Context-Moving Transformations for Function Verification[⋆]

Jürgen Giesl

Computer Science Dept., University of New Mexico, Albuquerque, NM 87131, USA,
E-mail: giesl@cs.unm.edu

Abstract. Several induction theorem provers have been developed which support mechanized verification of functional programs. Unfortunately, a major problem is that they often fail in verifying tail recursive functions (which correspond to imperative programs). However, in practice imperative programs are used almost exclusively.
We present an automatic transformation to tackle this problem. It transforms functions which are hard to verify into functions whose correctness can be shown by the existing provers. In contrast to classical program transformations, the aim of our technique is not to increase efficiency, but to increase verifiability. Therefore, this paper introduces a novel application area for program transformations and it shows that such techniques can in fact solve some of the most urgent current challenge problems in automated verification and induction theorem proving.

1 Introduction

To guarantee the correctness of programs, a formal verification is required. However, mathematical correctness proofs are usually very expensive and time-consuming. Therefore, program verification should be *automated* as far as possible.

As *induction*[1] is the essential proof method for verification, several systems have been developed for automated induction proving. These systems are successfully used for *functional* programs, but a major problem for their practical application is that they are often not suitable for verifying *imperative* programs. The reason is that the translation of imperative programs into the functional input language of these systems always yields *tail recursive* functions which are particularly hard to verify. Thus, developing techniques for proofs about tail recursive functions is one of the most important research topics in this area.

In Sect. 2 we present our functional programming language and give a brief introduction to induction proving. We illustrate that the reason for the difficulties in verifying tail recursive functions is that their accumulator parameter is usually initialized with a fixed value, but this value is changed in recursive calls.

[⋆] *Proceedings of the 9th International Workshop on Logic-based Program Synthesis and Transformation (LOPSTR '99)*, Venice, Italy, Lecture Notes in Computer Science 1817, pages 293-312, Springer, 2000. This work was supported by the DFG under grant GI 274/4-1.

[1] In this paper, "induction" stands for *mathematical induction*, i.e., it should not be confused with induction in the sense of machine learning.

This paper introduces a new framework for mechanized verification of such functions by first transforming them into functions which are better suitable for verification and by afterwards applying the existing induction provers for their verification. To solve the verification problems with tail recursive functions, the *context* around recursive accumulator arguments has to be shifted away, such that the accumulator parameter is no longer changed in recursive calls. For that purpose, we introduce two automatic transformation techniques in Sect. 3 - 5. While of course our transformations are not always applicable, they proved successful on a representative collection of tail recursive functions, cf. [Gie99b]. In this way, correctness of many imperative programs can be proved *automatically* without inventing loop invariants or generalizations.

2 Functional Programs and their Verification

We consider a first order functional language with eager semantics and (non-parameterized and free) algebraic data types. As an example, regard the data type nat for natural numbers whose objects are built with the *constructors* $\mathsf{0}$ and $\mathsf{s} : \mathsf{nat} \to \mathsf{nat}$ (for the successor function). Thus, the constructor ground terms represent the data objects of the respective data type. In the following, we often write "1" instead of "$\mathsf{s}(\mathsf{0})$", etc. For every n-ary constructor c there are n *selector* functions $d_1, \ldots, d_n$ which serve as inverse functions to c (i.e., $d_i(c(x_1, \ldots, x_n)) \equiv x_i$). For example, for the unary constructor s we have the selector function p such that $\mathsf{p}(\mathsf{s}(m)) \equiv m$ (i.e., p is the **p**redecessor function).

In particular, every program F contains the type bool whose objects are built with the (nullary) constructors true and false. Moreover, there is a built-in equality function $= : \tau \times \tau \to \mathsf{bool}$ for every data type τ. To distinguish the function symbol $=$ from the equality predicate symbol, we denote the latter by "$\equiv$". The *functions* of a *functional program* F have the following form.

```
function $f\,(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau \;\Leftarrow$
  if $b_1$  then $r_1$
        ⋮
  if $b_m$ then $r_m$
```

Here, "**if** b_i **then** r_i" is called the i-th *case* of f with *condition* b_i and *result* r_i. For functions with just one case of the form "**if** true **then** r" we write "**function** $f\,(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau \;\Leftarrow\; r$". To ease readability, if b_m is true, then we often denote the last case by "**else** r_m". As an example, consider the following function (which calls an auxiliary algorithm $+$ for addition).

```
function $\mathsf{times}\,(x, y : \mathsf{nat}) : \mathsf{nat} \;\Leftarrow$
  if $x \neq \mathsf{0}$ then $y + \mathsf{times}(\mathsf{p}(x), y)$
              else  $\mathsf{0}$
```

If a function f is called with a tuple of ground terms t^* as arguments, then t^* is evaluated first (to constructor ground terms q^*). Now the condition $b_1[x^*/q^*]$

of the first case is checked. If it evaluates to true, then $r_1[x^*/q^*]$ is evaluated. Otherwise, the condition of the second case is checked, etc. So the conditions of a functional program as above are tested from top to bottom.

Our aim is to verify statements about the algorithms of a functional program. We only consider universally quantified equations $\forall_{\dots}\ s \equiv t$ and we often omit the quantifiers to ease readability. Let s, t contain the tuple of variables x^*. Then $s \equiv t$ is *inductively true* for the program F, denoted $F \models_{\mathrm{ind}} s \equiv t$, if for all those data objects q^* where evaluation of $s[x^*/q^*]$ or evaluation of $t[x^*/q^*]$ is defined, evaluation of the other term $t[x^*/q^*]$ resp. $s[x^*/q^*]$ is defined as well, and if both evaluations yield the same result. For example, the conjecture

$$\mathsf{times}(\mathsf{times}(x, y), z) \equiv \mathsf{times}(x, \mathsf{times}(y, z)) \tag{1}$$

is inductively true, since $\mathsf{times}(\mathsf{times}(x, y), z)$ and $\mathsf{times}(x, \mathsf{times}(y, z))$ evaluate to the same result for all instantiations with data objects. Similar notions of inductive truth are widely used in program verification and induction theorem proving. For an extension of inductive truth to more general formulas and for a model theoretic characterization see e.g. [ZKK88,Wal94,BR95,Gie99c].

To prove inductive truth automatically, several *induction theorem provers* have been developed, e.g. [BM79,KM87,ZKK88,BSH+93,Wal94,BR95,BM98]. For instance, these systems can prove conjecture (1) by *structural* induction on the variable x. If we abbreviate (1) by $\varphi(x, y, z)$, then in the induction base case they would prove $\varphi(\mathsf{0}, y, z)$ and in the step case (where $x \neq \mathsf{0}$), they would show that the induction hypothesis $\varphi(\mathsf{p}(x), y, z)$ implies the induction conclusion $\varphi(x, y, z)$.

However, one of the main problems for the application of these induction theorem provers in practice is that most of them can only handle functional algorithms with recursion, but they are not designed to verify imperative algorithms containing loops.

