variable y by a term using a skolemfunction f and then to prove the resulting formula:

(2) $\forall x_1,...,x_n\ \psi[x_1...x_n\ f(x_1...x_n)]$

instead. This technique is sound because

(3) $\forall x_1,...,x_n\ \psi[x_1...x_n\ f(x_1...x_n)] \rightarrow \forall x_1,...,x_n\ \exists y\ \psi[x_1...x_n y]$

is valid for each formula ψ.

But why is it easier to prove formula (2) than formula (1)? Suppose in our set of axioms AX we have definition formulas for the addition of natural numbers denoted by the function symbol plus. Substituting *plus* for f in

(4) $\forall x,y : nat\ diff(f(xy)y) = x$

formula (4) is easily proved by induction. The reason for this is that the existentially quantified variable y in (0) which causes a search for a solution is replaced by a term using a function which is constructively defined and so a

search is obsolete.

But what should be done if it is not known which of the given functions for which definition axioms are present should replace the skolem function f such that (4) becomes provable (or even worse what should be done if no such function is in the axiom set AX at all)?

The answer is, and this is the central idea of our work, that we attempt to synthezise definition formulas for the skolem function f such that (4) becomes provable where we use formula (4) as a specification.

For the present purpose we shall restrict our attention to formulas of the form:

$\forall x_1 \ldots x_n \; \exists y \; \psi[x_1 \ldots x_n y]$ with ψ being a first-order formula without any quantifier.

Now we shall demonstrate our technique by an example: Given the formula

(5) $\forall xy$: nat $\exists z$: nat pred(z) = diff(pred(x)y).

By skolemization we obtain the formula

(6) $\forall xy$: nat pred(f(xy)) = diff(pred(x)y)

which we try to prove by induction:

Base case: y = 0

$\forall x$: nat pred(f(x0)) = diff(pred(x)0)

↓ by AX6

$\forall x$: nat pred(f(x0)) = pred(x).

Defining

(7) $\forall xy$: nat y = 0 → f(xy) = x

the base case is proved. □

Induction Step:

Our induction hypothesis is to assume that

(8) ∀x : nat pred(f(xy)) = diff(pred(x)y).

We have to prove

∀x : nat pred(f(xs(y))) = diff(pred(x)s(y))

↓ by AX 7

∀x : nat pred(f(xs(y))) = pred(diff(pred(x)y))

↓ by (8)

∀x : nat pred(f(xs(y))) = pred(pred(f(xy)))

Defining

(9) ∀x,y : nat f(xs(y)) = pred(f(xy))

the induction step is proved. □

The formulas (7) and (9) constitute the definition formulas of a new function expression

```
FUNCTION  f(x,y : nat): nat =
  IF  y = O  THEN  x
  IF  ∃v : nat y = s(v) THEN pred(f(xv))
```

i.e. we have synthesized the difference function on natural numbers. Now as a final step we have to verify that f fulfills the consistency conditions (thus guaranteeing that such a function f in fact exists) and after a successful verification a proof of the initially given formula (5) can easily be done.

In the well-known induction theorem proving systems [BM79, Aub79] the problem of existence proofs is solved rather radically by simply inhibiting the explicit and implicit usage of existential quantifiers:

"Advocates of quantification may feel that our lack of quantification makes it difficult for us to state certain conjectures. We agree, but we observe that the use of explicit existential quantification makes it more difficult to find constructive proofs." [BM79, p. 84].

In these systems the user is forced to decide which known function has to replace the skolem function or even worse to define a new one.

We want to shift the burden of searching for skolem functions and their constructive definitions from the user to the system: This paper presents an outline of a subsystem of an induction theorem prover which synthesizes the definition of skolem functions (in our example the formulas (7) and (9)) in order to replace existentially quantified variables in an in-constructive formula (5) obtaining a formula (6) which can be proved automatically by induction using the definition formulas (7) and (9) as axioms.

In general a function definition for the skolem function f is generated and then it has to be proved that the skolemized formula (2) becomes true.

An important point is, and this will be explained in detail later on, that in almost all cases the synthesis process it-self can be viewed as an induction proof of formula (2).

In section 1.4 it will be pointed out that a proof of formula (2) represents a solution of the initial problem: a proof of formula (1).

1.3 Program Synthesis

Deductive program synthesis [MW79/1, MW79/2, Bi80] is one of the proposed methods of deriving constructive (algorithmic) definitions of functions.

It consists in transforming a specification (i.e. a description of a function given in a specific high level specification language) according to certain rules, the transformation rules [MW79/1, MW79/2] into an algorithm which computes the specified function. The specification may be completely inconstructive giving no hint of how to "implement" the function. The transformation rules represent knowledge about the program's subject domain and their application is guided by several strategies and heuristics [MW79/2].

Goad [Go80] shows how to extract algorithmic definitions of skolem functions from already existing (hand-written) proofs.

These techniques can now be applied for our purpose as follows:

The specification and programming language is first-order predicate logic.

Let us consider a relation R(xyz), for instance

(1) $$z - y = x \quad .$$

The goal is to find a program (in logic) which for each pair of input variables x and y computes a value for an output variable z such that (1) is satisfied (if (1) has a solution at all).

Using the proper quantifiers to express this fact

(2) $$\forall xy \; \exists z \; z - y = x$$

means that (1) must be a total relation. Transforming (2) into

(3) $$\varphi_o :\equiv \forall xy \; f(xy) - y = x$$

(i.e. the introduction of a skolem function) causes a further restriction: the relation (1) has to be unique wrt. z.

Now various transformation rules are applied to φ_o

$$(*) \qquad \varphi_o \equiv\!> \ldots \varphi_i \equiv\!> \varphi_{i+1} \equiv\!> \ldots \varphi_k$$

until a formula φ_k is derived (which includes the function symbol f) and which can be interpreted as a recursive algorithm computing f.

φ_k is a conjunction of so-called definition formulas and may look like this:

$$[y = 0 \rightarrow f(xy) = x] \wedge [y = s(v) \rightarrow f(xy) = s(f(xv))]$$

φ_k has to fulfill the consistency conditions described in chapter 1.1, i.e. the definition of f has to be complete, unique and terminating and f has to be a new function symbol.

Finally it has to be proved that φ_k is correct in the sense that it is a solution of the given problem: One has to prove (possibly by induction): $\varphi_k \rightarrow \varphi_o$, i.e. that φ_k satisfies the specification.

The transformation rules we shall use (see section 2.1) have the following property:

R: A transformation rule TR has the property R iff for all formulas φ, φ':

$$\varphi \underset{TR}{\equiv\!>} \varphi' \qquad \text{implies} \quad \varphi' \rightarrow \varphi \ .$$

This is very important because R holding for each rule used in a transformation (*) frees us from having to prove explicitly the correctness of φ_k: the correctness proof is implicitly part of the synthesis process which in general involves some kind of induction.

1.4 The Correctness of "Proving by Synthesis"

In this section it will finally be demonstrated that a successful synthesis of a skolem function (i.e. having synthesized a definition φ_k such that φ_k fulfills the consistency check and φ_k has been proved correct indeed represents a proof not only of the skolemized formula but of the original existentially quantified one.

Let us suppose that given:

- a formula Φ with
  $\Phi := \forall x_1 \ldots x_n \; \exists y \;\; \psi[x_1 \ldots x_n y]$,
- a finite set of axioms:
  among others the definition formulas for all sort, function, and predicate symbols occurring in Φ, and
- an infinite set of induction axioms IND^*.

The task is to prove

$$(1) \qquad IND^* \cup AX \vdash \Phi \quad .$$

First of all formula Φ will be skolemized to eliminate the existential quantification. The result will be a formula $\hat{\Phi}$ with

$$\hat{\Phi} := \forall x_1 \ldots x_n \; \psi[x_1 \ldots x_n \; f(x_1 \ldots x_n)] \quad \text{and}$$

$$(2) \qquad \vdash \hat{\Phi} \rightarrow \Phi \quad .$$

Now a definition for the skolem function f (i.e. a set of definition formulas DEF_f) will be synthesized from the specification $\hat{\Phi}$ and its consistency will be proved:

$$(3) \quad IND^* \cup AX \vdash total(f) \wedge unique(f) \wedge terminating(f).$$

Let us assume there exists an interpretation I with
$I \vdash IND^* \cup AX$.

With (3) we have shown that for each S-interpretation I with $I \models IND^* \cup AX$ there exists a $S \cup \{f\}$-interpretation I_f with

(4) $\qquad I_f \models IND^* \cup AX \cup DEF_f \quad ,$

which differs from I only in the interpretation of the function symbol f.

Now we have to prove that:

(5) $\quad IND^* \cup AX \cup DEF_f \models \hat{\Phi} \quad .$ (Correctness)

Let I be a S-interpretation with $I \models IND^* \cup AX$. Then

$I_f \models IND^* \cup AX \cup DEF_f$ by (4),

$I_f \models \hat{\Phi}$ by (5),

$I_f \models \Phi$ by (2), and

$I_f \models \Phi \quad ,$ because f does not occur in Φ.

Thus for each S-interpretation I we have:

If $I \models IND^* \cup AX$

then $I \models \Phi \quad ,$

i.e. $IND^* \cup AX \models \Phi \quad .$ □

2. The Synthesis System

2.1 Transformation Rules

In this section the transformation rules will be presented which we use to synthesize the definitions of various skolem functions. They are classified according to their functions.

