Verification of the Redecoration Algorithm for Triangular Matrices

Ralph Matthes* and Martin Strecker

C. N. R. S. et Université Paul Sabatier (Toulouse III)
Institut de Recherche en Informatique de Toulouse (IRIT)
118 route de Narbonne, F-31062 Toulouse Cedex 9
{matthes,strecker}@irit.fr

December 21, 2007

Abstract. Triangular matrices with a dedicated type for the diagonal elements can be profitably represented by a nested datatype, i. e., a heterogeneous family of inductive datatypes. These families are fully supported since the version 8.1 of the Coq theorem proving environment, released in 2007. Redecoration of triangular matrices has a succinct implementation in this representation, thus giving the challenge of proving it correct. This has been achieved within Coq, using also induction with measures. An axiomatic approach allowed a verification in the Isabelle theorem prover, giving insights about the differences of both systems.

1 Introduction

Nested datatypes [9] may keep certain invariants (see also the illuminating [11]) even without employing a dependently-typed system where types may also depend on objects, thus e. g., maintaining size information in the types.

Redecoration for triangular matrices by means of a nested datatype has first been studied in the case of infinite triangles [2]. Its finitary version has been programmed iteratively in the subsequent journal version [3] and through primitive recursion in Mendler-style [1]. In all these cases, no attempt was made to verify properties other than termination.

We put forward the example of redecoration of triangular matrices as a prototypical situation, where nested datatypes yield concise and elegant programs that are verifiable. The price to pay is a more complex framework that is needed in order to formulate the programs and a more complex logical apparatus for verifying them. Moreover, as in all formal verification tasks, a major challenge is to develop an appropriate correctness criterion. We have chosen to give a very precise intuition about the algorithm. Even though this might satisfy the experienced programmer, we felt the need for a subsequent verification against a completely different model: a model that is just based on the ordinary type of lists and thus does not impose the aforementioned complex machinery.

* with financial support by the European Union FP6-2002-IST-C Coordination Action 510996 "Types for Proofs and Programs"

Since we chose not to use dependent types, the list-based model speaks about a too large datatype, namely "triangles" that may be quite degenerate. Nevertheless, we may fully profit from tool support for lists that is well-developed in interactive theorem provers. In this case study, we deployed the Coq and the Isabelle proof assistants. Coq has a very strong type system that is fully adequate for representing nested datatypes and reasoning about them. The latest release 8.1 of Coq explicitly supports nested datatypes: definition by pattern-matching, induction principles, ... Even though Isabelle, which is based on simply-typed lambda-calculus with type variables, does not accept nested datatypes as such, it permits to simulate essential aspects of the development in an axiomatic manner.

Two critical aspects can be ensured by the use of theorem provers such as the two systems under study: termination of the algorithm (a non-functional specification) and functional correctness with respect to a chosen correctness criterion, in our case the relation with the list-based model. The axiomatic approach that we had to use in Isabelle cannot ensure termination of the algorithms on nested datatypes and cannot justify their induction schemes from first principles. However, the development of the list-based model is entirely derivable from a small logical core.

The main challenge of the verification is to find the right lemmas that allow to get the inductive proof of the simulation theorem through. The whole explanations and the semi-formal development in standard mathematical style of the proof in the next three sections have only been possible with the aid of a proof engineering effort in the proof assistants. Simplification and rewriting are tasks that are error-prone for humans and where the tool support is particularly well-developed and helpful. In the light of two complete formalizations in two entirely independent systems, it does not seem necessary to reproduce such proof steps in the main part of this article. The curious reader is invited to consult the full proof scripts that are available online [12].

The article is structured as follows: Section 2 introduces the problem and gives the intuitive justification of redecoration for triangular matrices, viewed as a nested datatype. In Section 3, the list-based model is developed, against which the original model is verified in Section 4. Highlights of the formalizations in the proof assistants are presented in Section 5 for Coq and Section 6 for Isabelle. We conclude in Section 7.

2 Triangular Matrices

The "triangular matrices" of the present article are finite square matrices, where the part below the diagonal has been cut off. Equivalently, one may see them as symmetric matrices where the redundant information below the diagonal has been omitted. The elements on the diagonal play a different role than the other elements in many mathematical applications, e. g., one might require that the diagonal elements are invertible (non-zero). This is modeled as follows: A type E of elements outside the diagonal is fixed throughout, and there is a type of diagonal elements that enters all definitions as a parameter. More technically, if

A is the type of diagonal elements, then $\mathsf{Tri}\,A$ shall denote the type of triangular matrices with A's on the diagonal and E's outside. Then, Tri becomes a family of types, indexed over all types, hence a type transformation. Moreover, the different $\mathsf{Tri}\,A$ are inductive datatypes that are all defined simultaneously, hence they are an "inductive family of types" or "nested datatype".

We do not consider empty triangles here. So, the smallest element of $\mathsf{Tri}\,A$ contains a single element that thus is a diagonal element and hence taken from the type A. This is materialized by the datatype constructor

$$\mathsf{sg} : A \to \mathsf{Tri}\,A,$$

that constructs these "singletons". Here, A is meant to be a type variable, i. e., a variable of type Set, the universe of computational types. The reader may just conceive that sg were given the quantified type $\forall A.\, A \to \mathsf{Tri}\,A$.

The non-singleton case can be visualized like this:

$$\begin{array}{l|l} A & E\ E\ E\ E \\ & A\ E\ E\ E \\ & \quad A\ E\ E \\ & \quad\quad A\ E \\ & \quad\quad\quad A \\ & \quad\quad\quad\quad \ldots \end{array}$$

The vertical line cuts the triangle into one element of A and a "trapezium", with an uppermost row solely consisting of E's. One might now, for the purpose of having an inductive generation process, decompose that trapezium into the uppermost row and the triangle below, but it would be hard to keep the information that both have the same number of columns (unless making use of dependent types). The approach to be followed is the integration of the side diagonal (i. e., the elements just above the diagonal) into the diagonal. In this way, the trapeziums above are in one-to-one correspondence to triangles as follows:

$$\begin{array}{r} E\ E\ E\ E \\ A\ E\ E\ E \\ \quad A\ E\ E \\ \quad\quad A\ E \\ \quad\quad\quad A \\ \quad\quad \ldots \end{array} \qquad \begin{array}{c} \text{triangle} \\ \longrightarrow \\ \longleftarrow \\ \text{trapezium} \end{array} \qquad \begin{array}{llll} E \times A & E & E & E \\ & E \times A & E & E \\ & & E \times A & E \\ & & & E \times A \\ & & & \quad \ldots \end{array}$$

The trapezium to the left is then considered as the "trapezium view" of the triangle to the right. Vice versa, the triangle to the right is the "triangle view" of the trapezium to the left.