The classical techniques for the verification of imperative programs (like the so-called Hoare-calculus [Hoa69]) allow the proof of partial correctness statements of the form $\{\varphi_{\mathrm{pre}}\}\ \mathcal{P}\ \{\varphi_{\mathrm{post}}\}$. The semantics of this expression is that in case of termination, the program $\mathcal{P}$ transforms all program states which satisfy the precondition φ_{pre} into program states satisfying the postcondition φ_{post}. As an example, regard the following imperative program for multiplication.

```
procedure multiply ($x, y, z$ : nat) $\Leftarrow$
    $z := \mathsf{0}$;
    while $x \neq \mathsf{0}$ do   $x := \mathsf{p}(x)$;
                         $z := y + z$    od
```

To verify that this imperative program is equivalent to the functional program times, one has to prove the statement

$$\{x \equiv x_0 \wedge y \equiv y_0 \wedge z \equiv \mathsf{0}\}\ \ \textbf{while}\ x \neq \mathsf{0}\ \textbf{do}\ x := \mathsf{p}(x); z := y + z\ \textbf{od}\ \ \{z \equiv \mathsf{times}(x_0, y_0)\}.$$

Here, x_0 and y_0 are additional variables which represent the initial values of the variables x and y. However, in the Hoare-calculus, for that purpose one needs a *loop invariant* which is a consequence of the precondition and which (together with the exit condition $x = 0$ of the loop) implies the postcondition $z \equiv \mathsf{times}(x_0, y_0)$. In our example, the proof succeeds with the loop invariant

$$z + \mathsf{times}(x, y) \equiv \mathsf{times}(x_0, y_0). \tag{2}$$

The search for loop invariants is the main difficulty when verifying imperative programs. Of course, it would be desirable that programmers develop suitable loop invariants while writing their programs, but in reality this is still often not the case. Thus, for an *automation* of program verification, suitable loop invariants would have to be discovered mechanically. However, while there exist some heuristics and techniques for the choice of loop invariants [SI98], in general this task seems difficult to mechanize [Dij85].

Therefore, in the following we present an alternative approach for automated verification of imperative programs. For that purpose our aim was to use the existing powerful induction theorem provers. As the input language of these systems is restricted to functional programs, one first has to translate imperative programs into functional ones. Such a translation can easily be done automatically, cf. [McC60,Gie99a].

In this translation, every **while**-loop is transformed into a separate function. For the loop of the procedure multiply we obtain the following algorithm mult which takes the input values of x, y, and z as arguments. If the loop-condition is satisfied (i.e., if $x \neq 0$), then mult is called recursively with the new values of x, y, and z. Otherwise, mult returns the value of z. The whole imperative procedure multiply corresponds to the following functional algorithm with the same name which calls the auxiliary function mult with the initial value $z \equiv 0$.

```
function multiply ($x, y$ : nat) : nat ⇐
  mult($x, y, 0$)
```

```
function mult ($x, y, z$ : nat) : nat ⇐
  if $x \neq 0$ then mult(p($x$), $y$, $y + z$)
               else $z$
```

Thus, while the above functions may look unnatural on their own, verification of such functions is indeed an important practical problem, since this is required in order to verify (very natural) imperative procedures like multiply.

Now induction provers may be used to prove conjectures about the functions multiply and mult. However, it turns out that the functional algorithms resulting from this translation have a certain characteristic form which makes them unsuitable for verification tasks. In fact, this difficulty corresponds to the problem of finding loop invariants for the original imperative program.

To verify the equivalence of multiply and times using the transformed functions multiply and mult, one now has to prove $\mathsf{multiply}(x, y) \equiv \mathsf{times}(x, y)$, i.e.,

$$\mathsf{mult}(x, y, 0) \equiv \mathsf{times}(x, y). \tag{3}$$

Using structural induction on x, the base formula $\mathsf{mult}(0, y, 0) \equiv \mathsf{times}(0, y)$ can easily be proved, but there is a problem with the induction step. In the case $x \neq 0$ we have to show that the induction hypothesis

$$\mathsf{mult}(\mathsf{p}(x), y, \mathsf{0}) \equiv \mathsf{times}(\mathsf{p}(x), y) \tag{IH}$$

implies the induction conclusion $\mathsf{mult}(x, y, \mathsf{0}) \equiv \mathsf{times}(x, y)$. Using the algorithms of mult and times, the induction conclusion can be transformed into

$$\mathsf{mult}(\mathsf{p}(x), y, y) \equiv y + \mathsf{times}(\mathsf{p}(x), y). \tag{IC}$$

However, the desired proof fails, since the induction hypothesis (IH) cannot be successfully used for the proof of (IC).

The reason for this failure is due to the *tail recursive* form of mult (i.e., there is no context around mult's recursive call). Instead, its result is computed in the *accumulator* parameter z. The accumulator z is initialized with 0, but this value is changed in the recursive calls of mult. For that reason the induction hypothesis (where $z \equiv \mathsf{0}$) does not correspond to the induction conclusion (where $z \equiv y$).

The classical solution for this problem is to *generalize* the conjecture (3) to a stronger conjecture which is easier to prove. For instance, in our example one needs the following generalization which can be proved by a suitable induction.

$$\mathsf{mult}(x, y, z) \equiv z + \mathsf{times}(x, y) \tag{4}$$

Thus, developing generalization techniques is one of the main challenges in induction theorem proving [Aub79,BM79,HBS92,Wal94,IS97,IB99]. Note that the generalization (4) corresponds to the loop invariant (2) that one would need for a direct verification of the imperative program multiply in the Hoare-calculus. So in fact, finding suitable generalizations is closely related to the search for loop invariants.[2]

In this paper we propose a new approach to avoid the need for generalizations or loop invariants. The idea is to transform functions like mult, which are difficult to verify, into algorithms like times which are much better amenable to automated induction proofs. For example, the well-known induction theorem proving system NQTHM [BM79,BM98] fails in proving (3), whereas after a transformation of multiply and mult into times this conjecture becomes trivial. This approach of verifying imperative programs via a translation into functional programs is based on the observation that in functional languages there often exists a formulation of the algorithms which is easy to verify (whereas this formulation cannot be expressed in iterative form). The aim of our technique is to find such a formulation automatically.

Our approach has the advantage that the transformation solves the verification problems resulting from a tail recursive algorithm once and for all. On the other hand, when using generalizations or loop invariants one has to find a new generalization (or a new loop invariant, respectively) for every new conjecture

[2] A difference between verifying functional programs by induction and verifying imperative programs by loop invariants and inductive assertions is that for imperative programs one uses a "forward" induction starting with the initial values of the program variables and for functional programs a "reversed" induction is used which goes back from their final values to the initial ones. However, the required loop invariants resp. the corresponding generalizations are easily interchangeable, cf. [RY76].

about such an algorithm. Moreover, most techniques for finding generalizations or loop invariants have to be guided by the system user, since they rely on the presence of suitable lemmata. By these lemmata the user often has to provide the main idea for the generalization resp. the loop invariant. In contrast, our transformation works automatically.

In particular, automatic generalization techniques fail for many conjectures which contain *several* occurrences of a tail recursive function. As an example, regard the associativity of $\mathsf{multiply}$ or, in other words,

$$\mathsf{mult}(\mathsf{mult}(x, y, \mathsf{0}), z, \mathsf{0}) \equiv \mathsf{mult}(x, \mathsf{mult}(y, z, \mathsf{0}), \mathsf{0}). \tag{5}$$

Similar to (3), a direct proof by structural induction on x does not succeed. So again, the standard solution would be to generalize the conjecture (5) by replacing the fixed value $\mathsf{0}$ by suitable terms. For example, one may generalize (5) to

$$\mathsf{mult}(\mathsf{mult}(x, y, \underline{v}), z, \mathsf{0}) \equiv \mathsf{mult}(x, \mathsf{mult}(y, z, \mathsf{0}), \underline{\mathsf{mult}(v, z, \mathsf{0})}).$$

To ease readability, we have underlined those terms where the generalization took place. While the proof of this conjecture is not too hard (using the distributivity of $+$ over $\mathsf{multiply}$), we are not aware of any technique which would find this generalization (or the corresponding loop invariant) automatically, because it is difficult to synthesize the correct replacement of the third argument in the right-hand side (by $\mathsf{mult}(v, z, \mathsf{0})$). The problem is that the disturbing $\mathsf{0}$'s occurring in (5) cannot just be generalized to new variables, since this would yield a flawed conjecture. Thus, finding generalizations for conjectures with several occurrences of a tail recursive function is often particularly hard, as different occurrences of an instantiated accumulator may have to be generalized to different new terms.[3] On the other hand, our transformation allows us to prove such conjectures without user interaction. Essentially, the reason is that while generalizations and loop invariants depend on both the algorithms and the conjectures to be proved, the transformation only depends on the algorithms.

