Martin Hofmann's case for non-strictly positive data types – reloaded

Ralph Matthes

CNRS, Institut de Recherche en Informatique de Toulouse (IRIT),
University of Toulouse

2nd Worksh. on Proof Theory and its Applications, The Proof Society
Department of Computer Science, Swansea University , U. K.
September 12, 2019

Abstract – first page

We describe the breadth-first traversal algorithm by Martin Hofmann that uses a non-strictly positive data type and carry out a simple verification in an extensional setting. Termination is shown by implementing the algorithm in the strongly normalising extension of system F by Mendler-style recursion.

We then analyze the same algorithm by alternative verifications

- in an intensional setting,
- in a setting with non-strictly positive inductive definitions (not just non-strictly positive data types), and one
- by algebraic reduction.

Abstract – contd.

The verification approaches are compared in terms of notions of simulation and should elucidate the somewhat mysterious algorithm and thus make a case for other uses of non-strictly positive data types.

Except for the termination proof, which cannot be formalised in Coq, all proofs were formalised in Coq.

The present talk is based on a paper in the forthcoming LIPIcs post-proceedings of TYPES 2018 with Ulrich Berger and Anton Setzer (both Swansea University) and should demonstrate how much progress the three authors could make on understanding this old proposal of Hofmann, made between 1993 and 1995, since the [intended] speaker [R. M.] gave a tribute to him at TYPES 2018.

Outline

1. Obtaining fancy breadth-first traversal
2. Analyzing fancy breadth-first traversal
3. Other non-strictly positive datatypes in use

Outline

1. Obtaining fancy breadth-first traversal
2. Analyzing fancy breadth-first traversal
3. Other non-strictly positive datatypes in use

Breadth-first traversal

Binary trees with leaf labels and node labels in $\mathbb{N}$. Call this data type Tree, with constructors $\mathit{Leaf} : \mathbb{N} \to \mathit{Tree}$ and $\mathit{Node} : \mathit{Tree} \to \mathbb{N} \to \mathit{Tree} \to \mathit{Tree}$.

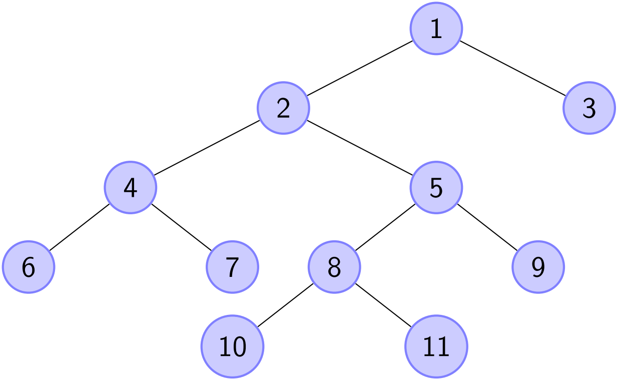
(For simplicity and to avoid pseudo-generality, we restrict the type of labels to be the natural numbers but any other type could be used instead.)

The type of homogeneous lists with elements from type A is called $\mathit{List}\, A$.

(We need parameter A, it will often be $\mathit{List}\, \mathbb{N}$.)

The task: go through $t : \mathit{Tree}$ in breadth-first order and collect the labels in $\mathit{breadthfirst}\, t : \mathit{List}\, \mathbb{N}$.

Illustration: result $[1, 2, \ldots, 11]$



The function is not compositional! (Does not only depend on its values for the subtrees.)

Less intuitive specification

Create the list of labels for every line, i. e., level-wise. This function is called $\mathit{niv} : \mathit{Tree} \to \mathit{List}^2\mathbb{N}$. ($\mathit{niv}$ refers to the French word "niveaux" for levels)
Result for our example: $[[1],[2,3],[4,5],[6,7,8,9],[10,11]]$
niv can be obtained by iteration over Tree: in the Node case, zip the results for the sub-trees with list concatenation, vulgo append.
Define $\mathit{flatten} : \mathit{List}^2\mathbb{N} \to \mathit{List}\,\mathbb{N}$ as concatenation of all those lists (the monad multiplication of the list monad).
$\mathit{breadthfirst}\, t$ has to evaluate to the result of $\mathit{flatten}(\mathit{niv}\, t)$. The latter is not the algorithm but an executable specification.