Since we are about to define what triangles are, it is now comfortable to refer to the triangle view of the trapezium in the second datatype constructor

$$\mathsf{constr} : A \to \mathsf{Tri}(E \times A) \to \mathsf{Tri}A\ ,$$

again with A a type variable. Hence, the non-singleton triangles are conceived to consist of the topmost leftmost element, taken from A, and a triangle with

diagonal elements taken from $E \times A$. Abbreviate $\mathsf{Trap}\, A := \mathsf{Tri}(E \times A)$. Therefore, $\mathsf{constr} : \forall A.\, A \to \mathsf{Trap}\, A \to \mathsf{Tri}\, A$, but this is just a point of view to refer to the trapezium view of the argument that "is" a triangle.

From sg and constr, the inductive family Tri is now fully defined as everything that is finitely generated from these two constructors. As is usual with nested datatypes, one cannot understand one specific family member $\mathsf{Tri}\, A$ for some type A in isolation, but the recursive structure will include $\mathsf{Tri}(E \times A)$, $\mathsf{Tri}(E \times (E \times A))$, ..., hence infinitely many family members (with indices of increasing size).

Naturally, also the induction principle for Tri cannot speak about one instance $\mathsf{Tri}\, A$ in isolation. The following induction principle is intuitively justified.[1] Given a predicate P that speaks about all triangles with all types of diagonal elements, i. e., $P : \forall A.\, \mathsf{Tri}\, A \to \mathsf{Prop}$, where Prop is the universe of propositions, the aim is to assure that P holds universally, i. e., $\forall A \forall t : \mathsf{Tri}\, A.\, P_A\, t$ holds true. (Here and everywhere, we suppress the type information that A is a type variable, and we write the type argument to P as a subscript.) An inductive proof of this universal statement now only requires a proof of the two following statements:

- $\forall A \forall a : A.\, P_A(\mathsf{sg}\, a)$.
- $\forall A \forall a : A \forall r : \mathsf{Trap}\, A.\, P_{E \times A}\, r \to P_A(\mathsf{constr}\, a\, r)$.

The inductive hypothesis $P_{E \times A}\, r$ refers to the instantiation of the predicate P with type argument $E \times A$. Except from this, the principle is no more difficult in nature than the induction principle for (homogeneous) lists.

The redecoration algorithm whose verification is the aim of this article can be described as the binding operation of a comonad[2]. In other words, we will organize the material in the form of a comonad for the type transformer Tri that might then be called the redecoration comonad. The function $\mathsf{top} : \forall A.\, \mathsf{Tri}\, A \to A$ that computes the top left element is programmed as follows:

$$\mathsf{top}(\mathsf{sg}\, a) := a \ , \qquad \mathsf{top}(\mathsf{constr}\, a\, r) := a \ .$$

This is a simple non-recursive instance of definition by pattern-matching. The function top will be the counit of the comonad that we are defining. Redecoration in general is dual to substitution, see [15]. Following this view, redecoration for triangular matrices will be defined as Kleisli coextension operation: Given types A, B and a function $f : \mathsf{Tri}\, A \to B$ – the "redecoration rule" – define the function $\mathsf{redec}\, f : \mathsf{Tri}\, A \to \mathsf{Tri}\, B$. In the formulation of [15], $\mathsf{Tri}\, A$ becomes the type of A-decorated structures. So, the redecoration rule f assigns B-decorations to A-decorated structures, and $\mathsf{redec}\, f$ "co-extends" this to an assignment of B-decorated structures to A-decorated structures.

The intuitive idea of redecoration in the case of triangles is to go recursively through the triangle and to replace each diagonal element by the result of applying f to the sub-triangle that extends to the right and below the diagonal element.

[1] A formal justification is provided by Coq, see section 5.

[2] No category-theoretic knowledge is required to follow the article. The laws are given in our concrete situation, but they are mentioned to be instances of the general comonad notions.

$$
\begin{array}{l}
\mathsf{redec} : \forall A \forall B.\, (\mathsf{Tri}\, A \to B) \to \mathsf{Tri}\, A \to \mathsf{Tri}\, B \ , \\
\mathsf{redec}\, f\ (\mathsf{sg}\, a) \qquad\quad := \mathsf{sg}(f\, (\mathsf{sg}\, a)) \ , \\
\mathsf{redec}\, f\ (\underbrace{\mathsf{constr}\, a\, r}_{t:=}) := \mathsf{constr}(f\, t)\, (\mathit{rest}(\mathsf{redec}\, f\, t)) \ .
\end{array}
$$

Here, $\mathit{rest}(\mathsf{redec}\, f\, t)$ is a meta-notation for the element of type $\mathsf{Tri}(E \times B)$ yet to be defined. However, already at this stage of definition, the first comonad law becomes apparent:

$$\mathsf{top}(\mathsf{redec}\, f\, t) = f\, t \ .$$

It remains to define $\mathit{rest}(\mathsf{redec}\, f\, t) : \mathsf{Tri}(E \times B)$ for $f : \mathsf{Tri}\, A \to B$ and $t = \mathsf{constr}\, a\, r$ with $a : A$ and $r : \mathsf{Tri}(E \times A)$. The type of r "is" a triangle, but, from the description of constr, r ought to be seen in trapezium view in order to follow the above intuition of redecoration. Going back to the illustration on page 3, the uppermost row is to be cut off, then the topmost A be replaced by f applied to the remaining triangle, and then redecoration has to be carried on recursively. Finally, the uppermost row has to be recovered.

First, define the operation cut that cuts off the top row from the trapezium view of its argument:

$$
\begin{array}{l}
\mathsf{cut} : \forall A.\, \mathsf{Trap}\, A \to \mathsf{Tri}\, A \ , \\
\mathsf{cut}(\mathsf{sg}\, (e, a)) \qquad\quad := \mathsf{sg}\, a \ , \\
\mathsf{cut}(\mathsf{constr}\, (e, a)\ r) := \mathsf{constr}\, a\, (\mathsf{cut}\, r) \ .
\end{array}
$$

Here, $(e, a) : E \times A$ denotes pairing. Note that r is of type $\mathsf{Trap}(E \times A)$ in the second clause. The definition principle is thus "polymorphic recursion" where the type parameter can change in the recursive calls. Since it is even a definition that exploits that the argument is not an arbitrary $\mathsf{Tri}\, A$, it goes beyond the iteration schemes proposed in [3].[3]

Note that, in the recursive equation for cut, no change of view is necessary since the arguments are always seen as trapeziums.