The area of program transformation is a well examined field which has found many applications in software engineering, program synthesis, and compiler construction. For surveys see e.g. [BW82,Par90,MPS93,PP96,PP98]. However, the transformations developed for these applications had a goal which is fundamentally different from ours. Our aim is to transform programs into new programs which are easier to verify. In contrast to that, the classical transformation methods aim to increase efficiency. Such transformations are unsuitable for our purpose, since a more efficient algorithm is often harder to verify than a less efficient easier algorithm. Moreover, we want to transform tail recursive algorithms into non-tail recursive ones, but in the usual applications of program transformation,

[3] An alternative generalization of (5) is $\mathsf{mult}(\mathsf{mult}(x, y, \mathsf{0}), z, \underline{v}) \equiv \mathsf{mult}(x, \mathsf{mult}(y, z, \mathsf{0}), \underline{v})$. This generalization is easier to find (as we just replaced both third arguments of the left- and right-hand side by the same new variable v). However, it is not easy to verify (its proof is essentially as hard as the proof of the original conjecture (5)).

non-tail recursive programs are transformed into tail recursive ones ("recursion removal", cf. e.g. [Coo66,DB76,BD77,Wan80,BW82,AK82,HK92]).

As the goals of the existing program transformations are often opposite to ours, a promising approach is to use these classical transformations *in the reverse direction*. To our knowledge, such an application of these transformations for the purpose of verification has rarely been investigated before. In this way, we indeed obtained valuable inspirations for the development of our transformation rules in Sect. 3 - 5. However, our rules go far beyond the reversed standard program transformation methods, because these methods had to be modified substantially to be applicable for the programs resulting in our context.

3 Context Moving

The only difference between mult and times is that the context $\underline{y+}\ \ldots$ to compute times' result is outside of the recursive call, whereas in mult the context $\underline{y+}\ \ldots$ is in the recursive argument for the accumulator z. This change of the accumulator in recursive calls is responsible for the verification problems with mult.

For that reason, we now introduce a transformation rule which allows to *move* the context away from recursive accumulator arguments to a position outside of the recursive call. In this way, the former result $\mathsf{mult}(\mathsf{p}(x), y, \underline{y+}z)$ can be replaced by $\underline{y+}\,\mathsf{mult}(\mathsf{p}(x), y, z)$. So the algorithm mult is transformed into

```
function mult $(x, y, z : \mathsf{nat}) : \mathsf{nat} \Leftarrow$
  if $x \neq 0$ then $y + \mathsf{mult}(\mathsf{p}(x), y, z)$
              else $z$.
```

To develop a rule for context moving, we have to find sufficient criteria which ensure that such a transformation is equivalence preserving. For our rule, we regard algorithms of the form (6) where the last argument z is used as an accumulator. Our aim is to move the contexts $r_1, \ldots, r_k$ of the recursive accumulator arguments to the top, i.e., to transform the algorithm (6) into (7).

```
function $f\,(x^* : \tau^*, z : \tau) : \tau \Leftarrow$
  if $b_1$      then $f(r_1^*, r_1)$
                  ⋮
  if $b_k$      then $f(r_k^*, r_k)$                (6)
  if $b_{k+1}$  then $r_{k+1}$
                  ⋮
  if $b_m$      then $r_m$
```

```
function $f\,(x^* : \tau^*, z : \tau) : \tau \Leftarrow$
  if $b_1$      then $r_1[z/f(r_1^*, z)]$
                  ⋮
  if $b_k$      then $r_k[z/f(r_k^*, z)]$          (7)
  if $b_{k+1}$  then $r_{k+1}$
                  ⋮
  if $b_m$      then $r_m$.
```

We demand $m > k \geq 1$, but the order of the f-cases is irrelevant and the transformation may also be applied if the accumulator z is not f's *last* parameter. (We just used the above formulation to ease readability.)

First of all, note that the intermediate values of the parameter z are not the same in the two versions of f. Thus, to guarantee that evaluation of both versions of f leads to the same cases in the same order, we must demand that the accumulator z does not occur in the conditions $b_1, \ldots, b_m$ or in $r_1^*, \ldots, r_k^*$.

Let u^*, w be constructor ground terms. Now for both versions of f, evaluation of $f(u^*, w)$ leads to the same f-cases $i_1, \ldots, i_d$ where $i_1, \ldots, i_{d-1} \in \{1, \ldots, k\}$ and $i_d \in \{k+1, \ldots, m\}$ (provided that the evaluation is defined). Let $t[r^*, s]$ abbreviate $t[x^*/r^*, z/s]$ (where for terms t containing at most the variables x^*, we also write $t[r^*]$) and let $a_h^* = r_{i_h}^*[r_{i_{h-1}}^*[\ldots[r_{i_1}^*[u^*]]\ldots]]$, where $a_0^* = u^*$. Then with the old definition of f we obtain the result (8) and with the new definition we obtain (9).

$$r_{i_d}[a_{d-1}^*, r_{i_{d-1}}[a_{d-2}^*, \ldots r_{i_2}[a_1^*, r_{i_1}[a_0^*, w]] \ldots]] \tag{8}$$

$$r_{i_1}[a_0^*, r_{i_2}[a_1^*, \ldots r_{i_{d-1}}[a_{d-2}^*, r_{i_d}[a_{d-1}^*, w]] \ldots]]. \tag{9}$$

For example, the original algorithm mult computes a result of the form

$$y_x + (y_{x-1} + (\ldots(y_2 + (y_1 + z))\ldots))$$

where y_i denotes the number which is added in the i-th execution of the algorithm. On the other hand, the new version of mult computes the result

$$y_1 + (y_2 + (\ldots(y_{x-1} + (y_x + z))\ldots)).$$

Therefore, the crucial condition for the soundness of this transformation is the *left-commutativity* of the contexts $r_1, \ldots, r_k$ moved, cf. [BW82]. In other words, for all $i \in \{1, \ldots, m\}$ and all $i' \in \{1, \ldots, k\}$ we demand

$$r_i[x^*, r_{i'}[y^*, z]] \equiv r_{i'}[y^*, r_i[x^*, z]].$$

Then (8) and (9) are indeed equal as can be proved by subsequently moving the inner $r_{i_j}[a_{j-1}^*, \ldots]$ contexts of (8) to the top. So for mult, we only have to prove $x + (y + z) \equiv y + (x + z)$ and $y + z \equiv y + z$ (which can easily be verified by the existing induction theorem provers).

Note also that since in the schema (6), $r_1, \ldots, r_m$ denote arbitrary *terms*, such a context moving would also be possible if one would exchange the arguments of $+$ in mult's recursive call. Then r_1 would be $z + y$ and the required left-commutativity conditions would read $(z+y)+x \equiv (z+x)+y$ and $z+y \equiv z+y$.

However, context moving may only be done, if all terms $r_1, \ldots, r_m$ contain the accumulator z. Otherwise f's new definition could be total although the original definition was partial. For example, if f has the (first) case

if $x \neq 0$ **then** $f(x, 0)$

then $f(x, z)$ does not terminate for $x \neq 0$. However, if we would not demand that z occurred in the recursive accumulator argument, then context moving could transform this case into "**if** $x \neq 0$ **then** 0". The resulting function is clearly not equivalent to the original one, because now the result of $f(x, z)$ is 0 for $x \neq 0$.

