Deaccumulation Techniques for Improving Provability *

Jürgen Giesl a,∗,1 Armin Kühnemann b Janis Voigtländer b,2

aLuFG Informatik 2, RWTH Aachen, Ahornstr. 55, 52074 Aachen, Germany
bInstitut für Theoretische Informatik, TU Dresden, 01062 Dresden, Germany

Abstract

Several induction theorem provers were developed to verify functional programs mechanically. Unfortunately, automatic verification often fails for functions with accumulating arguments. Using concepts from the theory of tree transducers and extending on earlier work, the paper develops automatic transformations from accumulative functional programs into non-accumulative ones, which are much better suited for mechanized verification. The overall goal is to reduce the need for generalizing induction hypotheses in (semi-)automatic provers. Via the correspondence between imperative programs and tail-recursive functions, the presented approach can also help to reduce the need for inventing loop invariants in the verification of imperative programs.

Key words: tree transducers, induction theorem proving, tail recursion, program transformation, program verification

1 Introduction

Automatic transformation of programs is a key technology in software engineering, as it enables programmers to work at a higher level of abstraction

* This work extends the research reported by the same authors in [24].
* Corresponding author. Phone: +49 241 8021230, Fax: +49 241 8022217

Email addresses: giesl@informatik.rwth-aachen.de (Jürgen Giesl),
kuehne@tcs.inf.tu-dresden.de (Armin Kühnemann),
voigt@tcs.inf.tu-dresden.de (Janis Voigtländer).
1 The research of this author was supported by the “Deutsche Forschungsgemeinschaft” under grant GI 274/5-1.
2 The research of this author was supported by the “Deutsche Forschungsgemeinschaft” under grants KU 1290/2-1 and 2-4.

Preprint submitted to Elsevier Science 3 November 2006
than would otherwise be possible and thus raises their productivity. Another important trend, in particular for safety-critical applications, is formal verification of programs. This paper combines these two paradigms, employing an automatic program transformation to improve the amenability of programs to automatic verification. So while most classical program transformations aim at improving the efficiency, our goal is to develop transformations which improve the provability. This goal is detailed in the following.

To automate correctness proofs about programs as much as possible, several powerful induction theorem provers have been developed, which can be used for mechanized reasoning about program properties (e.g., NQTHM [8], ACL-2 [33], RRL [32], CLAM [10,9], INKA [2,61], SPIKE [7]). While their most successful application area is that of functional programming, such provers can in principle also be used for the verification of imperative programs. To this end, imperative programs are translated into the functional input language of induction provers. Unfortunately, this leads to a certain form of programs that poses severe problems for the existing provers.

As an example, we consider the calculation of a decreasing list containing the first $x_1$ even numbers (i.e., $[2 \cdot x_1 - 2, \ldots, 4, 2, 0]$). This problem can be solved by the following part $p_{even}$ of an imperative program (in C-like syntax [34]):

```c
[int] even (int x1)
{
    int y1 = 0; [int] y2 = [];
    while (x1!=0) { y2 = y1:y2; y1 = y1+2; x1--; };
    return y2;
}
```

Here, [int] denotes the type of integer lists, [] denotes the empty list, and : denotes list insertion, i.e., $y_1 : y_2$ inserts the element $y_1$ in front of the list $y_2$. In the absence of pointers, as above, imperative programs can easily be translated into functional ones by transforming every while-loop into a separate function whose parameters record the changes during a run through the while-loop [42]. For our program $p_{even}$ we obtain the following tail-recursive program $p_{acc}$ (in Haskell-like syntax [47]) together with an initial call $r_{acc} = (lev \ x_1 \ 0 \ [])$. Here, “lev” stands for “list of even numbers”. The program $p_{acc}$ represents natural numbers with the constructors 0 and $S$ for the successor function, and uses pattern matching on $lev$’s first argument, called recursion argument:

```haskell
p_{acc} :  lev (S \ x_1) \ y_1 \ y_2 = lev x_1 (S (S \ y_1)) \ (y_1 : y_2)
lev 0 \ y_1 \ y_2 = y_2
```

The described translation of imperative into functional programs always yields tail-recursive functions that compute their result using accumulators. For instance, the decreasing list of the first three even numbers is computed by $p_{acc}$
as follows (where \( \Rightarrow_{\text{acc}} \) denotes the reduction relation w.r.t. \( p_{\text{acc}} \)):

\[
lev (S^3 0) 0 [] \Rightarrow_{\text{acc}} \lev (S^2 0) (S^2 0) (0 : [])
\]

\[
\Rightarrow_{\text{acc}} \lev (S 0) (S^4 0) ((S^2 0) : (0 : []))
\]

\[
\Rightarrow_{\text{acc}} \lev 0 (S^6 0) ((S^4 0) : ((S^2 0) : (0 : [])))
\]

\[
\Rightarrow_{\text{non}} (S^4 0) : ((S^2 0) : (0 : []))
\]

As one can see, \( lev \) accumulates values in its context arguments (arguments different from the recursion argument, i.e., \( lev \)'s second and third argument). A function is called *accumulative* if at least one of its context arguments is modified in a recursive call. For instance, \( lev \) is accumulative because both the second and the third argument do not remain unchanged in the recursive call. A program like \( p_{\text{acc}} \), containing an accumulative function, is itself called *accumulative*.

Now assume that our aim is to verify the equivalence of \( r_{\text{acc}} \) and \( r_{\text{spec}} = (\lev_2 x_1) \) for all natural numbers \( x_1 \), where \( p_{\text{spec}} \) is the following specification of our problem. Here, \( (\lev_2 x_1) \) calculates the desired list and \( (\text{doub } x_1) \) computes \( 2 \cdot x_1 \):

\[
\begin{align*}
\lev_2 (S x_1) &= (\text{doub } x_1) : (\lev_2 x_1) & \text{doub } (S x_1) &= S (S (\text{doub } x_1)) \\
\lev_2 0 &= [] & \text{doub } 0 &= 0
\end{align*}
\]

Note that even if there exists such a “natural” non-accumulative recursive specification of a problem, imperative programs are typically written using loops, which translate into accumulative programs like \( p_{\text{acc}} \) above. The accumulative version may also be more efficient than a non-accumulative implementation (see, e.g., Appendix B).

But unfortunately, accumulative programs pose serious problems for mechanized verification. For example, an automatic proof of

\[
lev x_1 0 [] = \lev_2 x_1
\]

by induction (using this equation for fixed \( x_1 \) as induction hypothesis) fails because in the induction step \( (x_1 \mapsto (S x_1)) \) the induction hypothesis cannot be successfully applied to prove the equality of \( (\lev (S x_1) 0 []) \) and \( (\lev_2 (S x_1)) \). The reason is that \( lev \) uses accumulators: the context arguments of the term \( (\lev x_1 (S (S 0)) (0 : [])) \), which originates from rule application to \( (\lev (S x_1) 0 []) \), do not fit to the context arguments of the term \( (\lev_2 0 []) \) in the induction hypothesis! So the problem is that accumulating arguments are typically initialized with some fixed values (like 0 and []), which then appear also in the conjecture to be proved and hence in the induction hypothesis. But since accumulators are changed in recursive calls, after rule application we have different values (like \( (S (S 0)) \) and \( (0 : []) \)) in the statement to be proved in the induction step.
In induction theorem proving this problem is usually solved by transforming the conjecture to be proved. More precisely, the aim is to invent a suitable generalization (see, e.g., [1,8,9,29,30,33,61]). So, as a replacement for the original conjecture, one tries to find a stronger conjecture that however is easier to prove. In our example, the original conjecture may be generalized to

\[ \text{lev} \ x_1 \ y_1 \ y_2 = (\text{lev}'_2 \ x_1 \ y_1) + + y_2, \]

where + + denotes list concatenation and where \( \text{lev}'_2 \) and \( \text{doub}' \) are defined as follows:

\[ \begin{align*}
\text{lev}'_2 \ (S \ x_1) & \quad y_1 = (\text{doub}' \ x_1 \ y_1) : (\text{lev}'_2 \ x_1 \ y_1) \\
\text{lev}'_2 \ 0 & \quad y_1 = [] \\
\text{doub}' \ (S \ x_1) & \quad y_1 = S \ (S \ (\text{doub}' \ x_1 \ y_1)) \\
\text{doub}' \ 0 & \quad y_1 = y_1
\end{align*} \]

However, finding successful generalizations automatically is often very hard. The ACL-2 prover [33], for instance, performs a series of generalizations for the above original conjecture that do not increase verifiability, and it ends up with consuming all memory available. This corresponds to the problem of inventing suitable loop invariants in classical approaches to direct verification of imperative programs [26]. While there are heuristics for discovering good loop invariants [12,30,39,48,51], in general this task is hard to mechanize [14]. Since discovering good generalizations of conjectures is equally difficult, the development of techniques to verify accumulative functions is one of the most important research topics in the area of inductive theorem proving [29].

In contrast to the classical approach of generalizing conjectures, we suggest an automatic, semantics-preserving program transformation. It transforms functions for which conjectures are hard to verify into functions that are much more suitable for mechanized verification. The advantage of this approach is that by transforming a function definition the verification problems with this function are typically solved once and for all (i.e., for all conjectures one would like to prove about this function). This is unlike the situation when using the generalization approach, where one has to find a new generalization for every new conjecture to be proved. In particular, finding generalizations is very difficult for conjectures with several occurrences of an accumulative function (see, e.g., [23] and Appendices A and B).

The transformation to be presented in this paper transforms the original pro-
gram \( p_{\text{acc}} \) with initial call \( r_{\text{acc}} \) into the following equivalent program \( p_{\text{non}} \):

\[
p_{\text{non}} : \quad \text{lev}'(S \ x_1) = \text{sub}(\text{lev}' \ x_1) (S (S \ 0)) (0 : [])
\]

\[
\text{lev}' \ 0 = []
\]

\[
\text{sub}(x_1 : x_2) \ y_1 \ y_2 = (\text{sub} \ x_1 \ y_1 \ y_2) : (\text{sub} \ x_2 \ y_1 \ y_2) \quad \text{sub} \ 0 \ y_1 \ y_2 = y_1
\]

\[
\text{sub}(S \ x_1) \ y_1 \ y_2 = S (\text{sub} \ x_1 \ y_1 \ y_2) \quad \text{sub} \ [] \ y_1 \ y_2 = y_2
\]

with initial call \( r_{\text{non}} = (\text{lev}' \ x_1) \). Since \( p_{\text{non}} \) contains a function \( \text{lev}' \) without context arguments and a function \( \text{sub} \) with unchanged context arguments in recursive calls, \( p_{\text{non}} \) is a non-accumulative program and our transformation technique is called deaccumulation. An application \((\text{sub} \ t \ s_1 \ s_2)\) of the substitution function \( \text{sub} \) replaces all occurrences of 0 and \([\underline{\underline{.}}]\) in \( t \) by \( s_1 \) and \( s_2 \), respectively. For instance, the decreasing list of the first three even numbers is computed by \( p_{\text{non}} \) as follows (where the superscript of \( \Rightarrow_{p_{\text{non}}} \) indicates the number of reduction steps):

\[
\text{lev}'(S^3 \ 0)
\]

\[
\Rightarrow^4_{p_{\text{non}}} \ \text{sub}(\text{sub}(\text{sub} [] (S^2 \ 0) (0 : []))) (S^2 \ 0) (0 : [])) (S^2 \ 0) (0 : [])
\]

\[
\Rightarrow^3_{p_{\text{non}}} \ \text{sub}(\text{sub} (0 : [])) (S^2 \ 0) (0 : [])) (S^2 \ 0) (0 : [])
\]

\[
\Rightarrow^3_{p_{\text{non}}} \ \text{sub}((S^2 \ 0) : (0 : [])) (S^2 \ 0) (0 : [])
\]

\[
\Rightarrow^7_{p_{\text{non}}} (S^4 \ 0) : ((S^2 \ 0) : (0 : []))
\]

This computation shows that the constructors 0 and \([\underline{\underline{.}}]\) in \( p_{\text{non}} \) are used as “placeholders”, which are repeatedly substituted by \((S^2 \ 0)\) and \((0 : []))\), respectively. Our transformation is meant to be applied as an explicit pre-processing step preceding actual verification attempts.

Now, the statement

\[
\text{lev}' \ x_1 = \text{lev}_2 \ x_1
\]

(1)

can be proved by three nested inductions as follows. We only give the induction step \((x_1 \mapsto (S \ x_1))\), omitting the simple base case \((x_1 = 0)\). We have to prove \( \text{lev}'(S \ x_1) = \text{lev}_2(S \ x_1) \). For the left-hand side \( \text{lev}'(S \ x_1) \), exhaustive rewriting with the (directed) equations from \( p_{\text{non}} \) and application of the induction hypothesis \((IH)\) \( \text{lev}' \ x_1 = \text{lev}_2 \ x_1 \) yields

\[
\text{lev}'(S \ x_1)
\]

\[
= \text{sub}(\text{lev}' \ x_1) (S (S \ 0)) (0 : [])
\]

\[
= \text{sub}(\text{lev}_2 \ x_1) (S (S \ 0)) (0 : []).
\]

\((IH)\)

For the right-hand side \( \text{lev}_2(S \ x_1) \), we obtain

\[
\text{lev}_2(S \ x_1) = (\text{doub} \ x_1) : (\text{lev}_2 \ x_1)
\]

5
by rewriting. So to finish the proof, we have to show the conjecture

\[ \text{sub} \ (\text{lev}_2 \ x_1) \ (S \ (S \ 0)) \ (0 : []) = (\text{doub} \ x_1) : (\text{lev}_2 \ x_1). \]  

(2)

Note that there is no need to invent such subgoals manually here because they show up automatically as proof obligations during the course of the proof. For the proof of (2), we again only give the induction step \((x_1 \mapsto (S \ x_1))\). We apply the same strategy as above by exhaustively rewriting both sides of the equation and by applying the induction hypothesis afterwards. For the left-hand side, this yields

\[ \text{sub} \ (\text{lev}_2 \ (S \ x_1)) \ (S \ (S \ 0)) \ (0 : []) = \text{sub} \ ((\text{doub} \ x_1) : (\text{lev}_2 \ x_1)) \ (S \ (S \ 0)) \ (0 : [])) = (\text{doub} \ x_1) : (\text{lev}_2 \ x_1) \quad \text{(IH)} \]

and for the right-hand side, we obtain

\[(\text{doub} \ (S \ x_1)) : (\text{lev}_2 \ (S \ x_1)) = (S \ (S \ (\text{doub} \ x_1))) : ((\text{doub} \ x_1) : (\text{lev}_2 \ x_1)).\]

Since the tails of the two resulting list expressions are identical, we have to show the following conjecture to finish the proof:

\[ \text{sub} \ (\text{doub} \ x_1) \ (S \ (S \ 0)) \ (0 : []) = S \ (S \ (\text{doub} \ x_1)). \]

(3)

Conjecture (3) is again proved by induction. In the step case, the left-hand side is transformed as follows:

\[ \text{sub} \ (\text{doub} \ (S \ x_1)) \ (S \ (S \ 0)) \ (0 : [])) = S \ (S \ (\text{doub} \ x_1)) \quad \text{(IH)} \]

and rewriting the right-hand side yields

\[ S \ (S \ (\text{doub} \ (S \ x_1))) = S \ (S \ (S \ (\text{doub} \ x_1))). \]

Thus, Conjecture (3) is verified. This also proves (2) and the original conjecture (1). A similar proof can also be generated automatically by existing induction theorem provers like ACL-2, if provided with the transformed program.

In this paper we consider the definition of \(\text{lev}\) in \(p_{acc}\) as a \textit{macro tree transducer} (for short \(\text{mtt}\)) \cite{16,18,22} with one function. In general, such a function is defined by equations which perform a case analysis on the root symbol of its recursion argument \(t\). The right-hand side of such a defining equation may,
beside constructors and context arguments, only contain (extended) primitive-recursive function calls, i.e., ones in which the function being defined is called with a recursion argument that is a variable referring to a subtree of \( t \). The functions \( lev' \) and \( sub \) together are viewed as a 2-modular tree transducer (for short modtt) [19], where it is allowed that a function in module 1 (here \( lev' \)) calls a function in module 2 (here \( sub \)) non-primitive-recursively.

In [24] we have simplified a decomposition technique from [37], which itself is based on results in [16,18,19] and transforms mtts like \( lev \) into modtts like \( lev' \) and \( sub \) without accumulators. It turned out that the programs obtained right after decomposition are still not suitable for automatic verification. Since their verification problems are caused only by the form of the new initial calls, which still contain initial values, we developed another transformation step, called (basic) constructor replacement, which yields initial calls of the innocuous form \((f' \ x_1)\). An implementation of decomposition and basic constructor replacement (up to now without integration into an induction theorem prover) is described in [50].

The second transformation step, however, imposed a quite strong restriction on the original program, namely that the initial values for the context arguments of the function \( f \) to be transformed are pairwise distinct nullary constructors not occurring in the right-hand sides of defining equations for \( f \). While this restriction is fulfilled for 0 and \([\ ]\) in the example considered above, it is not hard to envisage other examples (cf. Section 3.2) where this is not the case, but where performing an automatic deaccumulation to improve the suitability for verification would still be desirable. Extending on our earlier work [24], we show how to overcome this restriction in the current paper. The idea is to allow more control to take place in module 2 of the resulting modtt than is the case for the kind of simple substitution function seen above. The need to determine the exact way in which this (finite) control is to be exercised leads to an analysis problem regarding the original program, which is solved using a fixpoint construction. Compared to the preliminary version of this paper [24], this program analysis, the advanced deaccumulation technique based on it, and the associated correctness proofs constitute the main additional contributions of the current article. We also developed an implementation of the fixpoint construction, though again it is not integrated into an induction theorem prover up to now.

Also independent of the tail-recursive embedding of imperative programs, the

---

3 The reason is that the function \( sub \) is used to replace these nullary constructors by context arguments of \( f \). The restriction on \( f \) and its initial call ensures that after the transformation, for each occurrence of a nullary constructor it is clear whether it is just a placeholder that must be replaced by a former context argument or whether it stands for the nullary constructor itself. A more detailed motivation for the restriction is given in Section 3.1.2.
accumulating style is a quite common programming idiom in functional languages (cf., e.g., Chapter 6 of [20]). Therefore, the topic of transforming accumulative functions (not necessarily into non-accumulative ones) has received much attention in recent years [13,23,27,31,43,44,52], partly drawing on concepts from the theory of tree transducers as well [35,36,56,57,59]. The present work continues this line of research, with the interesting twist that our aim is not the classical one of improving the efficiency of programs. Choosing mtts as model for functional programs with accumulating arguments opens the way to deal with a large class of typical functions on algebraic data types, which are indeed often defined by structural descent on a distinguished argument. For example, manipulation of abstract syntax trees in compilers often follows the recursion scheme of mtts [22,54], and the “tree transformation core” of XML processing languages can be compiled into compositions of mtts [17,41]. Accordingly, we will demonstrate by examples that deaccumulation can not only be useful for functions resulting from the translation of imperative programs, but also for accumulative functional programs in general.

Beside this introduction, the paper contains four further sections and four appendices. Section 2 introduces necessary notions and notations, our functional language, and tree transducers. Section 3 develops basic and advanced deaccumulation. Section 4 considers related work. Section 5 concludes with a discussion of our approach and its future implementation, and points out directions for further research. Three additional examples demonstrating the application of our results can be found in Appendices A, B, and C. Appendix D contains full proofs.

2 Preliminaries, Language, and Tree Transducers

We denote by \( \mathbb{N} \) the set of natural numbers including 0. For every \( n \in \mathbb{N} \), \([n]\) denotes the set \( \{1, \ldots, n\} \) and \([0,n]\) denotes the set \( \{0\} \cup [n] \). For every finite subset of \( \mathbb{N} \), the mapping \( \text{max} \) gives the maximum of that subset’s elements, where by convention \( \text{max}(\emptyset) = 0 \). Let \( S \) be a set. We denote by \( S^* \) the set of finite sequences of elements of \( S \). The power set of \( S \) is denoted by \( \mathcal{P}(S) \). If \( S \) is finite, then the number of its elements is denoted by \( |S| \).