We will define $\mathit{rest}(\mathsf{redec}\, f\, t)$ just from f and $r : \mathsf{Trap}\, A$ where the latter might be called $\mathit{rest}\, t$. In "reality", r is no trapezium so that a recursive call to redec for r will need a dedicated redecoration rule for the trapezium view, to be obtained from the "original" redecoration rule f in (ordinary) triangle view:

$$
\begin{array}{l}
f' : \mathsf{Trap}\, A \to E \times B \ , \\
f'\, r := (\mathsf{fst}(\mathsf{top}\, r), f(\mathsf{cut}\, r)) \ ,
\end{array}
$$

with fst the left/first projection out of a pair. Note that the target type $E \times B$ of f' is the type parameter of $\mathsf{Tri}(E \times B) = \mathsf{Trap}\, B$. The left component of $f'\, r$

[3] Basically, that article only allows to define iterative functions of type $\forall A.\, \mathsf{Tri}\, A \to X\, A$ for some type transformation X. Through the use of syntactic Kan extensions for X, this can be relaxed somewhat, and an iterative function (called fcut' on page 49 of that article) with the more general type $\forall A \forall B.\, (B \to E \times A) \to \mathsf{Tri}\, B \to \mathsf{Tri}\, A$ had to be defined before instantiating it to cut by using the identity on $E \times A$ as the functional parameter (hence with $B := E \times A$).

instructs to keep the leftmost element of the uppermost row in the trapezium view. Note that the original definition of the right component of $f'\,r$ in [2] did not use a cut function but just lifted the second projection via a mapping function for Tri. Correct types do not suffice, and verification would have been welcome to exclude such an error.

The operation that associates f' with f is named lift, so $\mathsf{lift}\,f := f'$ and

$$\mathsf{lift} : \forall A \forall B.\, (\mathsf{Tri}\,A \to B) \to \mathsf{Trap}\,A \to E \times B.$$

The definition of redecoration is finished by setting

$$\mathit{rest}(\mathsf{redec}\,f\,t) := \mathsf{redec}\,(\mathsf{lift}\,f)\,r,$$

whence (without using the abbreviations), the recursive equation for redec becomes

$$\mathsf{redec}\,f\,(\mathsf{constr}\,a\,r) = \mathsf{constr}\,(f(\mathsf{constr}\,a\,r))\,(\mathsf{redec}\,(\mathsf{lift}\,f)\,r)\ \ ,$$

which is the equational form of the reduction behaviour established through Mendler recursion in earlier work [1].

A very typical phenomenon of nested datatypes are recursive functions that take an additional functional parameter – here the f – that is modified during the recursion.

The major question is now: Did we come up with the right definition?

By fairly easy inductive reasoning, using some auxiliary lemmas about cut and lift, the other two comonad laws can be established for the triple consisting of Tri, top and redec:[4]

- $\mathsf{redec}\,\mathsf{top}\,t = t$,
- $\mathsf{redec}\,(g \circ (\mathsf{redec}\,f))\,t \;=\; \mathsf{redec}\,g\,(\mathsf{redec}\,f\,t)$ for f, g, t of appropriate types, namely $f : \mathsf{Tri}\,A \to B$, $g : \mathsf{Tri}\,B \to C$, $t : \mathsf{Tri}\,A$. In general, $\circ$ denotes functional composition $\lambda g \lambda f \lambda x.\, g(f\,x)$, but is written in infix notation.

However, these laws and the textual description do not yet confirm a computational intuition that might have been formed through the experience with simpler datatypes such as lists. Therefore, we will set out to relate the behaviour of redec to a function redecL that does not involve nested datatypes but is based on just the ubiquitous datatype of lists.

3 A List-Based Model

We assume the type transformation List, where for any type A, the type of all finite lists with elements taken from A is $\mathsf{List}\,A$. Although it is also a family of inductive types, List is not a nested datatype since there is no relation between any $\mathsf{List}\,A$ and $\mathsf{List}\,B$ for $A \neq B$ in the *definition* of List. Clearly, such relations

[4] We also need that redec is extensional in its function argument, see the discussion in the implementation-related sections.

occur with the usual mapping function $\mathsf{map} : \forall A \forall B.\, (A \to B) \to \mathsf{List}\, A \to \mathsf{List}\, B$ that maps its function argument over all the elements of the second argument, but this is only after the definition of List.

The list-based representation of triangles is now a simply parameterized family of inductive types, defined explicitly by reference to List:

$$\mathsf{TriL}\, A := \mathsf{List}(\mathsf{List}\, E \times A) \ .$$

Any element of some $\mathsf{TriL}\, A$ is a finite list of "columns", and each column consists of the finite list of E elements above the diagonal and the A element on the diagonal. Note that the argument $\mathsf{List}\, E \times A$ of List has to be parenthesized to the left, i. e., as $(\mathsf{List}\, E) \times A$. We visualize an element of $\mathsf{TriL}\, A$ as a generalized triangle, with the A's still in the diagonal, but always the list of E's above each diagonal element, with the first element the farthest away from the diagonal. An example with 4 columns would be:

$$\begin{array}{cccc} E & & & E \\ E & & & E \\ A & & & E \\ & A & E & E \\ & & A & E \\ & & & A \end{array}$$

Triangularity is not expressed since, again, we do not want to make use of dependent types by which this could be controlled through the lengths of the E lists.