For the following the letters A, B, C, ... denote quantifier-free first-order formulas; t, t_1, ... and τ, τ_1, ... denote terms.

The rules are of form

$$\frac{A}{\begin{array}{l} B_1 \\ B_2 \\ \vdots \\ B_n \end{array}}$$

and each of the formulas A, B_1, ..., B_n has to be read as if it is prefixed by a universal quantification of all variables occurring in it.

A[object] means that object occurs in A and A[$\text{object}_1 \leftarrow \text{object}_2$] is the formula obtained from A by replacement of object_1 by object_2.

2.1.1 The Induction Rule

The definition of a skolem function f, like those of the already defined functions and predicates occurring in the specification φ_o, has to be done by case analysis and by recursion.

The problem of how to get a suitable recursion scheme for f is quite similar to the problem of generating an induction scheme for a formula φ using the definitions of the functions and predicates occurring in it [BM79].

One successful heuristic is to use the recursion scheme of one of the most nested functions of φ_o. A most nested function is a function occurring at an innermost position in φ_o. If no such function is available the recursion scheme of a predicate occurring in φ_o can be taken as well.

The way to generate the case analysis and the recursion for the definition of f from the definition of a most nested function or a suitable predicate is given by the induction rule.

It reads as follows:

$$\text{IND}\ (\text{"induction"})\qquad \frac{A[f(x_1 \ldots x_n)\ \rho(t_1 \ldots x_i \ldots t_m)]}{\begin{array}{l}\gamma_1 \rightarrow A \\ \quad\vdots \\ \gamma_t \rightarrow A \\ \gamma_k \wedge \forall z_1 \ldots z_l\ A[x_i \leftarrow \hat{t}_i'] \rightarrow A\end{array}}$$

where

- ρ is a recursively defined function or predicate
- f is the skolem function
- $x_i \in \{x_1, \ldots, x_n\}$, i.e. the variable x_i occurs in A as an argument of f as well as an argument of ρ
- x_i is a recursion argument of ρ

- the definition formulas of ρ are:

$$\psi_1 \rightarrow \rho(y_1 \ldots y_i \ldots y_m) \underset{(\leftrightarrow)}{=\!=} \tau_1$$
$$\vdots$$
$$\psi_k \rightarrow \rho(y_1 \ldots y_i \ldots y_m) \underset{(\leftrightarrow)}{=\!=} \tau_k[\rho(\hat{t}_1 \ldots \hat{t}_i \ldots \hat{t}_m)]$$
$$\vdots$$
$$\psi_r \rightarrow \rho(y_1 \ldots y_i \ldots y_m) \underset{(\leftrightarrow)}{=\!=} \tau_r$$

- $\psi_1, \ldots, \psi_r$ are conjunctions of atomar or negated atomar formulas
- $\tau_1, \ldots, \tau_r$ are terms or atomar or negated atomar formulas resp. and ρ neither occurs in τ_s ($s \in \{1,\ldots,r\}\backslash\{K\}$) nor in $\hat{t}_1, \ldots, \hat{t}_m$
- for all $j \in \{1,\ldots,t\}$ is
  $j \in \{p \mid p \in \{1,\ldots,r\}\backslash\{K\}$ and there exists an $x \in \{x_1,\ldots,x_n\}$ with: x occurs in $\psi_p[*]\}$
- $\psi_j[*] := \psi_j[y_1 \leftarrow t_1] \ldots [y_i \leftarrow x_i] \ldots [y_m \leftarrow t_m]$
  $j \in \{1,\ldots,r\}$
- with $\psi_j[*] = B_1 \wedge \ldots \wedge B_q$
  is $\gamma_j := \bigwedge_K B_K$ for all $K \in \{1,\ldots,q\}$ with: there exists an $x \in \{x_1,\ldots,x_n\}$ which occurs in B_K
- $\gamma_1 \vee \ldots \vee \gamma_t \vee \gamma_k$ is a theorem
- $\{z_1,\ldots,z_l\} = \text{vars}(A)\backslash\text{vars}(\hat{t}_i')$
  where vars (object) is the set of all variables occurring in object
- $\hat{t}_i' = \hat{t}_i[*]$.

The induction rule is the first rule to be applied to a specification φ_o to start the process of synthesis. If its application fails because no suitable function or predicate

is available a complete recursion scheme will be constructed by selecting a variable parameter of f and setting up cases and recursion according to the associated structure definition.

Given, for instance, the following structure definition of lists of natural numbers:

STRUCTURE empty cons(nat list): list.

Let us now assume that f(xyz) is the skolem term of a formula φ_o and rule IND cannot be applied; let y be the selected parameter of sort list.

The formulas generated to synthesize the definition of f for the base case or the recursion case resp. would be:

$$y = \text{empty} \rightarrow \varphi_o$$

$$y = \text{cons}(\text{nl}) \wedge \forall x z z_1 \ldots z_n \; \varphi_o[y \leftarrow \text{l}] \rightarrow \varphi_o$$

where $\{x, z, z_1, \ldots, z_n\} = \text{vars}(\varphi_o) \setminus \{y\}$.

But this technique shall not be considered here in detail.

2.1.2 Evaluation Rules

Performing the synthesis of a skolem function presupposes having at one's disposal a set of lemmata about properties of functions and predicates occurring in the specification φ_o, for example, lemmata expressing communtativity, associativity, injectivity of a function or symmetry, reflexivity etc. of a predicate. The definition formulas of course play the most important role. The following two "evaluation"- or "rewrite"-rules serve to use such lemmata or definitions to simplify an (already modified) specification or make other rules applicable.

RWT ("rewrite term")

$$\frac{A \wedge B \rightarrow C[t_1]}{\begin{array}{l} B \rightarrow t_1 = t_1 \quad (*) \\ A \wedge B \rightarrow C[t_1 \leftarrow t_2] \end{array}}$$

RWF ("rewrite formula")

$$\frac{A \wedge B \rightarrow C[D]}{\begin{array}{l} B \rightarrow [D \leftrightarrow D'] \quad (*) \\ A \wedge B \rightarrow C[D \leftarrow D'] \end{array}}$$

The formulas marked with (*) are presumed to be among the axioms.

2.1.3 Using Lemmata

There are three further transformation rules which serve to use already existing lemmata or to generate them. The most important of them is the implication rule:

IMPL ("implication")

$$\frac{A \to C}{\begin{array}{l} B \to C \quad (*) \\ A \to B \end{array}} \quad .$$

This makes it possible to weaken a formula $A \to C$ by replacing C by a formula which implies C.

A transformation rule used to strengthen a given formula $A \wedge B \to C$ is the specialization rule:

SPEC ("specialization")

$$\frac{A \wedge B \to C}{\begin{array}{l} B \rightsquigarrow B' \\ A \wedge B' \to C \end{array}} \quad .$$

The extension rule is a very helpful tool if additional propositions, for example about terms occurring in a formula, are needed.

EXT ("extension")

$$\frac{A \wedge B \to C}{\begin{array}{ll} B \to D & (*) \\ A \wedge B \to C \wedge D & \end{array}} \quad .$$

2.1.4 Isolating Terms

The following rules are used to extract terms from formulas to obtain equalities which finally help to constitute the desired definition formulas.

The most important of these rules is the equality rule which carries out the very last step in almost every synthesis.

EQ ("equality")

$$\frac{B \wedge t = t_1 \rightarrow (-)\ t = t_2}{B \rightarrow (-)\ t_1 = t_2}$$

Very similar is the predicate rule which is also helpful in eleminating predicates:

PRED ("predicate")

$$\frac{B \wedge p(t_1 \ldots t_n) \rightarrow p(t_1' \ldots t_n')}{B \rightarrow t_1 = t_1' \wedge \ldots \wedge t_n = t_n'}$$

where p is a n-ary predicate.

The elimination of functions is done by the function rule:

FUN ("function")

$$\frac{A \rightarrow f(t_1 \ldots t_n) = f(t_1' \ldots t_n')}{A \rightarrow t_1 = t_1' \wedge \ldots \wedge t_n = t_n'} \quad .$$

The equal term rule serves to eliminate redundant terms:

EQT ("equal terms")

$$\frac{A \rightarrow [t = t_1 \leftrightarrow t = t_2]}{A \rightarrow t_1 = t_2}$$

Extraction of formulas analogous to the extraction of terms by the equality rule is done by the equivalence rule:

EQV ("equivalence")

$$\frac{B \wedge [A \leftrightarrow C] \rightarrow [A \leftrightarrow D]}{B \rightarrow [C \leftrightarrow D]} \quad .$$

2.1.5 Special Rules

Finally some rules will be described which belong to none of the above categories or are very special.

The case analysis rule:

CA ("case analysis")

$$\frac{A \to B}{\begin{array}{l} A \wedge C_1 \to B \\ \quad \vdots \\ A \wedge C_n \to B \end{array}}$$

provided $C_1 \vee \ldots \vee C_n$ is a theorem.

The strengthening rule serves to eliminate implications:

STR ("strengthening")

$$\frac{D \quad [A \to B \wedge C]}{D \quad [A \wedge B \wedge C]} \quad .$$

The premise elimination rule is used to eliminate redundant premises from a synthesized definition formula:

EL ("elimination")

$$\frac{A \wedge B \to C}{\begin{array}{l} A \to B \\ A \to C \end{array}} \quad .$$

In addition to the transformation rules described up to now equivalent logical transformations are allowed such as the elimination of <u>true</u> and <u>false</u> and substitutions like

(*)

$$\frac{t_1 = t_2 \wedge B \to C[t_1]}{t_1 = t_2 \wedge B \to C[t_2]}$$

and

$$(**) \qquad \frac{A \to t_1 = t_2 \wedge C[t_1]}{A \to t_1 = t_2 \wedge C[t_2]} \quad .$$

From this we can see that all transformation rules have the property R.