This is good for functional programmers. Imperative programming would suggest to use a queue of binary trees. We type theoreticians want language-based termination guarantees.

Martin Hofmann's 1993 proposal

A post to the TYPES mailing list, which is still in the TYPES archives (checked on September 11, 2019).

Assumes a data type Rou with constructors

$$\begin{array}{lcl} \mathit{Over} & : & \mathit{Rou} \\ \mathit{Next} & : & ((\mathit{Rou} \to \mathit{List}\,\mathbb{N}) \to \mathit{List}\,\mathbb{N}) \to \mathit{Rou} \end{array}$$

Martin viewed the elements as continuations, but I [R. M.] learned from Olivier Danvy in 2002 that they are rather coroutines, but in 2018 we came to the conclusion that they do not have specific coroutine features but are just routines, hence the name Rou chosen here (Over and Next suggested by Danvy). Over: nothing more to be done; Next: its argument f takes a "continuation" argument $k : \mathit{Rou} \to \mathit{List}\,\mathbb{N}$ and computes a list.

Working with routines

Specify $unfold : Rou \to (Rou \to List\,\mathbb{N}) \to List\,\mathbb{N}$ by distinguishing the two cases (the second one is indeed an “unfolding”):

$$\begin{array}{lcl} unfold\ Over & \simeq & \lambda k.\, k\ Over \\ unfold\,(Next\ f) & \simeq & f \end{array}$$

Relation $\simeq$ is used for definitional equality, i. e., convertibility. This spec. does not by itself constitute a definition.
Martin Hofmann (called *unfold* rather *apply*) recasts the breadth-first traversal as a transformation on routines controlled by the input tree:

$$\begin{array}{lcl} br : Tree \to Rou \to Rou & & \\ br\,(Leaf\ n)\,c & = & Next\,(\lambda k\,.\,n :: unfold\,c\,k) \\ br\,(Node\,tl\,n\,tr)\,c & = & Next\,\big(\lambda k\,.\,n :: unfold\,c\,(k \circ br\ tl\ \circ\ br\ tr)\big) \end{array}$$

($\circ$ denotes composition of functions)

Extraction of the final result

$br\,t\,Over$ is a routine, and we want $breadthfirst\,t$ to be the list extracted from it by the function $extract : Rou \to List\,\mathbb{N}$, specified as

$$\begin{array}{lcl} extract\ Over & \simeq & [] \\ extract(Next\ f) & \simeq & f\,extract \end{array}$$

No problem with subject reduction—recall $f : (Rou \to List\,\mathbb{N}) \to List\,\mathbb{N}$. In our view, $extract$ is a “continuation”, and so the argument f to $Next$ can be naturally applied to it.

Why is this recursion scheme safe, i. e., why does it not present the risk of non-termination?

Outline

Termination of *extract*

For Martin Hofmann, this was the main motivation. One can see Rou as least fixed point of the "functor" $RouF$ that a Haskell programmer could define as

```
data RouF rou = Over | Next ((rou -> List nat) -> List nat)
```

The variable `rou` for the datatype to be defined is twice to the left of `->`, hence at a positive position, even if not at a strictly positive position. Martin argues that the specification of $extract$ can be ensured by the usual Church encoding of data types in system F; in categorical terms, $extract$ can be obtained as catamorphism for a certain $RouF$-algebra. However, only weak initiality is obtained, unless one uses parametric equality. In more computational terms, this means that $extract$ is defined by pure iteration.

Pitfall concerning termination

The argument on $extract$ is valid, even if Mendler-style iteration would more directly allow to program $extract$ precisely according to the specification, and likewise with termination guarantee (as instance of Mendler-style iteration).
However, the function $unfold : Rou \to (Rou \to List\,\mathbb{N}) \to List\,\mathbb{N}$ has to be defined with the same ontology for Rou. To recall:

$$\begin{array}{lcl} unfold\ Over & \simeq & \lambda k.\, k\ Over \\ unfold\,(Next\ f) & \simeq & f \end{array}$$

No recursion but the patterns are distinguished and the argument f extracted. This is not compatible with weakly initial algebras, as obtained with the Church encoding. Martin was satisfied with parametric equality theory, but $unfold$ should have constant execution time, if the proposed algorithm is meant to have advantages over the executable specification.

