Concept Formation

Christoph Walther *

Fachgebiet Programmiermethodik
Technische Universität Darmstadt

1 Introduction

The frequent need of user interaction is commonly considered as one of the main obstacles for accepting theorem proving technology in applications, like e.g. program verification. User interaction is often needed to support a system in the construction of a proof. This requires a certain competence in logic and in proof engineering, a competence non-expert users neither possess nor are willing to acquire. Therefore many efforts in the community aim to improve the inference machinery of a system in order to relieve a user from the burden of proof construction as far as possible.

But user interaction is also needed to provide a system with useful lemmas and concepts which are necessary to prove a certain theorem. These kind of interactions often requires a deep insight into the domain of discourse, and working out the missing notions can be a tedious and frustrating effort, even for users well familiar with theorem proving.

In this note, we focus on the latter problem and illustrate the need of machine support for concept formation by four examples. By "concept formation" we understand the invention of mathematical notions which are not provided by a given formal presentation, i.e. an axiomatization of some theory, but are needed to formulate a required generalization of a statement or to speculate a required lemma.

2 Generalization and Lemma Speculation

When proving theorems by induction, often lemmas are required to complete the proofs of the induction formulas representing the base and the step cases of the induction. For a statement to be verified, a system creates the induction formulas and then calls a first-order theorem prover to compute the proofs for these formulas. However, quite often an induction formula is not *valid* because some lemma is missing, and then the theorem prover gets stuck. Now to continue, such a lemma has to be provided and verified, and then the failed proof attempt is resumed having the new lemma available now.

Sometimes the required lemma evolves by a *generalization* from the proof-goal under consideration, e.g. by replacing terms with (universally quantified)

* Chr.Walther@informatik.tu-darmstadt.de

Submitted to CADE-19 WS on Grand Challenges for Autom. Reasoning *(May 2003)*

variables, dropping disjunctive parts of a formula, strengthening the proof-goal by adding a conjunctive part etc., cf. e.g. [10].

For instance, when proving the ordering property of *Insertion Sort*, a prover may get stuck with the proof obligation

$$\forall k{:}list.\ k \neq \varepsilon \wedge ord(isort(tl(k)) \curvearrowright ord(insert(hd(k), isort(tl(k))))) \qquad (1)$$

from which the missing lemma

$$\forall k{:}list, n{:}nat.\ ord(k) \curvearrowright ord(insert(n, k)) \qquad (2)$$

is easily spotted by a generalization.[1]

In other cases, properties of the functions involved with a statement have to be speculated and formulated as a lemma. E.g. when verifying the correctness of a *Byte Adder*, a theorem prover may get stuck with the proof obligation[2]

$$\begin{array}{l}\forall c{:}bit, x{:}byte.\ c \neq O \wedge lsb(x) \neq O \curvearrowright \\ dbl(byte2nat(byte\text{-}inc(rshift(x)))) = succ(succ(dbl(byte2nat(rshift(x)))))\end{array} \qquad (3)$$

a proof of which requires a lemma about *byte-inc*, viz.

$$\forall x{:}byte.\ byte2nat(byte\text{-}inc(x)) = succ(byte2nat(x))\ . \qquad (4)$$

These problems are a bit harder than the generalization problems, because the missing lemmas do not evolve directly from the proof goal on which the theorem prover failed.

The generalization problem is a special case of the *lemma speculation* problem. Both problems are well recognized in the theorem proving community, and several proposals exist to spot the missing lemma by machine, cf. e.g. [2], [3], [4], [5], [6], [7], [8], [9], [11], [12].

3 Cases for Concept Formation

Matters become more complicated, if additional notions are required to formulate the missing lemma. Here a new concept has to be invented, i.e. some mathematical notion which is not directly available from the given presentation. We illustrate some cases for concept formation by four examples.

Example 1 In the first example, we consider the permutation property of *Mergesort*. When proving the step case, a theorem prover may get stuck with

[1] Throughout this note, linear lists `list` over natural numbers `nat` are defined by the data structures `structure nat <= 0, succ(pred:nat)` and `structure list <= ε, add(hd:nat,tl:list)`.