A \textit{ranked alphabet} is a pair \((\Sigma, \text{rank}_\Sigma)\), where \( \Sigma \) is a finite set of symbols and \( \text{rank}_\Sigma \) assigns to each of these symbols a natural number, its \textit{rank}. In the following, we usually omit the \( \text{rank}_\Sigma \)-function and only mention \( \Sigma \) when referring to a ranked alphabet. For every \( k \in \mathbb{N} \) we define \( \Sigma^{(k)} = \{\sigma \in \Sigma \mid \text{rank}_\Sigma(\sigma) = k\} \). The rank \( k \) of a symbol \( \sigma \) is also denoted by writing \( \sigma^{(k)} \). A \textit{nullary} symbol is one of rank 0, a \textit{unary} symbol is one of rank 1, and an \textit{n-ary} symbol (with \( n \in \mathbb{N} \)) is one of rank \( n \). For the sake of brevity, a quantification over a symbol in a ranked alphabet implicitly quantifies also its rank. For example, we write “for every \( \sigma \in \Sigma^{(k)} \)” instead of “for every \( k \in \mathbb{N} \)
and for every \( \sigma \in \Sigma^{(k)} \). We use the following sets of variables, denoted by lowercase letters. Let \( X \) be the set \( \{x_1, x_2, x_3, \ldots \} \) of variables, and for every \( k \in \mathbb{N}, X_k \) is the finite set \( \{x_1, \ldots, x_k\} \subseteq X \); analogously for \( Y \). Note that \( X_0 = Y_0 = \emptyset \). For a ranked alphabet \( \Sigma \) and a set \( V \) of variables disjoint from \( \Sigma \) we define the set \( T_\Sigma(V) \) of trees (or terms) over \( \Sigma \) indexed by \( V \) as the smallest set \( T \subseteq (\Sigma \cup V \cup \{(,\})^* \) such that (i) \( V \subseteq T \) and (ii) for every \( \sigma \in \Sigma^{(k)} \) and \( t_1, \ldots, t_k \in T \): \( (\sigma \, t_1 \cdots \, t_k) \in T \). If readability allows, outer brackets of trees are omitted. For a unary symbol \( \sigma, n \in \mathbb{N} \), and \( t \in T_\Sigma(V) \), we write \( (\sigma^n \, t) \) for the tree obtained by putting \( n \) occurrences of \( \sigma \) on top of \( t \). We denote \( T_\Sigma(\emptyset) \) by \( T_\Sigma \). We define the height of a (ground) tree by \( \text{height}(\sigma \, t_1 \cdots \, t_k) = 1 + \max(\{\text{height}(t_i) \mid i \in [k]\}) \) for every \( \sigma \in \Sigma^{(k)} \) and \( t_1, \ldots, t_k \in T_\Sigma \).

Let \( n \in \mathbb{N} \), let \( \alpha_1, \ldots, \alpha_n \in \Sigma^{(0)} \cup V \) be pairwise distinct, and let \( \Sigma' \) and \( V' \) be a ranked alphabet and a set of variables, respectively, where \( (\Sigma \cup \Sigma') \cap (V \cup V') = \emptyset \). For trees \( t_1', \ldots, t_n' \in T_{\Sigma'}(V') \), the tree substitution \( \bullet [\alpha_1, \ldots, \alpha_n \leftarrow t_1', \ldots, t_n'] \) (written postfix and also written using the alternative, set comprehension-like notation \( \bullet [\alpha_i \leftarrow t_i' \mid i \in [n]] \)), is a function mapping trees from \( T_{\Sigma'}(V) \) to trees from \( T_{\Sigma - \{\alpha_1, \ldots, \alpha_n\} \cup \Sigma'}(\Sigma - \{\alpha_1, \ldots, \alpha_n\} \cup V') \).

It is defined as follows:

\[
\begin{align*}
\alpha_j[\alpha_i \leftarrow t_i' \mid i \in [n]] &= t_j', \quad \text{for all } j \in [n], \\
v[\alpha_i \leftarrow t_i' \mid i \in [n]] &= v, \quad \text{for all } v \in V - \{\alpha_1, \ldots, \alpha_n\}, \\
(\sigma \, t_1 \cdots \, t_k)[\alpha_i \leftarrow t_i' \mid i \in [n]] &= \sigma \, t_1[\alpha_i \leftarrow t_i' \mid i \in [n]] \, \cdots \, t_k[\alpha_i \leftarrow t_i' \mid i \in [n]], \\
&\quad \text{for all } \sigma \in (\Sigma - \{\alpha_1, \ldots, \alpha_n\})^{(k)}, t_1, \ldots, t_k \in T_{\Sigma'}(V).
\end{align*}
\]

So a tree substitution permits the replacement of both variables and nullary symbols. (In our approach, we do not need more general tree substitutions that replace arbitrary trees. Such substitutions might be needed when extending our approach to more general forms of programs where the initial values of context arguments are not just nullary constructors.)

The following lemma will be needed repeatedly later on.

**Lemma 1 (properties of tree substitutions)** Let \( \Sigma \) be a ranked alphabet, \( V \) be a set of variables disjoint from \( \Sigma \), \( n \in \mathbb{N} \), and \( \alpha_1, \ldots, \alpha_n \in \Sigma^{(0)} \cup V \) be pairwise distinct. For every \( t, t_1, \ldots, t_n, t_1', \ldots, t_n' \in T_{\Sigma}(V) \):

(1) \( t[\alpha_i \leftarrow \alpha_i \mid i \in [n]] = t \),

(2) \( t[\alpha_i \leftarrow t_i \mid i \in [n]] [\alpha_j \leftarrow t_j' \mid j \in [n]] = t[\alpha_i \leftarrow t_i [\alpha_j \leftarrow t_j' \mid j \in [n]]] \mid i \in [n] \), and

(3) \( t[\alpha_i \leftarrow \beta_i \mid i \in [n]] [\beta_i \leftarrow t_i \mid i \in [n]] = t[\alpha_i \leftarrow t_i \mid i \in [n]] \) for pairwise distinct \( \beta_1, \ldots, \beta_n \in \Sigma^{(0)} \) that do not occur in \( t \).
PROOF. Straightforward, by induction on the structure of \( t \). \( \square \)

We consider a simple first-order, constructor-based functional programming language \( P \) as source and target language for the transformations. Every program \( p \in P \) consists of some modules. In every module some functions are defined by complete case analysis on the first argument (recursion argument) via pattern matching, where only flat patterns of the form \((c \ x_1 \ldots x_k)\) for constructors \( c \) and variables \( x_i \) are allowed. The other arguments are called context arguments. If, in a right-hand side of a function definition, there is a call of a function that is defined in the same module, then this call is (extended) primitive recursive, i.e. the first argument of this function call has to be a subtree \( x_i \) of the first argument in the corresponding left-hand side. Moreover, the variables \( x_1, \ldots, x_k \) may not occur anywhere else in the right-hand side of function definitions. Our transformation will only work on tree transducers, which are special programs satisfying these requirements. To ease readability, we choose an untyped ranked alphabet \( C_p \) of constructors, which is used to build up input trees and output trees (i.e., results) of every function in \( p \). In example programs and transformations we sometimes relax the completeness of function definitions on \( T_{C_p} \) by leaving out those equations which are not intended to be used in evaluations.

**Definition 2 (program, module, function definition, \( \text{rhs}_{p,f,c} \), RHS)**

Let \( C \) and \( F \) be ranked alphabets of constructors and defined function symbols, respectively, such that \( F^{(0)} = \emptyset \), and \( X, Y, C, F \) are pairwise disjoint. We define the sets \( P, M, D, R \) of programs, modules, function definitions, and right-hand sides as follows. Here, \( p, m, d, r, c, \) and \( f \) (also equipped with indices) range over the sets \( P, M, D, R, C, \) and \( F \), respectively.

- A program \( p \) is a set of modules \( m_1 \cdots m_l \).
- A module \( m \) is a set of function definitions \( d_1 \cdots d_h \).
- A function definition \( d \) is a set of defining equations of the form

  \[
  f \ (c \ x_1 \cdots x_k) \ y_1 \cdots y_n = r
  \]

- A right-hand side \( r \) is a tree of the following forms:
  - \( x_i \)
  - \( y_j \)
  - \( c \ r_1 \cdots r_k \)
  - \( f \ r_0 \ r_1 \cdots r_n \)

The sets of constructors and defined function symbols that occur in \( p \in P \) are denoted by \( C_p \) and \( F_p \), respectively. For every \( f \in F_p \), there is exactly one module \( m \) in \( p \) and exactly one function definition \( d \) in \( m \) such that \( f \) is defined in \( d \). The set of functions defined in \( m \in p \) is denoted by \( F_m \). For
every \( m \in p, f \in F_m^{(n+1)}, \) and \( c \in C_p^{(k)} \), there is exactly one equation of the form
\[
f (c \ x_1 \cdots x_k) \ y_1 \cdots y_n = rhs_{p,f,c},
\]
where \( rhs_{p,f,c} \in RHS(F_m, C_p \cup F_p - F_m, X_k, Y_n) \). Here, for every \( F' \subseteq F, C' \subseteq C \cup F, X' \subseteq X, \) and \( n \in \mathbb{N} \), \( RHS(F', C', X', Y_n) \) is the smallest set \( RHS \subseteq T_{F' \cup C'}(X' \cup Y_n) \) such that:

- \( Y_n \subseteq RHS \),
- for every \( c \in C'(a) \) and \( r_1, \ldots, r_a \in RHS : (c \ r_1 \ldots r_a) \in RHS \), and
- for every \( f \in F'(a+1), x_i \in X' \), and \( r_1, \ldots, r_a \in RHS : (f \ x_i r_1 \ldots r_a) \in RHS \).

Note that, in addition to constructors, defined function symbols may also be contained in the second argument \( C' \) of \( RHS \) in the previous definition. The functions in \( C' \) may then be called with arbitrary arguments in right-hand sides, whereas in calls of functions from \( F' - C' \), the recursion argument must be an \( x_i \). Hence, the latter calls are (extended) primitive recursive.

**Example 3 (the introductory example formalized in our language)**

Consider the programs \( p_{\text{acc}} \) and \( p_{\text{non}} \) from the introduction. Then:

- \( p_{\text{acc}} \in P \), where \( p_{\text{acc}} \) contains one module \( m_{\text{acc,lev}} \) with the definition of \( \text{lev} \), and
- \( p_{\text{non}} \in P \), where \( p_{\text{non}} \) contains two modules \( m_{\text{non,lev'}} \) and \( m_{\text{non,sub}} \), defining \( \text{lev'} \) and \( \text{sub} \), respectively.

For every program \( p \in P \), its evaluation (possibly on terms with variables) is described by a (nondeterministic) reduction relation \( \Rightarrow_p \) on \( T_{C_p \cup F_p}(Y) \), defined in the usual way by interpreting defining equations as **rewrite rules** [3]. We consider only **terminating** programs, i.e., ones for which there is no infinite chain \( s_1 \Rightarrow_p s_2 \Rightarrow_p s_3 \Rightarrow_p \cdots \). By their definition, programs in \( P \) never contain critical pairs, hence \( \Rightarrow_p \) is also confluent. As a consequence, for every \( s \in T_{C_p \cup F_p}(Y) \) there is a unique normal form with respect to \( \Rightarrow_p \), denoted by \( nf_p(s) \). By the completeness of function definitions, any element of \( T_{C_p \cup F_p} - T_{C_p} \) cannot be a normal form with respect to \( \Rightarrow_p \). Consequently, \( nf_p(s) \in T_{C_p} \) for every \( s \in T_{C_p \cup F_p} \).

Before introducing the classes of tree transducers relevant for this paper, we consider a special kind of program modules which will be needed for our deaccumulation technique, cf. the module with the function \( \text{sub} \) in the introduction. Since such a program module contains a substitution function which substitutes designated substitution constructors \( \pi_1, \ldots, \pi_n \) by parameters \( y_1, \ldots, y_n \), respectively, and retains other constructors (from a set \( C' \)), the module is
called sub-module induced by $C'$ and $\pi_1, \ldots, \pi_n$.

**Definition 4 (induced sub-module)** Let $C' \subseteq C$, sub $\in F^{(n+1)}$, and let $\pi_1, \ldots, \pi_n \in (C - C')^{(0)}$ be pairwise distinct. The sub-module induced by $C'$ and $\pi_1, \ldots, \pi_n$ consists of the following defining equations:

\[
\begin{align*}
\text{sub } \pi_j & \quad y_1 \cdots y_n = y_j, \text{ for every } j \in [n] \\
\text{sub } (c x_1 \cdots x_k) & \quad y_1 \cdots y_n = c \left( \text{sub } x_1 y_1 \cdots y_n \right) \cdots \left( \text{sub } x_k y_1 \cdots y_n \right), \text{ for every } c \in C^{(k)}.
\end{align*}
\]

For example, the function sub from the program $p_{non}$ in the introduction represents the sub-module induced by $C' = \{(\cdot, 2), S^{(1)}\}$ and $\pi_1 = 0, \pi_2 = \[]$.

The next lemma shows that the evaluation of a term sub $s_1 \cdots s_n$ replaces all occurrences of $\pi_j$ in $s$ by $s_j$, for all $j \in [n]$.

**Lemma 5 (semantics of substitution functions)** Let $p \in P$, sub $\in F^{(n+1)}_p$, and let $\pi_1, \ldots, \pi_n \in C^{(0)}_p$ be pairwise distinct. If $p$ contains the sub-module induced by $C_p - \{\pi_1, \ldots, \pi_n\}$ and $\pi_1, \ldots, \pi_n$, then for every $s, s_1, \ldots, s_n \in T_{C_p \cup F_p}$:

\[
nf_p(s s_1 \cdots s_n) = nf_p(s)[\pi_j \leftarrow nf_p(s_j) \mid j \in [n]].
\]

**PROOF.** Straightforward, by induction on the structure of $nf_p(s) \in T_{C_p}$. □

Some further useful information about substitution functions can be derived by additionally taking properties of tree substitutions into account.

**Lemma 6 (properties of substitution functions)** Let $p \in P$, sub $\in F^{(n+1)}_p$, and let $\pi_1, \ldots, \pi_n \in C^{(0)}_p$ be pairwise distinct. If $p$ contains the sub-module induced by $C_p - \{\pi_1, \ldots, \pi_n\}$ and $\pi_1, \ldots, \pi_n$, then for every $s, s_1, \ldots, s_n, s'_1, \ldots, s'_n$ in $T_{C_p \cup F_p}$:

\[
\begin{align*}
(1) \quad & \quad nf_p(s \pi_1 \cdots \pi_n) = nf_p(s), \\
(2) \quad & \quad nf_p(s_{s_1 \cdots s_n} s'_1 \cdots s'_n) \\
& \quad = nf_p(s_{s_1 s'_1 \cdots s'_n} \cdots (s_n s'_1 \cdots s'_n)), \text{ and} \\
(3) \quad & \quad nf_p(s s_1) = (c^{z_1 + z_2} \pi_1), \text{ if } n = 1 \text{ and } nf_p(s) = (c^{z_1} \pi_1), \text{ } nf_p(s_1) = (c^{z_2} \pi_1) \text{ for some } c \in C^{(1)}_p \text{ and } z_1, z_2 \in \mathbb{N}.
\end{align*}
\]

**PROOF.** Straightforward, using Lemma 5 and statements (1) and (2) of Lemma 1. □

Our transformations are based on the concepts of *macro tree transducers* [16,18] and *modular tree transducers* [19], which were motivated in the introduction and for which example computations were shown there. Since we
will present our deaccumulation technique only for modules defining exactly one function, we also project this restriction on the respective macro tree transducers. In the literature more general instances are studied which allow mutual recursion. Our transformations could also be defined for this case, but only with a considerable presentational overhead we seek to avoid here. The intermediate stages and final outputs of our transformation technique will be specialized modular tree transducers. We only introduce the required special cases rather than the general concept, again to simplify the presentation. Of course, the proofs in the literature about termination of the reduction relations induced by the tree transducer models under consideration carry over to our special cases. In contrast to (some of) the literature, we include an initial call in the definition of tree transducers which has the form of a right-hand side. Example 8 will illustrate the different classes of tree transducers, as well as the syntactic restrictions which are additionally introduced in the following definition.

Definition 7 (special mtts and modtts, and restrictions on them)

Let \( p \in P \).

- A pair \((m, r)\) with \( m \in p, |F_m| = 1, \) and \( r \in \text{RHS}(F_m, C_p, X_1, Y_0)\) is called a one-state macro tree transducer of \( p \) (for short 1-mtt of \( p \)) if for every \( c \in C_p^{(k)} \) we have \( \text{rhs}_{p,f,c} \in \text{RHS}(\{f\}, C_p, X_k, Y_n), \) where \( F_m = \{f^{(n+1)}\} \).

Thus, the single function \( f \) defined in module \( m \) may call itself in a primitive-recursive way, but it does not call any functions from other modules. Moreover, the initial call \( r \) is a term built from \( f, \) constructors, and the variable \( x_1 \) as first argument of all subterms rooted with \( f. \)

- A triple \((m_1, m_2, r)\) with \( m_1, m_2 \in p \) and \( |F_{m_1}| = 1 \) is called non-accumulative modular tree transducer of \( p \) (for short nmodtt of \( p \)) if:

1. \( F_{m_1} = F_{m_1}^{(1)} \) and \( F_{m_2} = F_{m_2}^{(n+1)} \) for some \( n \in \mathbb{N}, \)
2. for \( f \in F_{m_1} \) and every \( c \in C_p^{(k)} \): \( \text{rhs}_{p,f,c} \in \text{RHS}(F_{m_1}, C_p \cup F_{m_2}, X_k, Y_n), \)
3. for every \( g \in F_{m_2} \) and \( c \in C_p^{(k)} \) with \( k > 0 \) we have \( \text{rhs}_{p,g,c} = c \left( g_1 \ x_1 \ y_1 \cdots y_n \right) \cdots \left( g_k \ x_k \ y_1 \cdots y_n \right) \)
   for some (not necessarily pairwise distinct) \( g_1, \ldots, g_k \in F_{m_2}, \)
4. for every \( g \in F_{m_2} \) and \( c \in C_p^{(0)} \) we have \( \text{rhs}_{p,g,c} \in \{c\} \cup Y_n, \)
5. \( r \in \text{RHS}(F_{m_1}, C_p \cup F_{m_2}, X_1, Y_0). \)

Thus, the single function \( f \) defined in module \( m_1 \) is unary. In its right-hand sides, it may call itself primitive-recursively and it may call the functions defined in module \( m_2, \) all of which have the same rank, with arbitrary arguments. The function definitions in \( m_2 \) have a special form in that non-nullary constructors \( c \) in the input are reproduced in the output and their subtrees are traversed in order with unchanged context arguments, whereas nullary constructors \( c \) in the input are either also reproduced or replaced by one of the context arguments. The initial call \( r \) is as for 1-mtts, but it
may also contain the functions defined in $m_2$.

- An nmodtt $(m_1, m_2, r)$ of $p$ with $|F_{m_2}| = 1$ is called a substitution modular tree transducer of $p$ (for short smodtt of $p$) if there are pairwise distinct $\pi_1, \ldots, \pi_n \in C_p^{(0)}$, where the single function in $F_{m_2}$ has rank $n + 1$, such that:
  
  (1) $m_2$ is the sub-module induced by $C_p - \{\pi_1, \ldots, \pi_n\}$ and $\pi_1, \ldots, \pi_n$.
  
  (2) $r \in \text{RHS}(F_{m_1}, C_p - \{\pi_1, \ldots, \pi_n\} \cup \{\text{sub}\}, X_1, Y_0)$.