Finally, we also have to demand that in $r_1, \ldots, r_m$, the accumulator z may not occur within arguments of functions dependent on f. Here, every function is *dependent* on itself and moreover, if g is dependent on f and g occurs in the algorithm h, then h is also dependent on f. So in particular, this requirement excludes nested recursive calls with the argument z. Otherwise, the transformation would not preserve the semantics. As an example regard the following function, where the algorithm $\mathsf{one}(z)$ returns 1 for all arguments z.

```
function $\mathsf{f}(x, z : \mathsf{nat}) : \mathsf{nat} \Leftarrow$
  if $x \neq 0$ then $\mathsf{f}(\mathsf{p}(x), \mathsf{f}(z, 0))$
              else $\mathsf{one}(z)$
```

By moving the context $\mathsf{f}(\ldots, 0)$ to the top, the result of the first case would be transformed into $\mathsf{f}(\mathsf{f}(\mathsf{p}(x), z), 0)$. The original algorithm satisfies all previously developed conditions. However, the original algorithm is total, whereas after the transformation $\mathsf{f}(x, z)$ does not terminate any more for $x \neq 0$. Under the above requirements, the transformation of (6) into (7) is sound.

Theorem 1 (Soundness of Context Moving). *Let F be a functional program containing the algorithm (6) and let F' result from F by replacing (6) with (7). Then for all data objects t^*, t, and q, $f(t^*, t)$ evaluates to q in the program F iff it does so in F', provided that the following requirements are fulfilled:*

(A) $z \notin \mathcal{V}(b_1) \cup \ldots \cup \mathcal{V}(b_m)$
(B) $z \notin \mathcal{V}(r_1^*) \cup \ldots \cup \mathcal{V}(r_k^*)$
(C) For all $i \in \{1, \ldots, m\}$, $i' \in \{1, \ldots, k\}$: $F \models_{\mathrm{ind}} r_i[x^*, r_{i'}[y^*, z]] \equiv r_{i'}[y^*, r_i[x^*, z]]$
(D) $z \in \mathcal{V}(r_1) \cap \ldots \cap \mathcal{V}(r_m)$
(E) In $r_1, \ldots, r_m$, *z does not occur in arguments of functions dependent on f.*

Proof. We first prove the following context moving lemma for all constructor ground terms u^*, v^*, w and all $i' \in \{1, \ldots, k\}$:

$$F \models_{\mathrm{ind}} r_{i'}[v^*, f(u^*, w)] \equiv f(u^*, r_{i'}[v^*, w]). \tag{10}$$

We use an induction on u^* w.r.t. the relation $\succ_f$. Here, $u^* \succ_f q^*$ holds for the constructor ground terms u^* and q^* iff there exists a constructor ground term u such that $f(u^*, u)$ is defined in F and such that F-evaluation of $f(u^*, u)$ leads to a recursive call $f(q^*, q)$ for some constructor ground term q. The well-foundedness of $\succ_f$ is due to the requirements (A), (B), and (E).

If one of the two terms in the equation (10) is defined, then there is an $i \in \{1, \ldots, m\}$ such that $b_i[u^*] \equiv_F \mathsf{true}$ and $b_j[u^*] \equiv_F \mathsf{false}$ for all $1 \leq j < i$, where $s \equiv_F t$ abbreviates $F \models_{\mathrm{ind}} s \equiv t$. (Here we need condition (D) to infer the definedness of $f(u^*, w)$ from the definedness of $r_{i'}[v^*, f(u^*, w)]$.)
If $i \geq k + 1$, then

$$\begin{aligned} r_{i'}[v^*, f(u^*, w)] &\equiv_F r_{i'}[v^*, r_i[u^*, w]] \\ &\equiv_F r_i[u^*, r_{i'}[v^*, w]], \text{ by (C)} \\ &\equiv_F f(u^*, r_{i'}[v^*, w]), \text{ since } z \in \mathcal{V}(r_i) \text{ (by (D))}. \end{aligned}$$

If $i \leq k$, then we have

$$\begin{aligned} r_{i'}[v^*, f(u^*, w)] &\equiv_F r_{i'}[v^*, f(r_i^*[u^*], r_i[u^*, w])] \\ &\equiv_F f(r_i^*[u^*], r_{i'}[v^*, r_i[u^*, w]]), \text{ by the induction hypothesis} \\ &\equiv_F f(r_i^*[u^*], r_i[u^*, r_{i'}[v^*, w]]), \text{ by (C)} \\ &\equiv_F f(u^*, r_{i'}[v^*, w]), \qquad \text{since } z \in \mathcal{V}(r_i) \text{ (by (D))}. \end{aligned}$$

Thus, Lemma (10) is proved and now the "only if"-direction of Thm. 1 can also be shown by induction w.r.t. $\succ_f$. There must be an $i' \in \{1, \ldots, m\}$ such that $b_{i'}[t^*] \equiv_F \mathsf{true}$ and $b_{j'}[t^*] \equiv_F \mathsf{false}$ for all $1 \le j' < i'$. The induction hypothesis implies $b_{i'}[t^*] \equiv_{F'} \mathsf{true}$ and $b_{j'}[t^*] \equiv_{F'} \mathsf{false}$ as well.

If $i' \ge k+1$, then the conjecture follows from $f(t^*, t) \equiv_F r_{i'}[t^*, t]$, $f(t^*, t) \equiv_{F'} r_{i'}[t^*, t]$, and the induction hypothesis. If $i' \le k$, then we have $f(t^*, t) \equiv_F f(r^*_{i'}[t^*], r_{i'}[t^*, t]) \equiv_F q$ for some constructor ground term q. By Lemma (10) we obtain $r_{i'}[t^*, f(r^*_{i'}[t^*], t)] \equiv_F q$. Note that for all f-subterms $f(s^*, s)$ in this term, s^* evaluates to constructor ground terms q^* with $t^* \succ_f q^*$. For f-subterms where the root is in $r_{i'}$, this follows from Condition (E). Thus, the induction hypothesis implies $r_{i'}[t^*, f(r^*_{i'}[t^*], t)] \equiv_{F'} q$ and hence, we also have $f(t^*, t) \equiv_{F'} q$.

So the "only if"-direction of Thm. 1 is proved. The proof for the "if"-direction of Thm. 1 is completely analogous (where instead of $\succ_f$ one uses an induction on the length of $f(t^*, t)$'s evaluation in F'). $\square$

The algorithm obtained from mult by context moving is significantly easier to verify. As mult's (former) accumulator z is no longer changed, it can now be eliminated by replacing all its occurrences by 0. The semantics of the main function multiply remains unchanged by this transformation.

```
function multiply (x, y : nat) : nat ⇐        function mult (x, y : nat) : nat ⇐
  mult(x, y)                                     if x ≠ 0 then y + mult(p(x), y)
                                                          else 0
```

Now mult indeed corresponds to the algorithm times and thus, the complicated generalizations or loop invariants of Sect. 2 are no longer required. Thus, the verification problems for this algorithm are solved.

Similarly, context moving can also be applied to transform an algorithm like

```
function plus (x, z : nat) : nat ⇐                 function plus (x, z : nat) : nat ⇐
  if x ≠ 0 then plus(p(x), s(z))        into         if x ≠ 0 then s(plus(p(x), z))
           else z                                             else z,
```

which is much better suited for verification tasks. Here, for condition (C) we only have to prove $\mathsf{s}(\mathsf{s}(z)) \equiv \mathsf{s}(\mathsf{s}(z))$ and $\mathsf{s}(z) \equiv \mathsf{s}(z)$ (which is trivial).

