The Recursion Scheme of the Trace Function Method

Baltasar Trancón y Widemann
Department of Computer Science, University of Bayreuth, 95440 Bayreuth, Germany

Keywords: Formal Method, Trace Function Method, Executable Semantics, Recursion Theory, State System.

Abstract: The Trace Function Method (TFM) is a fundamental approach to the description of system behavior for requirements analysis, specification, and documentation. External behavior of systems or components is given in mathematically direct form, but with full abstraction from internal state, by defining output at discrete interface events as recursive functions of the complete history of previous interaction at the same interface, including both input and output. In order to understand and evaluate the semantics of the notation, and in particular the executable semantics, that is, the potential for automatic simulation and construction of prototype implementations from TFM descriptions, a recursion-theoretic analysis is given. It is demonstrated that a single run and the full reactive behavior of a TFM description can be presented as instances of first-order and higher-order course-of-value iteration, respectively. A simple sufficient condition for correct implementations of TFM descriptions in terms of state systems is given. The spectrum of possible state-based implementations of a TFM description, ranging from straightforward simulation to minimized state space, is explored. Implications for semantically calculated and hence formally verifiable prototype implementations are summarized.

1 INTRODUCTION

The trace function method (TFM), due to Parnas, is a fundamental approach to the description of system behavior. Its applications in software engineering range from requirements analysis (partial descriptions of required behavior) (Quinn et al., 2006), via design specification (complete descriptions of required behavior) (Baber et al., 2005; Liu et al., 2010) and simulation (Trancón y Widemann and Parnas, 2008) to documentation (descriptions of observed behavior at various levels of completeness and abstraction). As such, it is an integral part of the rational development process envisaged by (Parnas, 2009).

TFM abstracts fully from the internal structure and state of a system or component. All described behavior consists of discrete events at an interface. Interaction on events is specified by valuations of interface variables. Every interface variable is either input (controlled by the environment) or output (controlled by the system). The complete relevant history of behavior at the interface, organized as a list of events, most recent event first[1] is called a *trace*. A trace function is a total function[2], that maps traces to output values. The behavior of the system is described completely by giving a trace function for each output variable of the interface.

A trace that contains both input and output serves as precise documentation of a single instance of system behavior, independently of the trace functions that specify the general rules of behavior. A trace can be *validated* by comparing recorded outputs in the trace with specified outputs from trace functions; or it can be *reconstructed* from its input only, by recursively filling in specified outputs, oldest events first. Thus trace functions can act as either test oracles, or simulators or prototype implementations, respectively. In either case, there is a logical redundancy between historical records and abstract descriptions. The present article is an explication of the mathematical consequences of that redundancy. It is necessary to understand the way in which the redundancy is resolved, in order to give precise executable semantics to TFM, and hence derive correct implementations in a formally satisfactory manner.

[1]The original TFM style has the oldest event first. The reversal reflects the recursive structure better and will be justified in section 2.

[2]There is a relational extension of TFM that uses trace relations to describe nondeterministic behavior, which is not discussed here; the present arguments may be extended to cover the relational case nevertheless.

Trancón y Widemann B..
The Recursion Scheme of the Trace Function Method.
DOI: 10.5220/0003994501460155
In *Proceedings of the 7th International Conference on Evaluation of Novel Approaches to Software Engineering* (ENASE-2012), pages 146-155
ISBN: 978-989-8565-13-6

1.1 The Dilemma of Historical Records

Formally, output values of past events are recorded in a trace, even though they are redundant and can in principle be calculated from the output trace functions. That opens the syntactic possibility to have arbitrary pairings of input and output in trace events, not merely those that can actually be produced by the system and make up the graphs of the output trace functions. In philosophical terms, we may speak of pairings that arise from the given output functions as *factual*, and of those that do not as *counterfactual*. Then the question arises how to deal with the distinction when defining an output function: Should one assume the output values recorded in the trace to be factual, and retrieve them trustfully for subsequent computations? Or should one rather give a "skeptic" definition that calculates previous outputs recursively? Is there a semantic difference? Is there a difference in terms of system implementation effort and efficiency? The purpose of the present article is to use established mathematical theory to answer these questions.

1.2 Motivating Example and Discussion

As a running example, consider the case of a component with finite multiset functionality, keeping track of multiplicities of elements of some finite set X. The component supports four operations, namely counting the multiplicity of a given element, increasing the multiplicity of a given element by one up to an upper bound B, decreasing the multiplicity of a given element by one if present, and setting the multiplicity of all elements simultaneously to zero. Thus the input part of each event can be specified by a pair $(p,x) \in P \times X$, where $P = \{\mathsf{cnt}, \mathsf{inc}, \mathsf{dec}, \mathsf{clr}\}$, where x is irrelevant for $p = \mathsf{clr}$. The initial multiplicity of any element is zero. The output part of each event is an integer between zero and B. The TFM description of the component is depicted in Figure 1.[3] The trace function takes the "skeptic" approach discussed above, and assumes only input values are present in a trace. The variable T ranges over traces. The constant ε denotes the empty trace; an event prepended to a trace is denoted as $e \cdot T$ (cf. section 2.3). The auxiliary operation $T \triangleright x$ removes from a trace T the prefix of all events not concerning element x. Thus the result is always a sublist, or tail, of T.