Solution

Use primitive recursion in Mendler's style (invented by Nax Mendler for his 1987 PhD thesis whose advisor was Bob Constable). In fact, the only addition to system F that is really needed to preserve termination is positive fixed-points μF with a retraction between μF and $F(\mu F)$—the sequence from $F(\mu F)$ via μF back to $F(\mu F)$ has to be pointwise definitionally equal to the identity. Termination of more complex schemes can be obtained by simulation of reductions. (See my [R. M.] FICS'98 paper in RAIRO/ITA.)

Functional correctness

Given that the termination question is already solved, how can one see that the algorithm $\mathit{breadthfirst}\,t := \mathit{extract}(\mathit{br}\,t\,\mathit{Over})$ meets the specification, i. e., computes the right list?

Martin Hofmann provided in 1995 a proof by simple induction on Tree, by help of the routine transformer $\gamma : \mathit{List}^2\mathbb{N} \to \mathit{Rou} \to \mathit{Rou}$

$$\gamma\,[]\,c \;=\; c \qquad\qquad \gamma\,(l :: \mathit{ls})\,c \;=\; \mathit{Next}\Big(\lambda k\,.\,l +\!\!+ \big(\mathit{unfold}\,c\,(k \circ \gamma\,\mathit{ls})\big)\Big)$$

for which
(A) $\mathit{extract}(\gamma\,l\,\mathit{Over}) = \mathit{flatten}\,l$,
(B) composition of γ for two lists of lists is γ for the zipping with append,
which allows to prove that
(C) $\mathit{br}\,t$ does the same as $\gamma(\mathit{niv}\,t)$.
(A) and (C) then give correctness.

Alternative verifications—why?

The proof by Martin Hofmann is a typical mathematical proof in that it freely uses function extensionality: two functions are equal if they are pointwise equal. This principle is problematic for computational interpretations. E. g., when trying to represent the sketched proof in the Coq system, one has to load explicitly an extensionality axiom—this is not part of the core type theory.

Does γ reveal the “true nature” of this peculiar breadth-first traversal algorithm? We do not think so. γ convinces the proof checker in us that it is correct, but no more.

For at least these two reasons, we developed a variety of alternative proofs. All those proofs can be compared in mathematical ways by a notion of simulation—bear with me.

Verification by a non-strictly positive inductive relation

An intriguing idea: reflect the non-strictly positive nature of the routines in a likewise non-strictly positive inductive definition of a predicate that relates routines and lists of lists of natural numbers ("double lists") representing results of their execution.

We do not have the time here to go into detailed explanations of the concepts (cf. the forthcoming TYPES'18 post-proceedings paper). We rather restrict our attention to the syntactic aspects of the definition of representation.

Auxiliary step: we define when a continuation k is an extractor for a binary relation $R \subseteq \mathit{Rou} \times \mathit{List}^2\mathbb{N}$ (seen as a candidate for a representation relation) and an "initial" double list ls' (later set to $[]$).

$$\mathit{isextractor}(R, ls', k) := \forall c, ls''.\, R(c, ls'') \to k\,c = \mathit{flatten}\,(\mathit{zip}\; ls'\; ls'') \ .$$

(zip is the zipping function mentioned twice before; R only occurs negatively – to the left of the implication)

Non-strictly positive inductive definition of representation

$isextractor(R, ls', k) := \forall c, ls''\,.\, R(c, ls'') \to k\,c = flatten\,(zip\;ls'\;ls'')$ is used in the following inductive definition of $rep \subseteq Rou \times List^2\mathbb{N}$:

$$\frac{}{rep(Over, [])}\;\text{(over)}$$

$$\frac{\forall k, ls'\,.\, isextractor(rep, ls', k) \to f\,k = l \mathbin{+\!\!+} flatten\,(zip\;ls'\;ls)}{rep(Next\;f, l :: ls)}\;\text{(next)}$$

The premise of the rule (next) contains the predicate rep positively (though not strictly positively) and therefore depends monotonically on it. By Tarski's fixed point theorem it follows that the smallest relation rep closed under the rules (over) and (next) exists.