To apply context moving mechanically, the conditions (A) - (E) for its application have to be checked automatically. While the conditions (A), (B), (D), and (E) are just syntactic, the left-commutativity condition (C) has to be checked by an underlying induction theorem prover. In many cases, this is not a hard task, since for algorithms like plus the terms $r_i[x^*, r_{i'}[y^*, z]]$ and $r_{i'}[y^*, r_i[x^*, z]]$ are already *syntactically* equal and for algorithms like mult, the required left-commutativity follows from the associativity and commutativity of "+". To ease the proof of such conjectures about auxiliary algorithms, we follow the strategy to apply our transformations to those algorithms first which depend on few other algorithms. Thus, we would attempt to transform "+" before transforming mult. In this way, one can usually avoid the need for generalizations when performing the required left-commutativity proofs. Finally, note that of course, context moving should only be done if at least one of the recursive arguments $r_1, \dots, r_k$ is different from z (otherwise the algorithm would not change).

Our context moving rule has some similarities to the reversal of a technique known in program transformation (*operand commutation*, cf. e.g. [Coo66,DB76, BW82]). However, our rule generalizes this (reversed) technique substantially.

For example, directly reversing the formulation in [BW82] would result in a rule which would also impose applicability conditions on the functions *that call* the transformed function f (by demanding that f's accumulator would have to be initialized in a certain way). In this way, the applicability of the reversed rule would be unnecessarily restricted (and unnecessarily difficult to check). Therefore, we developed a rule where context moving is separated from the subsequent replacement of the (former) accumulator by initial values like $\mathsf{0}$. Moreover, in [BW82] the problems concerning the occurrence of the accumulator z and of nested recursive calls are not examined (i.e., the requirements (D) and (E) are missing there). Another important difference is that our rule allows the use of *several different* recursive arguments $r_1, \dots, r_k$ and the use of *several* non-recursive cases with *arbitrary* results (whereas reversing the formulation in [BW82] would only allow one single recursive case and it would only allow the non-recursive result z instead of the arbitrary terms $r_{k+1}, \dots, r_m$). Note that for this reason in our rule we have to regard *all* cases of an algorithm at once.

As an example where this flexibility of our transformation rule is needed consider the following algorithm to compute the multiplication of all elements in a list, where however occurring 0's are ignored. We use a data type list for lists of naturals with the constructors $\mathsf{nil} : \mathsf{list}$ and $\mathsf{cons} : \mathsf{nat} \times \mathsf{list} \to \mathsf{list}$, where $\mathsf{car} : \mathsf{list} \to \mathsf{nat}$ and $\mathsf{cdr} : \mathsf{list} \to \mathsf{list}$ are the selectors to cons. Moreover, "$*$" abbreviates a multiplication algorithm like times or multiply.

```
procedure prod ($l$ : list, $z$ : nat) $\Leftarrow$
    $z := \mathsf{s}(0)$;
    while $l \neq \mathsf{nil}$  do   if $\mathsf{car}(l) \neq 0$ then $z := \mathsf{car}(l) * z$;
                                  $l := \mathsf{cdr}(l)$                          od
```

This procedure can be translated automatically into the following functions (here, we re-ordered the cases of pr to ease readability).

```
function prod ($l$ : list) : nat $\Leftarrow$      function pr ($l$ : list, $z$ : nat) : nat $\Leftarrow$
  $\mathsf{pr}(l, \mathsf{s}(0))$                          if $l = \mathsf{nil}$      then $z$
                                             if $\mathsf{car}(l) \neq 0$ then $\mathsf{pr}(\mathsf{cdr}(l), \mathsf{car}(l) * z)$
                                                          else $\mathsf{pr}(\mathsf{cdr}(l), z)$
```

To transform the algorithm pr, we indeed need a technique which can handle algorithms with several recursive cases. Since $*$ is left-commutative, context moving and replacing z with $\mathsf{s}(0)$ results in

```
function prod ($l$ : list) : nat $\Leftarrow$      function pr ($l$ : list) : nat $\Leftarrow$
  $\mathsf{pr}(l)$                                 if $l = \mathsf{nil}$      then $\mathsf{s}(0)$
                                             if $\mathsf{car}(l) \neq 0$ then $\mathsf{car}(l) * \mathsf{pr}(\mathsf{cdr}(l))$
                                                          else $\mathsf{pr}(\mathsf{cdr}(l))$.
```

Further algorithms with several recursive and non-recursive cases where context moving is required are presented in [Gie99b].

Context moving is also related to a technique in [Moo75]. However, in contrast to our rule, his transformation is not equivalence-preserving, but it corresponds to a *generalization* of the conjecture. For that reason this approach faces the danger of over-generalization (e.g., the associativity law for multiply is generalized into a flawed conjecture). It turns out that for almost all algorithms considered in [Moo75] our transformation techniques can generate *equivalent* algorithms that are easy to verify. So for such examples, generalizations are no longer needed.

4 Context Splitting

Because of the required left-commutativity, context moving is not always applicable. As an example regard the following imperative procedure for uniting lists. We use a data type llist for lists of list's. Its constructors are empty and add with the selectors hd and tl. So $\mathsf{add}(z, k)$ represents the insertion of the list z in front of the list of lists k and $\mathsf{hd}(\mathsf{add}(z, k))$ yields z. Moreover, we use an algorithm app for list-concatenation. Then after execution of $\mathsf{union}(k, z)$, the value of z is the union of all lists in k.

```
procedure union($k$ : llist, $z$ : list) $\Leftarrow$
    $z$ := nil;
    while $k \neq$ empty do $z$ := app(hd($k$), $z$);
                            $k$ := tl($k$)          od
```

Translation of union into functional algorithms yields

```
function union ($k$ : llist) : list $\Leftarrow$
   uni($k$, nil)

function uni ($k$ : llist, $z$ : list) : list $\Leftarrow$
   if $k \neq$ empty then uni(tl($k$), app(hd($k$), $z$))
                     else $z$.
```

These functions are again unsuited for verification, because the accumulator z of uni is initially called with nil, but this value is changed in the recursive calls. Context moving is not possible, because the context $\mathsf{app}(\mathsf{hd}(k), \ldots)$ is not left-commutative. This motivates the development of the following *context splitting* transformation. Given a list of lists $k = [z_1, \ldots, z_n]$, the result of $\mathsf{uni}(k, \mathsf{nil})$ is

$$\mathsf{app}(z_n, \mathsf{app}(z_{n-1}, \ldots \mathsf{app}(z_3, \mathsf{app}(z_2, z_1)) \ldots)). \tag{11}$$

In order to move the context of uni's recursive accumulator argument to the top, our aim is to compute this result in a way such that z_1 is moved as far to the "outside" in this term as possible (whereas z_n may be moved to the "inside"). Although app is not commutative, it is at least *associative*. So (11) is equal to

$$\mathsf{app}(\mathsf{app}(\ldots \mathsf{app}(\mathsf{app}(z_n, z_{n-1}), z_{n-2}) \ldots, z_2), z_1). \tag{12}$$

This gives an idea on how the algorithm uni may be transformed into a new (unary) algorithm uni′ such that $\mathsf{uni}'(k)$ computes $\mathsf{uni}(k, \mathsf{nil})$. The result of

$\mathsf{uni}'([z_1,\ldots,z_n])$ should be $\mathsf{app}(\mathsf{uni}'([z_2,\ldots,z_n]),z_1)$. Similarly, $\mathsf{uni}'([z_2,\ldots,z_n])$ should yield $\mathsf{app}(\mathsf{uni}'([z_3,\ldots,z_n]),z_2)$, etc. Finally, $\mathsf{uni}'([z_n])$ is $\mathsf{app}(\mathsf{uni}'(\mathsf{empty}), z_n)$. To obtain the result (12), $\mathsf{app}(\mathsf{uni}'(\mathsf{empty}),z_n)$ must be equal to z_n. Hence, $\mathsf{uni}'(\mathsf{empty})$ should yield app's neutral argument nil. Thus, we obtain the following new algorithms, which are well suited for verification tasks.

```
function union (k : llist) : list ⇐
  uni'(k)

function uni' (k : llist) : list ⇐
  if k ≠ empty then app(uni'(tl(k)), hd(k))
               else nil
```