[2] Bits and bytes are defined here by the data structures `structure bit <= O, I` and `structure byte <= null, mkbyte(rshift:byte,lsb:bit)`.

the proof obligation

$$\begin{array}{l}\forall k{:}list, n{:}nat.\ k \neq \varepsilon \wedge tl(k) \neq \varepsilon \wedge n = hd(k) \wedge n = hd(tl(k)) \wedge \\ \quad \forall n'{:}nat.\ occurs(n', dis.ev(k)) = occurs(n', msort(dis.ev(k))) \wedge \\ \quad \forall n''{:}nat.\ occurs(n'', dis.odd(k)) = occurs(n'', msort(dis.odd(k))) \\ \curvearrowright occurs(n, merge(msort(dis.ev(k)), msort(dis.odd(k)))) \\ \quad = succ(succ(occurs(n, tl(tl(k))))) \ .\end{array} \tag{5}$$

Here the problem is, that the induction hypotheses cannot be applied by the presence of *merge* in the (simplified) induction conclusion.[3] Now, being familiar with the meaning of the procedures in (5), it is obvious that the lemma

$$\forall k, l{:}list, n{:}nat.\ occurs(n, merge(k, l)) = plus(occurs(n, k), occurs(n, l)) \tag{6}$$

will help, because it removes *merge* from the goal-equation in (5), thus making the induction hypotheses applicable. To complete the proof, further lemmas, and in particular the commutativity of *plus*, are required.

However, the definition of *plus* is not provided with the original problem, which means that the concept of addition has to be invented before starting to speculate lemma (6). To solve the problem by machine, a system may use an undefined procedure

```
function F(x:nat, y:nat):nat <= ?
```

,

start an induction proof of

$$\forall k, l{:}list, n{:}nat.\ occurs(n, merge(k, l)) = F(occurs(n, k), occurs(n, l)) \tag{7}$$

and retrieve the definition of F from the unsolved proof obligations in this induction proof. But unfortunately this approach fails, because F had to be synthesized as a *commutative* definition of addition.

Example 2 Sometimes, there is not an unique concept to invent (as in the *Mergesort* case), but alternatives exist. This introduces the risk to do more work than required, simply because one failed to discover are more superiour concept. We illustrate this problem by an example from the verification of *Quicksort*.

When verifying the ordering property of this algorithm, a theorem prover may get stuck (in the step case) with the proof obligation[4]

$$\begin{array}{l}\forall k{:}list.\ k \neq \varepsilon \wedge \\ \quad ord(qsort(sm(hd(k), tl(k)))) \wedge ord(qsort(lg(hd(k), tl(k)))) \\ \curvearrowright ord(app(qsort(sm(hd(k), tl(k))), add(hd(k), qsort(lg(hd(k), tl(k)))))) \ .\end{array} \tag{8}$$

[3] The procedure `dis.ev` computes a sublist of its argument list by disregarding all list elements on odd argument positions, and `dis.odd` performs a similar computation by disregarding all list elements on even positions.

[4] A procedure call `sm(`n`,`k`)` computes a sublist of k by disregarding all list elements which are *larger* than n, and `lg(`n`,`k`)` performs a similar computation by disregarding all list elements which are *smaller* than or *equal* to n.

Being familiar with the "idea" of *Quicksort*, it is rather obvious what is missing here: If l and k are ordered lists, then $app(k,l)$ is also ordered, provided the last element of k is less than or equal to the first element of l. To formulate this observation, the concept of the *last list element* is needed, and one obtains

$$\forall k,l{:}list.\ ord(k) \wedge last(k) \leq hd(l) \wedge ord(l) \curvearrowright ord(app(k,l)) \qquad (9)$$

as one of the key-lemmas for this verification problem. In course of the proof, another concept, viz. *list membership*, is also required to state, e.g., that $sm(n,k)$ and $qsort(k)$ both compute a sublist of k.

When proving (8) with the **√eriFun** system, cf. [1], [13], 9 auxiliary lemmas have to be formulated, all of which have an automated proof.[5] However, the proof of (8) requires 7 user interactions to tell the system which of the auxiliary lemmas has to be applied next.