The non-trivial results towards the verification of our algorithm through rep are: (A) $isextractor(rep, [], extract)$, proven by exploiting minimality of rep w. r. t. its defining clauses, and (B) that $rep(c, ls)$ implies $rep(br\,t\,c, zip\;(niv\;t)\;ls)$, proven mostly by structural induction on tree t.

Verification by interpreting routines as recursive programs

It has been known for a long time that breadth-first traversal can be profitably studied by extending the input type from trees to lists of trees: $\mathit{Forest} := \mathit{List}\ \mathit{Tree}$. Its executable specification is $\mathit{flatten} \circ \mathit{niv_f} : \mathit{Forest} \to \mathsf{List}\,\mathbb{N}$ where $\mathit{niv_f}$ zips all $\mathit{niv}\,t$ for t in ts, i. e.

$$\mathit{niv_f} : \mathit{Forest} \to \mathsf{List}^2\,\mathbb{N}$$
$$\mathit{niv_f}\ [] = [] \qquad \mathit{niv_f}\,(t :: \mathit{ts}) = \mathit{zip}\,(\mathit{niv}\,t)\,(\mathit{niv_f}\ \mathit{ts})$$

If the input forest consists of a single tree, the extended and the original specification agree.

This opens the possibility of an alternative verification of Hofmann's algorithm via an embedding of forests into routines that explains the roles of the functions $\mathit{br} : \mathit{Tree} \to \mathit{Rou} \to \mathit{Rou}$ and $\mathit{extract} : \mathit{Rou} \to \mathsf{List}\,\mathbb{N}$, and it appears simpler thanks to the richer data structure of forests.

Interpretation of routines as recursive programs

We define $c : \mathit{Forest} \to \mathit{Rou}$ by recursion on the depth—not detailed here—of the input forest.

$$c\;\mathit{ts} = \begin{cases} \mathit{Over} & \text{if } \mathit{ts} = [], \\ \mathit{next}\,(\mathit{addroots}\;\mathit{ts})\,(c\,(\mathit{sub}\;\mathit{ts})) & \text{otherwise.} \end{cases}$$

Here, we use:

- $\mathit{roots} : \mathit{Forest} \to \mathsf{List}\,\mathbb{N}$ with $\mathit{roots}\;[] = []$ and $\mathit{roots}\;(\mathit{Leaf}\;n :: \mathit{ts}) = \mathit{roots}\;(\mathit{Node}\;\mathit{tl}\,n\,\mathit{tr} :: \mathit{ts}) = n :: \mathit{roots}\;\mathit{ts}$
- $\mathit{addroots} : \mathit{Forest} \to \mathsf{List}\,\mathbb{N} \to \mathsf{List}\,\mathbb{N}$ with $\mathit{addroots}\;\mathit{ts} = \mathit{append}\,(\mathit{roots}\;\mathit{ts})$
- $\mathit{sub} : \mathit{Forest} \to \mathit{Forest}$ calculates the immediate subforest: $\mathit{sub}\;[] = []$, $\mathit{sub}\,(\mathit{Leaf}\;n :: \mathit{ts}) = \mathit{sub}\;\mathit{ts}$, and $\mathit{sub}\,(\mathit{Node}\;\mathit{tl}\,n\,\mathit{tr} :: \mathit{ts}) = \mathit{tl} :: \mathit{tr} :: \mathit{sub}\;\mathit{ts}$.
- $\mathit{next} : (\mathsf{List}\,\mathbb{N} \to \mathsf{List}\,\mathbb{N}) \to \mathit{Rou} \to \mathit{Rou}$ with $\mathit{next}\;g\;c = \mathit{Next}\,(\lambda k\,.\,g\,(k\,c))$.

Properties of this interpretation

We obviously obtain $\mathit{extract}\,(\mathit{next}\,g\,c) = g\,(\mathit{extract}\,c)$ besides $\mathit{extract}\ \mathit{Over} = []$.

Therefore, the c function provides routines whose extraction yields the desired result: $\mathit{extract}\,(c\,\mathit{ts}) = \mathit{flatten}\,(\mathit{niv}_f\ \mathit{ts})$. This is seen by induction on depth, using the easy auxiliary $\mathit{niv}_f\ \mathit{ts} = \mathit{roots}\ \mathit{ts} :: \mathit{niv}_f\ (\mathit{sub}\ \mathit{ts})$ for nonempty ts.