2.2 How to Apply the Transformation Rules

In this section we shall describe the sequence on which the transformation rules are applied in order to find a constructive definition of a skolem function f from a given specification φ_o.

Assume the definition formulas of all functions, predicates and structures occurring in φ_o as well as certain lemmata and information about the recursion of functions and predicates are given and available in a data base. These formulas together with the form of φ_o control the selection of the transformation rules to be applied according to the following strategy.

The rule to be applied first is the induction rule, yielding a set of formulas like

$$\{\gamma_1 \rightarrow \varphi_o, \ldots, \gamma_m \rightarrow \varphi_o\} \quad ,$$

where m is the number of definition cases.

The induction hypothesis $\forall z_1 \ldots z_1 \; A[x_i \leftarrow \hat{t}_i']$ will automatically be generated by the induction system.

Each of these formulas will be treated separately until a definition formula for f is found for this case.

The synthesis process terminates if a definition formula has been derived from each formula $\gamma_i \rightarrow \varphi_o$.

Now we shall try to evaluate symbolically the functions and predicates occurring in φ_o by applying the rules RWT and RWF to $\gamma_i \rightarrow \varphi_o$. The rules and definition formulas to be used are pattern directed selected ("pattern directed invocation" [MW79/2]) according to the function and predicate symbols occurring in φ_o.

This process continues until neither RWT nor RWF are applicable any more.

The examples of chapter 3 show that, if γ_i has been a formula characterizing a base case the synthesis has often already

finished at this point and a non-recursive definition of f, i.e. a definition formula

$$\gamma_i \rightarrow f(x_1 \ldots x_n) = t$$

is obtained.

If no definition formula has been derived at this point and some functions or predicates have remained unevaluated the application of the case analysis rule CA goal directed is attempted to enable applicability of RWT or RWF resp.

The way to do this is first to look up all definition formulas whose symbols occur in φ (where $\gamma_i \rightarrow \varphi$ is the formula derived from $\gamma_i \rightarrow \varphi_o$) and which have not yet been evaluated. Their premises are collected and the formulas ψ are determined which in conjunction with γ_i make the application of RWT or RWF possible. From the ψ the one which contains the highest number of structural propositions is chosen, i.e. propositions about the structure of the term $f(x_1 \ldots x_n)$ or a variable, in terms of constructor functions and/or base constants (e.g. $f(x_1 \ldots x_n) = s(v)$ or $y = cons(uv)$ etc.). Taking the selected ψ and the remaining cases $\psi_1, \ldots, \psi_k$ of the corresponding definition, the disjunction $\psi \vee \psi_1 \vee \ldots \vee \psi_k$ will be a theorem because the formulas $\psi, \psi_1, \ldots, \psi_k$ represent a complete case analysis.

The formulas resulting from application of CA are:

$$\begin{aligned} &\gamma_i \wedge \psi \rightarrow \varphi \\ &\gamma_i \wedge \psi_1 \rightarrow \varphi \\ &\quad \vdots \\ &\gamma_i \wedge \psi_k \rightarrow \varphi \quad . \end{aligned}$$

This transformation will be carried out in cases where not only the formula $\gamma_i \wedge \psi \rightarrow \varphi$ but also any of the others $\gamma_i \wedge \psi_j \rightarrow \varphi$ can be transformed by evaluation of the corresponding function or predicate.

If only the first formula $\gamma_i \wedge \psi \rightarrow \varphi$ can be transformed by RWT resp. RWF the cases $\psi_1,\ldots,\psi_k$ are ignored and the CA-rule simply yields:

$$\gamma_i \wedge \psi \rightarrow \varphi$$
$$\gamma_i \wedge \neg\psi \rightarrow \varphi \quad .$$

Evaluation steps which are possible only because a CA-rule was used previously with a formula ψ including predicates or functions which occur neither in γ_i nor in φ have a lower priority.

The next transformation after such a CA-step will always be carried out on that formula which makes the desired evaluation possible (eventually after another application of a CA-rule).

If γ_i is not a characterization of a base case and φ_o has been simplified by applying the evaluation rules RWT and RWF, possibly with the aid of CA, the next step is to consider the induction hypothesis of the original specification formula φ_o which is part of γ_i.

The main goal is now to transform the formula $\gamma_k \wedge I \wedge \psi \rightarrow \varphi$ (which has been derived from $\gamma_k \wedge I \rightarrow \varphi_o$) to obtain the applicability of one of the term isolating rules (section 2.1.4) in order to find a recursive definition of f.

This will be done by trying to match the induction hypothesis I with φ such that one of the rules PRED or FUN can be applied. A failure of this match causes goal-oriented transformations. They may consist in applying rules RWT, RWF, IMPL, EQ etc. together with (*) and (**) using lemmata (such as commutativity or associativity of functions), which will be selected pattern directed. Additionally the induction hypothesis I may be modified to I' by the SPEC-rule rendering a successful match of the resulting φ' and I'.

A good heuristic is to try matching the formulas after each step and then to decide which rule to apply next depending on the information obtained from the failed match. Transformations with

rules CA should be avoided if possible.

The process terminates if each of the formulas $\gamma_i \rightarrow \varphi_o$ (which are the results of the first transformation IND) has been transformed into a definition formula of form

$$\psi \rightarrow f(x_1 \ldots x_n) = t \quad .$$

The derived definition of the skolem function f must be checked for uniqueness, whereas completeness and termination are already guaranteed by the rules IND and CA. A correctness proof of the derived definition (i.e. the conjunction of all derived definition formulas implies the specification φ_o) is also obsolete because all rules defined in section 2.1 have the property R.

Finally all the formulas which have been generated in the synthesis process but have not lead to a definition formula have to be proved using the synthesized definition of f.

3. Examples

In this chapter some examples are presented to give a precise idea of how a synthesis system as described in chapter 2 works and in particular how the transformation rules are selected.

The first examples are very simple, whereas the last demonstrate some problems which occur in treating more complex specifications like conjunctions or equivalences.

3.1 Synthesis of function *plus*

The following example demonstrates the straight forward derivation of a function *plus* from its specification.

Let us suppose the definitions given to the system are those of the structure *nat* of natural numbers and two functions *pred* (*predecessor*) and *diff* (*difference*).

```
STRUCTURE   0 s(nat): nat
FUNCTION    pred(x:nat): nat =
      IF    x = 0  THEN  0
      IF    ∃u : nat  x = s(u)  THEN  u

FUNCTION    diff(x,y:nat): nat =
      IF    y = 0  THEN  x
      IF    ∃v : nat  y = s(v)  THEN  pred(diff(xv)) .
```

For the following assume that all formulas are prefixed by a universal quantification of all variables occurring in it.

The definition formulas generated by the system are then:

AX1 $x = 0 \vee \exists y : nat \;\; x = s(y)$

AX2 $s(x) \neq 0$

AX3 $s(x) = s(y) \leftrightarrow x = y$

AX4 $x = 0 \rightarrow pred(x) = 0$

AX5 $x = s(u) \rightarrow pred(x) = u$

AX6 $y = 0 \rightarrow diff(xy) = x$

AX7 $y = s(v) \rightarrow diff(xy) = pred(diff(xv))$

The formula to be proved is:

$$\forall x,y : nat \; \exists z : nat \; diff(zy) = x \quad .$$

Skolemization yields the specification:

$$diff(f(xy)y) = x \quad .$$

The rule first to be applied is the induction rule. The recursive function *diff* satisfies all premises for its application.

$$\text{(IND)}\frac{diff(f(xy)y) = x}{}$$

1. $y = 0 \quad diff(f(xy)y) = x$
2. $y = s(v) \wedge \forall z : nat \; diff(f(zv)v) = z \rightarrow diff(f(xy)y) = x$

Now formula 1. will be transformed by symbolical evaluation of function *diff*:

1. $y = 0 \rightarrow diff(f(xy)y) = x$

$$\text{(RWT)}\frac{}{}$$

1.1 $y = 0 \rightarrow f(xy) = x$

1.2 $y = 0 \rightarrow diff(f(xy)y) = f(xy)$
(instance of AX6)

With formula 1.1 the first definition formula has been derived and the process continues with transformation of formula 2. The first step again will be the symbolical evaluation of function *diff*:

2. $y = s(v) \wedge \forall z : nat\ diff(f(zv)v) = z \rightarrow diff(f(xy)y) = x$

(RWT)

2.1 $y = s(v) \wedge \forall z : nat\ diff(f(zv)v) = z \rightarrow pred(diff(f(xy)v)) = x$

2.2 $y = s(v) \rightarrow diff(f(xy)y) = pred(diff(f(xy)v))$ (instance of AX7)

Now no further evaluation step is possible and the attempt to match formula $pred(diff(f(xy)v) = x$ with the induction hypothesis I: $\forall z : nat\ diff(f(zv)v) = z$ fails because of the occurrence of function *pred*. Therefore its elimination is attempted with the aid of its definition and the implication rule:

2.1 $y = s(v) \wedge \forall z : nat\ diff(f(zv)v) = z \rightarrow pred(diff(f(xy)v) = x$

(IMPL)

2.1.1 $y = s(v) \wedge \forall z : nat\ diff(f(zv)v) = z \rightarrow diff(f(xy)v) = s(x)$

2.1.2 $diff(f(xy)v) = s(x) \rightarrow pred(diff(f(xy)v)) = x$ (instance of AX5)

Obviously the match between formula $diff(f(xy)v) = s(x)$ and the induction hypothesis I will succeed, in that rule FUN becomes applicable, if we take the instance of I with z replaced by the term $s(x)$ and then apply the equality rule:

2.1.1 $y = s(v) \wedge \forall z : nat\ diff(f(zv)v) = z \rightarrow diff(f(xy)v) = s(x)$

(SPEC)

2.1.1.1 $y = s(v) \wedge diff(f(s(x)v)v) = s(x) \rightarrow diff(f(xy)v) = s(x)$

2.1.1.2 $\forall z : nat\ diff(f(zv)v) = z \rightarrow diff(f(s(x)v)v) = s(x)$

$$2.1.1.1 \quad y = s(v) \wedge diff(f(s(x)v)v) = s(x) \rightarrow diff(f(xy)v) = s(x)$$

(EQ)

$$y = s(v) \rightarrow diff(f(s(x)v)v) = diff(f(xy)v)$$

Now application of the function rule leads to the second definition formula:

$$y = s(v) \rightarrow diff(f(s(x)v)v) = diff(f(xy)v)$$

(FUN)

$$y = s(v) \rightarrow f(xy) = f(s(x)v)$$

(The formula v = v has been simplified to <u>true</u> and then has been eliminated.)

At this point the synthesis process terminates. The derived definition formulas for the skolem function f (i.e. *plus*) are:

$y = 0 \rightarrow f(xy) = x$

$y = s(v) \rightarrow f(xy) = f(s(x)y)$, i.e.

the definition of f is:

```
FUNCTION  f(x,y:nat): nat =
      IF  y = 0  THEN  x
      IF  y = s(v)  THEN  f(s(x)v)
```

Now the definition has to be checked for uniqueness and finally the formula 2.1.1.2: $\forall xv : nat[\forall z:nat\ diff(f(zv)v) = z \rightarrow diff(f(s(x)v)v) = s(x)]$ has to be proved.

3.2 Synthesis of function *rest*

This example shows how some lemmata have to be used skillfully to derive the definition formulas of function *rest*.

Let us assume the following definitions of natural numbers and functions and predicates defined on them are given:

```
STRUCTURE  0  s(nat): nat

FUNCTION   plus(x,y:nat): nat =
      IF   x = 0  THEN  y
      IF   ∃u : nat x = s(u)  THEN  s(plus(uy))

FUNCTION   times(x,y:nat): nat =
      IF   x = 0  THEN  0
      IF   ∃u : nat x = s(u)  THEN  plus(times(uy)y)

FUNCTION   sub(x,y:nat): nat =
      IF   y = 0  THEN  x
      IF   x = 0  THEN  0
      IF   ∃u,v : nat  x = s(u) ∧ y = s(v)  THEN  sub(uv)

PREDICATE  lt(x,y:nat) =
       IF  y = 0  THEN  false
       IF  x = 0 ∧ ∃v : nat  y = s(v)  THEN  true
       IF  ∃u,v : nat  x = s(u) ∧ y = s(v)  THEN  lt(uv)

PREDICATE  ge(x,y:nat) = not lt(xy)

FUNCTION   quot(x,y:nat): nat =
      IF   y = 0  THEN  0
      IF   lt(xy)  THEN  0
      IF   ge(xy) ∧ y ≠ 0  THEN  s(quot(sub(xy)y))
```

The definition formulas generated by the system are:

AX1 $x = 0 \vee \exists y : nat \; x = s(y)$

AX2 $s(x) \neq 0$

AX3 $s(x) = s(y) \leftrightarrow x = y$

AX4 $x = 0 \rightarrow plus(xy) = y$

AX5 $x = s(u) \rightarrow plus(xy) = s(plus(uy))$

AX6 $x = 0 \rightarrow times(xy) = 0$

AX7 $x = s(u) \rightarrow times(xy) = plus(times(uy)y)$

AX8 $y = 0 \rightarrow sub(xy) = x$

AX9 $x = 0 \rightarrow sub(xy) = 0$

AX10 $x = s(u) \wedge y = s(v) \rightarrow sub(xy) = sub(uv)$

AX11 $y = 0 \rightarrow [lt(xy) \leftrightarrow \underline{false}]$

AX12 $x = 0 \wedge y = s(v) \rightarrow [lt(xy) \leftrightarrow \underline{true}]$

AX13 $x = s(u) \wedge y = s(v) \rightarrow [lt(xy) \leftrightarrow lt(uv)]$

AX14 $ge(xy) \leftrightarrow \underline{not}\ lt(xy)$

AX15 $y = 0 \rightarrow quot(xy) = 0$

AX16 $lt(xy) \rightarrow quot(xy) = 0$

AX17 $ge(xy) \wedge y \neq 0 \rightarrow quot(xy) = s(quot(sub(xy)y))$

Additionally the following lemmata will be given:

L1 $plus(xy) = plus(yx)$

L2 $plus(plus(xy)z) = plus(x\ plus(yz))$

L3 $ge(uv) \rightarrow [plus(wv) = u \leftrightarrow w = sub(uv)]$

The formula to be proved is:

$\forall x,y : nat \; \exists z : nat \; plus(times(quot(xy)y)z) = x$

Skolemization yields the specification:

$$plus(times(quot(xy)y)\ f(xy)) = x \quad .$$

The function *quot* will be taken to apply the induction rule because its parameter x is a recursion argument and all premises for the application of IND are fulfilled. The result is three formulas:

```
      plus(times(quot(xy)y) f(xy)) = x
(IND)-------------------------------------------------------------

      1.  y = 0 → plus(times(quot(xy)y) f(xy)) = x

      2.  lt(xy) → plus(times(quot(xy)y) f(xy)) = x

      3.  ge(xy) ∧ y ≠ 0 ∧ plus(times(quot(sub(xy)y)y)
                                  f(sub(xy)y)) = sub(xy)
            → plus(times(quot(xy)y) f(xy)) = x
```

Formula 1. will be transformed by symbolical evaluation of functions *quot*, *times* and *plus*:

```
      1.  y = 0   plus(times(quot(xy)y) f(xy)) = x
(RWT)-------------------------------------------------------------

      1.1 y = 0 → plus(times(0y) f(xy)) = x

      1.2 y = 0 → quot(xy) = 0
                              (instance of AX15)

      1.1 y = 0   plus(times(0y) f(xy)) = x
(RWT)-------------------------------------------------------------

      1.1.1 y = 0 → plus(0 f(xy)) = x

      1.1.2 0 = 0 → times(0y) = 0
                                  (instance of AX6)

      1.1.1 y = 0 → plus(0 f(xy)) = x
(RWT)-------------------------------------------------------------

      1.1.1.1 y = 0 → f(xy) = x

      1.1.1.2 0 = 0 → plus(0 f(xy)) = f(xy)
                                  (instance of AX4)
```

With formula 1.1.1.1 the first definition formula has been derived.

Formula 2 will now be treated very similarly:

2. lt(xy) → plus(times(quot(xy)y) f(xy)) = x

(RWT)

2.1 lt(xy) → plus(times(Oy) f(xy)) = x

2.2 lt(xy) → quot(xy) = O

(instance of AX16)

2.1 lt(xy) → plus(times(Oy) f(xy)) = x

(RWT)

2.1.1 lt(xy) → plus(O f(xy)) = x

2.1.2 O = O → times(Oy) = O

(instance of AX6)

2.1.1 lt(xy) → plus(O f(xy)) = x

(RWT)

2.1.1.1 lt(xy) → f(xy) = x

2.1.1.2 O = O → plus(O f(xy)) = f(xy)

(instance of AX4)

With formula 2.1.1.1 the second definition formula has been derived.

Now formula 3 will be transformed and the first steps to be taken will be symbolical evaluation of functions *quot* and *times*. But first the following abbreviations shall be made.

$\underline{r}$:= quot(sub(xy)y)

I := plus(times($\underline{r}$y) f(sub(xy)y)) = sub(xy) .

3. $ge(xy) \wedge y \neq 0 \wedge I \rightarrow plus(times(quot(xy)y)\ f(xy)) = x$

(RWT)

3.1 $ge(xy) \wedge y \neq 0 \wedge I \rightarrow$

$plus(times(s(quot(sub(xy)y))y)\ f(xy)) = x$

3.2 $ge(xy) \wedge y \neq 0 \rightarrow quot(xy) = s(quot(sub(xy)y))$

(instance of AX17)

3.1 $ge(xy) \wedge y \neq 0 \wedge I \rightarrow plus(times(s(\underline{r})y)\ f(xy)) = x$

(RWT)

3.1.1 $ge(xy) \wedge y \neq 0 \wedge$

$plus(times(\underline{r}y)\ f(sub(xy)y)) = sub(xy)$

$\rightarrow\ plus(plus(times(\underline{r}y)y)\ f(xy)) = x$

3.1.2 $s(\underline{r}) = s(\underline{r}) \rightarrow times(s(\underline{r})y) = plus(times(\underline{r}y)y)$

(instance of AX7)

Now no further evaluation would be helpful and therefore we try to match I with the formula

$F := plus(plus(times(\underline{r}y)y)\ f(xy)) = x$.