As this is an unusual bad performance of the system, we looked for an alternative for proving (8). To this effect, the concepts of a *lower* and an *upper bound* of a list are invented, and using these notions, one of the key-lemmas for proving (8) is formulated as

$$\forall k,l{:}list.\ ord(k) \wedge upper.bound(k,hd(l)) \wedge ord(l) \curvearrowright ord(app(k,l))\ . \qquad (10)$$

This time, 13 auxiliary lemmas about upper and lower bounds are needed, stating e.g. that an upper bound of k also is an upper bound of $sm(n,k)$ and of $qsort(k)$. All these lemmas have an automated proof too, but now the proof of (8) requires only *one* user interaction. Moreover, this interaction is not needed, if the system is given more resources for search, whereas with the former attempt, the need for all interactions persists.

As the auxiliary lemmas in both attempts do not express deep thoughts and are rather obvious to spot, the concepts of an upper and an lower bound are obviously more useful here than the concepts of the last list element and list membership. Hence this example rises the question, how to find a *good* concept *a priori*, i.e. without exploring several alternatives.

Example 3 Having an idea for a good concept sometimes is not enough. Also a good *representation* matters, in particular when the domain of discourse is not so simple as for the sorting algorithms. To illustrate this point, let us consider an example from the verification of *Binary Search*. Imagine a procedure

```
function find(key:nat, a:list, i:nat, j:nat):bool <= ...
```

which decides whether a *key* is stored between the indices i and j in an array a with dimension 0..|a|-1 (represented by a linear list). The procedure $find$ operates according to the binary search method: If $i > j$, then `false` is returned, and the result is $a[i] = key$ if $i = j$. In the remaining case $i < j$ holds, and $find(key, a, i, i + \lfloor \frac{j-i}{2} \rfloor - 1)$ is recursively computed if $key < a\left[i + \lfloor \frac{j-i}{2} \rfloor\right]$,

[5] The statements about system performance refer to system version *2.6.1*.

$find(key, a, i + \lfloor \frac{j-i}{2} \rfloor + 1, j)$ is recursively computed if $a\,[i + \lfloor \frac{j-i}{2} \rfloor] < key$, and otherwise `true` is returned as result.

The completeness property of $find$ is stated as

$$\forall a{:}list, key{:}nat.\ ord(a) \wedge key \in a \curvearrowright find(key, a, 0, |a|\text{-}1) \tag{11}$$

where $key \in a$ is computed by searching key in a step by step. The difficulty in proving (11) is typical for a statement involving a tail-recursive procedure like $find$ stemming from the body of a loop: The variables which vary in a procedure (or in the loop-body respectively), viz. i and j in the present case, are replaced in the statement with non-variable terms representing the initialization of the loop-variables, viz. 0 and $|a|$-1 here, thus preventing an induction upon these variables. So to enable an induction, (11) has to be generalized, and to formulate the required generalization a new concept, viz. the *partition* $a\,[i..j]$ of an array has to be invented. This concept allows to formulate the generalization

$$\begin{array}{l} \forall a{:}list, key, i, j{:}nat. \\ \quad j < |a| \wedge ord(a) \wedge key \in a\,[i..j] \curvearrowright find(key, a, i, j) \end{array} \tag{12}$$

from which a proof of (11) immediately is obtained, by using a further lemma, viz.

$$\forall a{:}list, key{:}nat.\ key \in a \curvearrowright key \in a\,[0..\,|a|\text{-}1] \quad . \tag{13}$$

Having spotted the missing concept, a representation has to be developed for it such that subsequent proofs are supported. A straightforward representation of $key \in a\,[i..j]$ is obtained by defining a procedure

```
function partition(a:list,i:nat,j:nat):list <= ...
```

representing $a\,[i..j]$ by computing the sublist of a between the indices i and j. Now the lemmas

$$\begin{array}{l} \forall a{:}list, key, i, j{:}nat. \\ \quad j < |a| \wedge ord(a) \wedge key \in partition(a, i, j) \curvearrowright find(key, a, i, j) \end{array} \tag{14}$$

and

$$\forall a{:}list, key{:}nat.\ key \in a \curvearrowright key \in partition(a, 0, |a|\text{-}1) \tag{15}$$

have to be proved. However, we failed in proving (14) when following this idea. New subgoals were created involving the creation of new lemmas, which in turn raised new subgoals etc., until we gave up. An analysis of this tedious and frustrating effort revealed that the problem is the use of *partition*. Roughly speaking, this procedure (using `tl`) strips of the list elements in a between positions 0 and $i-1$ from the beginning of a and cuts of (using a procedure *butlast*) the list elements between $|a|$-1 and $j+1$ working from the end of a towards the beginning. The problem caused by this approach is that by the presence of *butlast*, the use of the induction hypotheses and auxiliary lemmas is spoiled so that the proofs do not get through, cf. [15] for details.