The main technical lemma states $\mathit{br}\,t\,(c\,\mathit{ts}) = c\,(t :: \mathit{ts})$, with proof by induction on the depth of t.

The lemma elucidates the purpose of the routine transformer br: $\mathit{br}\,t$ transforms the routine so that t as first element of the input forest is treated in addition—which obviously requires the forest view taken in this approach.

Verification of Hofmann's algorithm is by instantiating the main lemma with empty ts and the lemma before with the singleton forest $[t]$.

Verification by successive refinements

First step: $br : \mathit{Tree} \to \mathit{Rou} \to \mathit{Rou}$ can be shrunk down to a structurally recursive definition of a function $br' : \mathit{Tree} \to \mathit{Rou}' \to \mathit{Rou}'$ with $\mathit{Rou}' := \mathit{List}(\mathit{List}\,\mathbb{N} \to \mathit{List}\,\mathbb{N})$. The crucial definition is a purely iterative function $\Phi : \mathit{Rou}' \to \mathit{Rou}$, for which one tries to obtain

$$br\,t\,(\Phi\,l) = \Phi(br'\,t\,l)$$

This guides the definition process for br'.

Second step: $br' : \mathit{Tree} \to \mathit{Rou}' \to \mathit{Rou}'$ can be further shrunk down to a structurally recursive definition of a function $br'' : \mathit{Tree} \to \mathit{List}^2\mathbb{N} \to \mathit{List}^2\mathbb{N}$. Crucial observation: for the mapping over lists of the append function of type $\mathit{List}\,\mathbb{N} \to (\mathit{List}\,\mathbb{N} \to \mathit{List}\,\mathbb{N})$, let's call it $\Psi : \mathit{List}^2\mathbb{N} \to \mathit{Rou}'$, one can obtain

$$br'\,t\,(\Psi\,l) = \Psi(br''\,t\,l)$$

Verification by successive refinements—the end

The function $br'' : \mathit{Tree} \to \mathit{List}^2\mathbb{N} \to \mathit{List}^2\mathbb{N}$ thus obtained is easy to grasp in terms of list operations:

$$br''\, t\, l = \mathit{zip}\,(\mathit{niv}\, t)\, l$$

And $\mathit{extract}(\Phi(\Psi\, l)) = \mathit{flatten}\, l$.

Modulo the exciting presentation in terms of the non-strictly positive data type of routines, the outcome of this (predicative) analysis is that br adopts an “accumulation trick” for computing the levels, and that the extraction process takes care of flattening.

Formal comparison of the obtained algorithms and proofs

We have exposed four different ways to verify breadth-first traversal à la Martin Hofmann. Can something interesting be said about the relations between these proofs? Other than mere qualitative observations such as that the proof with rep and the proof with c do not need the extensionality axiom, that the proof with rep uses heavier meta-theory, etc.

Yes, we identify four components in each of those proofs that play the same role, and we can relate these 4-tuples in a systematic way.

Systems and simulations between systems

Definition

- A system is a quadruple $S = (A, \mathsf{Nil}, g, e)$ where $A : \mathsf{Set}$, $\mathsf{Nil} : A$, $g : \mathit{Tree} \to A \to A$, and $e : A \to \mathit{List}\,\mathbb{N}$.
- S is correct (for breadth-first traversal) if $e\,(g\,t\,\mathsf{Nil}) = \mathit{flatten}(\mathit{niv}\,t)$ for all trees t.
- Let $S' = (A', \mathsf{Nil}', g', e')$ be another system. A relation R on $A \times A'$ is a simulation between S and S', $S \overset{R}{\sim} S'$, if (1) $R(\mathsf{Nil}, \mathsf{Nil}')$, and, whenever $R(a, a')$, then (2) $R(g\,t\,a, g'\,t\,a')$ for all trees t, and (3) $e\,a = e'\,a'$.
- Let S, S' be systems. S and S' are similar, $S \sim S'$, if there exists a simulation between S and S'.

Lemma: If $S \sim S'$ then S is correct if and only if S' is correct.