  Thus, the single function definition in $m_2$ now has the even more specialized form of a substitution function, and the initial call $r$ may not contain the corresponding substitution constructors.

- A 1-mtt $(m, r)$ of $p$ is called nullary constructor distinct (for short ncd) if there are pairwise distinct $c_1, \ldots, c_n \in C_p^{(0)}$ such that $r = (f \ x_1 \ c_1 \ldots c_n)$, where $F_m = \{f\}$, and $c_1, \ldots, c_n$ do not occur in right-hand sides of the function definition in $m$.

  An smodtt $(m_1, m_2, r)$ of $p$ is called ncd if $r = \text{sub} (f \ x_1 \ c_1 \ldots c_n)$ with pairwise distinct $c_1, \ldots, c_n \in C_p^{(0)} - \{\pi_1, \ldots, \pi_n\}$ that do not occur in right-hand sides of the definition of $f$ in $m_1$.

- An nmodtt $(m_1, m_2, r)$ of $p$ is called initial value free (for short ivf) if $r = (f \ x_1)$, where $F_{m_1} = \{f\}$.

**Example 8 (Example 3 continued)** Consider the programs $p_{\text{acc}}$ and $p_{\text{non}}$ from the introduction, and their modules $m_{\text{acc,lev}}, m_{\text{non,lev'}}$, and $m_{\text{non,sub}}$ as identified in Example 3. Then:

- $(m_{\text{acc,lev}}, r_{\text{acc}})$ with initial call $r_{\text{acc}} = (\text{lev} \ x_1 \ 0 \ [\])$ is a 1-mtt of $p_{\text{acc}}$ that is ncd.

- Our basic transformation, to be presented in Section 3.1, consists of the two steps “decomposition” and “constructor replacement”. Decomposition transforms $p_{\text{acc}}$ into a program $p_{\text{dec}} \in P$ containing the following two modules $m_{\text{dec,lev'}}$ and $m_{\text{dec,sub}}$:

$$
\begin{align*}
\text{lev'} (S \ x_1) &= \text{sub} (\text{lev'} \ x_1) (S (S \ \pi_1)) \ (\pi_1 : \pi_2) \\
\text{lev'} 0 &= \pi_2 \\
\text{sub} (x_1 : x_2) y_1 y_2 &= (\text{sub} x_1 \ y_1 \ y_2) : (\text{sub} x_2 \ y_1 \ y_2) \\
\text{sub} (S \ x_1) y_1 y_2 &= S (\text{sub} x_1 \ y_1 \ y_2) \\
\text{sub} 0 y_1 y_2 &= 0 \\
\end{align*}
$$

Here, $(m_{\text{dec,lev'}}, m_{\text{dec,sub}}, r_{\text{dec}})$ with initial call $r_{\text{dec}} = (\text{sub} (\text{lev'} \ x_1) 0 \ [\])$ is an smodtt (and hence also an nmodtt) of $p_{\text{dec}}$ that is ncd, but not ivf.
\[- (m_{\text{non}, \text{lev}^{'}}; m_{\text{non}, \text{sub}}, r_{\text{non}}) \text{ with initial call } r_{\text{non}} = (\text{lev}^{'}, x_1) \text{ is an smodtt (with } \ n = 2, \pi_1 = 0, \text{ and } \pi_2 = []) \text{ of } p_{\text{non}} \text{ that is ivf.}\]

3 Deaccumulation

To improve verifiability, we transform accumulative programs into non-accumulative programs by transforming 1-mtts into ivf nmodtts. The defined functions of the resulting programs have no context arguments at all or they have context arguments that are not accumulating. Moreover, the resulting initial calls have no (initial values in) context argument positions. In Section 3.1, we present a first deaccumulation technique for 1-mtts that are ncd. Section 3.2 introduces a deaccumulation technique which can also handle many 1-mtts that are not ncd.

3.1 Basic Deaccumulation

Conceptually, the transformation proceeds in two steps: “decomposition” (Section 3.1.1) and “constructor replacement” (Section 3.1.2). For the extension presented in Section 3.2 we will integrate the two steps into a single one.

3.1.1 Decomposition

In [16,18,19] it was shown that every mtt (with possibly several functions of arbitrary ranks) can be decomposed into a top-down tree transducer (an mtt with unary functions only [15,49,53]) plus a substitution device. In this paper we use a modification of this result, integrating the constructions of Lemmas 21 and 23 in [37]. The key idea is to simulate an \((n + 1)\)-ary function \(f\) by a new unary function \(f^{'}.\) To this end, all context arguments are deleted and only the recursion argument is maintained. Since \(f^{'}\) does not know the actual values of its context arguments, it uses a new constructor \(\pi_j\) whenever \(f\) uses its \(j\)-th (formal) context argument. For this purpose, every occurrence of \(y_j\) in the right-hand sides of equations for \(f\) is replaced by \(\pi_j.\) The current (actual) context arguments are integrated into the calculation by replacing every term of the form \((f \ x_i \ \cdots)\) in a right-hand side or in the initial call by \((\text{sub} \ (f^{'}, x_i) \ \cdots)\) for an appropriate substitution function. As explained before, \((\text{sub} \ t \ s_1 \cdots s_n)\) replaces every \(\pi_j\) in the first argument \(t\) of \(\text{sub}\) by the \(j\)-th context argument \(s_j.\) The transformation of right-hand sides described above will be formalized by the function \text{dec} in Transformation 10.

The essence of this transformation can also be stated in terms of \(f\) alone: a computation of \(f\) with arbitrary context arguments can always be simulated by a computation of \(f\) with the particular context arguments \(\pi_1, \ldots, \pi_n\) as
placeholders, which are only afterwards substituted with the appropriate values. Since the computation of $f$ in this simulation is performed with such fixed placeholders, it can just as well be performed by a unary function. The following lemma is proved in Appendix D using Lemma 1(3) and Lemma 5.

**Lemma 9 (key to the decomposition transformation)** Let $p \in P$ and $(m, r)$ be a 1-mtt of $p$, where $F_m = \{f^{(n+1)}\}$. Further, let $p' \in P$ with $C_{p'} = C_p \cup \{\pi_1^{(0)}, \ldots, \pi_n^{(0)}\}$ for pairwise distinct $\pi_1, \ldots, \pi_n \notin C_p$, and $\text{sub} \in F_{p'}^{(n+1)}$. Let $p'$ contain (at least) the sub-module induced by $C_p$ and $\pi_1, \ldots, \pi_n$, and a module $m'$ with all defining equations from $m$ and additional equations of the form 

$\text{"f\,\pi_j\,y_1\cdots y_n = \ldots"}$

that define $f$ for the new constructors $\pi_1, \ldots, \pi_n$ such that $(m', r)$ is a 1-mtt of $p'$. Then for every $t \in T_{C_p}$ and $s_1, \ldots, s_n \in T_{C_{p'} \cup F_{p'}}$:

\[
nf_{p'}(f \ t \ s_1 \cdots s_n) = nf_{p'}(\text{sub } (f \ t \ \pi_1 \cdots \pi_n) \ s_1 \cdots s_n).
\]

The following transformation and lemma formalize the above intuitions. Moreover, we will show that the ncd property is carried over from the original to the decomposed tree transducer. Note that the transformation retains the defining equations of $f$ from the original program. This is necessary because $f$ may be called from other modules. When giving examples, we do not show such retained function definitions.
Example 11 (decomposition for the introductory example) Consider the program \( p_{\text{acc}} \) from the introduction and its module \( m_{\text{acc,lev}} \) as identified in Example 3. Let \( r_{\text{acc}} = (\text{lev } x_1 0 []) \). Decomposition transforms the ncd 1-mtt \((m_{\text{acc,lev}}, r_{\text{acc}})\) of \( p_{\text{acc}} \) into the ncd smodtt \((m_{\text{dec,lev}}, m_{\text{dec,sub}}, r_{\text{dec}})\) of \( p_{\text{dec}} \) as given in Example 8.

The following lemma is proved in Appendix D using the principle of simultaneous induction (cf., e.g., [18,22,56]) and Lemma 9.

Lemma 12 (semantic correctness of the decomposition) For \( p, (m, r), p', (m_1, m_2, r') \) as in Transformation 10, for every \( t \in T_{C_p} \):

\[
\text{nf}_p(r[x_1 \leftarrow t]) = \text{nf}_{p'}(r'[x_1 \leftarrow t]).
\]

Moreover, if \((m, r)\) is ncd, then so is \((m_1, m_2, r')\).

However, we have not yet reached our goal to improve the verifiability of programs.
Example 13 (initial values are still problematic for verification)

Let \((m_{\text{dec,lev}'},m_{\text{dec,sub}},r_{\text{dec}})\) be the smodtt of \(p_{\text{dec}}\) from Example 8 which was created by decomposition in Example 11. We resume the first proof attempt from the introduction. Since the initial call has changed from \((\text{lev} \ x_1 \ 0 \ []\)) to \((\text{sub} \ (\text{lev}' \ x_1) \ 0 \ []\)), we have to prove

\[
\text{sub} \ (\text{lev}' \ x_1) \ 0 \ [] = \text{lev}_2 \ x_1
\]

by induction. Again, the automatic proof fails, because in the induction step \((x_1 \mapsto (S \ x_1))\) the induction hypothesis cannot be successfully applied to prove the equality of \((\text{sub} \ (\text{lev}' \ (S \ x_1)) \ 0 \ []\)) and \((\text{lev}_2 \ (S \ x_1))\). The problem is that the context arguments of \((\text{sub} \ (\text{lev}' \ x_1) \ (S \ (S \ \pi_1)) \ (\pi_1 : \pi_2))\), which originates as subterm from rule application to \((\text{sub} \ (\text{lev}' \ (S \ x_1)) \ 0 \ []\)) do not fit to the context arguments of the term \((\text{sub} \ (\text{lev}' \ x_1) \ 0 \ []\)) in the induction hypothesis.

3.1.2 Basic Constructor Replacement

We solve the problem observed above by avoiding applications of substitution functions (with specific context arguments like 0 and [] in Example 13) in initial calls. Then the initial call always has the form \(f' \ x_1\) for a unary function \(f'\). Hence, induction hypotheses can be applied without paying attention to context arguments. The idea, illustrated on Example 13, is to replace the substitution constructors \(\pi_1\) and \(\pi_2\) by 0 and [] from the initial call. Thus, the initial values of \(\text{sub}'\)'s context arguments are encoded into the equations of the program and the substitution in the initial call becomes superfluous.

We restrict ourselves to 1-mtts that are ncd. Then, after decomposition the initial calls have the form \((\text{sub} \ (f' \ x_1) \ c_1 \ldots c_n)\), where \(c_1, \ldots, c_n\) are nullary and pairwise distinct. Thus, when replacing each \(\pi_j\) by \(c_j\), there is a unique correspondence between the \(c_1, \ldots, c_n\) and the substitution constructors \(\pi_1, \ldots, \pi_n\).

(In the next section we will deal with the case of identical \(c_1, \ldots, c_n\).) After replacing each \(\pi_j\) by \(c_j\), the constructors \(c_1, \ldots, c_n\) show two different faces: If a \(c_j\) occurs within a first argument of \(\text{sub}\), then it acts like the former substitution constructor \(\pi_j\), i.e., it will be substituted by the \(j\)-th context argument of \(\text{sub}\). Thus, \(\text{sub}\) now has the defining equation \(\text{sub} \ c_j \ y_1 \ldots y_n = y_j\). Only occurrences of a \(c_j\) outside of \(\text{sub}'\)'s first argument are left unchanged, i.e., there the constructor \(c_j\) is interpreted as its original value. To make sure that there is no confusion between these two roles of \(c_j\), we again use the ncd-condition. It ensures that before the constructor replacement, \(c_j\) did not occur in right-hand sides of \(f'\)'s definition. Hence, the only occurrence of \(c_j\) which does not stand for the substitution constructor \(\pi_j\) is as context argument of \(\text{sub}\) in the initial call, where it is harmless. Actually, this whole substitution in the initial call can be omitted, because the call \((\text{sub} \ (f' \ x_1) \ c_1 \ldots c_n)\) would now just mean to replace every \(c_j\) in \((f' \ x_1)\) by \(c_j\) (cf. also Lemma 6(1)). Simplifying the initial call accordingly makes the resulting smodtt initial value free (ivf).
In the next section we will extend the basic idea in order to allow the (identical) constructors $c_1, \ldots, c_n$ to occur also in the right-hand sides of the original 1-mtt. But first we present the formalization for the simpler case discussed above.

**Transformation 14 (basic constructor replacement)** Let $p' \in P$ and $(m_1, m_2, r')$ be an smodtt of $p'$ as constructed in Transformation 10. Moreover, let $(m_1, m_2, r')$ be ncd, i.e., $r' = (\text{sub} (f' \cdot x_1) c_1 \cdots c_n)$ with pairwise distinct $c_1, \ldots, c_n \in C_{p'}^{(0)} - \{\pi_1, \ldots, \pi_n\}$ that do not occur in right-hand sides of the definition of $f'$ in $m_1$. We construct a program $p'' \in P$ which results from $p'$ by replacing $m_1$ and $m_2$ by the modules $m'_1$ and $m'_2$, defined below. Then, $(m'_1, m'_2, r'')$ is an smodtt of $p''$ that is ivf, where $r''$ is also defined below.

1. For every $c \in (C_{p'} - \{\pi_1, \ldots, \pi_n\})^{(k)}$ and equation $f' (c \cdot x_1 \cdots x_k) = \text{rhs}_{p', f', c}$ in $m_1$, the module $m'_1$ contains the equation $f' (c \cdot x_1 \cdots x_k) = \text{rhs}_{p', f', c}[\pi_j \leftarrow c_j \mid j \in [n]]$.

2. $m'_2$ is the sub-module induced by $C_{p'} - \{\pi_1, \ldots, \pi_n, c_1, \ldots, c_n\}$ and $c_1, \ldots, c_n$.

3. $r'' = (f' \cdot x_1)$.

The dummy equations for the $\pi_1, \ldots, \pi_n$ included in the other modules of $p'$ can now be dropped, so that $C_{p''} = C_{p'} - \{\pi_1, \ldots, \pi_n\}$.

**Example 15 (constructor replacement for introductory example)**

Let $(m\text{\_dec}, \text{lev}'', m\text{\_dec}, \text{sub}, r\text{\_dec})$ be the ncd smodtt of $p\text{\_dec}$ from Example 8 which was created by decomposition in Example 11. Basic constructor replacement transforms it into the ivf smodtt $(m\text{\_non}, \text{lev}'', m\text{\_non}, \text{sub}, r\text{\_non})$ of $p\text{\_non}$ as identified in Example 8. This resulting smodtt is exactly the program version for which the introduction demonstrated that automatic verification is easily possible.

The following lemma is proved in Appendix D using the principle of simultaneous induction, all three statements of Lemma 1, and Lemma 5.

**Lemma 16 (semantic correctness of basic constructor replacement)** For $p'$, $(m_1, m_2, r')$, $p''$, and $(m'_1, m'_2, r'')$ as in Transformation 14, for every $t \in T_{C_{p'}}$:

$$nf_{p'}(r'[x_1 \leftarrow t]) = nf_{p''}(r''[x_1 \leftarrow t]).$$

By combining Lemmas 12 and 16, we easily get the following theorem about the compound transformation.
Theorem 17 (semantic correctness of basic deaccumulation) Let $p \in P$ and $(m, r)$ be a 1-mtt of $p$ that is ncd. Let $p'$ and $(m_1, m_2, r')$ be the program and the smodtt constructed from $p$ and $(m, r)$ by Transformation 10. The smodtt is ncd. Further, let $p''$ and $(m'_1, m'_2, r'')$ be the program and the ivf smodtt constructed from $p'$ and $(m_1, m_2, r')$ by Transformation 14. For every $t \in T_{C_p}$:

$$nf_p(r[x_1 \leftarrow t]) = nf_{p''}(r''[x_1 \leftarrow t]).$$

Hence, for every 1-mtt that is ncd we can construct a semantically equivalent smodtt that uses no initial values and no accumulators. Thus, the resulting smodtt is well suited for verification.

3.2 Advanced Deaccumulation

The results of Section 3.1 were already given in the preliminary version of this paper [24]. However, in Appendix D we also present the full correctness proofs, which were omitted from [24].

Here we improve upon these results and develop an extension for 1-mtts violating the condition ncd. Thus, we now permit initial calls $(f \ x_1 \ c_1 \cdots \ c_n)$ where the nullary constructors $c_1, \ldots, c_n$ do no longer have to be pairwise distinct and where they may also occur in right-hand sides of $f$’s definition. To ease the presentation, in the following we restrict ourselves to the case where $c_1 = \cdots = c_n$. Note that the general case, in which there is no restriction on the nullary constructors in the initial call, is only technically, but not conceptually more complicated. This is due to the fact that we will use a system of “substitution-like” functions to distinguish between different meanings of the same constructor (i.e., they distinguish whether a constructor is used as a placeholder for some context argument, and for which, or whether the constructor should indeed have its original meaning). In the general case, parameter positions in the initial call with identical constructors can be grouped together and one can define and combine systems of substitution-like functions for the different groups.

To demonstrate the problems with deaccumulation for functions and initial calls as above, but also to motivate our approach to overcome these problems, we first consider two examples.

Example 18 (identical constructors in the initial call) Assume that the original 1-mtt, intended to compute the sum of the first $x_1$ natural numbers,
consists of the module

\[
\begin{align*}
    m_{\text{snat}, \text{sum}} & : \quad \text{sum} (S \ x_1) \ y_1 \ y_2 = \text{sum} \ x_1 \ (S \ y_1) \ (y_1 + y_2) \\
    \text{sum} \ 0 \ y_1 \ y_2 & = y_2
\end{align*}
\]

and the initial call \( r_{\text{snat}} = (\text{sum} \ x_1 \ 0 \ 0) \), where “snat” stands for “sum of natural numbers”. Due to the similarity in structure to the introductory example, analogous verification problems occur when trying to reason inductively about this specification. Attempting to improve the provability, we would first perform the decomposition transformation, which delivers an smodtt consisting of the function definitions

\[
\begin{align*}
    \text{sum'} (S \ x_1) & = \text{sub} \ (\text{sum'} \ x_1) \ (S \ \pi_1) \ (\pi_1 + \pi_2) \\
    \text{sum'} \ 0 & = \pi_2 \\
    \text{sub} \ (x_1 + x_2) \ y_1 \ y_2 & = (\text{sub} \ x_1 \ y_1 \ y_2) + (\text{sub} \ x_2 \ y_1 \ y_2) \quad \text{sub} \ \pi_1 \ y_1 \ y_2 = y_1 \\
    \text{sub} \ (S \ x_1) \ y_1 \ y_2 & = S \ (\text{sub} \ x_1 \ y_1 \ y_2) \quad \text{sub} \ \pi_2 \ y_1 \ y_2 = y_2 \\
    \text{sub} \ 0 \ y_1 \ y_2 & = 0
\end{align*}
\]

and the initial call \( \text{sub} \ (\text{sum'} \ x_1) \ 0 \ 0 \). The symbol + is treated as an ordinary binary constructor here. This is safe because clearly, if one can verify a conjecture (without negation) by treating + as a constructor (i.e., by using no information about it), then the conjecture also holds if + is a function defined by some equations elsewhere.

Note that still the same constructor 0 is the initial value for both context arguments. Now we perform the usual (but naïve, since it leads to a nondeterminism) replacement of the substitution constructors \( \pi_1 \) and \( \pi_2 \) by the corresponding values 0 and 0 from the initial call. In addition to the already existing equation \( \text{sub} \ 0 \ y_1 \ y_2 = 0 \), this leads to two more (different) equations with this left-hand side:

\[
\begin{align*}
    \text{sub} \ 0 \ y_1 \ y_2 & = y_1 \\
    \text{sub} \ 0 \ y_1 \ y_2 & = y_2
\end{align*}
\]

This kind of nondeterminism clearly conflicts with our aim of a semantics-preserving transformation of programs.