The trace function *bag* depends recursively on its own result for some strictly reduced arguments, although the amount of reduction depends nontrivially and dynamically on the pattern of usage. But apart from the recursion scheme, only very elementary mathematics are used.[4] Intuitively, there should be a unique function *bag* that solves these recursive equations, and there should be a straightforward algorithm which effectively computes that function, perhaps not with optimal efficiency, but viable as a simulation, rapid prototype or test oracle.

1.3 Enter Recursion Theory

Since the expressive power of TFM comes from the way past outputs are used recursively in the computation of present outputs, it seems only fair to turn to recursion theory for the analysis of the method. We shall demonstrate that the adequate scheme of recursive function definition is known theoretically as *course-of-value (cov) iteration*. The phrase "course of value", or "course of values", can be traced back to the works of Frege, where it is roughly synonymous with "extension".

The difference between cov iteration and the more familiar scheme of *primitive recursion* is illustrated by analogy with proof by induction: In order to prove $P(n)$ for all natural numbers n, ordinary complete induction uses a step of the form $P(n) \Rightarrow P(n+1)$ plus a base case, often $P(1)$ or $P(0)$. An alternative, equivalent method uses a step $\big(P(k)$ for all $k < n\big) \Rightarrow P(n)$ instead. This method is often more convenient to use, and generalizes to transfinite induction, where it is called the *Noetherian* induction scheme. Primitive recursion is similar to ordinary induction, in using just a *single* preceding instance to infer the next, whereas cov iteration is free to use *all* preceding instances. Analogous arguments regarding (unchanged) absolute power and (improved) convenience apply.

The use of cov iteration as a theoretical tool for "intentional" program analysis has been proposed recently (Bonfante, 2011), although we currently reserve judgement on the relevance for our own investigations.

Note that the auxiliary function $\triangleright$ is also recursive, but of a simpler form that will be formalized as ordinary *iteration* below. The presence of recursion also in the way a trace function accesses events is responsible for the complexity of the multiset example. TFM employs a variety of access operations, ranging from static (splitting a trace into most recent event and rest), via semi-dynamic (splitting at the most recent event from a pre-determined set) to fully dynamic (splitting at the most recent event that satisfies an arbitrary relation with current input; filtering certain events from a trace). The choice of ac-

[3] Figures are grouped on the last page for comparison.

[4] The reader is indeed invited to try and come up with a simpler, precisely equivalent description.

cess operations is made on a pragmatic, ad-hoc basis. Below we shall give results that indicate vastly different implementation costs for different classes of access. Hence it may be prudent, and in accordance with the principle of Occam's Razor, to classify access operations accordingly, and to prefer light-weight over heavy-weight ones where applicable.

2 STRUCTURED RECURSION THEORY

The classical theory of recursive functions comes in two subtly different variants that must not be confunded. The theory of *μ-recursive* functions deals with *partial* functions for which computation may diverge. The theory of *primitive* recursive functions deals with *total computable* functions, a strict subset of the former. Since μ-recursion is Turing-complete, the question whether a μ-recursive function is defined for an argument is undecidable. Hence they are a model of computation rather than of abstract behavioral description. Undecidability, and partial functions that do not terminate, are unsuitable for executable semantics of TFM, where a two-valued predicate logic that may speak about definedness is used in function definitions (Parnas, 1993), but descriptions must be decidable in order to be useful.

Turner has made a similar remark with regard to the algebraic analysis of functional programs:

> *The existing model of functional programming, although elegant and powerful, is compromised to a greater extent than is commonly recognised by the presence of partial functions. (Turner, 2004)*

As a corollary, an implementation of TFM in a Turing-complete (partial) functional language such as ML is not a simple matter of writing down the equations in a different syntax. The mismatch between partial and total functions pervades all formal reasoning, and one is hardly better off in terms of correctness than, say, with a C++ implementation. When full-scale verification of closed programs is out of the question, such as in the proposed agile derivation of oracles, simulators and prototypes, a disciplined approach to code generation is needed. We shall demostrate that appropriate constraints can be calculated from recursion-theoretic investigations; the choice of the actual back-end language is secondary.

Turner's conclusion is to focus on the special case of total recursive functions by way of theoretically informed, disciplined function definition styles. Unfortunately, primitive recursion on the natural numbers, the subject of the classical total theory, is a natural format for a small class of functions only. Some other functions can be encoded in obfuscated ways such as Gödel coding, while many cannot, the most famous example being the Ackermann function. Hence the strengthening of both recursion schemes and datatypes has been given much attention. An important case in point is the Bird–Meertens formalism (Bird and de Moor, 1997), also known derogatorily as *Squiggol*, that focuses on recursion schemes for list-like datatypes. It is a useful tool in the analysis and verification of recursive functional algorithms, and in the calculation of correct-by-construction functional programs. As such it serves as the role model for our present investigations of TFM.