In order to relate statements about Tri and TriL, we define the "list representation" of triangles of type $\mathsf{Tri}\, A$ as elements of type $\mathsf{TriL}\, A$. Assume we want the representation for some $\mathsf{constr}\, a\, r$ with $r : \mathsf{Trap}\, A$, then a recursive call to the representation function would yield an element of $\mathsf{TriL}(E \times A)$. So, we would have to push out those E's within the diagonal elements to the E lists. The columnwise operation is thus:

$$\begin{array}{l} \mathsf{shiftToE} : \forall A.\, \mathsf{List}\, E \times\ (E \times A) \to \mathsf{List}\, E \ \times A \ , \\ \mathsf{shiftToE}(es, (e, a)) := (es + [e], a) \ , \end{array}$$

where $+$ is used to denote list concatenation and $[e]$ is the list that consists of just the element e, while the empty list will be denoted by $[\,]$, and the "cons" operation will be denoted by infix "::". The mapping with shiftToE changes from "triangle view" to "trapezium view" in the list-based representation:

$$\begin{array}{l} \mathsf{shiftToEs} : \forall A.\, \mathsf{TriL}\, (E \times A) \to \mathsf{TriL}\, A \ , \\ \mathsf{shiftToEs} := \mathsf{map}\, \mathsf{shiftToE} \ . \end{array}$$

The list representation of triangles is given iteratively as follows:

$$\begin{array}{l} \mathsf{toListRep} : \forall A.\, \mathsf{Tri}\, A \to \mathsf{TriL}\, A \ , \\ \mathsf{toListRep}(\mathsf{sg}\, a) \qquad := [([\,], a)] \ , \\ \mathsf{toListRep}(\mathsf{constr}\, a\, r) := ([\,], a) :: \mathsf{shiftToEs}\, (\mathsf{toListRep}\, r) \ . \end{array}$$

The intention is to define a notion of redecoration also for the list-based representation, i. e., an operation

$$\mathsf{redecL} : \forall A \forall B.\, (\mathsf{TriL}\, A \to B) \to \mathsf{TriL}\, A \to \mathsf{TriL}\, B \ .$$

However, there will be no proper comonad structure since no counit $\mathsf{topL} : \forall A.\, \mathsf{TriL}\, A \to A$ can exist: A could be instantiated by an empty type A_0, and $\mathsf{TriL}\, A_0$ would still not be empty since it contains $[\,]$.

As a preparation for the definition of redecL, more operations on columns are introduced that allow to cut off and restore the topmost E element:

$$\begin{aligned}
&\mathsf{removeTopE} : \forall A.\, \mathsf{List}\, E \times A \to \mathsf{List}\, E \times A \ , \\
&\mathsf{removeTopE}([\,], a) \quad := ([\,], a) \ , \\
&\mathsf{removeTopE}(e :: es, a) \ := (es, a) \ , \\
&\mathsf{singletonTopE} : \forall A.\, \mathsf{List}\, E \times A \to \mathsf{List}\, E \ , \\
&\mathsf{singletonTopE}([\,], a) \quad := [\,] \ , \\
&\mathsf{singletonTopE}(e :: es, a) := [e] \ , \\
&\mathsf{appendEs} : \forall A.\, \mathsf{List}\, E \to \mathsf{List}\, E \times A \to \mathsf{List}\, E \times A \ , \\
&\mathsf{appendEs}\ es\ (es', a) \quad := (es + es', a) \ .
\end{aligned}$$

For all pairs $p : \mathsf{List}\, E \times A$, one has $\mathsf{appendEs}(\mathsf{singletonTopE}\, p)(\mathsf{removeTopE}\, p) = p$. The technical problem here is just that the E list can be empty, and so there is the need for the list with at most one element.

These operations can be canonically extended to multiple columns. For one-place functions, this is done via map, for the two-place function appendEs, the generic zipWith function known from the Haskell programming language (see `www.haskell.org`) comes into play:

$$\begin{aligned}
&\mathsf{zipWith} : \forall A \forall B \forall C.\, (A \to B \to C) \to \mathsf{List}\, A \to \mathsf{List}\, B \to \mathsf{List}\, C \ , \\
&\mathsf{zipWith}\ f\ (a :: \ell_1)\ (b :: \ell_2) := f\, a\, b :: \mathsf{zipWith}\ f\ \ell_1\ \ell_2 \ , \\
&\mathsf{zipWith}\ f\ \ell_1\ [\,] \qquad := [\,] \ , \\
&\mathsf{zipWith}\ f\ [\,]\ \ell_2 \qquad := [\,] \ .
\end{aligned}$$

The last auxiliary definitions for redecL are:

$$\begin{aligned}
&\mathsf{removeTopEs} : \forall A.\, \mathsf{TriL}\, A \to \mathsf{TriL}\, A \ , \\
&\mathsf{removeTopEs} \ := \mathsf{map}\ \mathsf{removeTopE} \ , \\
&\mathsf{singletonTopEs} : \forall A.\, \mathsf{TriL}\, A \to \mathsf{List}(\mathsf{List}\, E) \ , \\
&\mathsf{singletonTopEs} := \mathsf{map}\ \mathsf{singletonTopE} \ , \\
&\mathsf{zipAppendEs} : \forall A.\, \mathsf{List}(\mathsf{List}\, E) \to \mathsf{TriL}\, A \to \mathsf{TriL}\, A \ , \\
&\mathsf{zipAppendEs} \ := \mathsf{zipWith}\ \mathsf{appendEs} \ .
\end{aligned}$$

The following definition is by wellfounded recursion over the $\mathsf{TriL}\, A$ argument of redecL. Note that removeTopEs does not change the list length of its argument and that therefore, it is just the list length of the TriL argument of redecL that is smaller in the recursive call.

$$\begin{aligned}
&\mathsf{redecL} : \forall A \forall B.\, (\mathsf{TriL}\, A \to B) \to \mathsf{TriL}\, A \to \mathsf{TriL}\, B \ , \\
&\mathsf{redecL}\ f\ [\,] \qquad := [\,] \ , \\
&\mathsf{redecL}\ f\ ((es, a) :: r) := (es, f((es, a) :: r)) :: \mathsf{zipAppendEs}\, (\mathsf{singletonTopEs}\, r) \\
&\qquad\qquad\qquad\qquad\qquad\qquad (\mathsf{redecL}\ f\ (\mathsf{removeTopEs}\, r)) \ .
\end{aligned}$$

In comparison with redec, this definition contains a new trivial case for the empty list, and the redecoration rule f does not need to be adapted to a trapezium view in the recursive call. Thus, f is just a fixed parameter throughout the recursion, hence, also the type parameters stay fixed. Due to the less rigid constraints on the form in TriL, there may be E's above the leftmost A. This parameter *es* is taken into account when evaluating the redecoration rule, but still, only the diagonal elements are modified by the algorithm.