But fortunately, this problem immediately disappears if $key \in a\,[i..j]$ is implemented *directly*, i.e. by using a procedure

```
function in.partition(key:nat, a:list, i:nat, j:nat):bool <= ...
```

instead of *partition* as the new concept. Now the required generalization (and the auxiliary lemma) are expressed as

$$\begin{array}{l}\forall a{:}list, key, i, j{:}nat.\\ \quad j < |a| \wedge ord(a) \wedge in.partition(key, a, i, j) \curvearrowright find(key, a, i, j)\end{array} \tag{16}$$

and

$$\forall a{:}list, key{:}nat.\ key \in a \curvearrowright in.partition(key, a, 0, |a|\text{-}1)\ . \tag{17}$$

Both lemmas are easily proved, because the definition of *in.partition* is less complicated than the definition of *partition*. This example illustrates the general problem to distinguish a good *representation* of a new concept from a bad one *a priori*.

Example 4 In the final example, we consider the verification of a code generator, computing machine code from WHILE-programs. The language of WHILE-programs consists of conditional statements, while-loops, assignments, compound statements and a statement for doing nothing. WHILE-programs are defined by a data structure *wprog* and represent abstract syntax trees which e.g. are computed by a compiler from a program conforming to the concrete syntax of a programming language to be available for subsequent code generation.

An operational semantics for WHILE-programs is given by an interpreter

```
function eval(r:state, wp:wprog):state <= ...
```

which maps a program state and a WHILE-program to a program state. Instead of interpreting a WHILE-program *wp* by the interpreter *eval*, *wp* can be compiled into a machine program by some procedure

```
function code(wp:wprog):mprog <= ...
```

in order to execute the generated code by some target machine. Machine programs are defined by some data structure *mprog*, and the operation of the target machine is defined by another procedure, viz.

```
function exec(r:state, pc:nat, mp:mprog):state <= ... .
```

This procedure defines an operational semantics for machine programs, which are executed step-by-step by fetching and executing the instruction to which the program counter *pc* actually points in the machine program *mp*.

The correctness property for the code generator *code* is formally stated by

$$\forall wp{:}wprog, r{:}state.\ eval(r, wp) = exec(r, 0, code(wp))\ . \tag{18}$$

The induction proof of (18) develops into 3 base and 3 step cases. The base cases are easily proved, but the step cases require lemmas which are not that easy to spot. In the first step case, for instance, in which wp is a compound statement of the form $wp1; wp2$, the proof obligation

$$\begin{array}{l} \forall wp1, wp2{:}wprog, r{:}state. \\ \quad r \neq \bot \curvearrowright \begin{array}{l} exec(exec(r, 0, code(wp1)), 0, code(wp2)) \\ = exec(r, 0, app(code(wp1), code(wp2))) \end{array} \end{array} \tag{19}$$

is obtained after simplifying the induction conclusion. This proof obligation can be straightforwardly generalized to

$$\begin{array}{l} \forall mp_1, mp_2{:}mprog, r{:}state, pc{:}nat. \\ \quad exec(exec(r, pc, mp1), 0, mp2) = exec(r, pc, app(mp1, mp2)) \end{array} \tag{20}$$

in order to prove (19) with this equation. However, the generalization (20) does not hold. The "debugged" version of (20) reads as

$$\begin{array}{l} \forall mp_1, mp_2{:}mprog, r{:}state, pc{:}nat. \\ \quad pc < |mp1| \wedge HALT.free(mp1) \wedge \\ \quad closed^{+}(mp1) \wedge closed^{-}(mp1) \wedge closed^{-}(mp2) \\ \quad \curvearrowright exec(exec(r, pc, mp1), 0, mp2) = exec(r, pc, app(mp1, mp2)) \end{array} \tag{21}$$