So the idea is to *split up* the former context $\mathsf{app}(\mathsf{hd}(k),\ldots)$ into an *outer* part $\mathsf{app}(\ldots,\ldots)$ and an *inner* part $\mathsf{hd}(k)$. If the outer context is associative, then one can transform tail recursive results of the form $f(\ldots,\mathsf{app}(\mathsf{hd}(k),z))$ into results of the form $\mathsf{app}(f'(\ldots),\mathsf{hd}(k))$. In general, our context splitting rule generates a new algorithm (14) from an algorithm of the form (13).

```
function f (x* : τ*, z : τ) : τ ⇐
  if b_1      then f(r_1*, p[r_1, z])
                   ⋮
  if b_k      then f(r_k*, p[r_k, z])        (13)
  if b_{k+1}  then p[r_{k+1}, z]
                   ⋮
  if b_m      then p[r_m, z]

function f' (x* : τ*) : τ ⇐
  if b_1      then p[f'(r_1*), r_1]
                   ⋮
  if b_k      then p[f'(r_k*), r_k]          (14)
  if b_{k+1}  then r_{k+1}
                   ⋮
  if b_m      then r_m.
```

Here, p is a term of type τ containing exactly the two new variables x_1 and x_2 of type τ and $p[t_1,t_2]$ abbreviates $p[x_1/t_1,x_2/t_2]$. Then our transformation splits the contexts into their common top part p and their specific part r_i and it moves the common part p to the top of recursive results. (This allows an elimination of the accumulator z.) If there are several possible choices for p, then we use the heuristic to choose p as small and r_i as big as possible. Let e be a constructor ground term which is a neutral argument of p, i.e., $F \models_{\text{ind}} p[x,e] \equiv x$ and $F \models_{\text{ind}} p[e,x] \equiv x$. Then in (13), one may also have z instead of $p[e,z]$. For example, in uni we had the non-recursive result z instead of $\mathsf{app}(\mathsf{nil},z)$. Moreover we demand $m > k \geq 1$, but the order of the f-cases is again irrelevant and the rule may also be applied if z is not the *last* parameter of f.

We want to ensure that all occurrences of $f(t^*,e)$ in other algorithms g (that f is not dependent on) may be replaced by $f'(t^*)$. For the soundness of this transformation, similar to context moving, the accumulator z must not occur in conditions or in the subterms $r_1^*,\ldots,r_k^*$ or $r_1,\ldots,r_m$. Then for constructor ground terms u^*, the evaluation of $f(u^*,e)$ and of $f'(u^*)$ leads to the same cases $i_1,\ldots,i_d$ where $i_1,\ldots,i_{d-1} \in \{1,\ldots,k\}$ and $i_d \in \{k+1,\ldots,m\}$. For $1 \leq h \leq d$ let a_h be $r_{i_h}[r^*_{i_{h-1}}[\ldots[r^*_{i_1}[u^*]]\ldots]]$. Then the result of $f(u^*,e)$ is (15) and the result of $f'(u^*)$ is (16).

$$p[a_d, p[a_{d-1}, \ldots p[a_2, a_1]\ldots]] \tag{15}$$

$$p[p[\ldots p[p[a_d, a_{d-1}], a_{d-2}]\ldots a_2], a_1] \tag{16}$$

To ensure the equality of these two results, p must be associative. The following theorem summarizes our rule for context splitting.

Theorem 2 (Soundness of Context Splitting). *Let F be a functional program containing (13) and let F' result from F by adding the algorithm (14). Then for all data objects t^* and q, $f(t^*,e)$ evaluates to q in F iff $f'(t^*)$ evaluates to q in F', provided that the following requirements are fulfilled:*
(A) $z \notin \mathcal{V}(b_1) \cup \ldots \cup \mathcal{V}(b_m)$
(B) $z \notin \mathcal{V}(r_1^*) \cup \ldots \cup \mathcal{V}(r_k^*) \cup \mathcal{V}(r_1) \cup \ldots \cup \mathcal{V}(r_m)$
(C) $F \models_{\mathrm{ind}} p[p[x_1,x_2],x_3] \equiv p[x_1,p[x_2,x_3]]$
(D) $F \models_{\mathrm{ind}} p[x,e] \equiv x$ *and* $F \models_{\mathrm{ind}} p[e,x] \equiv x$.

Proof. Note that evaluation of f is the same in F and F'. Moreover, Conditions (C) and (D) also hold for F'. We prove the (stronger) conjecture

$$f(t^*,t) \equiv_{F'} q \;\text{ iff }\; p[f'(t^*),t] \equiv_{F'} q \tag{17}$$

for all constructor ground terms t^*, t, and q.

For the "only if"-direction of (17) we use induction on the length of $f(t^*,t)$'s evaluation. There must be a case i such that $b_i[t^*] \equiv_{F'}$ true and $b_j[t^*] \equiv_{F'}$ false for all $1 \leq j < i$. If $i \geq k+1$, then we have $f(t^*,t) \equiv_{F'} p[r_i[t^*],t] \equiv_{F'} p[f'(t^*),t]$.

If $i \leq k$, then $f(t^*,t) \equiv_{F'} f(r_i^*[t^*],p[r_i[t^*],t]) \equiv_{F'} p[f'(r_i^*[t^*]),p[r_i[t^*],t]]$ by the induction hypothesis. By (C), this is $\equiv_{F'}$-equal to $p[p[f'(r_i^*[t^*]),r_i[t^*]],t]$ which in turn is is $\equiv_{F'}$-equal to $p[f'(t^*),t]$. The "if"-direction of (17) is proved analogously (by induction w.r.t. the relation $\succ_{f'}$, where $u^* \succ_{f'} q^*$ holds for two tuples of constructor ground terms u^* and q^* iff evaluation of $f'(u^*)$ is defined and it leads to the evaluation of $f'(q^*)$). □

Context splitting is only applied if there is a term $f(t^*,e)$ in some other algorithm g that f is not dependent on. In this case, the conditions (C) and (D) are checked by an underlying induction theorem prover (where usually associativity is even easier to prove than (left-)commutativity). Conditions (A) and (B) are just syntactic. In case of success, f' is generated and the term $f(t^*,e)$ in the algorithm g is replaced by $f'(t^*)$.

Similar to context moving, a variant of the above rule if often used in the *reverse* direction (*re-bracketing*, cf. e.g. [Coo66,DB76,BD77,Wan80,BW82,PP96]). Again, instead of directly reversing the technique, we modified and generalized it, e.g., by regarding several tail recursive and non-tail recursive cases. An algorithm where this general form of our rule is needed will be presented in Sect. 5 and several others can be found in [Gie99b]. Moreover, the next section also introduces important refinements which increase the applicability of context splitting considerably and which have no counterpart in the classical re-bracketing rules.

5 Pre-Processing Transformations for Context Splitting

In examples where the context p is not yet in the right form, one can use suitable pre-processing transformations which in turn enable the application of context splitting. Regard the following imperative procedure for reversing lists.

```
procedure reverse($l, z$ : list) $\Leftarrow$
    $z$ := nil;
    while $l \neq$ nil do $z$ := cons(car($l$), $z$);
                         $l$ := cdr($l$)          od
```

By translating reverse into functional form one obtains

```
function reverse($l$ : list) : list $\Leftarrow$
  rev($l$, nil)

function rev($l, z$ : list) : list $\Leftarrow$
  if $l \neq$ nil then rev(cdr($l$), cons(car($l$), $z$))
                  else $z$.
```

In order to eliminate the accumulator z, we would like to apply context splitting. Here, the term p in (13) would be $\mathsf{cons}(x_1, x_2)$. But then x_1 would be a variable of type nat (instead of list as required) and hence, the associativity law is not even well typed.