4 Verification Against the List-Based Model

Theorem 1 (Simulation). *If E is non-empty, then for all types A, B, terms $t : \mathsf{Tri}\, A$ and $f : \mathsf{TriL}\, A \to B$:*

$$\mathsf{redecL}\, f\, (\mathsf{toListRep}\, t) = \mathsf{toListRep}(\mathsf{redec}\, (f \circ \mathsf{toListRep})\, t) \ .$$

This is the most natural theorem that relates redec and redecL through toListRep: If there were an operation topL to turn TriL and redecL into a comonad, this theorem would establish for toListRep one of the two properties of a comonad morphism from Tri to TriL. Unfortunately, it does not reduce redec to redecL or vice versa. The former direction seems already to be hampered by the need for a redecoration rule $f : \mathsf{TriL}\, A \to B$, hence with a much wider domain than prescribed for redec. However, this is not so due to the existence of a left inverse fromListRep of toListRep. As we will see in the main theorem at the end of this section, $\mathsf{redec}\, f\, t$ can be expressed in terms of redecL, toListRep and fromListRep.

For the proof of the simulation theorem, one has to replay cut and lift on the list representations: Define

$$\begin{aligned} &\mathsf{remsh} : \forall A.\, \mathsf{List}\, E \times (E \times A) \to \mathsf{List}\, E \times A \ , \\ &\mathsf{remsh} := \mathsf{removeTopE} \circ \mathsf{shiftToE} \ . \end{aligned}$$

Abbreviate $\mathsf{TrapL}\, A := \mathsf{TriL}(E \times A)$. Define

$$\begin{aligned} &\mathsf{cutL} : \forall A.\, \mathsf{TrapL}\, A \to \mathsf{TriL}\, A \ , \\ &\mathsf{cutL} := \mathsf{map}\ \mathsf{remsh} \ , \end{aligned}$$

where the operational intuition is just to put the argument in trapezium view and then to cut off the top row. This intuition is met thanks to the functor law for map stating preservation of composition, i. e., $\mathsf{map}\, (g \circ f)\, t = \mathsf{map}\, g\, (\mathsf{map}\, f\, t)$.

Lemma 1. $\mathsf{toListRep}(\mathsf{cut}\, r) = \mathsf{cutL}(\mathsf{toListRep}\, r)$ *for all A and terms $r : \mathsf{Trap}\, A$.*

Proof. This is by induction on Trap, hence a section of Tri. The induction principle is as follows: Given $P : \forall A.\, \mathsf{Trap}\, A \to \mathsf{Prop}$, one concludes its universality, i. e., $\forall A \forall t : \mathsf{Trap}\, A.\, P_A\, t$ from the following two clauses:

- $\forall A \forall a : E \times A.\, P_A(\mathsf{sg}\, a)$.
- $\forall A \forall a : E \times A \forall r : \mathsf{Trap}\, (E \times A).\, P_{E \times A}\, r \to P_A(\mathsf{constr}\, a\, r)$.

It should be as intuitive as the Tri induction principle. For formal justifications, see the later sections.

The inductive step of the lemma will need (for $r : \mathsf{TrapL}(E \times A)$)

$$\mathsf{shiftToEs}\,(\mathsf{cutL}\,r) = \mathsf{cutL}\,(\mathsf{shiftToEs}\,r) \ ,$$

that in turn follows from (for $r : \mathsf{List}\,E \times (E \times (E \times A))$)

$$\mathsf{shiftToE}(\mathsf{remsh}\,r) = \mathsf{remsh}(\mathsf{shiftToE}\,r)$$

and the above-mentioned functor law for map. □

The analogue liftL of lift can only be defined for non-empty E. We will assume some fixed e_0 of type E in the sequel. Define

$$\begin{array}{l} \mathsf{liftL} : \forall A \forall B.\, (\mathsf{TriL}\,A \to B) \to \mathsf{TrapL}\,A \to E \times B \ , \\ \mathsf{liftL}\,f\,[\,] \qquad\qquad\qquad\quad := (e_0, f(\mathsf{cutL}\,[\,])) \ , \\ \mathsf{liftL}\,f\,(\underbrace{(es, (e, a)) :: r'}_{r :=}) := (e, f(\mathsf{cutL}\,r)) \ . \end{array}$$

The following relation between lift and liftL is a consequence of the preceding lemma.

Lemma 2. $\mathsf{lift}\,(f \circ \mathsf{toListRep})\,r = \mathsf{liftL}\,f\,(\mathsf{toListRep}\,r)$ *for types* A, B *and terms* $f : \mathsf{TriL}\,A \to B$ *and* $r : \mathsf{Trap}\,A$. □

The major obstacle on the way to proving the theorem is the following lemma.

Lemma 3 (Main Lemma). *For any types* A, B *and terms* $f : \mathsf{TriL}\,A \to B$ *and* $r : \mathsf{TrapL}\,A$*, one has*

$$\mathsf{shiftToEs}\,(\mathsf{redecL}\,(\mathsf{liftL}\,f)\,r) = \mathsf{zipAppendEs}\,\ell_1\,\ell_2$$

with

$$\begin{array}{l} \ell_1 := \mathsf{singletonTopEs}(\mathsf{shiftToEs}\,r) : \mathsf{List}(\mathsf{List}\,E) \ , \\ \ell_2 := \mathsf{redecL}\,f\,(\mathsf{removeTopEs}(\mathsf{shiftToEs}\,r)) : \mathsf{TriL}\,B \ . \end{array}$$

Note that $r : \mathsf{TrapL}\,A$, but that r is nevertheless just a (generalized) triangle. Hence $\mathsf{liftL}\,f$ is the right redecoration rule that treats it in trapezium view. Redecoration will nevertheless produce a (generalized) triangle, so the result is finally transformed into trapezium view. On the right-hand side, r is first made into a (generalized) trapezium, then redecoration is done to the result after cutting off the top row, but then the cut off elements are restored, hence the outcome is also a (generalized) trapezium. Note also that, as argued before, by virtue of the functor law for map, the argument to $\mathsf{redecL}\,f$ in ℓ_2 is equal to $\mathsf{cutL}\,r$.

Proof. The function redecL can be understood as being defined by recursion over the list length of its argument, and also the proof can be done by induction on the list length of r. See more details in the following specific sections on Coq and Isabelle how it is done more elegantly. □

Theorem 1 follows from the main lemma by induction on Tri for t. □

We want to define a left inverse to $\mathsf{toListRep}$. The type $\forall A.\, \mathsf{TriL}\, A \to \mathsf{Tri}\, A$ cannot be inhabited since $\mathsf{TriL}\, A$ is never empty while $\mathsf{Tri}\, A$ inherits emptiness from A. Hence, we will only be able to define a function

$$\mathsf{fromListRep} : \forall A.\, A \to \mathsf{TriL}\, A \to \mathsf{Tri}\, A$$

such that $\mathsf{fromListRep}\, a_0\, (\mathsf{toListRep}\, t) = t$ for all $a_0 : A$, $t : \mathsf{Tri}\, A$.