A fairly general theory of recursive functions with a recursion scheme able to accommodate TFM has been given in terms of universal categorial algebra and coalgebra (Uustalu and Vene, 1999). Since only a special case is needed for the present discussion, the relevant notions are summarized in this section in a less general, but more accessible form. No knowledge of category theory is presupposed. Novel and TFM-specific results are given in the following sections.

Universal algebra, the mathematical foundation of algebraic specification, is based on three notions:

- a *signature* introduces operations with specified argument numbers and types,
- *algebras* are set-theoretic models of signatures, realizing the operations as functions on some carrier set,
- *homomorphisms* are functions between the carrier sets of two algebras that are compatible with the realizations of operations in either algebra.

It is quite natural to represent universal algebra in category theory, due to the observation that signatures and homomorphisms can be defined simultaneously as a *functor* on the category of sets. Such a functor is a mapping T that takes any set A to another set $T(A)$ and each total function $f : A \to B$ to another total function $T(f) : T(A) \to T(B)$, with the side conditions that identical functions are taken to identical functions, and compositions to compositions: $T(\mathrm{id}_A) = \mathrm{id}_{T(A)}$ and $T(g \circ f) = T(g) \circ T(f)$, where $(g \circ f)(x) = g\big(f(x)\big)$.

2.1 Algebras and Iteration

For a functor T that expresses a signature, the T-algebras are pairs (A, α) of a carrier set A and a combined realization of operations $\alpha : T(A) \to A$. A T-algebra homomorphism between two algebras (A, α)

and (B,β) is a function $h : A \to B$ such that $h \circ \alpha = \beta \circ T(h)$. The well-known result that the *term algebra* of a signature has a unique homomorphism to any other algebra, understood as bottom-up term evaluation, is rephrased categorially as an *initial* object in the category of T-algebras: There is a T-algebra $(\mu T, \mathrm{in}_T)$ such that a unique homomorphism $(\!|\alpha|\!)_T$ to any other T-algebra (A,α) can be found.[5] [6] Lambek's Lemma states that in_T is bijective, that is μT is a fixpoint of the domain equation $T(X) \cong X$, and in fact the least one.

As our first example, consider the Peano signature of natural numbers. It is traditionally given as two operations zero (nullary) and succ (unary). The same can be expressed by a functor N that takes a set A to the set $1+A$, the disjoint union of a singleton set 1 and A. We write $\star$ for the injection of the element of 1 and a' for the injection of an element $a \in A$. The function part of the functor is given by $N(f)(\star) = \star$ and $N(f)(x') = f(x)'$.

The obvious interpretation as natural numbers with the number zero and the successor function can be turned into an initial N-algebra, by virtue of Peano's axioms: $\mu N = \mathbb{N}$, $\mathrm{in}(\star) = 0$ and $\mathrm{in}(n') = n+1$. For any N-algebra (A,α), the function $\alpha : 1+A \to A$ can be decomposed into a constant $z \in A$ such that $\alpha(\star) = z$, and a function $s : A \to A$ such that $\alpha(a') = s(a)$. The unique homomorphism $(\!|\alpha|\!) : \mathbb{N} \to A$ satisfies $(\!|\alpha|\!)(0) = (\!|\alpha|\!)\big(\mathrm{in}(\star)\big) = \alpha\big(N(\!|\alpha|\!)(\star)\big) = \alpha(\star) = z$, and $(\!|\alpha|\!)(n+1) = (\!|\alpha|\!)\big(\mathrm{in}(n')\big) = \alpha\big(N(\!|\alpha|\!)(n')\big) = \alpha\big((\!|\alpha|\!)(n)'\big) = s\big((\!|\alpha|\!)(n)\big)$. It can then be shown that $(\!|\alpha|\!)$ computes the *iteration* of s starting from z: $(\!|\alpha|\!)(n) = s^n(z)$. For instance, for the N-algebra $(\mathbb{N},\alpha)$ with $\alpha(\star) = 1$ and $\alpha(n') = 2n$, the homomorphism $(\!|\alpha|\!)$ computes the *powers of two*.

In imperative programming languages, iterations feature as the semantics of certain side-effect free loops: z, s and n denote the initial state, loop body effect (as a state update) and number of iterations, respectively. The intended result is the final state.