where the new concepts of a $HALT.free$, a $closed^{+}$ and a $closed^{-}$ machine program had to be invented in order to formulate (21). Here a machine program mp is $HALT.free$ iff mp is free of `HALT` instructions, mp is $closed^{+}$ iff mp has no `JUMP-` or `BRANCH`-instructions moving the pc beyond the last instruction of mp, and mp is $closed^{-}$ iff mp has no `JUMP-` or `BRANCH`-instructions moving the pc before the first instruction of mp. Without demanding $HALT.free$, $closed^{+}$ and $closed^{-}$ for the machine programs in (21), counterexamples for (20) exist, see [14] for details.

The new concepts are also required to formulate auxiliary lemmas similar to (21), which are needed for the remaining step cases. This example illustrates, that sometimes a deep insight into the domain of discourse is required in order to develop the necessary concepts.

4 Summary

We have demonstrated the need of concept formation by four examples with increasing difficulty in the concept formation task. Computer support for problems of the kind given in Example 1 seem not that difficult to obtain, and examples of this kind provide a good starting point to analyze and tackle the problem. However, problems like in the remaining examples seem to be much harder to solve by machine, as they require much more insight into the domain of discourse than problems like in the first example.

It is our strong believe that the acceptance of theorem proving technology will grow significantly, if reasoning systems are able to solve concept formation problems of the kind illustrated by the examples of this note.

References

1. http://www.informatik.tu-darmstadt.de/pm/verifun/.
2. R. Aubin. Mechanizing structural induction Part II: Strategies. *Theoretical Computer Science*, 9(3):347–362, 1979.
3. R. S. Boyer and J. S. Moore. *A Computational Logic*. Academic Press, New York, 1979.
4. R. S. Boyer and J. S. Moore. Proving theorems about LISP functions. *J. ACM*, 22(1):129–144, 1979.
5. A. Bundy. The automation of proof by mathematical induction. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 13, pages 845–911. Elsevier Science, 2001.
6. A. Ireland and A. Bundy. Extensions to a Generalization Critic for Inductive Proofs. In M. McRobbie and J.Slaney, editors, *Proc. of the 13th Inter. Conf. on Automated Deduction (CADE-96)*, volume 1104 of *Lecture Notes in Artifical Intelligence*, pages 47–61, New Brunswick, 1996. Springer-Verlag.
7. A. Ireland and A. Bundy. Productive use of failure in inductive proof. *J. Automat. Reason.*, 16(1–2), 1996.
8. D. Kapur and M. Subramaniam. Lemma discovery in automating induction. In M. McRobbie and J.Slaney, editors, *Proc. 13th Intern. Conf. on Automated Deduction (CADE-96)*, volume 1104 of *Lecture Notes in Artificial Intelligence*, pages 538–552, New Brunswick, 1996. Springer-Verlag.
9. T. Walsh. A Divergence Critic for Inductive Proof. *J. Artifical Intelligence Research*, 4:209–235, 1996.
10. C. Walther. Mathematical Induction. In D. Gabbay, C. Hogger, and J. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 2, pages 127–227. Oxford University Press, Oxford, 1994.
11. C. Walther and T. Kolbe. On Terminating Lemma Speculations. *Information and Computation*, 162:96–116, 2000.
12. C. Walther and T. Kolbe. Proving theorems by reuse. *Artificial Intelligence*, 116:17–66, 2000.
13. C. Walther and S. Schweitzer. About √`VeriFun`. In F. Baader, editor, *Proc. of the 19th Inter. Conf. on Automated Deduction (CADE-19)*, volume 0000 of *Lecture Notes in Artifical Intelligence*, pages 1–5, Miami, 2003. Springer-Verlag.
14. C. Walther and S. Schweitzer. A Machine-Verified Code Generator. Technical Report VFR 03/01, Programmiermethodik, Technische Universität Darmstadt, 2003.
15. C. Walther and S. Schweitzer. A Verification of Binary Search. In D. Hutter and W. Stephan, editors, *Mechanizing Mathematical Reasoning: Techniques, Tools and Applications*, volume 2605 of *LNAI*, pages 1–18. Springer-Verlag, 2003.