The idea for overcoming this problem is based on an analysis of the decomposed program. Note that, outside its definition, the sub-function is only used with a call to \( \text{sum'} \) as first argument. Hence, the substitution only has to work properly for the results computed by \( \text{sum'} \) (i.e., for the “output trees” of \( \text{sum'} \)). Figure 1 shows (the first elements in) the sequence of these output trees for the above
example, with increasing height. There, \( \pi_1 \) occurs only in left subtrees of a + symbol, whereas \( \pi_2 \) never occurs in such positions. This information about the contexts in which the different substitution constructors \( \pi_1 \) and \( \pi_2 \) may occur can be used as a guide for performing the necessary substitution task, even after the difference between \( \pi_1 \) and \( \pi_2 \) has been blurred by replacing both by 0.

![Diagram of tree structures](image)

Fig. 1. \( \text{sum}' \ 0, \ \text{sum}' \ (S \ 0), \ \text{sum}' \ (S \ (S \ 0)), \ \text{sum}' \ (S \ (S \ (S \ 0))), \ldots \)

To employ the “context information”, we define two different “substitution-like” functions. The function \( \text{sub}_1 \) corresponds to positions in left subtrees of a + symbol and therefore, it interprets the symbol 0 like the substitution constructor \( \pi_1 \). Analogously, \( \text{sub}_2 \) corresponds to the other positions and interprets the symbol 0 like the substitution constructor \( \pi_2 \). Thus, we replace the above definitions of \( \text{sum}' \) and \( \text{sub} \) by the following (partial) ones:

\[
\begin{align*}
\text{sum}' \ (S \ x_1) &= \text{sub}_2 \ (\text{sum}' \ x_1) \ (S \ 0) \ (0 + 0) \\
\text{sum}' \ 0 &= 0 \\
\text{sub}_2 \ (x_1 + x_2) \ y_1 \ y_2 &= (\text{sub}_1 \ x_1 \ y_1 \ y_2) + (\text{sub}_2 \ x_2 \ y_1 \ y_2) \\
\text{sub}_1 \ (S \ x_1) \ y_1 \ y_2 &= S \ (\text{sub}_1 \ x_1 \ y_1 \ y_2) \\
\text{sub}_2 \ 0 \ y_1 \ y_2 &= y_2 
\end{align*}
\]

and the initial call is replaced by \( \text{sub}_2 \ (\text{sum}' \ x_1) \ 0 \ 0 \). If \( t \) is a tree as in Figure 1, but where \( \pi_1 \) and \( \pi_2 \) are replaced by 0, then starting with \( \text{sub}_2 \) at the root of \( t \) will lead to the same substitutions at leaf nodes as would have been performed by \( \text{sub} \). Therefore, evaluation of \( \text{sub}_2 \ (\text{sum}' \ x_1) \ 0 \ 0 \) with \( \text{sum}' \) as above will yield the same result as evaluation of \( \text{sub} \ (\text{sum}' \ x_1) \ 0 \ 0 \) with the former definition of \( \text{sum}' \), for every instantiation of \( x_1 \). Moreover, \( \text{sub}_2 \ (\text{sum}' \ x_1) \ 0 \ 0 \) can be simplified to \( \text{sum}' \ x_1 \) because a call to \( \text{sub}_2 \) substitutes every 0 in its recursion argument by either its first or its second context argument, which leads to an identity operation if both context arguments are themselves initialized with 0. Thus, finally, we have a program that is semantically equivalent to the original one but uses no initial values. It solves the verification problems for the original \text{sum}-function in the same way as demonstrated for \( p_{\text{acc}} \) and \( r_{\text{acc}} \) vs. \( p_{\text{non}} \) and \( r_{\text{non}} \) in the introduction (cf. Appendix C).

The previous example still relies on the fact that the nullary constructor 0 from
the initial call does not occur in the right-hand sides of defining equations for 
sum, and hence also not in the output of sum’ after decomposition. This would 
no longer be the case if, for example, we wanted to express the incrementation 
of y₁ with an explicit addition rather than an application of the successor 
symbol, that is, if we were to replace the first equation of m_{snat,sum} with the 
following one:

\[
\text{sum } (S \ x₁) \ y₁ \ y₂ = \text{sum } x₁ \ (y₁ + (S \ 0)) \ (y₁ + y₂).
\]

To discuss our strategy for such a situation, we first consider a simpler example 
in the following. Nevertheless, this example is considerably more interesting 
in terms of the obtained substitution-like functions. However, we will return 
to the above variation of m_{snat,sum} in Example 29.

Example 19 (initial value occurring in original right-hand sides)
Consider the 1-mtt consisting of the module

\[
m_{\text{string},f} : \quad f (A \ x₁) \ y₁ = f \ x₁ \ (A \ (A \ y₁))
\]
\[
f (B \ x₁) \ y₁ = f \ x₁ \ (A \ E)
\]
\[
f E \quad y₁ = y₁
\]

and the initial call (f \ x₁ E). If one regards trees as strings, then f computes the 
function with f ((A|B)^n B A^n E) y₁ = A^{2^n+1} E and f (A^n E) y₁ = A^{2^n} y₁.

Decomposition results in an smodtt consisting of the function definitions

\[
f (A \ x₁) = \text{sub } (f’ \ x₁) \ (A \ (A \ π₁))
\]
\[
f (B \ x₁) = \text{sub } (f’ \ x₁) \ (A \ E)
\]
\[
f E = π₁
\]
\[
\text{sub } (A \ x₁) \ y₁ = A (\text{sub } x₁ \ y₁) \quad \text{sub } E \ y₁ = E
\]
\[
\text{sub } (B \ x₁) \ y₁ = B (\text{sub } x₁ \ y₁) \quad \text{sub } π₁ \ y₁ = y₁
\]

and the initial call (sub (f’ \ x₁) E). The usual naïve replacement of π₁ by the 
corresponding value E from the initial call would lead to the sub-equation

\[
\text{sub } E \ y₁ = y₁.
\]

Note that the original equation sub E \ y₁ = E with the same left-hand side 
cannot be dropped since E may occur in the output of the function f’ above (if 
f’ is applied to an input in which B occurs). Again, the problem is tack-
led by analyzing the output trees of f’, as shown in Figure 2. We obtain 
f’ ((A|B)^n B A^n E) = A^{2^n+1} E and f’ (A^n E) = A^{2^n} π₁. Thus, the output 
always consists of a (possibly empty) string of A-symbols followed by either E 
or π₁, depending on whether the number of A-symbols is odd or even.
After replacing $\pi_1$ by $E$, this information can be employed by using two substitution-like functions that “count” the number of $A$-symbols. The function $\text{sub}_1$ corresponds to positions below an even number of $A$-symbols and $\text{sub}_0$ corresponds to positions below an odd number of $A$-symbols. Thus, depending on $\text{sub}_0$ or $\text{sub}_1$, an $E$ found at the end is to be interpreted as an actual $E$ or as a $\pi_1$. More precisely, we replace the above definitions of $f'$ and sub by the following (partial) ones:

$$f'(A \ x_1) = \text{sub}_1 (f' \ x_1) (A \ (A \ E))$$
$$f'(B \ x_1) = \text{sub}_1 (f' \ x_1) (A \ E)$$
$$f' \ E = E$$
$$\text{sub}_0 (A \ x_1) \ y_1 = A (\text{sub}_1 \ x_1 \ y_1) \quad \text{sub}_0 \ E \ y_1 = E$$
$$\text{sub}_1 (A \ x_1) \ y_1 = A (\text{sub}_0 \ x_1 \ y_1) \quad \text{sub}_1 \ E \ y_1 = y_1$$

and the initial call is replaced by $(\text{sub}_1 (f' \ x_1) \ E)$. The latter can be simplified to $(f' \ x_1)$, because $nf_p(\text{sub}_1 \ t \ E) = t$ holds for every tree $t$ over $\{A, E\}$ (assuming $p$ is the underlying program). Thus, we again have obtained a program that is equivalent to the original one but uses no initial values. This example demonstrates that substitution-like functions can not only distinguish between different argument positions of some symbol (as in Example 18), but they can also distinguish between positions according to the number of symbols occurring above them.

In the previous two examples, the definitions of the substitution-like functions used to overcome the limitations of basic constructor replacement were obtained by an ad-hoc analysis of the program after decomposition. Moreover, we did not formally prove that they serve their purpose for every input tree. In order to turn the above ideas into an automatic, semantics-preserving pro-
gram transformation, we should of course follow a more systematic approach and also provide a correctness proof.

As a first step, it seems reasonable to specify what exactly we mean by substitution-like functions. Intuitively, we want a group of mutually recursive functions that reproduce the shape of an input tree provided as recursion argument, leave the (non-nullary) labels of internal nodes unchanged, and at leaf nodes decide to either also leave the label unchanged or to replace the leaf with some context argument carried through unchanged from the root. That is, we want exactly the kind of functions that are allowed in the second module of an nmodtt (cf. Definition 7). But how many of them do we need? If the function defined in the original 1-mtt has \( n \) context parameters, then the smodtt obtained after decomposition uses \( n \) substitution constructors. To distinguish them even after each of them has been replaced by the same nullary constructor from the initial call, at least \( n \) “incarnations” of sub should be used, as in Example 18. If additionally we want to handle the case that the initial value may also occur in right-hand sides of the original 1-mtt, as in Example 19, we need a further sub\(_0\)-function that leaves the nullary constructor in question unchanged. While in principle one could use arbitrarily many mutually recursive substitution-like functions, we restrict ourselves to the \( n + 1 \) functions motivated above. This also reduces the search space when trying to find suitable substitution-like functions. Having fixed the number of substitution-like functions and their roles regarding the treatment of the nullary constructor acting as initial value, it remains to specify the ways in which they call each other when applied to non-nullary constructors in the recursion argument. The degrees of freedom we have in doing so can be captured as in the following definition.

**Definition 20 (candidate and induced sub-like module)** Let \( p \in P \) and \( n \in \mathbb{N} \). A candidate for \( p \) of rank \( n + 1 \) is a mapping

\[
K : \{(u, c, i) \mid u \in [0, n], c \in C_{p}^{(k)}, i \in [k]\} \rightarrow [0, n].
\]

For every \( \pi_0 \in C_{p}^{(0)} \), the sub-like module induced by \( K \) and \( \pi_0 \) consists of definitions for pairwise distinct functions sub\(_0\), \ldots, sub\(_n\) in \((F - F_{p})^{(n+1)}\), where for every \( u \in [0, n] \) the following equations are included:

\[
\begin{align*}
\text{sub}_u \pi_0 y_1 \cdots y_n &= \begin{cases} 
\pi_0 & \text{if } u = 0 \\
y_u & \text{otherwise}
\end{cases} \\
\text{sub}_u (c x_1 \cdots x_k) y_1 \cdots y_n &= c \left( \text{sub}_{K(u,c,1)} x_1 y_1 \cdots y_n \right) \cdots \\
& \quad \left( \text{sub}_{K(u,c,k)} x_k y_1 \cdots y_n \right),
\end{align*}
\]

for all \( c \in (C_{p} - \{\pi_0\})^{(k)} \).

**Example 21 (representing a candidate for Example 18)** Let \( p \) be a program with \( C_{p} = \{+^{(2)}, S^{(1)}, 0^{(0)}\} \). The following table specifies a candidate
$K$ for $p$ of rank 3, where the value of $K(u,c,i)$ is found in column $u$ of row $(c,i)$:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(+,1)</td>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>(+,2)</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>(S,1)</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

The sub-like module induced by $K$ and $\pi_0 = 0$ contains, among others, the defining equations for $\text{sub}_1$ and $\text{sub}_2$ given in Example 18. In particular, the boldface entries in the above table correspond to the equations for recursion arguments built with $+$ and $S$ in Example 18. The non-boldface entries correspond to the following equations (e.g., the entries $K(0,+1) = 0$ and $K(0,+2) = 0$ induce the two calls of $\text{sub}_0$ in the right-hand side of the first equation):

$\text{sub}_0(x_1 + x_2) y_1 y_2 = (\text{sub}_0 x_1 y_1 y_2) + (\text{sub}_0 x_2 y_1 y_2)$

$\text{sub}_0(S x_1) y_1 y_2 = S (\text{sub}_0 x_1 y_1 y_2)$

$\text{sub}_1(x_1 + x_2) y_1 y_2 = (\text{sub}_2 x_1 y_1 y_2) + (\text{sub}_1 x_2 y_1 y_2)$

$\text{sub}_2(S x_1) y_1 y_2 = S (\text{sub}_1 x_1 y_1 y_2)$

Moreover, by definition we have

$\text{sub}_0 0 y_1 y_2 = 0$.

Note that for a given program there are only finitely many candidates of a given rank. Hence, we can systematically check all candidates $K$ to find one that induces an appropriate replacement for the $\text{sub}$-module in the decomposed smodtt, in the sense that this new $\text{sub}$-like module can take over the work of the actual substitution function even after all occurrences of $\pi_1, \ldots, \pi_n$ in the definition of the function $f'$ from the decomposed smodtt have been replaced by the nullary constructor acting as initial value, called $\pi_0$. For a given candidate, this means to determine whether one of the functions in the induced $\text{sub}$-like module, say $\text{sub}_u$, has the property that whenever it is applied to an output tree computed by the function $f'$ from the decomposed smodtt, positions labeled with $\pi_v$ can only be reached by the $\text{sub}_v$-function, for every $v \in [0,n]$. This is both a sufficient and necessary condition to ensure that after replacing $\pi_1, \ldots, \pi_n$ by $\pi_0$ in the definition of $f'$, $\text{sub}_u$ performs the same substitutions which were previously done by $\text{sub}$.

**Example 22 (checking a candidate for an output tree of $\text{sum}'$)**

Consider the third output tree in Figure 1 of $\text{sum}'$ from the decomposed smodtt in Example 18. Further, consider the sub-like module induced by the candidate $K$ in Example 21. Figure 3 shows the actions of $\text{sub}_0$, $\text{sub}_1$, and $\text{sub}_2$, respec-
tively, on the output tree in question. For readability, the context arguments carried through unchanged from the root are not depicted.

As one can see, neither $sub_0$ nor $sub_1$ would be an appropriate choice for $sub_u$, because they violate the requirement that $\pi_1$ is only reached by $sub_1$ and that $\pi_2$ is only reached by $sub_2$. The function $sub_2$, on the other hand, might be an appropriate choice to use as replacement for $sub$. But to be sure, we would have to perform the above check for every output tree of $sum'$, not just for a single one.

Checking the behavior of a $sub$-like module for all (potentially infinitely many) output trees seems to be a hopeless endeavor at first. However, there are only finitely many possible outcomes of the analysis for any tree: for each $sub_u$ and each $\pi_v$ one has to determine the subset of those $sub_0, \ldots, sub_n$ that can reach $\pi_v$ if computation at the root is started with $sub_u$. Since there are $2^{n+1}$ subsets of $\{sub_0, \ldots, sub_n\}$ and since for every $sub_u$ and $\pi_v$ with $u, v \in [0, n]$ one subset is calculated, we obtain $(2^{n+1})^{(n+1)\cdot(n+1)}$ possible outcomes of the analysis. In order to effectively compute the finitely many outcomes for the infinitely many inputs to $f'$, we abstract from each output tree computed by $f'$ to the corresponding outcome of the analysis.

In order to base our analysis directly on the original 1-mtt with the function $f$
rather than on the decomposed smodt with the function \( f' \), we use the statement (\(*) from the proof of Lemma 12 in Appendix D that for every \( t \in T_{C_p} \), \((f' \ t)\) computes the same output as \((f \ t \ \pi_1 \cdots \pi_n)\). In addition, we use that the \( \pi_1, \ldots, \pi_n \) are different from \( \pi_0 \) and that none of them is ever produced by \( f \) itself. Therefore, instead of analyzing which of the \( sub_0, \ldots, sub_n \) reach a \( \pi_v \) (with \( v \in \{0, n\} \)) when evaluating \((sub_u \ (f \ t \ \pi_1 \cdots \pi_n) \cdots)\), one can equivalently analyze which of them reach \( \pi_0 \) or \( y_v \) (with \( v \in \{0, n\} \)) when evaluating \((sub_u \ (f \ t \ y_1 \cdots y_n) \cdots)\). This refines our task to determining the set of all “reachability functions” \( G : [0, n] \times [0, n] \rightarrow \mathcal{P}([0, n]) \). Here, a function \( G \) is a reachability function if there is a \( t \in T_{C_p} \) such that \( G(u, v) \) describes those \( sub_u \) which reach \( y_v \) when evaluating \((sub_u \ (f \ t \ y_1 \cdots y_n) \cdots)\). More precisely, for every \( u \in [0, n] \) we must have:

- for \( v \in \{n\} \), \( G(u, v) \) contains exactly those \( u' \in [0, n] \) where the \( v \)-th context argument \( y_v \) is reached by \( sub_u \) when evaluating \((sub_u \ (f \ t \ y_1 \cdots y_n) \cdots)\), and

- \( G(u, 0) \) contains exactly those \( u' \in [0, n] \) where \( \pi_0 \) is reached by \( sub_u \) when evaluating \((sub_u \ (f \ t \ y_1 \cdots y_n) \cdots)\).

The idea now is to compute the set of all these reachability functions \( G \) for trees of increasing height. Let \( G_0 \) denote the set of all reachability functions \( G \) for trees of height \( \leq h \). Clearly, we have \( G_0 = \emptyset \). In order to compute \( G_{h+1} \), note that the output produced by evaluating \((f \ t \ y_1 \cdots y_n)\) with \( \text{height}(t) = h + 1 \) is determined by evaluating an instance of \( rhs_{p,f,c} \), where \( c \) is the root symbol of \( t \). Further note that in every recursive call of \( f \) in \( rhs_{p,f,c} \), \( f \)'s first argument will be instantiated by some tree of height \( \leq h \). So to compute \( G_{h+1} \), we perform the above “reachability analysis” on all right-hand sides \( rhs_{p,f,c} \) and for recursive calls in \( rhs_{p,f,c} \), we draw on information from \( G_h \). More precisely, if \( c \) has rank \( k \), we consider every choice of functions \( G_1, \ldots, G_k \) from \( G_h \) to provide reachability information for calls of the form \((f \ x_1 \cdots), \ldots, (f \ x_k \cdots)\).

Formally, we use a function \( \text{rch}_{G_1, \ldots, G_k} \). Given a right-hand side \( \tilde{r} \) and a pair of values \( u, v \in [0, n] \), it describes those functions among \( sub_0, \ldots, sub_n \) which reach \( y_v \) (if \( v \neq 0 \)) resp. \( \pi_0 \) (if \( v = 0 \)) when evaluating an instance of \((sub_u \ \tilde{r} \cdots)\). In this instance, the variables \( x_1, \ldots, x_k \) in \( f \)'s recursion arguments may only be instantiated by trees whose corresponding reachability functions are \( G_1, \ldots, G_k \), respectively. Thus, for every recursive call \((f \ x_1 \cdots)\) in \( \tilde{r} \), we assume that \( G_i \) describes the result of the reachability analysis for \( x_i \). Then \( G_{h+1} \) can be computed by collecting \( \text{rch}_{G_1, \ldots, G_k} (rhs_{p,f,c}) \) for all \( c \in C(p) \) and all choices for \( G_1, \ldots, G_k \in G_h \). The definition of \( \text{rch}_{G_1, \ldots, G_k} (\tilde{r})(u, v) \) (formalized in Definition 23 below) is by induction on the structure of \( \tilde{r} \). We start with the base cases.

If \( \tilde{r} = \pi_0 \), then in instances of \((sub_u \ \pi_0 \cdots)\), \( \pi_0 \) can only be reached by \( sub_u \) itself and hence, \( \text{rch}_{G_1, \ldots, G_k} (\pi_0)(u, 0) = \{u\} \). Moreover, none of the context argu-
ments \(y_1, \ldots, y_n\) of \(f\) can be reached by any \(\text{sub}_w\) and hence, \(\text{rch}_{G_1,\ldots,G_k}(\pi_0)(u, v) = \emptyset\) for every \(v \in [n]\).