2.2 Coalgebras and Coiteration

As usual in category theory, the *dual* of a construction, obtained by reversing arrows, is also studied. The dual of a T-algebra is a T-coalgebra, a pair (C,γ) of a carrier set C and a realization of operations $\gamma : C \to T(C)$. A T-coalgebra homomorphism h between T-coalgebras (C,γ) and (D,δ) is a function $h : C \to D$ such that $T(h) \circ \gamma = \delta \circ h$. The dual of an initial algebra is a *final* coalgebra: There is a T-coalgebra $(\nu T, \mathrm{out}_T)$ such that a unique homomorphism $[\![\gamma]\!]_T$ from any T-coalgebra (C,γ) can be found. Dually to Lambek's Lemma, out_T is also bijective, and νT the greatest fixpoint of $T(X) \cong X$.

The final coalgebra view leads to an alternative interpretation of the Peano signature: N-coalgebras with realizations of operations of the type $\gamma : C \to 1+C$ can be understood as partial functions $\gamma : C \rightharpoonup C$, where a result of $\star$ or n' denotes undefinedness or the value n, respectively. The set $\overline{\mathbb{N}} = \mathbb{N} + \{\infty\}$ and the predecessor function can be turned into a final N-coalgebra: $\nu N = \overline{\mathbb{N}}$, $\mathrm{out}(0) = \star$, $\mathrm{out}(n+1) = n'$ and $\mathrm{out}(\infty) = \infty'$. The unique homomorphism $[\![\gamma]\!] : C \to \overline{\mathbb{N}}$ satisfies $[\![\gamma]\!](x) = 0$ if $\gamma(x) = \star$, and $[\![\gamma]\!](x) = [\![\gamma]\!]\big(\gamma(x)\big) + 1$ otherwise, with the special case $[\![\gamma]\!](x) = \infty$ if γ can be iterated indefinitely on x. It can then be shown that $[\![\gamma]\!]$ computes the *coiteration* of γ: $[\![\gamma]\!](c) = \min\{n \in \mathbb{N} \mid \gamma^{n+1}(c) = \star\}$, where powers of γ are obtained by strict composition of partial functions, and $\min \varnothing = \infty$. For instance, the *iterated logarithm* $\log^*$, as defined in algorithmic complexity theory, is the coiteration of the N-coalgebra $(\mathbb{R}, \log)$, where the function $\log x$ is restricted to arguments $x > 1$.

In imperative programming languages, coiterations feature as the semantics of certain side-effect free loops: The partial function γ denotes a combined state update and loop post-condition, starting from initial state c. The intended result is the number of iterations until the condition fails, with the possibility of non-termination.

2.3 From Numbers to Lists

The (finite or infinite) lists of elements from a set A are traditionally structured using the operations nil_A and $cons_A$. We abbreviate nil_A to ε and $cons_A(a,\ell)$ to $a \cdot \ell$, respectively.

The corresponding functor is $L_A(X) = 1 + (A \times X)$ on sets, and $L_A(f)(\star) = \star$ and $L_A(f)\big((a,\ell)'\big) = \big(a, f(\ell)\big)'$ on functions. The finite lists constitute an initial algebra: $\nu L_A = A^*$ with $\mathrm{in}_{L_A}(\star) = \varepsilon$ and $\mathrm{in}_{L_A}\big((a,\ell)'\big) = a \cdot \ell$. Dually, the finite and infinite lists constitute a final coalgebra: $\nu L_A = A^\infty = A^* \cup A^\omega$ with $\mathrm{out}_{L_A}(\varepsilon) = \star$ and $\mathrm{out}_{L_A}(a \cdot \ell) = (a,\ell)'$.

The similar functor $K_A(X) = A \times X$ is not very interesting from the algebraic viewpoint, because its initial algebra is empty. By contrast, the infinite lists constitute a final coalgebra: $\nu K_A = A^\omega$ with $\mathrm{out}_{K_A}(a \cdot \ell) = (a,\ell)$.

For numerous examples of iterative and coiterative functions on lists, their analysis and synthesis, see (Bird and de Moor, 1997).

[5] The "banana" bracket is due to (Meijer et al., 1991).

[6] Subscripts indicating the functor will be dropped where no confusion arises.

2.4 Generalizations

Iteration can be understood as a technique for the inductive definition of a function with initial algebra domain, $(|\alpha|)_T : \mu T \to A$, by giving a T-algebra (A, α) that defines one recursive step of the function. In the signature functor T, base and induction cases are dealt with together. The technique can easily be seen to subsume, for the Peano functor N, classical complete induction. The defining form $\alpha(a)$ is allowed to use only the recursive function results for immediate subarguments, $a-1$ in the case of N.

Not all functions are conveniently presented in this form, though: for instance, the *factorial* function $fac(n)$ requires both the recursive result $fac(n-1)$ and the original argument n, and the *Fibonacci* function $fib(n)$ requires two recursive results, $fib(n-1)$ and $fib(n-2)$. Consequently, more complicated patterns of recursion than the one catered for by iteration are also studied. Recursive functions that work like *fac* are handled with the *primitive* recursion pattern which, for the Peano functor, is the classical form of total recursion theory. Recursive functions that work like *fib*, or more generally, require recursive function results for arbitrary simpler arguments, are handled with the *course-of-value* iteration pattern, which we shall focus on for the following investigations.