Functional simulations between systems

Recall that a relation R on $A \times A'$ is a simulation between S and S', $S \overset{R}{\sim} S'$, if (1) $R(\mathsf{Nil}, \mathsf{Nil}')$, and, whenever $R(a, a')$, then (2) $R(g\,t\,a, g'\,t\,a')$ for all trees t, and (3) $e\,a = e'\,a'$.

Note that if R is functional, i. e., defined as the graph of a function $\phi : A' \to A$, by setting $R(a, a')$ iff $a = \phi\,a'$, then the simulation conditions become (1) $\mathsf{Nil} = \phi\,\mathsf{Nil}'$, (2) $g\,t \circ \phi \overset{ext}{=} \phi \circ g'\,t$ for all trees t, and (3) $e \circ \phi \overset{\text{ext}}{=} e'$. In this situation we write $S \overset{\phi}{\leftarrow} S'$. All but one of the simulations described below are functional.

Review of the zoo of proofs (abridged)

- The specification of breadth-first traversal corresponds to the correct system $S_{\mathsf{spec}} \overset{\text{Def}}{\equiv} (\mathit{List}^2\mathbb{N}, [], \mathit{zip} \circ \mathit{niv}, \mathit{flatten})$.
- Hofmann's algorithm is embodied by the system $S_{\mathsf{MH}} \overset{\text{Def}}{\equiv} (\mathit{Rou}, \mathit{Over}, \mathit{br}, \mathit{extract})$ and its verification amounts to showing that $S_{\mathsf{MH}} \overset{\gamma_{\mathit{Over}}}{\longleftarrow} S_{\mathsf{spec}}$ where $\gamma_{\mathit{Over}}\ \mathit{ls} \overset{\text{Def}}{\equiv} \gamma\ \mathit{ls}\ \mathit{Over}$.
- The verification by help of rep amounts to showing $S_{\mathsf{MH}} \overset{\mathit{rep}}{\sim} S_{\mathsf{spec}}$.
- The spec. with forests gives rise to a system S_{forest}, and the alternative verification in fact shows $S_{\mathsf{MH}} \overset{c}{\leftarrow} S_{\mathsf{forest}}$.
- The first refinement step yields system $S_{\mathsf{pred1}} \overset{\text{Def}}{\equiv} (\mathit{Rou}', [], \mathit{br}', \mathit{extract} \circ \Phi)$ and proves the simulation $S_{\mathsf{MH}} \overset{\Phi}{\leftarrow} S_{\mathsf{pred1}}$.
- The second refinement step yields system $S_{\mathsf{pred2}} \overset{\text{Def}}{\equiv} (\mathit{List}^2\mathbb{N}, [], \mathit{br}'', \mathit{flatten})$.

Overview of the simulations

$$\begin{tikzcd}
 & & S_{\mathsf{MH}} & & \\
S_{\mathsf{forest}} \arrow[urr, "\mathfrak{c}"] \arrow[rrrr, "traverse"] \arrow[dr, "niv_f"'] & & & & S_{\mathsf{pred1}} \arrow[ull, "\Phi"'] \\
 & S_{\mathsf{spec}} \arrow[uur, dash, "rep"] & \overset{ext}{=} & S_{\mathsf{pred2}} \arrow[uul, "\gamma_{Over}"'] \arrow[ur, "\Psi"'] &
\end{tikzcd}$$

The functions in the diagram are fully commutative assuming extensionality. In particular, the simulations $S_{\mathsf{MH}} \overset{\Phi}{\leftarrow} S_{\mathsf{pred1}} \overset{\Psi}{\leftarrow} S_{\mathsf{pred2}}$ provide a splitting of Hofmann's simulation $S_{\mathsf{MH}} \overset{\gamma_{Over}}{\longleftarrow} S_{\mathsf{spec}}$ into simpler components. (Function $traverse$ is a recursive optimization.)