The failure consists in the second *plus* of formula F and in the function *sub* occurring on the right side of the equality in I.

A look at our data base shows us that the lemma L3 makes it possible under certain conditions to exchange an equality involving a *plus*-term for one involving a *sub*-term. The only condition we still have to fulfill is to transform F such that y becomes the second argument of the first *plus* instead of that of the second one.

To reach this goal lemmata L1 and L2 will be helpful and the way we choose is the following:

Use commutativity of function *plus*:

3.1.1 ge(xy) ∧ y ≠ 0 ∧ I → plus(plus(times(ry)y) f(xy)) = x

(RWT)

3.1.1.1 ge(xy) ∧ y ≠ 0 ∧ I →
plus(plus(y times(ry)) f(xy)) = x

3.1.1.2 plus(times(ry)y) = plus(y times(ry))
(instance of L1)

Use associativity of function *plus*:

3.1.1.1 ge(xy) ∧ y ≠ 0 ∧ I →
plus(plus(y times(ry)) f(xy)) = x

(RWT)

3.1.1.1.1 ge(xy) ∧ y ≠ 0 ∧ I →
plus(y plus(times(ry) f(xy))) = x

3.1.1.1.2 plus(plus(y times(ry)) f(xy)) =
plus(y plus(times(ry) f(xy)))
(instance of L2)

Again use commutativity of function *plus*:

3.1.1.1.1 ge(xy) ∧ y ≠ 0 ∧ I →
plus(y plus(times(ry) f(xy))) = x

(RWT)

3.1.1.1.1.1 ge(xy) ∧ y ≠ 0 ∧ I →
plus(plus(times(ry) f(xy))y) = x

3.1.1.1.1.2 plus(y plus(times(ry) f(xy))) =
plus(plus(times(ry) f(xy))y)
(instance of L1)

Now with rule RWF and L3 formula 3.1.1.1.1.1 can be transformed such that a match between it and I succeeds in that after an EQ step rule FUN becomes applicable and yields a recursive definition of f:

3.1.1.1.1.1 ge(xy) ∧ y ≠ 0 ∧ I →
plus(plus(times(ry) f(xy))y) = x

RWF

3.1.1.1.1.1.1 ge(xy) ∧ y ≠ 0 ∧ I →
plus(times(ry) f(xy)) = sub(xy)

3.1.1.1.1.1.2 ge(xy) →
[plus(plus(times(ry) f(xy))y) = x ↔
plus(times(ry) f(xy)) = sub(xy)]
(instance of L3)

3.1.1.1.1.1.1
ge(xy) ∧ y ≠ 0 ∧ plus(times(ry) f(sub(xy)y)) = sub(xy)
→ plus(times(ry) f(xy)) = sub(xy)

(EQ)

ge(xy) ∧ y ≠ 0 → plus(times(ry) f(sub(xy)y)) =
plus(times(ry) f(xy))

ge(xy) ∧ y ≠ 0 → plus(times(ry) f(sub(xy)y)) =
plus(times(ry) f(xy))

(FUN)

ge(xy) ∧ y ≠ 0 → f(xy) = f(sub(xy)y)

Now the third definition formula of f has been derived and the synthesis process terminates.

The definition formulas of the skolem function f (i.e. *rest*) are:

$y = 0 \rightarrow f(xy) = x$

$lt(xy) \rightarrow f(xy) = x$

$ge(xy) \wedge y \neq 0 \rightarrow f(xy) = f(sub(xy)y)$,

i.e. the definition of f is:

```
FUNCTION   f(x,y:nat): nat =
      IF   y = 0  THEN  x
      IF   lt(xy)  THEN  x
      IF   ge(xy) ∧ y ≠ 0  THEN  f(sub(xy)y)
```

It still has to be checked for uniqueness but then it will be finished.

3.3 Synthesis of function *half*

This example has been chosen to demonstrate how a function definition can be derived from a more complex specification. It will be an equivalence with a predicate other than equality being involved. Suppose the definitions given to the system are:

STRUCTURE O s(nat): nat

FUNCTION double(x:nat): nat =

IF x = O THEN O

IF ∃u : nat x = s(u) THEN s(s(double(u)))

PREDICATE even(x:nat) =

IF x = O THEN true

IF x = s(O) THEN false

IF ∃u : nat x = s(s(u)) THEN even(u)

The definition formulas generated by the system are:

AX1 x = O ∨ ∃y : nat x = s(y)

AX2 s(x) ≠ O

AX3 s(x) = s(y) ↔ x = y

AX4 x = O → double(x) = O

AX5 x = s(u) → double(x) = s(s(double(u)))

AX6 x = O → [even(x) ↔ true]

AX7 x = s(O) → [even(x) ↔ false]

AX8 x = s(s(u)) → [even(x) ↔ even(u)]

The formula to be proved is:

∀x : nat ∃y : nat even(x) ↔ double(y) = x

Skolemization yields the specification:

even(x) ↔ double(f(x)) = x .

The induction rule will be applied and the recursive predicate *even* fulfills all premises which are required. We therefore get three definition cases:

$$\frac{even(x) \leftrightarrow double(f(x)) = x}{(IND)}$$

1. $x = 0 \rightarrow [even(x) \leftrightarrow double(f(x)) = x]$
2. $x = s(0) \rightarrow [even(x) \leftrightarrow double(f(x)) = x]$
3. $x = s(s(u)) \wedge [even(u) \leftrightarrow double(f(u)) = u] \rightarrow [even(x) \leftrightarrow double(f(x)) = x]$

The first formula will be transformed by symbolical evaluation of predicate *even*:

$$\frac{1.\ x = 0 \rightarrow [even(x) \leftrightarrow double(f(x)) = x]}{(RWF)}$$

1.1 $x = 0 \rightarrow [\underline{true} \leftrightarrow double(f(x)) = x]$

1.2 $x = 0 \rightarrow [even(x) \leftrightarrow \underline{true}]$

(instance of AX6)

Now no further evaluation rule can be applied because we have no information about the structure of f(x) to evaluate the term double(f(x)).

Therefore the case analysis rule will be goal directed applied to reach applicability of RWT:

$$\frac{1.1\ x = 0 \rightarrow double(f(x)) = x}{(CA)}$$

1.1.1 $x = 0 \wedge f(x) = 0 \rightarrow double(f(x)) = x$

1.1.2 $x = 0 \wedge f(x) = s(u) \rightarrow double(f(x)) = x$

$$\frac{1.1.1\ x = 0 \wedge f(x) = 0 \rightarrow double(f(x)) = x}{(RWT)}$$

1.1.1.1 $x = 0 \wedge f(x) = 0 \rightarrow 0 = x$

1.1.1.2 $f(x) = 0 \rightarrow double(f(x)) = 0$

(instance of AX4)

Applying rule EQ to formula 1.1.1.1 yields the first definition formula:

$$\frac{1.1.1.1 \quad x = 0 \wedge f(x) = 0 \rightarrow 0 = x}{(EQ)} \qquad x = 0 \rightarrow f(x) = x$$

Formula 2 will be transformed very similarly by first evaluating the predicate *even* and then goal directed applying the CA-rule to reach applicability of RWT to evaluate function *double*:

$$\frac{2. \quad x = s(0) \rightarrow [even(x) \leftrightarrow double(f(x)) = x]}{(RWF)}$$

$$2.1 \; x = s(0) \rightarrow [\underline{false} \leftrightarrow double(f(x)) = x]$$

$$2.2 \; x = s(0) \rightarrow [even(x) \leftrightarrow \underline{false}]$$

(instance of AX7)

$$\frac{2.1 \; x = s(0) \rightarrow - \; double(f(x)) = x}{(CA)}$$

$$2.1.1 \; x = s(0) \wedge f(x) = 0 \rightarrow - \; double(f(x)) = x$$

$$2.1.2 \; x = s(0) \wedge f(x) = s(u) \rightarrow - \; double(f(x)) = x$$

$$\frac{2.1.1 \; x = s(0) \wedge f(x) = 0 \rightarrow - \; double(f(x)) = x}{(RWT)}$$

$$2.1.1.1 \; x = s(0) \wedge f(x) = 0 \rightarrow - \; 0 = x$$

$$2.1.1.2 \; f(x) = 0 \rightarrow double(f(x)) = 0$$

(instance of AX4)

Applying rule EQ to formula 2.1.1.1 yields a very "weak" specification for our skolem function:

$$\frac{2.1.1.1 \; x = s(0) \wedge f(x) = 0 \rightarrow - \; 0 = x}{(EQ)}$$

$$2.1.1.1.1 \; x = s(0) \rightarrow - \; f(x) = x$$

Now our system has to choose a value for f(x) which is sufficient to make formula 2.1.1.1.1 true.