Whenever p has the form $c(x_1, \ldots, x_1, x_2)$ for some constructor c, where x_1 is of the "wrong" type, then one may use the following reformulation of the algorithm. (Of course, here x_2 does not have to be the *last* argument of c.) The idea is to "lift" $x_1, \ldots, x_1$ to an object $\mathsf{lift}_c(x_1, \ldots, x_1)$ of type τ and to define a new function $c' : \tau \times \tau \to \tau$ such that $c'(\mathsf{lift}_c(x_1, \ldots, x_1), x_2) \equiv c(x_1, \ldots, x_1, x_2)$. Moreover, in order to split contexts afterwards, c' should be associative.

As a heuristic, we use the following construction for lift_c and c', provided that apart from c the data type τ just has a constant constructor c_{con}. The function $\mathsf{lift}_c(x_1, \ldots, x_n)$ should yield the term $c(x_1, \ldots, x_n, c_{con})$ and the function c' is defined by the following algorithm (where $d_1, \ldots, d_{n+1}$ are the selectors to c).

```
function $c'(x, z : \tau) : \tau$ $\Leftarrow$
  if $x = c(d_1(x), \ldots, d_n(x), d_{n+1}(x))$ then $c(d_1(x), \ldots, d_n(x), c'(d_{n+1}(x), z))$
                                                  else $z$
```

Then $c'(\mathsf{lift}_c(x_1, \ldots, x_n), z) \equiv c(x_1, \ldots, x_n, z)$, c_{con} is a neutral argument for c', and c' is associative. So for rev, we obtain $\mathsf{lift}_{\mathsf{cons}}(x_1) \equiv \mathsf{cons}(x_1, \mathsf{nil})$ and

```
function cons'($x, z$ : list) : list $\Leftarrow$
  if $x$ = cons(car($x$), cdr($x$)) then cons(car($x$), cons'(cdr($x$), $z$))
                                  else $z$.
```

Note that in this example, cons′ corresponds to the concatenation function app.

Thus, the term $\mathsf{cons}(\mathsf{car}(l), z)$ in the algorithm rev may be replaced by $\mathsf{cons}'(\mathsf{lift}_{\mathsf{cons}}(\mathsf{car}(l)), z)$, i.e., by $\mathsf{cons}'(\mathsf{cons}(\mathsf{car}(l), \mathsf{nil}), z)$. Now the rule for context splitting is applicable which yields

```
function reverse($l$ : list) : list $\Leftarrow$
  rev'($l$)

function rev'($l$ : list) : list $\Leftarrow$
  if $l \neq$ nil then cons'(rev'(cdr($l$)), cons(car($l$), nil))
              else nil.
```

In contrast to the original versions of reverse and rev, these algorithms are well suited for verification.

Of course, there are also examples where the context p has the form $g(x_1, x_2)$ for some *algorithm* g (instead of a constructor c) and where x_1 has the "wrong" type. For instance, regard the following imperative procedure to filter all even elements out of a list l. It uses an auxiliary algorithm even and an algorithm $\mathsf{atend}(x, z)$ which inserts an element x at the end of a list z.

```
function  atend($x$ : nat, $z$ : list) : list $\Leftarrow$
  if $z$ = nil then cons($x$, nil)
               else  cons(car($z$), atend($x$, cdr($z$)))
```

Now the procedure filter reads as follows.

```
procedure filter($l, z$ : list) $\Leftarrow$
    $z$ := nil;
    while $l \neq$ nil   do   if even(car($l$)) then $z$ := atend(car($l$), $z$);
                              $l$ := cdr($l$)                                  od
```

Translating this procedure into functional algorithms yields

```
function  filter($l$ : list) : list $\Leftarrow$    function  fil($l, z$ : list) : list $\Leftarrow$
   fil($l$, nil)                          if $l$ = nil          then $z$
                                          if even(car($l$)) then fil(cdr($l$), atend(car($l$), $z$))
                                                            else  fil(cdr($l$), $z$).
```

To apply context splitting for fil, p would be $\mathsf{atend}(x_1, x_2)$ and thus, x_1 would be of type nat instead of list as required. While for constructors like cons, such a problem can be solved by the *lifting* technique described above, now the root of p is the algorithm atend. For such examples, the following *parameter enlargement* transformation often helps.

In the algorithm atend, outside of its own recursive argument the parameter x only occurs in the term $\mathsf{cons}(x, \mathsf{nil})$ and the value of $\mathsf{cons}(x, \mathsf{nil})$ does not change throughout the whole execution of atend (as the value of x does not change in any recursive call). Hence, the parameter x can be "enlarged" into a new parameter y which corresponds to the value of $\mathsf{cons}(x, \mathsf{nil})$. Thus, we result in the following algorithm atend′, where $\mathsf{atend}'(\mathsf{cons}(x, \mathsf{nil}), z) \equiv \mathsf{atend}(x, z)$.

```
function  atend′($y, z$ : list) : list $\Leftarrow$
  if $z$ = nil then $y$
               else  cons(car($z$), atend′($y$, cdr($z$)))
```

In general, let $h(x^*, z^*)$ be a function where the parameters x^* are not changed in recursive calls and where x^* only occur within the terms $t_1, \ldots, t_m$ outside of their recursive calls in the algorithm h. If $\mathcal{V}(t_i) \subseteq \{x^*\}$ for all i and if the t_i only contain total functions (like constructors), then one may construct a new algorithm $h'(y_1, \ldots, y_m, z^*)$ by enlarging the parameters x^* into $y_1, \ldots, y_m$. The algorithm h' results from h by replacing all t_i by y_i, where the parameters y_i again remain unchanged in their recursive arguments. Then we have $h'(t_1, \ldots, t_m, z^*) \equiv h(x^*, z^*)$. Thus, in all algorithms f that h is not dependent on, we may replace any subterm $h(s^*, p^*)$ by $h'(t_1[x^*/s^*], \ldots, t_m[x^*/s^*], p^*)$. (The only restriction for this replacement is that all possibly undefined subterms of s^* must still occur in some $t_i[x^*/s^*]$.)

Hence, in the algorithm fil, the term $\mathsf{atend}(\mathsf{car}(l), z)$ can be replaced by $\mathsf{atend}'(\mathsf{cons}(\mathsf{car}(l), \mathsf{nil}), z)$. It turns out that $\mathsf{atend}'(l_1, l_2)$ concatenates the lists l_2 and l_1 (i.e., it corresponds to $\mathsf{app}(l_2, l_1)$). Therefore, atend′ is associative and thus, context splitting can be applied to fil now. This yields the following algorithms which are well suited for verification.

```
function filter(l : list) : list ⇐   function fil′(l : list) : list ⇐
  fil′(l)                              if l = nil           then nil
                                       if even(car(l))  then atend′(fil′(cdr(l)), cons(car(l), nil))
                                                            else  atend′(fil′(cdr(l)), nil)
```

Of course, by subsequent unfolding (or "symbolic evaluation") of atend′, the algorithm fil′ can be simplified to

```
function  fil′(l : list) : list ⇐
  if l = nil           then nil
  if even(car(l))  then cons(car(l), fil′(cdr(l)))
                       else  fil′(cdr(l)).
```

Note that here we indeed needed a context splitting rule which can handle algorithms with *several* tail recursive cases. Thus, a direct reversal of the classical re-bracketing rules [BW82] would fail for both reverse and filter (since these rules are restricted to just one recursive case and moreover, they lack the concepts of lifting and of parameter enlargement).

The examples union, reverse, and filter show that context splitting can help in cases where context moving is not applicable. On the other hand for algorithms like plus, context moving is successful, but context splitting is not possible. So none of these two transformations subsumes the other and to obtain a powerful approach, we indeed need *both* of them. But there are also several algorithms where the verification problems can be solved by both context moving and splitting. For example, the algorithms resulting from mult by context moving or splitting only differ in the order of the arguments of $+$ in mult's first recursive case. Thus, both resulting algorithms are well suited for verification tasks.