If \(\bar{r} = y_j \in \mathcal{Y}_n\), then in instances of \(\text{(sub}_u y_j \cdots\), \(y_j\) can only be reached by \(\text{sub}_u\) itself, while neither \(\pi_0\) nor any of the \(y_1, \ldots, y_n\) other than \(y_j\) can be reached with any \(\text{sub}_w\). (Note that only variables in \(f\)'s recursion arguments may be instantiated, so \(y_j\) stays unchanged.) So we obtain \(\text{rch}_{G_1,\ldots,G_k}(y_j)(u, j) = \{u\}\) and \(\text{rch}_{G_1,\ldots,G_k}(y_j)(u, v) = \emptyset\) for every \(v \in [0, n] - \{j\}\).

In the first recursive case, let \(\bar{r} = (c \ r_1 \cdots r_a)\) for a constructor \(c \in C_p^{(a)}\) other than \(\pi_0\). As mentioned, \(\text{rch}_{G_1,\ldots,G_k}(c \ r_1 \cdots r_a)(u, v)\) should describe those functions among \(\text{sub}_0, \ldots, \text{sub}_n\) which reach \(y_v\) (or \(\pi_0\), if \(v = 0\)) when evaluating an instance of \(\text{(sub}_u (c \ r_1 \cdots r_a) \cdots)\). Due to the definition of the \(\text{sub}\)-like module induced by \(K\), the first evaluation step yields (a corresponding instance of)

\[
c \ (\text{sub}_{K(u, c, 1)} r_1 \cdots) \ (\text{sub}_{K(u, c, a)} r_a \cdots).
\]

Thus, by simply collecting the results of the reachability analysis for \(r_1, \ldots, r_a\), \(\text{rch}_{G_1,\ldots,G_k}(c \ r_1 \cdots r_a)(u, v)\) is defined as

\[
\text{rch}_{G_1,\ldots,G_k}(r_1)(K(u, c, 1), v) \cup \cdots \cup \text{rch}_{G_1,\ldots,G_k}(r_a)(K(u, c, a), v).
\]

In the other recursive case, we have \(\bar{r} = (f \ x_i r_1 \cdots r_n)\). Our goal is to describe those functions among \(\text{sub}_0, \ldots, \text{sub}_n\) which reach \(y_v\) (or \(\pi_0\), if \(v = 0\)) when evaluating an instance of \(\text{(sub}_u (f \ x_i r_1 \cdots r_n) \cdots)\), assuming that \(x_i\) is instantiated by a tree \(t'\) whose reachability function is \(G_i\). To properly collect, in a similar way as in the previous case, the reachability information recursively determined for \(r_1, \ldots, r_n\), we first need to know which of the \(\text{sub}_0, \ldots, \text{sub}_n\) will reach each \(r_i\) when evaluating \(\text{(sub}_u (f \ t' r_1 \cdots r_n) \cdots)\). This, of course, depends on the tree \(t'\) that \(x_i\) is instantiated with. However, we do not need to know that actual tree. Rather, the function \(G_i\) corresponding to \(x_i\), as carried along by \(\text{rch}_{G_1,\ldots,G_k}\), provides all necessary information. If, for example, that function \(\overline{G_i}\) maps \((u, 1)\) to a set containing \(u_1\), then we know that \(\text{sub}_{u_1}\) reaches \(r_1\), and hence (among others) we have to include \(\text{rch}_{G_1,\ldots,G_k}(r_1)(u_1, v)\). In a similar way, we have to proceed for \(r_2, \ldots, r_n\). So \(\text{rch}_{G_1,\ldots,G_k}(f \ x_i r_1 \cdots r_n)(u, v)\) must include the union of all \(\text{rch}_{G_1,\ldots,G_k}(r_i)(u_l, v)\) with \(l \in [n]\) and \(u_l \in G_i(u, l)\).

Further elements are only needed in the case \(v = 0\), when we have to determine all functions among \(\text{sub}_0, \ldots, \text{sub}_n\) which can reach \(\pi_0\) while evaluating \(\text{(sub}_u (f \ t' r_1 \cdots r_n) \cdots)\). In addition to those \(\pi_0\) which are contributed by the context arguments \(r_1, \ldots, r_n\), we then also need to account for those occurrences of \(\pi_0\) that would already be produced by the call \((f \ t' y_1 \cdots y_n)\). The necessary reachability information is again simply drawn from the function \(G_i\).

Note that the indexing of the \(\text{rch}\)-function with \(G_1, \ldots, G_k\) ensures that several recursive calls of \(f\) with the same recursion argument \(x_i\) in the same right-hand side always use the same \(G_i\).
Definition 23 (successful candidate)

Let \( p \in P \) and \((m, f, x_1, \pi_0 \cdots \pi_0)\) be a 1-\(m\)tt of \( p \), where \( F_m = \{ f^{(n+1)} \}\) and \( \pi_0 \in C_p^{(0)} \). Let \( K \) be a candidate for \( p \) of rank \( n + 1 \). For every \( h \in \mathbb{N} \) we define a set \( G_h \) of functions of type \([0, n] \times [0, n] \rightarrow \mathcal{P}([0, n])\) as follows:

\[
G_0 = \emptyset
\]

\[
G_{h+1} = \bigcup_{c \in C^{(e)}_p} \{ rch_{G_1, \ldots, G_k}((rhs_p, f, c)) \mid G_1, \ldots, G_k \in G_h \},
\]

where for every \( k \in \mathbb{N} \), \( \bar{r} \in \text{RHS}([f], C_p, X_k, Y_n) \), and functions \( G_1, \ldots, G_k \) of the above type, the function \( rch_{G_1, \ldots, G_k}(\bar{r}) \) of that type is obtained by case analysis on \( \bar{r} \). With \( y_j \in Y_n, c \in (C_p - \{ \pi_0 \})^{(a)}, x_i \in X_k, \) and \( r_l \in \text{RHS}([f], C_p, X_k, Y_n) \) for \( l \in \mathbb{N} \), it maps arguments \( u, v \in [0, n] \) to a result in \( \mathcal{P}([0, n]) \) as follows:

\[
rch_{G_1, \ldots, G_k}(\pi_0)(u, v) = \begin{cases} 
\{u\} & \text{if } v = 0 \\
\emptyset & \text{otherwise}
\end{cases}
\]

\[
rch_{G_1, \ldots, G_k}(y_j)(u, v) = \begin{cases} 
\{u\} & \text{if } v = j \\
\emptyset & \text{otherwise}
\end{cases}
\]

\[
rch_{G_1, \ldots, G_k}(c r_1 \cdots r_n)(u, v) = \bigcup_{l \in [a]} rch_{G_1, \ldots, G_k}(r_l)(K(u, c, l), v)
\]

\[
rch_{G_1, \ldots, G_k}(f x_1 r_1 \cdots r_n)(u, v) = \bigcup_{l \in [a]} \bigcup_{u' \in G_i, (u, l)} rch_{G_1, \ldots, G_k}(r_l)(u', v)
\]

\[
\cup \{ G_i(u, 0) \mid \text{if } v = 0 \}
\]

\[
\cup \{ \emptyset \mid \text{otherwise.} \}
\]

For some \( u \in [0, n] \) we say that the candidate \( K \) is successful for \((m, f, x_1, \pi_0 \cdots \pi_0)\) with \( \text{sub}_u \) if for every \( G \in \bigcup_{h \in \mathbb{N}} G_h \) and \( v \in [0, n] \):

\[
G(u, v) \subseteq \{v\}.
\]

Note that each (except the first) set in the sequence \( \emptyset, G_1, G_2, \ldots \) is computed in exactly the same way only from the previous one. This means that if some \( G_h \) and \( G_{h+1} \) are equal, then every further set in the sequence is also equal to them. Moreover, it is easy to see that \( \emptyset \subseteq G_1 \subseteq G_2 \subseteq \cdots \) because the operation computing \( G_{h+1} \) from \( G_h \) preserves set inclusion. Since there are only finitely many functions of type \([0, n] \times [0, n] \rightarrow \mathcal{P}([0, n])\), this implies that the fixpoint \( G_h = G_{h+1} \) is definitely reached. Then, we have actually computed the infinite union \( \bigcup_{h \in \mathbb{N}} G_h \) in finitely many iterations. Hence, the success or failure of a candidate can be decided effectively.

Example 24 (establishing success of the candidate from Ex. 21)

Let \( p_{\text{nat}} \) be a program with \( C_{p_{\text{nat}}} = \{ +^{(2)}, S^{(1)}, 0^{(0)} \} \) and let \( K \) be the candidate
from Example 21. Assume that $p_{snat}$ contains the module $m_{snat, sum}$ from Example 18. For completeness, $m_{snat, sum}$ is extended by an equation which handles the case when sum is applied to a $+$-term, e.g.

\[
\text{sum } (x_1 + x_2) y_1 y_2 = y_2.
\]

Then, $(m_{snat, sum}, r_{snat})$ with $r_{snat} = (\text{sum } x_1 0 0)$ is a 1-\text{mtt} of $p_{snat}$.

We have $\pi_0 = 0$. Since $G_0 = \emptyset$, we obtain

\[
G_1 = \bigcup_{c \in C_{G_0}^{(0)}} \{ \text{rch} (\text{rhs}_{p_{snat}, \text{sum}, c}) \} = \{ \text{rch} (y_2) \}.
\]

Using a representation of functions of type $\{0, 1, 2\} \times \{0, 1, 2\} \rightarrow P(\{0, 1, 2\})$ by tables, similar to Example 21, we get

\[
G_1 = \begin{pmatrix}
0 & 1 & 2 \\
0 & \text{rch}(y_2)(0, 0) & \text{rch}(y_2)(1, 0) & \text{rch}(y_2)(2, 0) \\
1 & \text{rch}(y_2)(0, 1) & \text{rch}(y_2)(1, 1) & \text{rch}(y_2)(2, 1) \\
2 & \text{rch}(y_2)(0, 2) & \text{rch}(y_2)(1, 2) & \text{rch}(y_2)(2, 2)
\end{pmatrix}
\]

\[
= \begin{pmatrix}
0 & 1 & 2 \\
0 & \emptyset & \emptyset & \emptyset \\
1 & \emptyset & \emptyset & \emptyset \\
2 & \{0\} & \{1\} & \{2\}
\end{pmatrix} = \{G\}.
\]

The fact that the single function $G$ in $G_1$ returns $\{0\}$ for the input $(0, 2)$ means that $\text{sub}_0$ is the only function which can reach $y_2$ when evaluating the term $(\text{sub}_0 y_2 \cdots)$.

In the next iteration, we have

\[
G_2 = \{ \text{rch} (\text{rhs}_{p_{snat}, \text{sum}, 0}) \} \\
\cup \{ \text{rch}_{G_1} (\text{rhs}_{p_{snat}, \text{sum}, S}) \mid G_1 \in G_1 \} \\
\cup \{ \text{rch}_{G_1, G_2} (\text{rhs}_{p_{snat}, \text{sum}, +}) \mid G_1, G_2 \in G_1 \} \\
= G_1 \cup \{ \text{rch}_{G_1} (\text{sum } x_1 (S y_1) (y_1 + y_2)), \text{rch}_{G, G}(y_2) \}.
\]

Obviously, $\text{rch}_{G, G}(y_2) = G$. So it remains to calculate $\text{rch}_{G} (\text{sum } x_1 (S y_1) (y_1 + y_2))$. We only show the calculation of a single entry in the table representing
that function:

\[
\text{rch}_{G}(\text{sum } x_1 (S y_1) (y_1 + y_2))(2, 1) = \bigcup_{u' \in G(2, 1)} \text{rch}_{G}(S y_1)(u', 1) \\
\bigcup_{u' \in G(2, 2)} \text{rch}_{G}(y_1 + y_2)(u', 1) \\
= \emptyset \cup \text{rch}_{G}(y_1 + y_2)(2, 1) \\
= \text{rch}_{G}(y_1)(K(2, +, 1), 1) \\
\cup \text{rch}_{G}(y_2)(K(2, +, 2), 1) \\
= \{K(2, +, 1)\} \cup \emptyset = \{1\}.
\]

This means that \( \text{sub}_1 \) is the only function which can reach \( y_1 \) when evaluating \((\text{sub}_2 (\text{sum } x_1 (S y_1) (y_1 + y_2)) \cdots)\) with \( x_1 \) instantiated by a tree of height \( \leq 1 \).

Calculating also the other entries leads to the following:

\[
G_2 = G_1 \cup \begin{bmatrix}
0 & 1 & 2 \\
0 & \emptyset & \emptyset & \emptyset \\
1 & \{0\} & \{2\} & \{1\} \\
2 & \{0\} & \{1\} & \{2\}
\end{bmatrix}
\]

The next iterations give

\[
G_3 = G_2 \cup \begin{bmatrix}
0 & 1 & 2 \\
0 & \emptyset & \emptyset & \emptyset \\
1 & \{0\} & \{1, 2\} & \{1\} \\
2 & \{0\} & \{1\} & \{2\}
\end{bmatrix}
\]

and \( G_4 = G_3 \). Thus, we have reached a fixpoint. Checking the three functions produced so far, it is now easy to see that \( K \) is successful for \((m_{\text{snat}}, \text{sum}, r_{\text{snat}})\) with \( \text{sub}_2 \) (but neither with \( \text{sub}_0 \) nor with \( \text{sub}_1 \)). The reason is that for every \( G \in G_3 \) and \( v \in \{0, 1, 2\} \) we have \( G(2, v) \subseteq \{v\} \).

In which sense the \( \text{sub} \)-like module induced by a successful candidate performs the required substitutions is made precise in the following key lemma. The proof in Appendix D is the technically most challenging one in this paper.

**Lemma 25 (key to the advanced deaccumulation transformation)**

Let \( p \in P, (m, r) \) be a 1-mtt of \( p \), where \( r = (f x_1 \pi_0 \cdots \pi_0) \) with \( F_m = \{f^{(n+1)}\} \)
and \( \pi_0 \in C_p^{(0)} \). Further, let \( K \) be a candidate for \( p \) of rank \( n + 1 \) and \( p' \in P \) be a program containing (at least) the module \( m \) and the sub-like module induced by \( K \) and \( \pi_0 \). For every \( u \in [0, n] \) such that \( K \) is successful for \( (m, r) \) with \( \sub_u \), for every \( t \in T_{C_p} \) and \( s_1, \ldots, s_n \in T_{C_{p'} \cup F_p} \):

\[
\nf_{p'}(f \ t \ s_1 \cdots s_n) = \nf_{p'}(\sub_u (f \ t \ \pi_0 \cdots \pi_0) \ s_1 \cdots s_n).
\]

The previous lemma carries the essence of the new transformation to be proposed now, similarly to the role that Lemma 9 played for decomposition. Indeed, Transformation 26 bears strong resemblance to Transformation 10, extending even to their correctness proofs.

**Transformation 26 (advanced deaccumulation)**

Let \( p \in P \) and \( (m, r) \) be a 1-mtt of \( p \), where \( r = (f \ x_1 \ \pi_0 \cdots \pi_0) \) with \( F_m = \{ f^{(n+1)} \} \) and \( \pi_0 \in C_p^{(0)} \). Let \( K \) be a candidate for \( p \) of rank \( n + 1 \), such that \( K \) is successful for \( (m, r) \) with some \( \sub_u \in \{ \sub_0, \ldots, \sub_n \} \). We construct a program \( p' \in P \) which results from \( p \) by adding the modules \( m_1 \) and \( m_2 \), defined below. Then, \( (m_1, m_2, r') \) is an ivf nmodtt of \( p' \), where \( r' \) is also defined below. Let \( f' \in (F - F_p)^{(1)} \) and \( \sub_0, \ldots, \sub_n \in (F - F_p)^{(n+1)} \) such that \( f', \sub_0, \ldots, \sub_n \) are pairwise distinct.

1. For every \( c \in C_p^{(k)} \) and every equation \( f \ (c \ x_1 \cdots x_k) \ y_1 \cdots y_n = \rsh_s f, c \) in \( m \), the module \( m_1 \) contains \( f' \ (c \ x_1 \cdots x_k) = \adv (\rsh_s f, c) \), where

   \[
   \adv : \rsh_s \{f\}, C_p, X_k, Y_n \rightarrow \rsh_s \{f'\} \cup \{\sub_u\}, X_k, Y_n
   \]

   \[
   \adv(f \ x_i \ r_1 \cdots r_n) = \sub_u (f' \ x_i) \adv(r_1) \cdots \adv(r_n),
   \]

   for all \( i \in [k], r_1, \ldots, r_n \in \rsh_s \{f\} \cup \{\sub_u\}, X_k, Y_n \)

   \[
   \adv(c' \ r_1 \cdots r_n) = c' \adv(r_1) \cdots \adv(r_n),
   \]

   for all \( c' \in C_p^{(a)}, r_1, \ldots, r_n \in \rsh_s \{f\}, X_k, Y_n \)

   \[
   \adv(y_j) = \pi_0, \quad \text{for all} \ j \in [n].
   \]

2. \( m_2 \) is the sub-like module induced by \( K \) and \( \pi_0 \).

3. \( r' = (f' \ x_1) \).

The following theorem is proved by essentially "recycling" the proof of Lemma 12, except for using Lemma 25 instead of Lemma 9. For the statements proved in the simultaneous induction, see Appendix D.

**Theorem 27 (semantic correctness of advanced deaccumulation)**

For \( p, (m, r), p', (m_1, m_2, r') \) as in Transformation 26, for every \( t \in T_{C_p} \):

\[
\nf_p(r'[x_1 \leftarrow t]) = \nf_{p'}(r'[x_1 \leftarrow t]).
\]

33
To experiment with advanced deaccumulation, we have implemented the analysis from Definition 23 in Haskell. The implementation generates all candidates and for each of them it performs the fixpoint computation to decide its success or failure. The outcome for the input programs discussed at the beginning of Section 3.2 is reported in the following.

Example 28 (advanced deaccumulation for Example 18) Let $p_{\text{snat}}$ be a program with $C_{p_{\text{snat}}} = \{ +^{(2)}, S^{(1)}, 0^{(0)} \}$. Assume that $p_{\text{snat}}$ contains the module $m_{\text{snat}, \text{sum}}$ from Example 18, for completeness’ sake again extended by the equation

$$\text{sum} (x_1 + x_2) y_1 y_2 = y_2.$$

Among the $3^{3\times3} = 19683$ candidates for $p_{\text{snat}}$ of rank 3, our implementation finds exactly 729 successful candidates for the 1-mtt $(m_{\text{snat}, \text{sum}}, \text{sum} x_1 0 0)$ of $p_{\text{snat}}$. Each of them is successful with (and only with) sub$_2$. Moreover, they all agree on the boldface entries in the table given in Example 21. Indeed, the 729 = $3^6$ successful candidates arise exactly from all possible choices for the non-boldface entries in that table. Choosing (randomly) the particular candidate $K$ given in Example 21, and performing advanced deaccumulation based on it, the transformed program contains an ivf nmott featuring the final set of equations given in Example 18, the additional equation

$$\text{sum}' (x_1 + x_2) = 0,$$

the equations for sub$_0$, sub$_1$, and sub$_2$ given in Example 21, and the initial call $(\text{sum'} x_1)$.