Recall that we have fixed an element e_0 of type E. The operation on columns is defined as

$$\begin{aligned}
&\mathsf{shiftFromE} : \forall A.\, \mathsf{List}\, E \times A \to \mathsf{List}\, E \times (E \times A)\ , \\
&\mathsf{shiftFromE}([\,], a) \quad := ([\,], (e_0, a))\ , \\
&\mathsf{shiftFromE}(e :: es, a) := (\mathsf{removelast}\, (e :: es), (\mathsf{last}\, (e :: es), a))\ ,
\end{aligned}$$

with functions $\mathsf{removelast}$ and last of the meaning suggested by their names. It is easy to establish that, for all pairs $p : \mathsf{List}\, E \times (E \times A)$, one has

$$\mathsf{shiftFromE}(\mathsf{shiftToE}\, p) = p\ .$$

This is extended to an operation on (generalized) triangles:

$$\begin{aligned}
&\mathsf{shiftFromEs} : \forall A.\, \mathsf{TriL}\, A \to \mathsf{TrapL}\, A\ , \\
&\mathsf{shiftFromEs} := \mathsf{map}\, \mathsf{shiftFromE}\ ,
\end{aligned}$$

and $\mathsf{shiftFromEs}(\mathsf{shiftToEs}\, r) = r$ for all $r : \mathsf{TrapL}\, A$ follows from the respective result on columns, the two functor laws for map (hence, also $\mathsf{map}\, (\lambda x.x)\, l = l$) and extensionality of map in its function argument.[5] The definition of $\mathsf{fromListRep}$ is by wellfounded recursion over the $\mathsf{TriL}\, A$ argument, and as for redecL, this is justified by the decrease of the list length of this argument in the recursive calls. However, unlike the situation of redecL, we need polymorphic recursion in that the function at type A calls itself at type $E \times A$:

$$\begin{aligned}
&\mathsf{fromListRep}\, a_0\, [\,] \qquad\qquad\qquad := \mathsf{sg}\, a_0\ , \\
&\mathsf{fromListRep}\, a_0\, [(es, a)] \qquad := \mathsf{sg}\, a\ , \\
&\mathsf{fromListRep}\, a_0\, ((es, a) :: p :: r) := \\
&\qquad\qquad \mathsf{constr}\, a\, \big(\mathsf{fromListRep}\, (e_0, a_0)\, (\mathsf{shiftFromEs}(p :: r))\big)\ .
\end{aligned}$$

Lemma 4 (left inverse). *For any type A, terms $a_0 : A$ and $t : \mathsf{Tri}\, A$, one has*

$$\mathsf{fromListRep}\, a_0\, (\mathsf{toListRep}\, t) = t.$$

Proof. By a use of the induction principle for Tri, exploiting the fact that any $\mathsf{toListRep}\, t$ is a non-empty list, hence of the form $p :: r$. □

Theorem 2 (Main Theorem). *If E is non-empty, then for all types A, B, and terms $a_0 : A$, $b_0 : B$, $f : \mathsf{Tri}\, A \to B$ and $t : \mathsf{Tri}\, A$:*

$$\mathsf{redec}\, f\, t = \mathsf{fromListRep}\, b_0\, \big(\mathsf{redecL}\, (f \circ (\mathsf{fromListRep}\, a_0))\, (\mathsf{toListRep}\, t)\big)\ .$$

Proof. An immediate consequence of the simulation theorem and the preceding lemma, by using extensionality of redec in its functional argument once more. □

[5] See the discussion on extensionality in Section 5.

5 Details on Formal Verification with Coq

In this section, a Coq development of the mathematical contents of the last three sections is discussed. The Coq vernacular file can be found at the web site [12].

We mentioned above that the Coq system [10][6] has a genuine pattern-matching support for nested datatypes like Tri since version 8.1, contributed by Christine Paulin. In version 8.0, there were subtle problems because such datatypes could only be specified through datatype constructors with universally quantified types that had to live in the universe Set as well, hence Set had to be made impredicative by an option to the Coq runtime system.

The following remarks concern Coq 8.1 at patch level 3, released in December 2007.

The nested datatype (a. k. a. inductive family) Tri is introduced as follows:

```
Inductive Tri (A:Set) : Set :=
  sg : A -> Tri A | constr : A -> Tri (E * A) -> Tri A.
```

Then, the appropriate induction principle is automatically generated, and one can check its type:

```
Check Tri_ind : forall P : forall A : Set, Tri A -> Prop,
(forall (A : Set) (a : A), P A (sg a)) ->
(forall (A : Set) (a : A) (r : Tri (E * A)),
                          P (E * A) r -> P A (constr a r)) ->
forall (A : Set) (t : Tri A), P A t.
```

This is exactly the induction principle of Section 2. However, the induction principle for Trap in Section 4 seems to need a (straightforward) proof via the `fix` construction for structurally recursive functions/proofs.

The definition of redecL by recursion over a measure and reasoning about redecL by "measure induction" uses an experimental feature of Coq 8.1 (one has to load separately the package `Recdef`), provided by Pierre Courtieu, Julien Forest and Yves Bertot [4–6].

```
Function redecL (A B:Set)(f:TriL A -> B)(t: TriL A)
                {measure length t} : TriL B :=
  match t with nil => nil
      | (es,a)::rest => (es,f((es,a)::rest))::
                        zipAppendEs (singletonTopEs rest)
                          (@redecL A B f (removeTopEs rest)) end.
```

The fact that the length is a measure that decreases in the recursive call has to be proven in order to get Coq to accept this as a definition. Thanks to the explicit form `@redecL A B` that reveals the Church-style syntax that underlies Coq although it is hidden from the user by the mechanism of implicit arguments,

[6] We will only presuppose concepts and features of Coq that are explained in the Coq textbook [8].

measure induction even works with this polymorphic function. Coq automatically generates an induction principle `redecL_ind`, called functional induction, that allows to argue about values of `redecL` directly along the recursive call scheme of its definition. The induction hypothesis is prepared with the argument `removeTopEs rest`, and there is no need to redo the justification by means of the decreasing length again. The proof of the main lemma is then an instance of `redecL_ind`, and this is interactively initiated by

```
functional induction (redecL (liftL e0 f) r).
```

In a simpler form, functional induction is used for the analysis of zipWith that, despite being structurally recursive in both list arguments, also profits from being defined by the "Function" command that again prepares the induction hypotheses, and this has already been available in Coq for years now.