This is very simple here. Because x is presumed to be a constructor term (s(0)) we simply choose the base constant of structure *nat* as value of f(x) and reach the second definition formula:

$$\frac{2.1.1.1.1 \quad x = s(0) \rightarrow - f(x) = x}{x = s(0) \rightarrow f(x) = 0} \text{ (choose value)}$$

Finally formula 3 will be transformed and the first rule to be applied to it will be RWF. I stands for the induction hypothesis, i.e. the formula $even(u) \leftrightarrow double(f(u)) = u$.

$$\frac{3. \quad x = s(s(u)) \wedge I \rightarrow [even(x) \leftrightarrow double(f(x)) = x]}{\begin{array}{l} 3.1 \; x = s(s(u)) \wedge I \rightarrow [even(u) \leftrightarrow double(f(x)) = x] \\ 3.2 \; x = s(s(u)) \rightarrow [even(x) \leftrightarrow even(u)] \end{array}} \text{ (RWF)}$$

(instance of AX8)

The function *double* can be evaluated symbolically after application of the CA-rule:

$$\frac{3.1 \; x = s(s(u)) \wedge I \rightarrow [even(u) \leftrightarrow double(f(x)) = x]}{\begin{array}{l} 3.1.1 \; x = s(s(u)) \wedge f(x) = 0 \wedge I \rightarrow [even(u) \leftrightarrow double(f(x)) = x] \\ 3.1.2 \; x = s(s(u)) \wedge f(x) = s(v) \wedge I \rightarrow \\ \qquad [even(u) \leftrightarrow double(f(x)) = x] \end{array}} \text{ (CA)}$$

3.1.2 $x = s(s(u)) \wedge f(x) = s(v) \wedge I \rightarrow$

$[even(u) \leftrightarrow double(f(x)) = x]$

(RWT)

3.1.2.1 $x = s(s(u)) \wedge f(x) = s(v) \wedge I \rightarrow$

$[even(u) \leftrightarrow s(s(double(v))) = x]$

3.1.2.2 $f(x) = s(v) \rightarrow double(f(x)) = s(s(double(v)))$

(instance of AX5)

With the equivalence rule EQV the formula *even(u)* can be eliminated from formula 3.1.2.1:

3.1.2.1 $x = s(s(u)) \wedge f(x) = s(v) \wedge$

$[even(u) \leftrightarrow double(f(u)) = u]$

$\rightarrow [even(u) \leftrightarrow s(s(double(v))) = x]$

(EQV)

$x = s(s(u)) \wedge f(x) = s(v) \rightarrow$

$[double(f(u)) = u \leftrightarrow s(s(double(v))) = x]$

The main goal now is to apply transformations which change the formula F := $s(s(double(v))) = x$ in such a way that rule EQT can be applied to eliminate the equivalence.

Replacing the x in formula F by the term $s(s(u))$, which can easily be done by the (*)-rule, makes it possible to reduce F by rule RWF and the injectivity of the constructor function such that EQT can be applied.

$x = s(s(u)) \wedge f(x) = s(v) \rightarrow$

$[double(f(u)) = u \leftrightarrow s(s(double(v))) = x]$

(*)

$x = s(s(u)) \wedge f(x) = s(v) \rightarrow$

$[double(f(u)) = u \leftrightarrow s(s(double(v))) = s(s(u))]$

$$x = s(s(u)) \wedge f(x) = s(v) \rightarrow$$
$$[double(f(u)) = u \leftrightarrow s(s(double(v))) = s(s(u))]$$

(RWF)

$$x = s(s(u)) \wedge f(x) = s(v) \rightarrow$$
$$[double(f(u)) = u \leftrightarrow s(double(v)) = s(u)]$$

$$s(s(double(v))) = s(s(u)) \leftrightarrow s(double(v)) = s(u)$$

(instance of AX3)

$$x = s(s(u)) \wedge f(x) = s(v) \rightarrow$$
$$[double(f(u)) = u \leftrightarrow s(double(v)) = s(u)]$$

(RWF)

$$x = s(s(u)) \wedge f(x) = s(v) \rightarrow$$
$$[double(f(u)) = u \leftrightarrow double(v) = u]$$

$$s(double(v)) = s(u) \leftrightarrow double(v) = u$$

(instance of AX3)

$$x = s(s(u)) \wedge f(x) = s(v) \rightarrow$$
$$[double(f(u)) = u \leftrightarrow double(v) = u]$$

(EQT)

$$x = s(s(u)) \wedge f(x) = s(v) \rightarrow double(f(u)) = double(v)$$

Now the function rule FUN can be applied:

$$x = s(s(u)) \wedge f(x) = s(v) \rightarrow double(f(u)) = double(v)$$

(FUN)

$$x = s(s(u)) \wedge f(x) = s(v) \rightarrow f(u) = v$$

We have now reached a kind of formula which very often occurs towards the end of a synthesis process.

We know that if f(x) is a constructor term then the argument of the constructor function is equal to a certain term. In such a situation the definition formula of f is obtained by

first applying the RWF rule with the injectivity of the constructor function and then applying the equality rule.

$$\text{(RWF)}\ \frac{x = s(s(u)) \wedge f(x) = s(v) \rightarrow f(u) = v}{\begin{array}{l} x = s(s(u)) \wedge f(x) = s(v) \rightarrow s(f(u)) = s(v) \\ f(u) = v \leftrightarrow s(f(u)) = s(v) \qquad \text{(instance of AX3)} \end{array}}$$

$$\text{(EQ)}\ \frac{x = s(s(u)) \wedge f(x) = s(v) \rightarrow s(f(u)) = s(v)}{x = s(s(u)) \rightarrow f(x) = s(f(u))}$$

Now the synthesis process terminates yielding the following definition formulas of function f (i.e. *half*):

$x = 0 \rightarrow f(x) = x$

$x = s(0) \rightarrow f(x) = 0$

$x = s(s(u)) \rightarrow f(x) = s(f(u))$,

i.e. the definition of f is:

FUNCTION f(x:nat): nat =

IF x = 0 THEN x

IF x = s(0) THEN 0

IF $\exists u$: nat x = s(s(u)) THEN s(f(u)) .

Finally this definition has to be checked for uniqueness and the following formulas have to be proved:

1.1.2 $\forall x$: nat[x = 0 $\wedge$ f(x) = s(u) $\rightarrow$ double(f(x)) = x]

2.1.2 $\forall x$: nat[x = s(0) $\wedge$ f(x) = s(u) $\rightarrow$ - double(f(x)) = x]

3.1.1 $\forall x,u$: nat[x = s(s(u)) $\wedge$ f(x) = 0 $\wedge$
[even(u) $\leftrightarrow$ double(f(u)) = u] $\rightarrow$
[even(u) $\leftrightarrow$ double(f(x)) = x]]

This is easily done using the definition formulas of function f together with the axioms AX2, AX4 and AX5.

3.4 Synthesis of function *sort*

The last example shows the synthesis of a sort-function. The specification is a conjunction of atomar formulas without any equality predicate. We will see that a terminating definition of the skolem function can only be obtained if the modified specification is extended by a lemma. The following shows how this will be managed.

First assume the definitions of the set of lists of natural numbers, the predicates *perm* (*permutation*), *ordered* and *member* and the function *min* (*minimum*) are given:

```
STRUCTURE  0  s(nat): nat

STRUCTURE  empty cons(nat list): list

PREDICATE  le(x,y:nat) =

       IF  x = 0  THEN  true

       IF  x ≠ 0 ∧ y = 0  THEN  false

       IF  ∃u,v : nat x = s(u) ∧ y = s(v)  THEN  le(uv)

PREDICATE  gt(x,y:nat) = not le(xy)

FUNCTION   min(x:list): nat =

       IF  x = empty  THEN  0

       IF  ∃n:nat x = cons(n empty)  THEN  n

       IF  ∃n,m : nat ∃y : list x = cons(n cons(my))
              ∧ le(nm)  THEN  min(cons(ny))

       IF  ∃n,m : nat ∃y : list x = cons(n cons(my))
              ∧ gt(nm)  THEN  min(cons(my))
```

```
FUNCTION   delete(n:nat x:list): list =
       IF  x = empty  THEN  x
       IF  ∃y : list x = cons(ny)  THEN  y
       IF  ∃m : nat ∃y : list x = cons(my)
            ∧ n ≠ m  THEN  cons(m delete(ny))

PREDICATE  ordered(x:list) =
       IF  x = empty  THEN  true
       IF  ∃n : nat ∃y : list x = cons(ny)
            ∧ n ≠ min(x)  THEN  false
       IF  ∃n : nat ∃y : list x = cons(ny)
            ∧ n = min(x)  THEN  ordered(y)

PREDICATE  perm(x,y:nat) =
       IF  x = empty  THEN  y = empty
       IF  y = empty  THEN  x = empty
       IF  ∃n : nat ∃z : list x = cons(nz)
            ∧ y ≠ empty  THEN  perm(z delete(ny))

PREDICATE  member(n:nat x:list) =
       IF  x = empty  THEN  false
       IF  ∃m : nat ∃y : list x = cons(my)
            ∧ n = m  THEN  true
       IF  ∃m : nat ∃y : list x = cons(my)
            ∧ n ≠ m  THEN  member(ny)
```

The definition formulas generated by the system are:

```
AX1   ∀x : nat [x = 0 ∨ ∃y : nat x = s(y)]

AX2   ∀x : nat s(x) ≠ 0

AX3   ∀x,y : nat [s(x) = s(y) ↔ x = y]
```

AX4 $\forall x : list\ [x = empty \vee \exists n : nat\ \exists y : list\ x = cons(ny)]$

AX5 $\forall x : list\ \forall n : nat\ cons(nx) \neq \underline{empty}$

AX6 $\forall x,y : list\ \forall n,m : nat\ [cons(nx) = cons(my) \leftrightarrow n = m \wedge x = y]$

The following axioms apply to the defined functions and predicates and are given without an outermost universal quantification. The sorts of the occurring variables become clear from the context.