Example 29 (advanced deaccumulation for variation of Ex. 18)

Let $p_{\text{snat}'}$ be a program with $C_{p_{\text{snat}'} } = \{ +^{(2)}, S^{(1)}, 0^{(0)} \}$, containing the following module, cf. the discussion after Example 18:

$$m_{\text{snat}', \text{sum}} : \begin{array}{l}
\text{sum} (S x_1) y_1 y_2 = \text{sum} x_1 (y_1 + (S 0)) (y_1 + y_2) \\
\text{sum} 0 y_1 y_2 = y_2 \\
\text{sum} (x_1 + x_2) y_1 y_2 = y_2
\end{array}$$

Among the $3^{3\times3} = 19683$ candidates for $p_{\text{snat}'}$ of rank 3, our implementation finds exactly 243 successful candidates for the 1-mtt $(m_{\text{snat}', \text{sum}}, \text{sum} x_1 0 0)$ of $p_{\text{snat}'}$. Each of them is successful with (and only with) sub$_2$, and is obtained by arbitrarily filling the empty positions in one of the following tables with values from $\{ 0, 1, 2 \}$:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(+,1)</td>
<td>1</td>
<td>1</td>
<td>(+,1)</td>
</tr>
<tr>
<td>(+,2)</td>
<td>0</td>
<td>2</td>
<td>(+,2)</td>
</tr>
<tr>
<td>(S,1)</td>
<td>0</td>
<td>(S,1)</td>
<td>0</td>
</tr>
</tbody>
</table>
Choosing the successful candidate corresponding to the first table filled up with 0-entries, and performing advanced deaccumulation based on it, the transformed program contains an ivf nmodtt featuring the following equations:

\[
\text{sum'} (S \ x_1) = \text{sub}_2 \ (\text{sum'} \ x_1) (0 + (S \ 0)) (0 + 0)
\]

\[
\text{sum'} 0 = 0
\]

\[
\text{sum'} (x_1 + x_2) = 0
\]

\[
\text{sub}_2 (x_1 + x_2) \ y_1 \ y_2 = (\text{sub}_1 \ x_1 \ y_1 \ y_2) + (\text{sub}_2 \ x_2 \ y_1 \ y_2) \quad \text{sub}_0 \ 0 \ y_1 \ y_2 = 0
\]

\[
\text{sub}_1 (x_1 + x_2) \ y_1 \ y_2 = (\text{sub}_1 \ x_1 \ y_1 \ y_2) + (\text{sub}_0 \ x_2 \ y_1 \ y_2) \quad \text{sub}_1 \ 0 \ y_1 \ y_2 = y_1
\]

\[
\text{sub}_0 (S \ x_1) \ y_1 \ y_2 = S (\text{sub}_0 \ x_1 \ y_1 \ y_2) \quad \text{sub}_2 \ 0 \ y_1 \ y_2 = y_2
\]

\[
\text{sub}_0 (x_1 + x_2) \ y_1 \ y_2 = (\text{sub}_0 \ x_1 \ y_1 \ y_2) + (\text{sub}_0 \ x_2 \ y_1 \ y_2)
\]

\[
\text{sub}_1 (S \ x_1) \ y_1 \ y_2 = S (\text{sub}_0 \ x_1 \ y_1 \ y_2)
\]

\[
\text{sub}_2 (S \ x_1) \ y_1 \ y_2 = S (\text{sub}_0 \ x_1 \ y_1 \ y_2)
\]

and the initial call \((\text{sum'} \ x_1)\).

**Example 30 (advanced deaccumulation for Example 19)** Let \(p_{\text{string}}\) be a program with \(C_{p_{\text{string}}} = \{A^{(1)}, B^{(1)}, E^{(0)}\}\). Assume that \(p_{\text{string}}\) contains the module \(m_{\text{string}, f}\) from Example 19. Among the \(2^{2^2} = 16\) candidates for \(p_{\text{string}}\) of rank 2, our implementation finds exactly the following four successful candidates for the 1-mtt \((m_{\text{string}, f}, f \ x_1 \ E)\) of \(p_{\text{string}}\):

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A, 1)</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>(B, 1)</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Each of them is successful with (and only with) \(\text{sub}_1\). Choosing the first successful candidate and performing advanced deaccumulation based on it, the transformed program contains an ivf nmodtt featuring the final set of equations given in Example 19, the equations

\[
\text{sub}_0 (B \ x_1) \ y_1 = B (\text{sub}_0 \ x_1 \ y_1)
\]

\[
\text{sub}_1 (B \ x_1) \ y_1 = B (\text{sub}_0 \ x_1 \ y_1)
\]

and the initial call \((f' \ x_1)\).

Thus, Transformation 26 successfully and systematically performs the deaccumulation tasks that could only be solved with an ad-hoc analysis at the beginning of Section 3.2. Of course, advanced deaccumulation does not necessarily succeed for every input function: it fails if the original 1-mtt admits no \(\text{sub}-like\)
module that can perform the required substitutions after the $\pi_0, \ldots, \pi_n$ have been equalized.

**Example 31 (possible failure of advanced deaccumulation)** Consider a program $p_{\text{fail}}$ to compute $\lceil \frac{x}{2} \rceil$, consisting of the module

\[
m_{\text{fail, div}}: \quad \text{div} (S \ x_1) \ y_1 \ y_2 = \text{div} \ x_1 \ (S \ y_2) \ y_1
\]

\[
\text{div} \ 0 \quad y_1 \ y_2 = y_1
\]

with initial call (\text{div} \ x_1 \ 0 \ 0). Among the $3^{3 \times 1} = 27$ candidates for $p_{\text{fail}}$ of rank 3, our implementation does not find a single successful one for the 1-\mtt ($m_{\text{fail, div}}, \text{div} \ x_1 \ 0 \ 0$) of $p_{\text{fail}}$. The intuitive reason is that the proper placement of context arguments in the output of $\text{div}$ cannot be determined solely from the shape of that output. More precisely, when called with the substitution constructors $\pi_1$ and $\pi_2$ as context arguments, $\text{div}$ may produce the output $(S \ \pi_2)$, for input $(S \ 0)$, as well as the output $(S \ \pi_1)$, for input $(S \ (S \ 0))$. These outputs have identical shape, but differ in the substitution constructor found at the leaf. Hence, in contrast to Examples 18 and 19 (and the variation of Example 18 considered in Example 29), here it is impossible to provide substitution-like functions that could properly decide, e.g., whether the leaf of $(S \ 0)$ is to be interpreted as an actual 0, as a $\pi_1$, or as a $\pi_2$.

4 Related Work

Program transformation is a well-established field in software engineering and compiler construction (see, e.g., [4,11,45,46]). There has also been a considerable amount of work on introducing accumulating arguments (see, e.g., [5,6,27,28,37,55,60]). While most of these transformations aim at increasing efficiency, we have explored a novel application area for program transformations by applying them in order to increase verifiability. This goal often runs counter to the classical aim of increasing efficiency, since a more efficient program is usually harder to verify. In particular, while composition techniques [18,21,35,37,40,59] from the theory of tree transducers can be applied to improve the efficiency of functional programs [36,38,56,58], we have demonstrated that also the corresponding decomposition techniques are not only of theoretical interest. Indeed, “inverting” existing transformation techniques seems to be a useful starting point in general to find transformations which increase verifiability. However, these inverted transformations may still have to be refined significantly in order to actually solve verification problems, as seen in our deaccumulation technique, where decomposition had to be combined with appropriate constructor replacement techniques.

Program transformations that improve verifiability have rarely been investigated before. A first step into this direction was taken in [23]. There, two
transformations were presented that can remove accumulators. They are based on associativity and commutativity properties of auxiliary functions like + occurring in accumulating arguments. The advantage of the approach in [23] is that it does not require the strict syntactic restrictions of 1-mtts. In particular, it does not require that functions from other modules may not be called in right-hand sides. Because of that restriction, in the present paper, we have to treat all auxiliary functions like + as constructors and exclude the use of any information about these functions during the transformation. On the other hand, the technique of [23] can essentially only remove one accumulating argument (e.g., in contrast to our method, it cannot eliminate both accumulators of \( p_{\text{acc}} \)). Moreover, the approach in [23] heavily relies on knowledge about auxiliary functions like +. Hence, it is not applicable if the contexts of accumulating arguments on the right-hand sides of equations are not associative or commutative. Thus, in contrast to our technique, it fails on examples like the following program \( p_{\text{exp}} \):

\[
\begin{align*}
\text{exp} (S \ x_1) \ y_1 &= \text{exp} \ x_1 (\text{exp} \ x_1 \ y_1) \\
\text{exp} 0 \ y_1 &= S \ y_1
\end{align*}
\]

The initial call is \((\text{exp} \ x_1 \ 0)\). We want to prove

\[
\text{exp} \ x_1 \ 0 = e \ x_1,
\]

where \((e \ (S^n \ 0))\) computes \((S^{2^n} \ 0)\), see below. Here, \((S^{z_1} \ 0) + (S^{z_2} \ 0)\) is assumed to compute \((S^{z_1+z_2} \ 0)\).

\[
\begin{align*}
e (S \ x_1) &= (e \ x_1) + (e \ x_1) \\
e 0 &= S \ 0
\end{align*}
\]

Since \( \text{exp} \) is a 1-mtt that is ncd, basic deaccumulation delivers the program:

\[
\begin{align*}
\text{exp'} (S \ x_1) &= \text{sub} (\text{exp'} \ x_1) (\text{sub} (\text{exp'} \ x_1) \ 0) \quad \text{sub} (S \ x_1) \ y_1 = S (\text{sub} \ x_1 \ y_1) \\
\text{exp'} 0 &= S \ 0 \\
\text{sub} 0 \ y_1 &= y_1
\end{align*}
\]

and the initial call \((\text{exp'} \ x_1)\), which are better suited for induction provers because there are no accumulating arguments anymore. For instance, instead of proving the above claim for the original program (which would require an ad-hoc generalization), now the statement

\[
\text{exp'} \ x_1 = e \ x_1
\]

can be proved automatically. We only show the induction step \((x_1 \mapsto (S \ x_1))\).

Note that the statements about substitution functions in Lemma 6 are often helpful for the verification of transformed programs (cf. also the examples in Appendices A and B). These statements require no extra proof effort, since they can be generated automatically during program transformation. Further
generic statements about substitution functions, depending only on the set of constructors but not on the accumulative function to be transformed, and how they can reduce verification effort, were discussed in [58].

\[
\begin{align*}
\text{exp}' (S \ x_1) \\
= \text{sub } (\text{exp}' \ x_1) (\text{sub } (\text{exp}' \ x_1) \ 0) \\
= \text{sub } (e \ x_1) (\text{sub } (e \ x_1) \ 0) \\
= \text{sub } (e \ x_1) (e \ x_1) \\
= (e \ x_1) + (e \ x_1) \\
= (S \ x_1)
\end{align*}
\]

The above example also demonstrates that, in contrast to [23], our technique can handle nested recursion. Indeed, deaccumulation is useful for functional programs in general (cf. also Appendix A, where the original program contains a recursive call with surrounding context) — not just for tail-recursive functions resulting from translating imperative programs.

5 Conclusions and Directions for Future Work

Conjectures about imperative programs and accumulative functional programs are hard to verify with induction theorem provers. The reason is that their proofs often require sophisticated generalizations which are difficult to find automatically. Therefore, we have introduced an automatic technique that transforms accumulative functions (for example, but not only, obtained by translating imperative programs) into non-accumulative ones, whose verification is usually significantly easier with existing proof tools.

While in many examples generalizations can be avoided by our technique, it does not render generalization techniques superfluous. There are accumulative functions where our transformations are not applicable, and even if they are, there are still conjectures that can only be proved via a suitable generalization. However, even then deaccumulation is advantageous because the generalizations for the transformed functions are usually much easier than the ones required for the original accumulative functions (cf. Appendix A).

An obvious direction for future work is to develop a transformation that subsumes both our basic and advanced deaccumulation techniques. Currently, basic deaccumulation requires the nullary constructors acting as initial values to be pairwise distinct and not to occur in right-hand sides of the relevant function definition. In contrast, advanced deaccumulation requires them to be all equal and poses no restriction on their occurrence in right-hand sides.
Instead, one might want to handle the general case that the initial values are arbitrary nullary constructors, where some (but not necessarily all) of them may be equal, and there is no restriction on right-hand sides. This is possible with a program analysis very much in the spirit of Definition 23, but complicated by more technicalities. Solely for the sake of accessibility of the key ideas, we restricted ourselves to the case of initial values being all equal here.

An interesting topic for future work is to couple the transformations directly with an induction theorem prover. To this end, we are working on a corresponding extension of the verification tool AProVE [25]. Moreover, also the current implementation of the presented fixpoint computation in Haskell certainly leaves room for improvement, even though it already uses some implementation tricks like integrating the success condition into the iterative computation to allow an early abort for non-successful candidates. Fortunately, the search space does not necessarily have to be explored in full. At least for the 1-mtt from Example 18 and its variation, each of the successful candidates reported in Examples 28 and 29 turns out to be equally suitable to automatically solve verification problems similar to that from the introduction (cf. Appendix C). Hence, the search process can be stopped once the first successful candidate is found. On the other hand, if there is not a single successful candidate for some 1-mtt, then a complete exploration is still necessary to detect this.

To improve the asymptotic complexity of the fixpoint computation, it is possible to simplify the domain for abstract interpretation (as implicitly used in Definition 23). For example, instead of a set of reachability functions, it would be possible to maintain only a single “superposed” function throughout the iteration process, and/or instead of arbitrary subsets of \{0, \ldots, n\}, one could allow only empty and singleton sets as function values, signaling non-success as soon as a set with at least two elements is produced. While these approximations might lead to some successful candidates being overlooked, correctness of those candidates being recognized as successful would still be guaranteed. And at least for the 1-mtts from Examples 18 and 19 (and the mentioned variation of Example 18), it turns out that successful candidates can be found even after the proposed modifications to the analysis process.

To increase the applicability of our approach, it could be extended to more general forms of algorithms. An obvious choice would be to handle mutually recursive functions, i.e., general mtts rather than 1-mtts only. For basic deaccumulation, such a generalization was already given in [50], along with an implementation. For advanced deaccumulation, it also seems to be unproblematic, using different sets of reachability functions for the different functions in the mt to be transformed in order to assure maximal accuracy in the fixpoint computation.

For simplicity, we have assumed an untyped language throughout. When introducing types, one would have to generate several substitution functions for
the different types of arguments, even in the case of basic deaccumulation. An extension beyond mtts seems to be possible as well. For example, the requirement of flat patterns on left-hand sides may be relaxed. Further extensions include transformations based on a decomposition that only removes those context arguments from a function that are modified in recursive calls. Finally, it would also be interesting to see whether it is possible to incorporate the transformations of [23] into our approach.

**Acknowledgment.** We would like to thank the anonymous referees for their valuable comments which helped us to improve the presentation of our results.

**Appendices**

In the next three appendices, we illustrate the advantages of our contributions with additional examples. Appendices A and B show that the classical approach of finding suitable generalizations is extremely hard for conjectures containing several occurrences of an accumulative function. Here, deaccumulation helps to simplify the proof tasks substantially. (After the deaccumulation, the proof works without generalizations in Appendix B and in Appendix A, the required generalization is now very easy to find.) While Appendices A and B illustrate the use of the basic deaccumulation technique, Appendix C demonstrates the advantage of the advanced deaccumulation technique for verification tasks. Finally, Appendix D contains full proofs.

**A Example: Splitting Monadic Trees**

The program

\[
\begin{align*}
\text{split} (A \ x_1) & \ y_1 = A (\text{split} \ x_1 \ y_1) \\
\text{split} (B \ x_1) & \ y_1 = \text{split} \ x_1 (B \ y_1) \\
\text{split} \ N & \ y_1 = y_1
\end{align*}
\]

with initial call \((\text{split} \ x_1 \ N)\) maps a monadic tree with \(n_1\) and \(n_2\) occurrences of the unary constructors \(A\) and \(B\), respectively, to the tree \((A^{n_1} (B^{n_2} N))\) by accumulating the \(B\)'s in the context argument of \(\text{split}\). By basic deaccumulation it is transformed into the program

\[
\begin{align*}
\text{split}' (A \ x_1) & = A (\text{sub} (\text{split}' \ x_1) \ N) & \text{sub} (A \ x_1) & y_1 = A (\text{sub} \ x_1 \ y_1) \\
\text{split}' (B \ x_1) & = \text{sub} (\text{split}' \ x_1) (B \ N) & \text{sub} (B \ x_1) & y_1 = B (\text{sub} \ x_1 \ y_1) \\
\text{split}' \ N & = N & \text{sub} \ N & y_1 = y_1
\end{align*}
\]
with initial call \((split' x_1)\). If we want to prove the idempotence of the splitting operation, then the proof for the original program requires a generalization from

\[
split (split x_1 N) N = split x_1 N
\]
to

\[
split (split x_1 (b x_2)) (b x_3) = split x_1 (b (x_2 + x_3)),
\]

where \((b n)\) computes \((B^n N)\). Such a generalization is difficult to find. On the other hand,

\[
split' (split' x_1) = split' x_1
\]
can be proved automatically. In the first step case \((x_1 \mapsto (A x_1))\), Lemma 6(1) is used to infer the equality of \((sub (split' x_1) N)\) and \((split' x_1)\). In the second step case \((x_1 \mapsto (B x_1))\), a straightforward generalization step is required by identifying two common subexpressions in a proof subgoal. More precisely, by applying the induction hypothesis, the induction conclusion is transformed into the proof obligation

\[
split' (sub (split' x_1) (B N)) = sub (split' (split' x_1)) (B N).
\]

Now, the two underlined occurrences of \((split' x_1)\) are generalized to a fresh variable \(x\), and then the proof works by induction on \(x\).

B Example: Reversing Monadic Trees

Consider the program

\[
rev (A x_1) y_1 = rev x_1 (A y_1)
\]
\[
rev (B x_1) y_1 = rev x_1 (B y_1)
\]
\[
rev N y_1 = y_1
\]

with initial call \((rev x_1 N)\). By basic deaccumulation, it is transformed into the program

\[
rev' (A x_1) = sub (rev' x_1) (A N) \quad sub (A x_1) y_1 = A (sub x_1 y_1)
\]
\[
rev' (B x_1) = sub (rev' x_1) (B N) \quad sub (B x_1) y_1 = B (sub x_1 y_1)
\]
\[
rev' N = N \quad sub N y_1 = y_1
\]

with initial call \((rev' x_1)\). Taking into account that \(sub\) is just the concatenation function on monadic trees, the above programs correspond to the efficient and the inefficient reverse function, which have linear and quadratic
time-complexity in the size of the input tree, respectively. Thus, this example shows that the aim of our technique runs counter to the aim of many classical program transformations, i.e., the efficiency is decreased, but the suitability for verification is improved: If we want to show that the reverse of two concatenated lists is the concatenation of the reversed lists in exchanged order, then the proof of

\[ \text{rev} \left( \text{sub} \ x_1 \ x_2 \right) \ N = \text{sub} \left( \text{rev} \ x_2 \ N \right) \left( \text{rev} \ x_1 \ N \right) \]

again requires considerable generalization effort, whereas

\[ \text{rev}' \left( \text{sub} \ x_1 \ x_2 \right) = \text{sub} \left( \text{rev}' \ x_2 \right) \left( \text{rev}' \ x_1 \right) \]

can be proved by a straightforward induction on \(x_1\), exploiting statements (1) and (2) of Lemma 6.