However, even the current extensions to functional induction in Coq 8.1 patch level 3 do not cover the definition of fromListRep because it combines recursion with decreasing measure with polymorphic recursion. In the development version of Coq, a proposal by Julien Forest works well where the type A and the elements $a_0 : A$ and $t : \mathsf{TriL}\, A$ are encapsulated in a record, see [12]. Our solution consists in defining an auxiliary function with an additional parameter n of type nat by ordinary recursion on n and then fixing n to the length of t. This works very well because the list length is just one less in the recursive call and because the proof of Lemma 4 only needs the defining equations of fromListRep immediately preceding that lemma.

In the middle of the proof of the second comonad law, $\mathsf{redec}\,\mathsf{top}\,t = t$, we have to prove $\mathsf{redec}\,\mathsf{top}\,t = \mathsf{redec}\,(\mathsf{lift}\,\mathsf{top})\,t$. It was easy to prove $\forall r.\, \mathsf{lift}\,\mathsf{top}\,r = \mathsf{top}\,r$ before that. It would be easy to conclude if this implied $\mathsf{lift}\,\mathsf{top} = \mathsf{top}$, but this typically cannot be done in intensional type theory to which the underlying system of Coq belongs, namely the Calculus of Inductive Constructions. But we do not even need that equality since, in general, $\mathsf{redec}\,f\,t$ only depends on the values of f (the "extension" of f) and not its definition (or "intension"). More precisely, one can show by Tri induction on t that redec is "extensional":

$$\forall f\, f'.\, (\forall t'.\, f\, t' = f'\, t') \to \mathsf{redec}\, f\, t = \mathsf{redec}\, f'\, t \ .$$

This property is also needed for the proofs of the third comonad law, the simulation theorem and the main theorem, and the analogous property for map enters the proof of the main lemma, its auxiliary lemmas and the proof that shiftFromEs is a left-inverse of shiftToEs.

6 Details on Formal Verification with Isabelle

We are going to sketch an alternative to the Coq implementation, described in the previous section. This is done within the system Isabelle (more precisely, Isabelle 2007 of November 2007), and the script with the theory development is also available from the web site [12].

The type system of Isabelle is less expressive than the type system of Coq: it is a simply typed lambda-calculus with ML-style polymorphism [13]. Type parameters of polymorphic functions need not be supplied explicitly, but can be inferred by the system, and universal quantification over types on the top-level is provided through schematic type variables. The datatype definition mechanism currently implemented in Isabelle is described in more detail in [7].

To be consistent with the Coq formalization, we would like to fix a type constant `E` by declaring "`typedecl E`" and define the polymorphic `tri` datatype as follows:

```
datatype 'a tri  =  sg ('a)  |  constr ('a) ((E * 'a) tri)
```

As spelled out in Section 2, in the resulting induction principle

```
∀ P. (∀ a. P (sg a)) ⟶
        (∀ a r. P r ⟶ P (constr a r)) ⟶ ∀ t. P t
```

the universally quantified induction predicate `P` would then be applied both to a `(E * 'a) tri` and a `'a tri`, thus overstraining Isabelle's type system. Therefore, such a datatype definition is not valid in Isabelle.

We circumvent this and related problems by not conceiving `sg` and `constr` as constructors of an inductive type, but just as constants declared by

```
consts sg :: 'a ⇒ ('a,'e) tri
       constr :: 'a ⇒ ('a, 'e) trap ⇒ ('a, 'e) tri
```

As above, `('a, 'e) trap` abbreviates `('e * 'a,'e) tri`. (And the fixed parameter `E` is replaced by a second type parameter `'e`. For the whole theoretical development, this difference does not play any role, but it facilitates concrete programming examples that are also provided in the Isabelle script.)

For carrying out proofs, we have to provide appropriate instances of the induction predicate. In order to obtain the desired computational behaviour, we manually have to add reduction rules, as will be shown in the following.

As an example, take the `cut` function of Section 2. We declare the function `cut` by:

```
consts cut :: ('a, 'e) trap ⇒ ('a, 'e) tri
```

The primitive-recursive function definition is accomplished by providing the following characteristic equations:

```
axioms  cut_sg [simp]: cut (sg (e,a)) = sg a
        cut_constr [simp]: cut (constr (e,a) r) = constr a (cut r)
```

Note that in the second equation, `cut` is applied to expressions of different types: on the left, to a term of type `'a tri`, on the right, to a term `('a,'e) tri`. Here, we exploit an essential difference between a universally quantified variable (as in the induction predicate above), which can only be applied to elements of the same type, and a globally declared constant such as `cut`, which can be applied

to instances of different type. This distinction is reminiscent of the difference, in an ML-style type system, between the term

$$\lambda id : {'a} \Rightarrow {'a}. \lambda f : nat \Rightarrow bool \Rightarrow nat.\ f\,(id\,0)\,(id\,True)$$

(which is not well-typed) and

$$\text{let } id = (\lambda x : {'a}. x) \text{ in } \lambda f : nat \Rightarrow bool \Rightarrow nat.\ f\,(id\,0)\,(id\,True)$$

(which is).

Of course, this axiomatization does not provide the guarantees of a genuine primitive recursive definition, such as termination.

As mentioned above, for typing reasons, we cannot state a general induction principle. We can, however, exploit the same mechanism as for function definitions and provide instances of the induction principle for proving individual theorems.

We illustrate the procedure for the proof of the following (where `#` is the "cons" operation and `snd` is the right/second projection out of a pair)

```
lemma toListRep_cons_inv:
    toListRep t = a # list ⟶ top t = snd a
```

We notice that the proof can be carried out using the following instance of the induction predicate:

```
(∀a. P1 (sg a)) ⟶ (∀a r. P1 r ⟶ P1 (constr a r)) ⟶ ∀t. P1 t
```

where `P1` is defined as

```
λt. (∀ a list. toListRep t = a # list ⟶  top t = snd a)
```

The proof of the lemma is now very easy: unfold the definition of `P1` and carry out elementary term simplification.

Altogether, the proof of Theorem 1 requires four instances of the induction schema. This approach is not difficult, but suffers from the well-known drawbacks of code duplication: it is error-prone and the resulting theories are hard to maintain.