AX7 $x = 0 \rightarrow [le(xy) \leftrightarrow \underline{true}]$

AX8 $x \neq 0 \wedge y = 0 \rightarrow [le(xy) \leftrightarrow \underline{false}]$

AX9 $x = s(u) \wedge y = s(v) \rightarrow [le(xy) \leftrightarrow le(uv)]$

AX10 $gt(xy) \leftrightarrow \underline{not}\ le(xy)$

AX11 $x = \underline{empty} \rightarrow min(x) = 0$

AX12 $x = cons(n\ \underline{empty}) \rightarrow min(x) = n$

AX13 $x = cons(n\ cons(my)) \wedge le(nm) \rightarrow min(x) = min(cons(ny))$

AX14 $x = cons(n\ cons(my)) \wedge gt(nm) \rightarrow min(x) = min(cons(my))$

AX15 $x = \underline{empty} \rightarrow delete(nx) = x$

AX16 $x = cons(ny) \rightarrow delete(nx) = y$

AX17 $x = cons(my) \wedge n \neq m \rightarrow delete(nx) = cons(m\ delete(ny))$

AX18 $x = \underline{empty} \rightarrow [ordered(x) \leftrightarrow \underline{true}]$

AX19 $x = cons(ny) \wedge n \neq min(x) \rightarrow [ordered(x) \leftrightarrow \underline{false}]$

AX20 $x = cons(ny) \wedge n = min(x) \rightarrow [ordered(x) \leftrightarrow ordered(y)]$

AX21 $x = \underline{empty} \rightarrow [perm(xy) \leftrightarrow y = \underline{empty}]$

AX22 $y = \underline{empty} \rightarrow [perm(xy) \leftrightarrow x = \underline{empty}]$

AX23 $x = cons(nz) \wedge y \neq \underline{empty} \rightarrow [perm(xy) \leftrightarrow perm(z\ delete(ny))]$

AX24 $x = \underline{empty} \rightarrow [member(nx) \leftrightarrow \underline{false}]$

AX25 $x = cons(my) \wedge n = m \rightarrow [member(nx) \leftrightarrow \underline{true}]$

AX26 $x = cons(my) \wedge n \neq m \rightarrow [member(nx) \leftrightarrow member(ny)]$

Additionally the following lemma will be available:

L1 $\forall x,y : list\ \forall n : nat\ member(nx) \wedge perm(y\ delete(nx)) \rightarrow min(x) = min(cons(ny))$.

The formula to be proved is:

$$\forall x : list\ \exists y : list\ perm(yx) \wedge ordered(y) \quad .$$

Skolemization yields the specification:

$$perm(f(x)x) \wedge ordered(f(x)) \quad .$$

The induction rule is applied using the recursive predicate *perm*. The induction hypothesis generated for this specification has a more complex structure because taking only its second argument into consideration the predicate perm doesn't terminate except under a certain condition. However the problem of generating a correct induction hypothesis for a formula will not be discussed here.

The first transformation step yields two formulas:

$$\frac{perm(f(x)x) \wedge ordered(f(x))}{(IND)}$$

1. $x = \underline{empty} \rightarrow perm(f(x)x) \wedge ordered(f(x))$

2. $- x = \underline{empty} \wedge \forall m : nat\ [member(mx) \rightarrow perm(f(delete(mx))\ delete(mx)) \wedge ordered(f(delete(mx)))] \rightarrow perm(f(x)x) \wedge ordered(f(x))$

The first formula will be transformed by symbolical evaluation of predicate *perm*:

$$\frac{1.\ x = \underline{empty} \rightarrow perm(f(x)x) \wedge ordered(f(x))}{(RWF)}$$

1.1 $x = \underline{empty} \rightarrow f(x) = \underline{empty} \wedge ordered(f(x))$

1.2 $x = \underline{empty} \rightarrow [perm(f(x)x) \leftrightarrow f(x) = \underline{empty}]$

(instance of AX22)

After a (**)-step the predicate *ordered* will be evaluated:

1.1 x = empty → f(x) = empty ∧ ordered(f(x))

(**)———————————————————————————————————————

1.1.1 x = empty → f(x) = empty ∧ ordered(empty)

1.1.1 x = empty → f(x) = empty ∧ ordered(empty)

(RWF)———————————————————————————————————————

1.1.1.1 x = empty → f(x) = empty

1.1.1.2 empty = empty → [ordered(empty) ↔ true]

(instance of AX18)

With formula 1.1.1.1 the first definition formula has been derived.

For the following assume I to be an abbreviation for the induction hypothesis ∀m : nat [member(mx) → perm(f(delete(mx)) delete(mx)) ∧ ordered(f(delete(mx)))].

A symbolical evaluation of a predicate occurring in formula 2 can only be made if some information about the structure of the term f(x) is available. Therefore a case-analysis-step is carried out first:

2. x ≠ empty ∧ I → perm(f(x)x) ∧ ordered(f(x))

(CA)———————————————————————————————————————

2.1 x ≠ empty ∧ f(x) = empty ∧ I → perm(f(x)x) ∧ ordered(f(x))

2.2 x ≠ empty ∧ f(x) = cons(nz) ∧ I → perm(f(x)x) ∧ ordered(f(x))

2.2 $x \neq \underline{empty} \wedge f(x) = cons(nz) \wedge I \rightarrow perm(f(x)x) \wedge ordered(f(x))$

(RWF)

2.2.1 $x \neq \underline{empty} \wedge f(x) = cons(nz) \wedge I \rightarrow perm(z\ delete(nx)) \wedge ordered(f(x))$

2.2.2 $f(x) = cons(nz) \wedge x \neq \underline{empty} \rightarrow perm(f(x)x) \leftrightarrow perm(z\ delete(nx))$

(instance of AX23)

A second CA-step enables symbolical evaluation of predicate *ordered*:

2.2.1 $x \neq \underline{empty} \wedge f(x) = cons(nz) \wedge I \rightarrow perm(z\ delete(nx)) \wedge ordered(f(x))$

(CA)

2.2.1.1 $x \neq \underline{empty} \wedge f(x) = cons(nz) \wedge n = min(f(x)) \wedge I \rightarrow perm(z\ delete(nx)) \wedge ordered(f(x))$

2.2.1.2 $x \neq \underline{empty} \wedge f(x) = cons(nz) \wedge n \neq min(f(x)) \wedge I \rightarrow perm(z\ delete(nx)) \wedge ordered(f(x))$

2.2.1.1 $x \neq \underline{empty} \wedge f(x) = cons(nz) \wedge n = min(f(x)) \wedge I \rightarrow perm(z\ delete(nx)) \wedge ordered(f(x))$

(RWF)

2.2.1.1.1 $x \neq \underline{empty} \wedge f(x) = cons(nz) \wedge n = min(f(x)) \wedge I \rightarrow perm(z\ delete(nx)) \wedge ordered(z)$

2.2.1.1.2 $f(x) = cons(nz) \wedge n = min(f(x)) \rightarrow [ordered(f(x)) \leftrightarrow ordered(z)]$

(instance of AX20)

In order to eventually obtain the applicability of the predicate-rule to arrive at a value of the variable z we use an instance of our induction hypothesis. This we will obtain by the specialization-rule:

2.2.1.1.1 $x \neq \underline{empty} \wedge f(x) = cons(nz) \wedge n = min(f(x)) \wedge$
$\forall m : nat\ [member(mx) \rightarrow perm(f(delete(mx))$
$delete(mx)) \wedge ordered(f(delete(mx)))] \rightarrow$
$perm(z\ delete(nx)) \wedge ordered(z)$

(SPEC)

2.2.1.1.1.1 $x \neq \underline{empty} \wedge f(x) = cons(nz) \wedge n = min(f(x)) \wedge$
$[member(nx) \rightarrow perm(f(delete(nx))\ delete(nx)) \wedge$
$ordered(f(delete(nx)))] \rightarrow perm(z\ delete(nx))$
$ordered(z)$

2.2.1.1.1.2 $\forall m : nat\ [member(mx) \rightarrow perm(f(delete(mx))$
$delete(mx)) \wedge ordered(f(delete(mx)))] \rightarrow$
$[member(nx) \rightarrow perm(f(delete(nx))\ delete(nx)) \wedge$
$ordered(f(delete(nx)))]$

Now we use the strengthening-rule to eliminate the implication in the antecedent of formula A := 2.2.1.1.1.1.

A $x \neq \underline{empty} \wedge f(x) = cons(nz) \wedge n = min(f(x)) \wedge$
$[member(nx) \rightarrow perm(f(delete(nx))$
$delete(nx)) \wedge ordered(f(delete(nx)))] \rightarrow$
$perm(z\ delete(nx)) \wedge ordered(z)$

(STR)

A.1 $x \neq \underline{empty} \wedge f(x) = cons(nz) \wedge n = min(f(x)) \wedge$
$member(nx) \wedge perm(f(delete(nx))$
$delete(nx)) \wedge ordered(f(delete(nx))) \rightarrow$
$perm(z\ delete(nx)) \wedge ordered(z)$

Now we shall replace the term $f(x)$ in the formula $n = min(f(x))$ by the term $cons(nz)$. By doing this we are able to achieve that the term $f(x)$ occurs only once in the antecedent. From there it can be removed (after some further transformations) by the EQ-rule.