C Example: Summing up Natural Numbers

In Examples 18 and 28, advanced deaccumulation was used to transform the program

\[
\begin{align*}
\text{sum} \ (S \ x_1) \ y_1 \ y_2 &= \text{sum} \ x_1 \ (S \ y_1) \ (y_1 + y_2) \\
\text{sum} \ 0 \ y_1 \ y_2 &= y_2
\end{align*}
\]

with initial call \((\text{sum} \ x_1 \ 0 \ 0)\) into the program

\[
\begin{align*}
\text{sum}' \ (S \ x_1) &= \text{sub} \ 2 \ (\text{sum}' \ x_1) \ (S \ 0) \ (0 + 0) \\
\text{sum}' \ 0 &= 0 \\
\text{sub} \ 2 \ (x_1 + x_2) \ y_1 \ y_2 &= (\text{sub} \ 1 \ x_1 \ y_1 \ y_2) + (\text{sub} \ 2 \ x_2 \ y_1 \ y_2) \quad \text{sub} \ 1 \ 0 \ y_1 \ y_2 = y_1 \\
\text{sub} \ 1 \ (S \ x_1) \ y_1 \ y_2 &= S \ (\text{sub} \ 1 \ x_1 \ y_1 \ y_2) \quad \text{sub} \ 2 \ 0 \ y_1 \ y_2 = y_2
\end{align*}
\]

with initial call \((\text{sum}' \ x_1)\), omitting some superfluous equations here. Our goal is to verify the equivalence of the original program and the following alternative specification of summing up natural numbers:

\[
\begin{align*}
\text{sum}_2 \ (S \ x_1) &= x_1 + (\text{sum}_2 \ x_1) \\
\text{sum}_2 \ 0 &= 0
\end{align*}
\]

The automatic proof of

\[ \text{sum} \ x_1 \ 0 \ 0 = \text{sum}_2 \ x_1 \]
fails in the induction step \((x_1 \mapsto (S \ x_1))\). For the deaccumulated program, however, the statement

\[ \text{sum'} \ x_1 = \text{sum}_2 \ x_1 \]

can be proved without any problems. We only give the induction step \((x_1 \mapsto (S \ x_1))\), omitting the simple base case \((x_1 = 0)\). For the left-hand side of the equation, we obtain

\[
\begin{align*}
\text{sum'} (S \ x_1) \\
= \text{sub}_2 (\text{sum'} \ x_1) (S \ 0) (0 + 0) \\
= \text{sub}_2 (\text{sum}_2 \ x_1) (S \ 0) (0 + 0) \quad (IH)
\end{align*}
\]

and for the right-hand side, we have

\[ \text{sum}_2 (S \ x_1) = x_1 + (\text{sum}_2 \ x_1). \]

So to finish the proof, we have to show the conjecture

\[ \text{sub}_2 (\text{sum}_2 \ x_1) (S \ 0) (0 + 0) = x_1 + (\text{sum}_2 \ x_1). \]

We again use induction, omitting the simple base case. In the step case, for the left-hand side we obtain

\[
\begin{align*}
\text{sub}_2 (\text{sum}_2 (S \ x_1)) (S \ 0) (0 + 0) \\
= \text{sub}_2 (x_1 + (\text{sum}_2 \ x_1)) (S \ 0) (0 + 0) \\
= (\text{sub}_1 x_1 (S \ 0) (0 + 0)) + (\text{sub}_2 (\text{sum}_2 \ x_1) (S \ 0) (0 + 0)) \\
= (\text{sub}_1 x_1 (S \ 0) (0 + 0)) + (x_1 + (\text{sum}_2 \ x_1)) \quad (IH)
\end{align*}
\]

and for the right-hand side, we have

\[ (S \ x_1) + (\text{sum}_2 (S \ x_1)) = (S \ x_1) + (x_1 + (\text{sum}_2 \ x_1)). \]

Since the second summands of the two resulting expressions are identical, it remains to show the conjecture

\[ \text{sub}_1 x_1 (S \ 0) (0 + 0) = S \ x_1. \]

Here, in the step case, we obtain

\[
\begin{align*}
\text{sub}_1 (S \ x_1) (S \ 0) (0 + 0) \\
= S (\text{sub}_1 x_1 (S \ 0) (0 + 0)) \\
= S (S \ x_1) \quad (IH),
\end{align*}
\]

which (together with the straightforward base case) proves the conjecture.
D Proofs

First, we prove an auxiliary lemma, which also serves to illustrate the principle of proof by simultaneous induction (cf., e.g., [18,22,56]).

Lemma 32 (auxiliary, actual vs. formal parameters) Let $p \in P$ and $(m, r)$ be a 1-mlt of $p$, where $F_m = \{f^{(n+1)}\}$. For every $t \in T_C$, and $s_1, \ldots, s_n \in T_{C_P \cup F_p}(Y_n)$:

$$nf_p(f \ t \ s_1 \ldots \ s_n) = nf_p(f \ t \ y_1 \ldots \ y_n)[y_j \leftarrow nf_p(s_j) \mid j \in [n]].$$

PROOF. We prove the following two statements by simultaneous induction, where the first coincides with the statement of the lemma:

(*) For every $t \in T_C$, and $s_1, \ldots, s_n \in T_{C_P \cup F_p}(Y_n)$:

$$nf_p(f \ t \ s_1 \ldots \ s_n) = nf_p(f \ t \ y_1 \ldots \ y_n)[y_j \leftarrow nf_p(s_j) \mid j \in [n]].$$

(**) For every $k \in \mathbb{N}$, $t_1, \ldots, t_k \in T_C$, $s_1, \ldots, s_n \in T_{C_P \cup F_p}(Y_n)$, and $\bar{r} \in RHS(\{f\}, C_p, X_k, Y_n)$:

$$nf_p(\bar{r}[x_i, y_j \leftarrow t_i, s_j \mid i \in [k], j \in [n]])$$
$$= nf_p(\bar{r}[x_i \leftarrow t_i \mid i \in [k]][y_j \leftarrow nf_p(s_j) \mid j \in [n]].$$

To prove (*) for $t = (c \ t_1 \ldots \ t_k)$ with $c \in C_p^{(k)}$ and $t_1, \ldots, t_k \in T_C$, under the assumption that (***) holds for $k$ and $t_1, \ldots, t_k$, we instantiate $\bar{r}$ in (***) to $rhs_{p,f,c}$. To prove (**) for $k \in \mathbb{N}$ and $t_1, \ldots, t_k \in T_C$, under the assumption that (*) holds for each of the $t_1, \ldots, t_k$, we perform an induction on the structure of $\bar{r}$, for fixed $s_1, \ldots, s_n \in T_{C_P \cup F_p}(Y_n)$. The cases $\bar{r} \in Y_n$ and $\bar{r} = (c \ r_1 \ldots r_n)$ for some $c \in C_p^{(a)}$ and $r_1, \ldots, r_n \in RHS(\{f\}, C_p, X_k, Y_n)$ are straightforward. The validity in the remaining case is proved as follows.

Case $\bar{r} = (f \ x_{i'} \ r_1 \ldots \ r_n)$ for some $x_{i'} \in X_k$ and $r_1, \ldots, r_n \in RHS(\{f\}, C_p, X_k, Y_n)$:

$$nf_p((f \ x_{i'} \ r_1 \ldots \ r_n)[x_i, y_j \leftarrow t_i, s_j \mid i \in [k], j \in [n]])$$
$$= (\text{by substitution, (*) for } t_{i'}, \text{ and the induction hypotheses for } r_1, \ldots, r_n)$$
$$nf_p(f \ t_{i'} \ y_1 \ldots \ y_n)[y_j' \leftarrow nf_p(r_j'[x_i \leftarrow t_i \mid i \in [k]][y_j \leftarrow nf_p(s_j) \mid j \in [n]]$$
$$\mid j' \in [n])$$
$$= (\text{by Lemma } 1(2))$$
$$nf_p(f \ t_{i'} \ y_1 \ldots \ y_n)[y_j' \leftarrow nf_p(r_j'[x_i \leftarrow t_i \mid i \in [k]) \mid j' \in [n]]$$
$$[y_j \leftarrow nf_p(s_j) \mid j \in [n]]$$
$$= (\text{by substitution and (*) for } t_{i'})$$
\[nf_p((f \ x_i \ r_1 \cdots r_n)[x_i \leftarrow t_i \mid i \in [k]])[y_j \leftarrow nf_p(s_j) \mid j \in [n]] \quad \Box\]

**Proof of Lemma 9.** The lemma is established by the following calculation:

\[\begin{align*}
nf_p(f \ t \ s_1 \cdots s_n) &= (\text{by Lemma 32}) \\
nf_p(f \ t \ y_1 \cdots y_n)[y_j \leftarrow nf_p(s_j) \mid j \in [n]] &= (\text{by Lemma 1(3), using that}) \\
nf_p(f \ t \ y_1 \cdots y_n) &= nf_p(f \ t \ y_1 \cdots y_n) \in TC_p(Y_n) \\
&\text{does not contain any of the } \pi_1, \ldots, \pi_n) \\
nf_p(f \ t \ y_1 \cdots y_n)[y_j \leftarrow \pi_j \mid j \in [n]][\pi_j \leftarrow nf_p(s_j) \mid j \in [n]] &= (\text{by Lemma 32}) \\
nf_p(f \ t \ \pi_1 \cdots \pi_n)[\pi_j \leftarrow nf_p(s_j) \mid j \in [n]] &= (\text{by Lemma 5}) \\
nf_p(\text{sub} (f \ t \ \pi_1 \cdots \pi_n) \ s_1 \cdots s_n) &\quad \Box
\end{align*}\]

**Proof of Lemma 12.** We prove the following two statements by simultaneous induction:

(1) For every \(t \in TC_p\): \(nf_p(f \ t \ \pi_1 \cdots \pi_n) = nf_p(f' t)\).

(2) For every \(k \in \mathbb{N}, t_1, \ldots, t_k \in TC_p\), and \(\bar{r} \in RHS(\{f\}, C_p, X_k, Y_n)\):

\[nf_p(\bar{r}[x_i; y_j \leftarrow t_i, \pi_j \mid i \in [k], j \in [n]]) = nf_p(\text{dec}(\bar{r})[x_i \leftarrow t_i \mid i \in [k]])\).

The first statement of the lemma then follows from (2) with \(k = 1, t_1 = t\), and \(\bar{r} = r\), taking into account that \(nf_p(r[x_1 \leftarrow t]) = nf_p(r[x_1, y_j \leftarrow t, \pi_j \mid j \in [n]])\) due to the facts that \(r \in RHS(\{f\}, C_p, X_1, Y_0)\) contains no \(y_j\) for any \(j \in [n]\), and that the equations defining \(f\) in module \(m\) of \(p\) were taken over to \(p'\). Regarding the second statement of the lemma, note that if \((m, r)\) is ncd, then there are pairwise distinct \(c_1, \ldots, c_n \in C_p^{(0)} = C_p^{(0)} - \{\pi_1, \ldots, \pi_n\}\) such that \(r = (f \ x_1 \ c_1 \cdots c_n)\) and \(c_1, \ldots, c_n\) do not occur in right-hand sides of the function definition in \(m\). Thus, in this case \(r' = (\text{sub} (f' x_1) c_1 \cdots c_n)\) and by the definition of the \text{dec}-function and by the form of the dummy equations that we add for \(f'\) at the new constructors \(\pi_1, \ldots, \pi_n\), it is clear that \(c_1, \ldots, c_n\) do not occur in right-hand sides of the function definition in \(m_1\).

Now we give the proof of (1) and (2). To prove (1) for \(t = (c \ t_1 \cdots t_k)\) with \(c \in C_p^{(k)}\) and \(t_1, \ldots, t_k \in TC_p\) under the assumption that (2) holds for \(k\) and \(t_1, \ldots, t_k\), we instantiate \(\bar{r}\) in (2) to \(rhs_{p', f, c}\) and use that \(rhs_{p', f' e} = \text{dec}(rhs_{p', f, c})\) by construction. To prove (2) for \(k \in \mathbb{N}\) and \(t_1, \ldots, t_k \in TC_p\) under the assumption that (1) holds for each of the \(t_1, \ldots, t_k\), we perform an induction on the structure of \(\bar{r}\). The cases \(\bar{r} \in Y_n\) and \(\bar{r} = (c \ r_1 \cdots r_n)\) for some \(c \in C_p^{(a)}\) and \(r_1, \ldots, r_n \in RHS(\{f\}, C_p, X_k, Y_n)\) are straightforward. The
validity in the remaining case is proved as follows.

Case \( \bar{r} = (f \ x_i \ r_1 \ldots r_n) \) for some \( x_i \in X_k \) and \( r_1, \ldots, r_n \in \text{RHS}(\{f\}, C_p, X_k, Y_n) \):

\[
\text{nf'}((f \ x_i \ r_1 \ldots r_n)[x_i, y_j \leftarrow t_i, \pi_j \mid i \in [k], j \in [n]]) = \text{nf'}(\text{sub} (f \ t_i \ \pi_1 \ldots \pi_n) \ r_1[x_i, y_j \leftarrow t_i, \pi_j \mid i \in [k], j \in [n])
\]

\[
\ldots
\]

\[
r_n[x_i, y_j \leftarrow t_i, \pi_j \mid i \in [k], j \in [n])
\]

= \(\text{nf'}(\text{sub} (f' \ t_i) \text{dec}(r_1)[x_i \leftarrow t_i \mid i \in [k]] \ldots \text{dec}(r_n)[x_i \leftarrow t_i \mid i \in [k]])\)

= \(\text{nf'}(\text{sub} (f' \ t_i) \text{dec}(r_1)[x_i \leftarrow t_i \mid i \in [k]])\)

\(\square\)

**PROOF of Lemma 16.** We prove the following two statements by simultaneous induction:

\((*)\) For every \( t \in T_{C_{p'}} \): \( \text{nf'}(f' \ t) = \text{nf'}(f' \ t)[c_j \leftarrow \pi_j \mid j \in [n]]\).

\((**)\) For every \( k \in \mathbb{N} \) and \( t_1, \ldots, t_k \in T_{C_{p'}} \), for every \( \bar{r} \in \text{RHS}(\{f'\}, C_{p'} - \{c_1, \ldots, c_n\} \cup \{\text{sub}\}, X_k, Y_0) \) that is in the image of \( \text{dec} \) from Transformation 10:

\[
\text{nf'}(\bar{r}[x_i \leftarrow t_i \mid i \in [k]]) = \text{nf'}(\bar{r}[x_i, \pi_j \leftarrow t_i, c_j \mid i \in [k], j \in [n])][c_j \leftarrow \pi_j \mid j \in [n]].
\]

The lemma is then established by the following calculation for every \( t \in T_{C_{p'}} \):

\[
\text{nf'}(r'x_1 \leftarrow t)
\]

= \(\text{by substitution}\)

\[
\text{nf'}(\text{sub} (f' \ t) c_1 \ldots c_n)
\]

= \(\text{by Lemma 5}\)

\[
\text{nf'}(f' \ t)[\pi_j \mid j \in [n]]
\]

= \(\text{by }(*)\)

\[
\text{nf'}(f' \ t)[c_j \leftarrow \pi_j \mid j \in [n]][\pi_j \leftarrow c_j \mid j \in [n]]
\]

= \(\text{by Lemma 1(3), using that } \text{nf'}(f' \ t) \in T_{C_{p'}} \) does not contain any of the \( \pi_1, \ldots, \pi_n \)

\[
\text{nf'}(f' \ t)[c_j \leftarrow c_j \mid j \in [n]]
\]

= \(\text{by Lemma 1(1)}\)

\[
\text{nf'}(f' \ t)
\]

= \(\text{by substitution}\)

\[
\text{nf'}(r''x_1 \leftarrow t)
\]

46
To prove (*) for \( t = (c \ t_1 \cdots t_k) \) with \( c \in C^{(k)}_\rho \) and \( t_1, \ldots, t_k \in T_{C^{(k)}_{\rho'}} \), under the assumption that (**) holds for \( k \) and \( t_1, \ldots, t_k \), we instantiate \( \bar{r} \) in (**) to \( \text{rhs}_{\rho', f', c} \) and use that \( \text{rhs}_{\rho', f', c} = \text{rhs}_{\rho', f', \bar{c}} \). By (**) we obtain an induction hypothesis for \( t_1, \ldots, t_k \), and we perform an induction on the structure of \( t \). The case \( \bar{r} \in \{ \pi_1, \ldots, \pi_n \} \) is straightforward, as is the case \( \bar{r} = (c r_1 \cdots r_n) \) for some \( c \in (C^{(k)}_\rho \setminus \{ \pi_1, \ldots, \pi_n, c_1, \ldots, c_n \})^{(a)} \) and some \( r_1, \ldots, r_n \in \text{RHS}(\{ f' \}, C^{(k)}_\rho \setminus \{ c_1, \ldots, c_n \} \cup \{ \text{sub} \}, X_k, Y_0) \) that are in the image of \( \text{dec} \). Since \( \bar{r} \) is restricted to be in the image of \( \text{dec} \), the only remaining case is given (and proved) as follows.

**Case \( \bar{r} = (\text{sub} \ (f' \ x_{i'}) r_1 \cdots r_n) \) for some \( x_{i'} \in X_k \) and \( r_1, \ldots, r_n \in \text{RHS}(\{ f' \}, C^{(k)}_\rho \setminus \{ c_1, \ldots, c_n \} \cup \{ \text{sub} \}, X_k, Y_0) \) that are in the image of \( \text{dec} \):**

\[
n_{f'}((\text{sub} \ (f' \ x_{i'}) r_1 \cdots r_n)[x_i \leftarrow t_i \mid i \in [k]]) = (\text{by substitution and Lemma 5})
\]
\[
n_{f'}(f' \ t_{i'})[\pi_j \leftarrow n_{f'}(r_j[x_i \leftarrow t_i \mid i \in [k]]) \mid j \in [n]] = (\text{by (*) for } t_{i'} \text{ and the induction hypotheses for the } r_1, \ldots, r_n)
\]
\[
n_{f'}(f' \ t_{i'})[c_j \leftarrow \pi_j \mid j \in [n]]
\]
\[
\quad \quad [\pi_j \leftarrow n_{f'}(r_j[x_i, \pi_{j'} \leftarrow t_i, c_{j'} \mid i \in [k], j' \in [n]])[c_{j'} \leftarrow \pi_{j'} \mid j' \in [n]]] \\
\quad \quad \mid j \in [n]
\]
\[
= (\text{by Lemma 1(3), using that } n_{f'}(f' \ t_{i'}) \in T_{C^{(k)}_\rho} \text{ does not contain any of the } \{ \pi_1, \ldots, \pi_n \})
\]
\[
n_{f'}(f' \ t_{i'})[c_j \leftarrow n_{f'}(r_j[x_i, \pi_{j'} \leftarrow t_i, c_{j'} \mid i \in [k], j' \in [n]])[c_{j'} \leftarrow \pi_{j'} \mid j' \in [n]]] \\
\quad \quad \mid j \in [n]
\]
\[
= (\text{by Lemma 1(2)})
\]
\[
n_{f'}(f' \ t_{i'})[c_j \leftarrow n_{f'}(r_j[x_i, \pi_{j'} \leftarrow t_i, c_{j'} \mid i \in [k], j' \in [n]]) \mid j \in [n]]
\]
\[
= (\text{by substitution and Lemma 5})
\]
\[
n_{f'}((\text{sub} \ (f' \ x_{i'}) r_1 \cdots r_n))[x_i, \pi_{j'} \leftarrow t_i, c_{j'} \mid i \in [k], j' \in [n]]
\]
\[
\quad \quad [c_{j'} \leftarrow \pi_{j'} \mid j' \in [n]] \quad \square
\]

**Definition 33 (nondeterministic tree substitution)** Let \( \Sigma \) be a ranked alphabet and \( V, V' \) be sets of variables, where \( \Sigma \cap (V \cup V') = \emptyset \). Let \( n \in \mathbb{N} \) and let \( \alpha_1, \ldots, \alpha_n \in \Sigma^{(0)} \cup V \) be pairwise distinct, where \( \{ \alpha_1, \ldots, \alpha_n \} \supseteq V \). Then, for sets \( T_1, \ldots, T_n \subseteq T_{\Sigma}(V') \), the nondeterministic tree substitution \( \bullet [\alpha_1, \ldots, \alpha_n \leftarrow T_1, \ldots, T_n] \) (or \( \bullet [\alpha_i \leftarrow T_i \mid i \in [n]] \)) is a function mapping...
each tree from $T_{Σ}(V)$ to a set of trees from $T_{Σ}(V')$. It is defined as follows:

$$α_j[α_i ← T_i \mid i \in [n]] = T_j,$$

for all $j \in [n]$

$$(σ t_1 \cdots t_k)[α_i ← T_i \mid i \in [n]] = \{σ s_1 \cdots s_k \mid ∀j ∈ [k]. s_j ∈ t_j[α_i ← T_i \mid i \in [n]]\},$$

for all $σ ∈ (Σ - \{α_1, \ldots, α_n\})^k, t_1, ..., t_k ∈ T_{Σ}(V)$.