Note that, in a setting with arbitrary datatypes definable as functors, all total recursion schemes are essentially equivalent in the sense that they can simulate each other in extended datatypes. Note also that *higher-order* functions, operating on datatypes containing computable functions, are strictly more powerful than the first-order functions, which are dealt with comprehensively in the classical theory via natural numbers and Gödel encoding. For instance, the archetypical example of a computable function that is not primitively recursive in the classical sense, the *Ackermann* function *ack*, turns out to be not just primitively recursive, but even iterative, in a higher-order setting: Consider the higher-order N-algebra $([\mathbb{N} \to \mathbb{N}], \alpha)$ over the space of iterative functions on $\mathbb{N}$, where $\alpha(\star)(n) = n+1$ and $\alpha(f') = (|\beta_f|)$, referring to the f-indexed family of nested first-order N-algebras $(\mathbb{N}, \beta_f)$ where $\beta_f(\star) = f(1)$ and $\beta_f(n') = f(n)$. To verify that $ack(m,n) = (|\alpha|)(m)(n)$ holds is left as an exercise to the reader.

2.5 Course-of-Value Iteration

The categorial form of course-of-value (cov) iteration (Uustalu and Vene, 1999) manages recursive function results by considering, besides the functor T whose initial algebra is the intended function domain, an extended functor T^C, defined as $T^C(X) = C \times T(X)$ and $T^C(f) = \mathrm{id}_C \times T(f)$, where C is the intended function range. It can be understood as specifying the same algebraic signature as T, but with an additional annotation of a value from C. The intuition is that T specifies one level of structure of a function argument, and T^C pairs that with the function result. Then νT^C is the space of arbitrarily (even infinitely) deeply nested function arguments with results annotated at all levels of nesting, and $T(\nu T^C)$ is the same, except that the result for the top level is missing. The whole point of cov iteration is to fill that hole. The cov induction principle can hence be put as follows:

> For any set C and generator $\varphi : T(\nu T^C) \to C$, there is a unique recursive function $\{|\varphi|\}_T : \mu T \to C$.[7]

This terse statement is much illuminated by an example. For the Peano functor N, the extension N^C looks very similar to the list functor L_C, except that the structure elements $(C\times)$ and $(1+)$ are exchanged. Consequently, the final N^C-coalgebra is almost the same, except that the empty list is excluded, making deconstruction total: $\nu N^C = C^{+\infty} = C^+ \cup C^\omega$ and $\mathrm{out}_{N^C}(c \cdot \ell) = (c, \ell)$. The set $N(\nu N^C)$ can then be understood as re-including the empty list, denoted by $\star$: $N(\nu N^C) \cong C^\infty$. We take the liberty to implicitly conflate the two sets. Now let $C = \mathbb{N}$ and consider the generator defined as $\varphi(\varepsilon) = 0$, $\varphi(a \cdot \varepsilon) = 1$ and $\varphi(a \cdot b \cdot \varepsilon) = a + b$. This defines the Fibonacci function: $fib = \{|\varphi|\}_N$.

Compared with ordinary iteration, cov functions often have multiple base cases, because their dependence on results for subarguments may exceed the nesting depth of the argument given. For the Fibonacci function, each value depends on the two preceding ones, so base cases for arguments less than two are required. The presentation can often be simplified, and all base cases handled with a single mathematical object, by specifying a default, infinite *surrogate* history that is appended to any finite input to the cov generator. Formally, the generator $\varphi : C^\infty \to C$ can be decomposed into an infinite list $h \in C^\omega$ and a generator without base cases $\overline{\varphi} : C^\omega \to C$, such that $\varphi = \overline{\varphi} \circ append(h)$. For some cov iterations, especially time-invertible ones, there is a natural candidate for h that eliminates the base cases completely. For the Fibonacci function, set $\overline{\varphi}(a \cdot b \cdot \ell) = a + b$ and $h = (+1) \cdot (-1) \cdots$; surrogate elements after the second are never used and hence arbitrary. Where no such natural surrogate history can be found, add a distinguished "end" element to C and handle base cases

[7] The exact relation of $\{|\cdot|\}$ to unique (co)algebra homomorphisms is technically involved and out of scope here.

internally. In either case, we may assume without loss of generality that cov iterators for the functor N are functions of infinite lists.

If a list functor L_A is used in the first place instead of N, we obtain $\nu L_A^C = (C \times A)^{+\infty}$ and $L_A(\nu L_A^C) = 1 + \big(A \times (C \times A)^{+\infty}\big)$. Hence $L_A(\nu L_A^C)$ corresponds to finite or infinite $(C \times A)$-lists where the C-part of the first element, if any, is missing.