This is even more true since, for the proof of Lemma 3, in order to get the induction through, we have to quantify over the function f as well, and this implicitly requires to quantify over its additional type variable B. Since the latter quantification cannot be expressed, we cannot just use the above induction axiom for `P1` with the respective new predicate in place of `P1` but have to copy its definition to the four occurrences in the induction formula, giving rise to the axiom `Tri_ind_MAIN_appl2` in the Isabelle script. Even though it is possible in principle to generate the required induction schemas, the discussion shows that the result tends to be artificial and, by excessive code duplication, contrary to good practice. On the good side, the difference between the induction principles for Tri and Trap becomes invisible in this approach while in Coq, the former is provided and the latter has to be defined by structural recursion.

In the list-based model, we enjoy the full support from Isabelle for datatypes, here for lists. The proof of the main lemma can just follow the structure of the recursive calls in the definition, expressed in the generated theorem `redecL.induct` that is a version of the respective "functional induction" scheme in Coq, without dependent types and hence without the need to reference `redecL` in it. This functionality has been developed by Konrad Slind [14].

Also note that proving extensionality of `redec` in its function argument becomes a triviality in Isabelle, thanks to its rule `expand_fun_eq` that assumes `(?f = ?g) = (∀ x. ?f x = ?g x)`, i. e., functions are equal *if* and only if they are point-wise equal.

Finally, we remark that Isabelle's type system only allows inhabited types, hence the type parameters only range over nonempty types. Thus, all the elements denoted by a_0, b_0 and e_0 in our informal description (and present in the Coq development) could have been obtained by the ε operator and therefore do not show up in the Isabelle scripts [12].

7 Conclusions

This article has presented a mathematical formalization of redecoration in triangular matrices by means of a nested datatype. Redecoration provides a comonad structure for this datatype. Moreover, we have established a precise relationship with a model that is only based on lists. For its verification, we have contrasted two formalizations in the proof assistants Coq and Isabelle and discussed their different approaches, in particular recursion and induction that do not just follow the datatype definition. An important difficulty has been the necessity of polymorphic recursion, but this is intrinsic to nested datatypes.

We would hope for some Isabelle extension with a full support of nested datatypes, i. e., where induction axioms and equational specifications of recursive functions are generated and justified in the kernel of Isabelle, just as in the existing datatype package.

Interesting future work would treat the original *infinite* triangular matrices of [2, 3] or even specify and verify a datatype-generic definition of `redec`.

Acknowledgements: With the help of Stefan Berghofer, we overran the more subtle problems with variables of different kinds in Isabelle. Mamoun Filali has provided valuable suggestions for the elimination of functional induction in the Coq development as an alternative to Julien Forest's construction that is no longer supported by the current Coq version. The referees' suggestions helped to substantially strengthen the main theorem.

References

1. Abel, A., Matthes, R.: Fixed points of type constructors and primitive recursion. In: Marcinkowski, J., Tarlecki, A., eds., Computer Science Logic: 18th International Workshop, CSL 2004, 13th Annual Conference of the EACSL. Proceedings, volume 3210 of Lecture Notes in Computer Science. Springer Verlag (2004), 190–204

2. Abel, A., Matthes, R., Uustalu, T.: Generalized iteration and coiteration for higher-order nested datatypes. In: Gordon, A., ed., Foundations of Software Science and Computational Structures, 6th International Conference, FoSSaCS 2003, volume 2620 of Lecture Notes in Computer Science. Springer Verlag (2003), 54–68
3. Abel, A., Matthes, R., Uustalu, T.: Iteration and coiteration schemes for higher-order and nested datatypes. Theoretical Computer Science **333** (2005) 3–66
4. Balaa, A., Bertot, Y.: Fix-point equations for well-founded recursion in type theory. In: Aagaard, M., Harrison, J., eds., Theorem Proving in Higher-Order Logics, TPHOLs 2000, Proceedings, volume 1869 of Lecture Notes in Computer Science. Springer Verlag (2000), 1–16
5. Barthe, G., Courtieu, P.: Efficient reasoning about executable specifications in Coq. In: Carreño, V. A., Muñoz, C. A., Tahar, S., eds., Theorem Proving in Higher Order Logics, TPHOLs 2002, Proceedings, volume 2410 of Lecture Notes in Computer Science. Springer Verlag (2002), 31–46
6. Barthe, G., Forest, J., Pichardie, D., Rusu, V.: Defining and reasoning about recursive functions: A practical tool for the Coq proof assistant. In: Hagiya, M., Wadler, P., eds., Functional and Logic Programming, 8th International Symposium, FLOPS 2006, Proceedings, volume 3945 of Lecture Notes in Computer Science. Springer Verlag (2006), 114–129
7. Berghofer, S., Wenzel, M.: Inductive datatypes in HOL - lessons learned in formal-logic engineering. In: Bertot, Y., Dowek, G., Hirschowitz, A., Paulin, C., Théry, L., eds., Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs'99, volume 1690 of Lecture Notes in Computer Science. Springer Verlag (1999), 19–36
8. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science, Springer Verlag (2004)
9. Bird, R., Meertens, L.: Nested datatypes. In: Jeuring, J., ed., Mathematics of Program Construction, MPC'98, Proceedings, volume 1422 of Lecture Notes in Computer Science. Springer Verlag (1998), 52–67
10. Coq Development Team: The Coq Proof Assistant Reference Manual Version 8.1. Project LogiCal, INRIA (2006). System available at `coq.inria.fr`.
11. Hinze, R.: Manufacturing datatypes. Journal of Functional Programming **11** (2001) 493–524
12. Matthes, R., Strecker, M.: Coq and Isabelle development for "Verification of the Redecoration Algorithm for Triangular Matrices". `http://www.irit.fr/~Ralph.Matthes/CoqIsabelle/TYPES07/` (2007)
13. Nipkow, T., Paulson, L., Wenzel, M.: Isabelle/HOL. A Proof Assistant for Higher-Order Logic, volume 2283 of Lecture Notes in Computer Science. Springer Verlag (2002)
14. Slind, K.: Wellfounded schematic definitions. In: McAllester, D. A., ed., Automated Deduction – CADE-17, 17th International Conference on Automated Deduction, Proceedings, volume 1831 of Lecture Notes in Computer Science. Springer Verlag (2000), 45–63
15. Uustalu, T., Vene, V.: The dual of substitution is redecoration. In: Hammond, K., Curtis, S., eds., Trends in Functional Programming 3. Intellect, Bristol / Portland, OR (2002), 99–110