The problem which we have to solve now is that we may indeed obtain a value for the variable z by applying the PRED-rule, but what about n? No formula in the succedent of our modified specification makes the application of the EQ-rule possible to attain a suitable value for n.

Therefore we apply the extension-rule with an instance of lemma L1:

A.1 x ≠ empty ∧ f(x) = cons(nz) ∧ n = min(cons(nz)) ∧
member(nx) ∧ perm(f(delete(nx)) delete(nx)) ∧
ordered(f(delete(nx))) → perm (z delete(nx)) ∧
ordered(z)

(EXT)———————————————————————————————

A.1.1 member(nx) ∧ perm(f(delete(nx)) delete(nx)) →
min(cons(n f(delete(nx)))) = min(x)
(instance of lemma L1)

A.1.2 x ≠ empty ∧ f(x) = cons(nz) ∧ n = min(cons(nz)) ∧
member(nx) ∧ perm(f(delete(nx)) delete(nx)) ∧
ordered(f(delete(nx))) → perm(z delete(nx)) ∧
ordered(z) ∧ min(cons(n f(delete(nx)))) = min(x)

Applying the PRED-rule twice yields:

A.1.2

(PRED)
(PRED) ——————————————————————————————————————

A.1.2.1 $x \neq \underline{empty} \wedge f(x) = cons(nz) \wedge n = min(cons(nz)) \wedge$
$member(nx) \rightarrow z = f(delete(nx)) \wedge$
$min(cons(n\ f(delete(nx)))) = min(x)$

Applicability of the equality-rule to get a value of n is attained by a (**)-step:

A.1.2.1

(**) ——————————————————————————————————————

A.1.2.1.1 $x \neq \underline{empty} \wedge f(x) = cons(nz) \wedge n = min(cons(nz)) \wedge$
$member(nx) \rightarrow z = f(delete(nx)) \wedge$
$min(cons(nz)) = min(x)$

A.1.2.1.1

(EQ) ——————————————————————————————————————

A.1.2.1.1.1 $x \neq \underline{empty} \wedge f(x) = cons(nz) \wedge member(nx) \rightarrow$
$z = f(delete(nx)) \wedge n = min(x)$

Rule (**) will now replace the variable n in $z = f(delete(nx))$ by the term $min(x)$:

$x \neq \underline{empty} \wedge f(x) = cons(nz) \wedge member(nx) \rightarrow$
$z = f(delete(nx)) \wedge n = min(x)$

(**) ——————————————————————————————————————

$x \neq \underline{empty} \wedge f(x) = cons(nz) \wedge member(nx) \rightarrow$
$z = f(delete(min(x)x)) \wedge n = min(x)$

As in example 3.3 the definition formula of f is now obtained by first applying the RWF-rule with the injectivity of the constructor function and then applying the equality-rule:

$x \neq \underline{empty} \wedge f(x) = cons(nz) \wedge member(nx) \rightarrow$

$z = f(delete(min(x)x)) \wedge n = min(x)$

(RWF)

$x \neq \underline{empty} \wedge f(x) = cons(nz) \wedge member(nx) \rightarrow cons(nz) =$

$cons(min(x)f(delete(min(x)x)))$

$n = min(x) \wedge z = f(delete(min(x)x)) \leftrightarrow cons(nz) =$

$cons(min(x)f(delete(min(x)x)))$

(instance of AX6

$x \neq \underline{empty} \wedge f(x) = cons(nz) \wedge member(nx) \rightarrow cons(nz) =$

$cons(min(x)f(delete(min(x)x)))$

(EQ)

$x \neq \underline{empty} \wedge member(nx) \rightarrow f(x) = cons(min(x)$

$f(delete(min(x)x)))$

Finally we are left with the fact that the antecedent of the derived definition formula F includes a formula *(member(nx))* with a variable (n) occurring in it which appears nowhere else in F.

This calls for an application of the elimination-rule:

$x \neq \underline{empty} \wedge member(nx) \rightarrow f(x) = cons(min(x)$

$f(delete(min(x)x)))$

(EL)

$x \neq \underline{empty} \rightarrow f(x) = cons(min(x)f(delete(min(x)x)))$

$x \neq \underline{empty} \rightarrow \exists n : nat\ member(nx)$

Now the synthesis process terminates. The definition formulas of function f (i.e. *sort*) are:

$x = \underline{empty} \rightarrow f(x) = \underline{empty}$

$x \neq \underline{empty} \rightarrow f(x) = cons(min(x)f(delete(min(x)x)))$.

They correspond to the following definition:

FUNCTION f(x:list): list =

IF x = empty THEN empty

IF x ≠ empty THEN cons(min(x)f(delete(min(x)x)))

The definition has to be checked for uniqueness and finally the following formulas have to be proved:

2.1 $x \neq \underline{empty} \wedge f(x) = \underline{empty} \wedge I \rightarrow perm(f(x)x) \wedge ordered(f(x))$

2.2.1.2 $x \neq \underline{empty} \wedge f(x) = cons(nz) \wedge n \neq min(f(x)) \wedge I \rightarrow$
$perm(z\ delete(nx) \wedge ordered(f(x))$

2.2.1.1.1.2 $\forall m : nat\ [member(mx) \rightarrow perm(f(delete(mx))$
$delete(mx)) \wedge ordered(f(delete(mx)))] \rightarrow$
$[member(nx) \rightarrow perm(f(delete(nx))\ delete(nx)) \wedge$
$ordered(f(delete(nx)))]$

- $x \neq \underline{empty} \rightarrow \exists n : nat\ member(nx)$.

The first and second formulas are easily proved using the derived definition formulas of function f. The third formula is very simple and the last one can be proved by AX4 and the definition formulas of predicate *member*.

4. Conclusion

In order to eliminate existential quantifiers a method has been presented to show how to synthesize a constructive definition of the corresponding skolem function.

So far only atomar formulas with the equality predicate and with only one occurrence of the skolem function have been considered. For these specifications synthesis processes have been carried out as demonstrated in examples 3.1 and 3.2. Now further work on this topic consists in exactly specifying a synthesis system which will be able to deal with such specifications and then implementing it.

Later more complex specifications involving implications or conjunctions of predicates different from equality will be considered. This should solve problems like those occurring in examples 3.3 and 3.4.

Another problem which will become particularly important for more complex specifications is the great number of CA-steps which leads to an enormous increase in the number of formulas to be transformed. It may become necessary to carefully select those formulas which can possibly provide a definition. A way of doing this is to first prove the premises of the formulas resulting from a CA-step to exclude those whose premises are false. This restricts the number of formulas to be considered and helps to avoid useless derivations.

Additionally the first transformation step may be modified to make any suitable case analysis possible (as discussed at the end of section 2.1.1). It may be helpful, for instance, to first look at the specification and all function and predicate definitions involved and then to decide which case analysis would be helpful (cf. example 3.4).

Further problems are those of weak specifications as in example 3.3 which force the system to find explicit values.

Presently the lemmata to be used are assumed to be available in a data base and are pattern directed selected.
Yet it would be helpful to enable the system to generate lemmata, for instance, about properties of functions, particularly while trying to reach a successful match between the formulas derived from the specification and the induction hypotheses.

Finally there is the problem of treating inconstructive specifications like the following one of the *gcd*-function.

$$\forall x,y,z : \mathrm{nat}[\mathrm{divides}(f(xy)x) \wedge \mathrm{divides}(f(xy)y)$$

$$\wedge\ [\mathrm{divides}(zx) \wedge \mathrm{divides}(zy) \rightarrow \mathrm{le}(zf(xy))]] \quad .$$

These problems together with an extension of the strategies for applying the transformation rules will be the subject of our future work.

References

[Aub79] R. Aubin
Mechanizing Structural Induction
Part I: Formal System
Part II: Strategies
Theoretical Computer Science 9 (1979)

[BES81] K. Bläsius, N. Eisinger, J. Siekmann, G. Smolka,
A. Herold and C. Walther
The Markgraf Karl Refutation Procedure
Proc. of the 7th International Joint Conference on
Artificial Intelligence 1981

[BI80] W. Bibel
Syntax-Directed,
Semantics-Supported Program Synthesis
Artificial Intelligence 14 (1980)

[BM79] R.S. Boyer and J S. Moore
A Computational Logic
Academic Press (1979)

[DMW81] W. Dilger, J. Müller and W. Womann
Einführung in die Markgraf Karl Refutation Procedure
Interner Bericht 40/81
Fachbereich Informatik, Universität Kaiserslautern (1981)

[GO80] C.A. Goad
Computational Uses of the Manipulation of Formal Proofs
PhD Thesis, Stanford University (1980)

[GTW77] J.A. Goguen, J.W. Thatcher and E.G. Wagner
An Initial Algebra Approach to the Specification,
Correctness and Implementation of Abstract Data Types
IBM Research Report RCG 487 (1977)

[MW79/1] Z. Manna and R. Waldinger
A Deductive Approach to Program Synthesis
Proc. of the 6th International Joint Conference on
Artificial Intelligence (1979)

[MW79/2] Z. Manna and R. Waldinger
Synthesis: Dreams → Programs
IEEE Transactions on Software Engineering
Vol. SE-5 No. 4 (1979)

[Ohl82] H.J. Ohlbach
The Markgraf Karl Refutation Procedure:
The Logic Engine
Interner Bericht 24/82
Institut für Informatik I, Universität Karlsruhe
(1982)

[WH81] P.H. Winston and B.K.P. Horn
LISP
Addison Wesley (1981)