3 APPLICATION TO TFM

3.1 Single Trace: First-order Iteration

Traces in TFM are lists of interface events, most recent events first, each consisting of input and output part, formalized as elements of sets $\mathbb{I}$ and $\mathbb{O}$, respectively. Hence *complete* traces are elements of $(\mathbb{O} \times \mathbb{I})^*$ in the finite case, or $(\mathbb{O} \times \mathbb{I})^\infty$ more generally. A trace function computes the output for the current input, and may depend on previous outputs, but of course not cyclically on the current output. Hence the arguments to trace functions are *incomplete* traces: either the trace is empty, or the first (current) element has no $\mathbb{O}$-part. Hence trace function definitions φ match the type of cov generators: $\varphi : L_{\mathbb{I}}(\nu L_{\mathbb{I}}^{\mathbb{O}}) \to \mathbb{O}$, giving rise to behavior functions $\{|\varphi|\}_{L_{\mathbb{I}}} : \mathbb{I}^* \to \mathbb{O}$ that map a complete history of inputs to the current output, computing previous outputs recursively behind the scenes.

Figure 2 shows the cov version of the multiset component description. The differences with respect to Figure 1 are subtle but significant: The type of bag_{i} is now such that $\{|bag_{\mathrm{i}}|\}_{L_{\mathbb{I}}} = bag_{\mathrm{r}}$ is the desired trace function. By virtue of the cov iteration operator, no explicit recursion is necessary. The auxiliary function $(\blacktriangleright)$ does not merely reduce the trace to be used as a recursive argument to *bag*, but may directly retrieve the results interspersed in the trace structure.

The first-order cov iterative form of trace functions already settles some of our research questions: The existence and uniqueness of solutions of recursive equations in TFM format are guaranteed. Furthermore, the functions thus defined are total and computable, with no regard to efficiency, by straightforward evaluation algorithms. The distinction between factual and counterfactual pairings of input and output is meaningful: Only the factual ones are relevant to the function to be defined, as evident from the fact that the type $\mathbb{O}$ does not occur in the domain of $\{|\varphi|\} : \mathbb{I}^* \to \mathbb{O}$. From the perspective of model parsimony, it seems best to disregard counterfactual pairings completely, and adopt the skeptic style, where past outputs are always computed recursively and never retrieved.

3.2 Full Behavior: Higher-order Iteration

The presentation of TFM in the preceding paragraphs is adequate from a first-order, relational viewpoint. Nevertheless, it may be worthwhile to consider a different presentation from a higher-order, functional viewpoint. Though technically much more ambitious, it provides additional insights, in particular concerning the relationship of TFM with other mathematical models of history-dependent system behavior; see below.

The first step is to separate the concerns of input and output, which are dealt with in logically different ways, and of the end of the trace, which are all conflated when a trace function is given as a cov generator $\varphi : L_{\mathbb{I}}(\nu L_{\mathbb{I}}^{\mathbb{O}}) \to \mathbb{O}$. To that end, we borrow a trick from practical stream programming, and embed finite lists into infinite lists (streams) by assuming a distinguished element $\square$ that occurs only and infinitely often at the end of streams (cf. surrogate histories above). Assume that $\mathbb{I}$ and $\mathbb{O}$ contain $\square$. A trace function is then given as a generator $\psi : \mathbb{I}^\omega \times \mathbb{O}^\omega \to \mathbb{O}$ that takes infinite backwards streams of inputs and outputs, respectively, where the inputs do include the current event but the outputs do not, and yields the current output. Since such streams will play a major role in the following, we call all infinite backwards streams *histories*, and additionally *recent* if they include the current value, and *ancient* if they do not.

It is easy to see that ψ is a natural rearrangement with respect to φ (recall the end of section 2.5). No information is lost; traces are merely split into independent input and output histories, recent and ancient, respectively, and padded with $\square$.

The second step is to observe also from the type of the iteration $\{|\varphi|\}_{L_{\mathbb{I}}} : \mathbb{I}^* \to \mathbb{O}$ that two pieces of information are required simultaneously, namely the number of steps (the *length* of the input trace) and the actual input of events (the *content* of the input trace). By systematic rearrangement, a cov iteration equivalent to $\{|\varphi|\}_{L_{\mathbb{I}}}$ but for the simpler functor N, thus only being recursive in the number of steps, can be constructed by "plugging" ψ into a generic construction F. The price for the simpler recursion signature is a more complicated carrier set, namely the space $[\mathbb{I}^\omega \to \mathbb{O}]$ of functions taking recent input histories to current outputs, called *responses*. Note that, since a trace function generally depends not only on input history but output history as well, it will be represented by a different response for every point in time marked by an event.

Higher-order cov iteration will be used to construct a recursive list of responses. Like in the Ack-

ermann example, we shall make use of the natural one-to-one correspondence between functions of types $X \times Y \to Z$ and $X \to [Y \to Z]$, respectively. As customary, we write $curry(f)$ for the invertible translation of a function f from the former to the latter type. The generic construction F is specified in terms of two auxiliary operations.