Note that substitution by $* \{α_i ← T_i \mid i ∈ [n]\}$ is independently nondeterministic for different occurrences of the same $α_i$. For example, $(σ x_1 x_1)[x_1 ← \{β, γ\}]$ does not only contain $(σ β β)$ and $(σ γ γ)$, but also $(σ β γ)$ and $(σ γ β)$.

**Lemma 34 (properties of nondeterministic tree substitutions)** Let $Σ$ be a ranked alphabet, $V$ be a set of variables disjoint from $Σ$, $n ∈ \mathbb{N}$, and $α_1, \ldots, α_n ∈ Σ^{(0)} ∪ V$ be pairwise distinct, where $V ⊆ \{α_1, \ldots, α_n\}$. For every $t ∈ T_Σ(V)$, $T ⊆ T_Σ$, finite set $J$, and $t_i, t'_i ∈ T_Σ(V)$ and $T_i, T'_i ⊆ T_Σ$ for every $i ∈ [n]$ and $j ∈ J$:

1. $t[α_i ← \{t_i\} \mid i ∈ [n]] = \{t[α_i ← t_i \mid i ∈ [n]]\}$,
2. $t[α_i ← t_i \mid i ∈ [n]][β, α' ← T, T' \mid i' ∈ [n]]$
   $$= t[β, α_i ← T, t_i[β, α' ← T, T' \mid i' ∈ [n]] \mid i ∈ [n]]$$
   for every $β ∈ Σ^{(0)} - \{α_1, \ldots, α_n\}$,
3. $t[α_i ← T_i \mid i ∈ [n]] ⊆ t[α_i ← T'_i \mid i ∈ [n]]$ if $T_i ⊆ T'_i$ for every $i ∈ [n]$, and
4. $∪_{j ∈ J} t[α_i ← T_{i,j} \mid i ∈ [n]] ⊆ t[α_i ← ∪_{j ∈ J} T_{i,j} \mid i ∈ [n]]$.

**PROOF.** Statements (1), (2), and (3) have straightforward proofs by induction on the structure of $t$. Statement (4) follows easily from statement (3) (by $T_{i,j} ⊆ ∪_{j ∈ J} T_{i,j}$).

**PROOF of Lemma 25.** First, consider Definition 33 and Lemma 34 above. For fixed $s_1, \ldots, s_n ∈ T_{C_p(∪F_p)}$, we will prove that for every $h ∈ \mathbb{N}$ and $t ∈ T_{C_p}$ with $height(t) ≤ h$ there is a $G ∈ G_h$ (where $G_h$ is defined as in Definition 23, based on $K$ and $π_0$) such that for every $v ∈ [0, n]$ and $θ_1, \ldots, θ_n ∈ T_{C_p(∪F_p)}$:

$$nf_p(f \ t \ θ_1 \cdots \ θ_n) \ s_1 \cdots s_n$$

$$∈ nf_p(f \ t \ y_1 \cdots y_n)π_0 ← \{π_0\} ∪ \{nf_p(s_{v'}) \mid v' ∈ [n] ∩ G(v, 0)\}, \quad \text{(D.1)}$$

$$y_j ← \{nf_p(\text{sub}_{v'} θ_j s_1 \cdots s_n) \mid v' ∈ G(v, j)\} \mid j ∈ [n].$$

Instantiating $v$ to $u$, setting $θ_1, \ldots, θ_n = π_0, \ldots, π_0$, and using that if $K$ is successful for $(m, r)$ with $\text{sub}_u$, then for every $G ∈ \bigcup_{h ∈ \mathbb{N}} G_h$ and $j ∈ [0, n]$,
for some $I_0, \ldots, I_n$ with $I_j \subseteq \{j\}$ for every $j \in [0, n]$. Using $[n] \cap I_0 = \emptyset$ and applying Lemma 34(3), this implies:

$$nf'_p(sub_u (f \ t \ \pi_0 \cdots \pi_0) \ s_1 \cdots s_n) \ \in \ nf'_p(f \ t \ y_1 \cdots y_n)[\pi_0 \leftarrow \{\pi_0\} \cup \{nf'_p(s_{v'}) \mid v' \in [n] \cap I_j\}, \ y_j \leftarrow \{nf'_p(sub_{v'} \ \pi_0 \ s_1 \cdots s_n) \mid v' \in I_j\} \mid j \in [n]]$$

from which the statement of the lemma follows by Lemma 32.

Now we prove (D.1) by induction on $h$. For $h = 0$ there is nothing to prove because there are no trees of height 0 or smaller. For the induction step ($h \mapsto h+1$) it suffices to show that for every $c \in C_p^{(k)}$ and $t_1, \ldots, t_k \in T_{C_p}$ with $\text{height}(t_i) \leq h$ for every $i \in [k]$, there is a $G' \in \mathcal{G}_{h+1}$ such that for every $v \in [0, n]$ and $\theta_1, \ldots, \theta_n \in T_{C_p \cup \{f\}}$:

$$nf'_p(sub_v (f \ c \ t_1 \cdots t_k) \ \theta_1 \cdots \theta_n) \ s_1 \cdots s_n) \ \in \ nf'_p(f \ c \ t_1 \cdots t_k) \ y_1 \cdots y_n)[\pi_0 \leftarrow \{\pi_0\} \cup \{nf'_p(s_{v'}) \mid v' \in [n] \cap G(v, 0)\}, \ y_j \leftarrow \{nf'_p(sub_{v'} \ \theta_1 \cdots \theta_n) \mid v' \in G(v, j)\} \mid j \in [n]],$$

where by the induction hypothesis for $h$ we may assume that there are $G_1, \ldots, G_k \in \mathcal{G}_h$ such that for every $i \in [k]$, $v \in [0, n]$, and $\theta'_1, \ldots, \theta'_n \in T_{C_p \cup \{f\}}$:

$$nf'_p(sub_v (f \ t_i \ \theta'_1 \cdots \theta'_n) \ s_1 \cdots s_n) \ \in \ nf'_p(f \ t_i \ y_1 \cdots y_n)[\pi_0 \leftarrow \{\pi_0\} \cup \{nf'_p(s_{v'}) \mid v' \in [n] \cap G_i(v, 0)\}, \ y_j \leftarrow \{nf'_p(sub_{v'} \ \theta'_1 \cdots \theta'_n) \mid v' \in G_i(v, j)\} \mid j \in [n]].$$

By Definition 23, $\mathcal{G}_{h+1}$ contains the function $G = \text{rhh}_{G_1, \ldots, G_k}(\text{rhs}_{p, f, c})$ for the particular $G_1, \ldots, G_k$ assumed for (D.3). Hence, using that $\text{rhs}_{p', f, c} = \text{rhs}_{p, f, c}$, to establish (D.2) it suffices to show that for every $\bar{r} \in \text{RHS}(\{\bar{f}\}, C_p, X_k, Y_n)$,

$$G(u, j) \subseteq \{j\},$$

we obtain for every $t \in T_{C_p}$:

$$nf'_p(sub_u (f \ t \ \pi_0 \cdots \pi_0) \ s_1 \cdots s_n) \ \in \ nf'_p(f \ t \ y_1 \cdots y_n)[\pi_0 \leftarrow \{\pi_0\} \cup \{nf'_p(s_{v'}) \mid v' \in [n] \cap I_0\}, \ y_j \leftarrow \{nf'_p(sub_{v'} \ \pi_0 \ s_1 \cdots s_n) \mid v' \in I_j\} \mid j \in [n]]$$
\(v \in [0, n]\), and \(\theta_1, \ldots, \theta_n \in T_{C_p \cup \{f\}}:\)
\[
\begin{align*}
  nf_{\pi}(sub_\nu \bar{r}[x_i, y_j \leftarrow t_i, \theta_j \mid i \in [k], j \in [n]] s_1 \cdots s_n) \\
  \in nf_{\pi}(\bar{r}[x_i \leftarrow t_i \mid i \in [k]])[\pi_0 = \{\pi_0\} \cup \{nf_{\pi}(s_{w'}) \mid v' \in [n] \cap \text{RHS}_{G_1, \ldots, G_k}(\bar{r}(v, 0))\}, \\
  y_j = \{nf_{\pi}(sub_\nu \theta_j s_1 \cdots s_n) \mid v' \in \text{RHS}_{G_1, \ldots, G_k}(\bar{r}(v, j)) \} \mid j \in [n].
\end{align*}
\]

The proof is by induction on the structure of \(\bar{r}\) as follows.

Case \(\bar{r} = \pi_0\):
\[
\begin{align*}
  nf_{\pi}(sub_\nu \pi_0[x_i, y_j \leftarrow t_i, \theta_j \mid i \in [k], j \in [n]] s_1 \cdots s_n) \\
  = (\text{using the equation for } sub_\nu \text{ at } \pi_0 \text{ in the}) \\
  \text{sub-like module induced by } K \text{ and } \pi_0) \\
  \begin{cases}
    \pi_0 & \text{if } v = 0 \\
    nf_{\pi}(s_\nu) & \text{otherwise}
  \end{cases} \\
  \in (\text{by substitution and } \text{RHS}_{G_1, \ldots, G_k}(\pi_0)(v, 0) = \{v\}) \\
  nf_{\pi}(\pi_0[x_i \leftarrow t_i \mid i \in [k]])[\pi_0 = \{\pi_0\} \cup \{nf_{\pi}(s_{w'}) \mid v' \in [n] \cap \text{RHS}_{G_1, \ldots, G_k}(\pi_0)(v, 0))}, \\
  y_j = \{nf_{\pi}(sub_\nu \theta_j s_1 \cdots s_n) \mid v' \in \text{RHS}_{G_1, \ldots, G_k}(\pi_0)(v, j)) \} \mid j \in [n]
\end{align*}
\]

Case \(\bar{r} = y_j \in Y_n\):
\[
\begin{align*}
  nf_{\pi}(sub_\nu y_j'[x_i, y_j \leftarrow t_i, \theta_j \mid i \in [k], j \in [n]] s_1 \cdots s_n) \\
  \in (\text{by substitution and } \text{RHS}_{G_1, \ldots, G_k}(y_j')(v, j') = \{v\}) \\
  nf_{\pi}(y_j'[x_i \leftarrow t_i \mid i \in [k]])[\pi_0 = \{\pi_0\} \cup \{nf_{\pi}(s_{w'}) \mid v' \in [n] \cap \text{RHS}_{G_1, \ldots, G_k}(y_j')(v, 0))}, \\
  y_j = \{nf_{\pi}(sub_\nu \theta_j s_1 \cdots s_n) \mid v' \in \text{RHS}_{G_1, \ldots, G_k}(y_j')(v, j)) \} \mid j \in [n]
\end{align*}
\]

Case \(\bar{r} = (c \prod_{i=1}^{a} r_i) \prod_{i=1}^{b} c \prod_{i=1}^{a} r_i\) for some \(c \in (C_p - \{\pi_0\})^{(a)}\) and \(r_1, \ldots, r_a \in RHS(\{f\}, C_p, X_k, Y_n)\):
\[
\begin{align*}
  nf_{\pi}(sub_\nu (c \prod_{i=1}^{a} r_i)[x_i, y_j \leftarrow t_i, \theta_j \mid i \in [k], j \in [n]] s_1 \cdots s_n) \\
  = (\text{using the equation for } sub_\nu \text{ at } c \text{ in the}) \\
  \text{sub-like module induced by } K \text{ and } \pi_0) \\
  c \prod_{i=1}^{a} nf_{\pi}(sub_{K(v,c,l)} r_i[x_i, y_j \leftarrow t_i, \theta_j \mid i \in [k], j \in [n]] s_1 \cdots s_n) \cdots \\
  \in (\text{see below}) \\
  nf_{\pi}((c \prod_{i=1}^{a} r_i)[x_i \leftarrow t_i \mid i \in [k]]) \\
  [\pi_0 = \{\pi_0\} \cup \bigcup_{l \in [a]} \{nf_{\pi}(s_{w'}) \mid v' \in [n] \cap \text{RHS}_{G_1, \ldots, G_k}(r_l)(K(v, c, l), 0))}, \\
  y_j = \bigcup_{l \in [a]} \{nf_{\pi}(sub_\nu \theta_j s_1 \cdots s_n) \mid v' \in \text{RHS}_{G_1, \ldots, G_k}(r_l)(K(v, c, l), j)) \} \mid j \in [n]
\end{align*}
\]

(by definition of \(\text{RHS}_{G_1, \ldots, G_k}(c \prod_{i=1}^{a} r_i)\))

50
\[nf'_r((c \, r_1 \cdots r_a)[x_i \leftarrow t_i \mid i \in [k]])\]
\[\pi_0 \leftarrow \{\pi_0\} \cup \{nf'_r(s_{v'}) \mid v' \in [n] \cap \text{rgh}_{G_1, \ldots, G_k}(c \, r_1 \cdots r_a)(v, 0)\},\]
\[y_j \leftarrow \{nf'_r(sub_{v'} \, \theta_j \, s_1 \cdots s_n) \mid v' \in \text{rgh}_{G_1, \ldots, G_k}(c \, r_1 \cdots r_a)(v, j)\} \mid j \in [n]\]

The above gap can be closed if for every \(l' \in [a]\) we can establish:

\[nf'_r(\text{sub}_{K(v, c, \ell')} \, r_{l'}[x_i, y_j \leftarrow t_i, \theta_j \mid i \in [k], j \in [n]] \, s_1 \cdots s_n)\]
\[\in nf'_r(r_{l'}[x_i \leftarrow t_i \mid i \in [k]])\]
\[\pi_0 \leftarrow \{\pi_0\} \cup \bigcup_{l' \in [a]} \{nf'_r(s_{v'}) \mid v' \in [n] \cap \text{rgh}_{G_1, \ldots, G_k}(r_l)(K(v, c, l), 0)\},\]
\[y_j \leftarrow \bigcup_{l' \in [a]} \{nf'_r(sub_{v'} \, \theta_j \, s_1 \cdots s_n) \mid v' \in \text{rgh}_{G_1, \ldots, G_k}(r_l)(K(v, c, l), j)\} \mid j \in [n].\]

But this is a consequence of the induction hypothesis for \(r_{l'}\) and Lemma 34(3).

Case \(r = (f \, x_i \, r_1 \cdots r_n)\) for some \(x_i \in X_k\) and some \(r_1, \ldots, r_n \in \text{RHS}(\{f\}, C_p, X_k, Y_n)\):

\[nf'_r(f \, t_i \, y_1 \cdots y_n)\]
\[\pi_0 \leftarrow \{\pi_0\} \cup \{nf'_r(s_{v'}) \mid v' \in [n] \cap G_i(v, 0)\},\]
\[y_j \leftarrow \{nf'_r(sub_{v'} \, r_{j'}[x_i, y_j' \leftarrow t_i, \theta_{j'} \mid i' \in [k], j' \in [n]] \, s_1 \cdots s_n) \mid v' \in G_i(v, j)\} \mid j \in [n]\]
\[\subseteq \text{(by the induction hypotheses for the} \, r_1, \ldots, r_n \text{and Lemma 34(3))}\]

\[nf'_r(f \, t_i \, y_1 \cdots y_n)\]
\[\pi_0 \leftarrow \{\pi_0\} \cup \{nf'_r(s_{v'}) \mid v' \in [n] \cap G_i(v, 0)\},\]
\[y_j \leftarrow \bigcup_{v' \in G_i(v, j)} \{nf'_r(r_{j'}[x_i \leftarrow t_i] \mid i' \in [k])\}\]
\[\pi_0 \leftarrow \{\pi_0\} \cup \{nf'_r(s_{v''}) \mid v'' \in [n] \cap \text{rgh}_{G_1, \ldots, G_k}(r_j)(v', 0)\},\]
\[y'_j \leftarrow \{nf'_r(sub_{v''} \, \theta_{j'} \, s_1 \cdots s_n) \mid v'' \in \text{rgh}_{G_1, \ldots, G_k}(r_j)(v', j')\} \mid j' \in [n]\]
\[\subseteq \text{(by Lemma 34(4) and Lemma 34(3), twice)}\]
\[ n_f'(f, t_1, y_1 \ldots y_n) \]
\[ \{\pi_0 \leftarrow \{\pi_0\} \cup \{n_f'(s_{v'}) | v' \in [n] \cap G_i(v,0)\} \}, \]
\[ y_j \leftarrow n_f'(r_j[x_{1'} \leftarrow t_{1'} \ldots \nu' \in [k]] \]
\[ \{\pi_0 \leftarrow \{\pi_0\} \cup \bigcup_{v' \in G_i(v,j)} \{n_f'(s_{v'}) | v'' \in [n] \cap rch_{G_i, \ldots, G_k}(r_j)(v',0)\}, \]
\[ y_j' \leftarrow \bigcup_{l \in [n]} \bigcup_{v' \in G_i(v,l)} \{n_f'(sub_{v'} \theta_j' s_1 \ldots s_n) | v'' \in rch_{G_i, \ldots, G_k}(r_l)(v',j') \}
\]
\[ (\text{by Lemma 34(2)}) \]
\[ n_f'(f, t_1, y_1 \ldots y_n) \]
\[ [\pi_0 \leftarrow \{\pi_0\} \cup \{n_f'(s_{v'}) | v'' \in [n] \cap \bigcup_{l \in [n]} \bigcup_{v' \in G_i(v,l)} rch_{G_i, \ldots, G_k}(r_l)(v',0)\}], \]
\[ y_j' \leftarrow \bigcup_{l \in [n]} \bigcup_{v' \in G_i(v,l)} \{n_f'(sub_{v'} \theta_j' s_1 \ldots s_n) | v'' \in rch_{G_i, \ldots, G_k}(r_l)(v',j') \} \]
\[ (\text{by substitution, Lemma 32, and definition of } rch_{G_i, \ldots, G_k}(f, x_i r_1 \ldots r_n)) \]
\[ n_f'(f, x_i r_1 \ldots r_n) \]
\[ [\pi_0 \leftarrow \{\pi_0\} \cup \{n_f'(s_{v'}) | v'' \in [n] \cap rch_{G_i, \ldots, G_k}(f, x_i r_1 \ldots r_n)(v,0)\}], \]
\[ y_j' \leftarrow \{n_f'(sub_{v'} \theta_j' s_1 \ldots s_n) | v'' \in rch_{G_i, \ldots, G_k}(f, x_i r_1 \ldots r_n)(v,j') \}
\]
\[ \square \]

**PROOF of Theorem 27** By copying the simultaneous induction from the proof of Lemma 12, except for using Lemma 25 instead of Lemma 9, we can
prove the following two statements:

(∗) For every \( t \in T_{C_p} \): 
\[
\text{nf}_{p'}(f \cdot \pi_0 \cdot \pi_0) = \text{nf}_{p'}(f' \cdot t).
\]

(∗∗) For every \( k \in \mathbb{N} \), \( t_1, \ldots, t_k \in T_{C_p} \), and \( r \in \text{RHS}(\{f\}, C_p, X_k, Y_n) \):
\[
\text{nf}_{p'}(r[x_i, y_j \leftarrow t_i, \pi_0 | i \in [k], j \in [n]]) = \text{nf}_{p'}(\text{adv}(r)[x_i \leftarrow t_i | i \in [k]]).
\]

The theorem then follows from (∗) since \( r = (f \cdot \pi_0 \cdot \pi_0) \) and \( r' = (f' \cdot x_1) \). Here one has to take into account that \( \text{nf}_{p}(f \cdot \pi_0 \cdot \pi_0) = \text{nf}_{p'}(f' \cdot \pi_0 \cdot \pi_0) \) due to the fact that the module \( m \) of \( p \), defining \( f \), was taken over to \( p' \). \( \square \)

References