6 Conclusion

We have presented a new transformational approach for the mechanized verification of imperative programs and tail recursive functions, which consists of the following transformations:

- *context moving* for left-commutative contexts of accumulators (Sect. 3)
- *context splitting* for (partly) associative contexts of accumulators (Sect. 4)
- *lifting* of arguments in order to enable context splitting (Sect. 5)
- *parameter enlargement* to enable context splitting (Sect. 5)

By our technique, functions that are hard to verify are automatically transformed into functions where verification is significantly easier. Hence, for many programs the invention of loop invariants or of generalizations is no longer required and an automated verification is possible by the existing induction theorem provers. As our transformations generate *equivalent* functions, this transformational verification approach is not restricted to partial correctness, but it can also be used to simplify total correctness and termination proofs [Gie95,Gie97,GWB98,BG99,AG00]. See [Gie99b] for a collection of examples that demonstrates the power of our approach. It shows that our transformation indeed simplifies the verification tasks substantially for many practically relevant algorithms from different areas of computer science (e.g., arithmetical algorithms or procedures for processing (possibly multidimensional) lists including algorithms for matrix multiplication and sorting algorithms like selection-, insertion-, and merge-sort, etc.). Based on the rules and heuristics presented, we implemented a system to perform such transformations automatically [Gie99a].

The field of mechanized verification and induction theorem proving represents a new application area for program transformation techniques. It turns out that our approach of transforming algorithms often seems to be superior to the classical solution of generalizing theorems. For instance, our technique automatically transforms all (first order) tail recursive functions treated in recent generalization techniques [IS97,IB99] into non-tail recursive ones whose verification is very simple. On the other hand, the techniques for finding generalizations are mostly semi-automatic (since they are guided by the system user who has to provide suitable lemmata). Obviously, by formulating the right lemmata (interactively), in principle generalization techniques can deal with almost every conjecture to be proved. But in particular for conjectures which involve *several* occurrences of a tail recursive function, finding suitable generalizations is often impossible for fully automatic techniques. Therefore, our approach represents a significant contribution for mechanized verification of imperative and tail recursive functional programs. Nevertheless, of course there also exist tail recursive algorithms where our automatic transformations are not applicable. For such examples, (interactive) generalizations are still required.

Further work will include an examination of other existing program transformation techniques in order to determine whether they can be modified into transformations suitable for an application in the program verification domain. Moreover, in future work the application area of program verification may also give rise to new transformations which have no counterpart at all in classical program transformations.

References

[AK82] J. Arsac and Y. Kodratoff. Some techniques for recursion removal from recursive functions. *ACM Trans. Prog. Languages Systems*, 4:295–322, 1982.

[AG00] T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 2000. To appear.

[Aub79] R. Aubin. Mechanizing structural induction. *TCS*, 9:347–362, 1979.

[BW82] F. L. Bauer and H. Wössner. *Algorithmic Language and Program Development.* Springer, 1982.

[BR95] A. Bouhoula and M. Rusinowitch. Implicit induction in conditional theories. *Journal of Automated Reasoning*, 14:189–235, 1995.

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

[BM98] R. S. Boyer and J S. Moore. *A Computational Logic Handbook.* Academic Press, 2nd edition, 1998.

[BG99] J. Brauburger and J. Giesl. Approximating the domains of functional and imperative programs. *Science of Computer Programming*, 35:113-136, 1999.

[BSH$^+$93] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. *Artif. Int.*, 62:185–253, 1993.

[BD77] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24:44–67, 1977.

[Coo66] D. Cooper. The equivalence of certain computations. *Comp. J.*, 9:45–52, 66.

[DB76] J. Darlington and R. M. Burstall. A system which automatically improves programs. *Acta Informatica*, 6:41–60, 1976.

[Dij85] E. W. Dijkstra. Invariance and non-determinacy. In *Mathematical Logic and Programming Languages*, chapter 9, pages 157–165. Prentice-Hall, 1985.

[Gie95] J. Giesl. Termination analysis for functional programs using term orderings. In *Proc. SAS' 95*, LNCS 983, pages 154–171, Glasgow, UK, 1995.

[Gie97] J. Giesl. Termination of nested and mutually recursive algorithms. *Journal of Automated Reasoning*, 19:1–29, 1997.

[GWB98] J. Giesl, C. Walther, and J. Brauburger. Termination analysis for functional programs. In Bibel and Schmitt, eds., *Automated Deduction – A Basis for Applications, Vol. III*, Applied Logic Series 10, pages 135–164. Kluwer, 1998.

[Gie99a] J. Giesl. Mechanized verification of imperative and functional programs. Habilitation Thesis, TU Darmstadt, 1999.

[Gie99b] J. Giesl. Context-moving transformations for function verification. Technical Report IBN 99/51, TU Darmstadt. Available from http://www.inferenzsysteme.informatik.tu-darmstadt.de/~giesl/ibn-99-51.ps

[Gie99c] J. Giesl. Induction proofs with partial functions. *Journal of Automated Reasoning.* To appear. Preliminary version appeared as Technical Report IBN 98/48, TU Darmstadt. Available from http://www.inferenzsysteme.informatik.tu-darmstadt.de/~giesl/ibn-98-48.ps

[HK92] P. Harrison and H. Khoshnevisan. A new approach to recursion removal. *Theoretical Computer Science*, 93:91–113, 1992.

[HBS92] J. Hesketh, A. Bundy, and A. Smaill. Using middle-out reasoning to control the synthesis of tail-recursive programs. In *Proc. CADE-11*, LNAI 607, pages 310–324, Saratoga Springs, NY, 1992.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–583, 1969.

[IS97] A. Ireland and J. Stark. On the automatic discovery of loop invariants. *4th NASA Langley Formal Methods Workshop*, NASA Conf. Publ. 3356, 1997.

[IB99] A. Ireland and A. Bundy. Automatic verification of functions with accumulating parameters. *Journal of Functional Programming*, 9:225-245, 1999.

[KM87] D. Kapur and D. R. Musser. Proof by consistency. *AI*, 31:125–158, 1987.

[McC60] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3, 1960.

[MPS93] B. Möller, H. Partsch, and S. Schuman. *Formal Program Development.* LNCS 755, Springer, 1993.

[Moo75] J S. Moore. Introducing iteration into the Pure LISP theorem prover. *IEEE Transactions on Software Engineering*, 1:328–338, 1975.

[Par90] H. Partsch. *Specification and Transformation of Programs.* Springer, 1990.

[PP96] A. Pettorossi and M. Proietti. Rules and strategies for transforming functional and logic programs. *ACM Computing Surveys*, 28:360–414, 1996.

[PP98] A. Pettorossi and M. Proietti. Transformations of logic programs. *Handbook of Logic in AI and Logic Programming, Vol. 5*, Oxford University Pr., 1998.

[RY76] C. Reynolds and R. T. Yeh. Induction as the Basis for Program Verification. *IEEE Transactions on Software Engineering*, SE-2(4):244–252, 1976.

[SI98] J. Stark and A. Ireland. Invariant discovery via failed proof attempts. In *Proc. LOPSTR '98*, LNCS 1559, Manchester, UK, 1998.

[Wal94] C. Walther. Mathematical induction. *Handbook of Logic in Artificial Intelligence and Logic Programming, Vol. 2.* Oxford University Press, 1994.

[Wan80] M. Wand. Continuation-based program transformation strategies. *Journal of the ACM*, 27:164–180, 1980.

[ZKK88] H. Zhang, D. Kapur, and M. S. Krishnamoorthy. A mechanizable induction principle for equational specifications. *CADE-9*, LNCS 310, Argonne, 1988.