The first auxiliary operation has the type $F_1 : Q \to \mathbb{O} \times Q$ where $Q = [\mathbb{I}^\omega \to \mathbb{O}]^\omega \times \mathbb{I}^\omega$. It maps pairs of recent response and input histories to triples of the corresponding current output, and ancient response and input histories; formally: $F_1(r \cdot R, i \cdot I) = \big(r(i \cdot I), (R, I)\big)$. Note that (Q, F_1) has the form of a $K_\mathbb{O}$-coalgebra. The unique homomorphism $[\![F_1]\!]_{K_\mathbb{O}} : Q \to \mathbb{O}^\omega$ maps pairs of (recent/ancient) response and input histories to the corresponding output history. The higher-order function $curry\big([\![F_1]\!]_{K_\mathbb{O}}\big) : [\mathbb{I}^\omega \to \mathbb{O}]^\omega \to [\mathbb{I}^\omega \to \mathbb{O}^\omega]$ then maps response histories to the corresponding black-box behavior of the system, namely functions from input history to output history.

The second auxiliary operation has the type $F_2(\psi) : [\mathbb{I}^\omega \to \mathbb{O}^\omega] \times \mathbb{I}^\omega \to \mathbb{O}$. It maps pairs of previous black-box behavior and recent input history to current output, by passing recent input history and ancient output history, obtained by applying the behavior to ancient input history, to the generator ψ; formally: $F_2(\psi)(b, i \cdot I) = \psi\big(i \cdot I, b(I)\big)$. The higher-order function $curry\big(F_2(\psi)\big) : [\mathbb{I}^\omega \to \mathbb{O}^\omega] \to [\mathbb{I}^\omega \to \mathbb{O}]$ then maps the previous black-box behavior to the current response.

In synopsis the composition $F(\psi) = curry\big(F_2(\psi)\big) \circ curry\big([\![F_1]\!]_{K_\mathbb{O}}\big)$ maps ancient response histories to the current response. It can be proven equivalent with the first-order variant: $\{|\varphi|\}_{L_\mathbb{O}} \circ take(n) = \{|F(\psi) \circ pad|\}_N(n)$, where $take(n)$ discards all but the first n elements of history. Figure 3 shows the higher-order cov version of the multiset example.

The higher-order cov iteration form reveals a striking similarity (that has not been noticed before) between TFM and a class of stochastic models of system behavior that is immensely popular in time series modeling: The *auto-regressive moving average* (ARMA) models (Box and Jenkins, 1970) have a common vector space (most often simply real numbers) as both input and output, and calculate current output as a linear combination of recent input and ancient output history. Thus they subsume the Fibonacci example (where the linear combination is simply addition of the two preceding outputs) and are in turn subsumed by TFM (where the linearity assumption is lifted).

ARMA models can be used directly as filters that add auto-correlation to a signal in a controlled way. Or they can be used inversely, estimating the linear coefficients from given output, assuming that the input is purely random and of minimal variance. ARMA models may handle input and output histories differently: The most common case are models that have nonzero coefficients for the p most recent outputs and q most recent inputs, respectively, for finite ranks p and q. But models with genuinely infinite dependence on past outputs, effected by *fractional differencing* (Granger and Joyeux, 1980), are popular in economical and ecological applications (Montanari et al., 1997), because they exhibit the fractal properties often apparent in data from these areas.

4 TOWARDS IMPLEMENTATIONS

Structured total recursion schemes establish the well-definedness, uniqueness, totality and general computability of a function. Naive function evaluation, however, may not be appropriate to actually and efficiently compute values. Although the recursive form of the Fibonacci function is featured in virtually every textbook on recursive programming, it is widely acknowledged that the recursive algorithm is not useful at all in practice. A well-known equivalent algorithm, that reduces complexity from exponential to linear, can be given in terms of ordinary iteration or a loop, and two state variables instead of one. Mathematically it is specified concisely by iterated matrix multiplication: $ffib(n) = \binom{1}{0}^{\mathrm{T}} \binom{0\ 1}{1\ 1}^n \binom{0}{1}$. The implementation in an actual programming language is left as an exercise to the reader.

Some general questions arise: Is naive evaluation is generally inefficient for cov iterations? Can a more efficient, ordinary iteration be proven equivalent? If so, can it be found automatically? The short answers are, respectively: Yes, except for degenerate cases. Yes, but it may have unbounded space requirements. Yes, for certain disciplined patterns of recursive dependency.

The theoretical results about implementations of cov iteration discussed in the following section are novel and being published in a companion article (Trancón y Widemann, 2012).

4.1 Simulation by State Systems

For the discussion of simulation of cov iteration by state systems and ordinary iteration, we consider the functor N only for simplicity. The results then apply

to the higher-order cov representation of TFM. The function space lurking in the carrier set is not a technical problem: Object-level functions can either be represented intensionally by code (there is no need, for instance, to decide equality), or eliminated by *defunctionalization* (Reynolds, 1972), a standard technique to transform functional program expressions to first order. Similar results could be achieved in principle for the functor $L_{\mathbb{O}}$ to handle the first-order cov representation of TFM directly.