Outline

1. Obtaining fancy breadth-first traversal
2. Analyzing fancy breadth-first traversal
3. Other non-strictly positive datatypes in use

Non-strictly positive datatypes to understand classical logic

For a given type A, the type $\sharp A := \mu X.(A + \neg\neg X)$ is “a bit bigger” than $\neg\neg A$: the second constructor ensures

$$\neg\neg\sharp A \to \sharp A \ ,$$

i. e., double negation elimination for $\sharp A$. $\neg\neg A$ also has double negation elimination, however, $\sharp A$ is freely constructed with this property—called the “stabilization” of A. Being “bigger” (as target of an embedding) is better since several proofs of strong normalization of variants of $\lambda\mu$-calculus suffered from erasure problems. Second-order $\lambda\mu$-calculus can be simulated inside system F with these types $\sharp A$ and their iteration principle, see my [R. M.] TLCA’01 paper and subsequent work.

Another case for verification by successive refinement?

In 2002, Danvy communicated to me [R. M.] a coroutine solution (again, according to his conceptual analysis) to the same fringe problem.

Example

They have the same fringe. For this problem, inner nodes are unlabeled.

Same fringe with routines—preparation

Let $\mathbb{B} := \{\mathsf{t}, \mathsf{f}\}$ and Rou now have the constructors $Over : Rou$ and $Next : \mathbb{N} \to ((Rou \to \mathbb{B}) \to \mathbb{B}) \to Rou$. Variable convention: $k : Rou \to \mathbb{B}$ "continuations", and $f : (Rou \to \mathbb{B}) \to \mathbb{B}$.

The critical function that needs elimination principles for Rou is $skim : Rou \to Rou \to \mathbb{B}$ with $skim\ Over\ Over \simeq \mathsf{t}$, result f for two arguments with different constructor and

$$skim (Next\, n_1\, f_1)(Next\, n_2\, f_2) \simeq if\, n_1 \neq n_2\, then\, \mathsf{f}\, else\, f_1(\lambda c^{Rou}.f_2(skim\, c))$$

This is an instance of Mendler-style iteration, but needs the same addition we needed before for $unfold$. It also nicely type-checks with sized types, as developed in the PhD thesis of Andreas Abel.

Same fringe program

Define $\mathit{walk} : \mathit{Tree} \to (\mathit{Rou} \to \mathbb{B}) \to ((\mathit{Rou} \to \mathbb{B}) \to \mathbb{B}) \to \mathbb{B}$ by

$$\begin{array}{lcl} \mathit{walk}\,(\mathit{Leaf}\ n)\,k\,f & \simeq & k(\mathit{Next}\,n\,f) \\ \mathit{walk}\,(\mathit{Node}\,l\,r)\,k\,f & \simeq & \mathit{walk}\,l\,k\,(\lambda k_1.\,\mathit{walk}\,r\,k_1\,f) \end{array}$$

Define $\mathit{canf} := \lambda k.k\,\mathit{Over}$ and $\mathit{init} : \mathit{Tree} \to (\mathit{Rou} \to \mathbb{B}) \to \mathbb{B}$ by $\mathit{init}\,t\,k := \mathit{walk}\,t\,k\,\mathit{canf}$. Finally, the function to detect if the trees have the same fringe, $\mathit{smf} : \mathit{Tree} \to \mathit{Tree} \to \mathbb{B}$, is defined by

$$\mathit{smf}\,t_1\,t_2 := \mathit{init}\,t_1\,(\lambda c_1.\,\mathit{init}\,t_2(\mathit{skim}\,c_1))$$

Is there a verification by successive refinement to demystify these operations?

Implementation in Coq

In a message to the Coq club—`https://sympa.inria.fr/sympa/arc/coq-club/2018-06/msg00096.html`—on the day following my [R. M.] TYPES 2018 talk, Simon Boulier, who attended the conference, announced the availability of a Coq plugin to deactivate the checks for strict positivity. He had already tested it with Martin Hofmann's program on the day of that talk.

Using Boulier's plugin, all the developments of the presented work other than the justification of termination by Mendler-style recursion can be directly replayed in Coq (but not in vanilla Coq that already rejects Rou).

This includes the definition of smf, so one can play with it and try to formulate and prove lemmas on it.

Conclusion

From the published abstract of TYPES 2018, but still worth remembering:
And this talk should remind the audience how much Martin's scientific insights were able to fascinate other researchers, even if they were not considered as ready to be published by Martin. Sadly, we have to live with these memories without further opportunities to get new notes from Martin or to work with him. May he rest in peace.