Fix a function range C. A *C-state system* is a triple (S,σ,τ), where S is a state space, $\sigma : N(\nu N^C) \to S$ is called the *abstraction* function and $\tau : C \times S \to S$ is called the *transition* function. It is called an *epi-state system*[8] if and only if σ is surjective. A state system is said to *factor* a cov generator $\varphi : N(\nu N^C) \to C$ if and only if two conditions hold: Firstly there is a function $\widetilde{\varphi} : S \to C$ such that $\varphi = \widetilde{\varphi} \circ \sigma$. For epi-state systems, $\widetilde{\varphi}$ is determined uniquely. A state system can correctly simulate the cov iteration of a generator only if this condition holds; otherwise the state is too coarse to make all the relevant distinctions. Secondly, it is required that $\tau\big(\varphi(\ell),\sigma(\ell)\big) = \sigma\big(\varphi(\ell)\cdot\ell\big)$ (note the similarity to homomorphism properties).

Whereas the first factoring condition is clearly necessary for simulation, it can be shown that adding the second one is sufficient. If the state system (S,σ,τ) factors the generator φ, then an ordinary iteration can be constructed from σ, τ and $\widetilde{\varphi}$ that simulates $\{|\varphi|\}$: There is a N-algebra $(C \times S, \rho)$ such that for every $\ell \in N^C(\nu N^C)$ there is some final state $s!$ such that $(|\rho|) = \big(\{|\varphi|\}(\ell), s!\big)$. The operation ρ is most concisely given in two parts $\rho = \rho_2 \circ \rho_1$, with $\rho_1(\star) = \sigma(\varepsilon)$ and $\rho_1\big((c,s)'\big) = s$, and $\rho_2(s) = \big(\widetilde{\varphi}(s), \tau(\widetilde{\varphi}(s), s)\big)$. The proof is too technical to be repeated here.

4.2 Regular Course-of-Value Iteration

A cov generator φ is called *k-regular*, for some natural number $k \geq 0$, if and only if it is determined completely by exactly k previous results; formally, there is a surrogate history $h \in C^k$ and $\widehat{\varphi} : C^k \to C$ such that $\varphi = \widehat{\varphi} \circ \mathit{take}(k) \circ \mathit{append}(h)$.[9] Then $\{|\varphi|\}$ can be simulated on constant space, namely with a first-in-first-out buffer of k elements of C as state: the state system (C^k,σ,τ) with $\sigma = \mathit{take}(k) \circ \mathit{append}(h)$ and $\tau\big(c_0,(c_1,\ldots,c_k)\big) = (c_0,\ldots,c_{k-1})$ satisfies the conditions given above, with $\widetilde{\varphi} = \widehat{\varphi}$. The proof involves straightforward manipulations of the iterative functions *take* and *append* and is left as an exercise to the reader. A real implementation would probably use a *ring* buffer, where the elements are not shifted, but remain stationary and are addressed modulo k. This is easily shown to be equivalent to the former representation.

The Fibonacci function is regular: set $k = 2$ and $h = (+1,-1)$ to retrieve the efficient iterative algorithm, albeit in the slightly modified form $\mathit{ffib}(n) = \binom{1}{1}^{\mathrm{T}} \binom{0\ 1}{1\ 1}^{n} \binom{-1}{+1}$. TFM descriptions that refer statically to the k most recent events are k-regular as well. TFM descriptions that refer to the most recent event fulfilling some predicate require more complicated state spaces. In the multiset example, an X-indexed family of the most recent multiplicity per element would do. Of course, a good programmer would obviously come up with an implementation as an array or hashtable. But it can also be inferred automatically from the recursive access pattern. Similar state constructions can be given for many of access patterns of TFM, and organized in a "compiler" that realizes trace functions as ordinary iterations.

4.3 Category of Implementations

The algebras for a functor T and their homomorphisms constitute a category. The initial T-algebra is just the initial object in that category, and serves as a generic model of *syntax* with respect to the signature T. Dually, the T-coalgebras and their homomorphisms constitute another category. The final T-coalgebra is the final object in that category, and serves as a generic model of *semantics* with respect to the signature T. This duality has been used, for instance, to give structured semantics to abstract data types (Erwig, 1998).

The state systems that factor a fixed cov generator φ can also be organized as a category, with both initial and final objects, and analogous interpretations as syntax and semantics, respectively. Abbreviate the function which the second factoring condition is about as $\delta(\ell) = \tau\big(\varphi(\ell),\sigma(\ell)\big)$. A homomorphism between two state systems (S_1,σ_1,τ_1) and (S_2,σ_2,τ_2), both factoring a common φ, is a function $h : S_1 \to S_2$ such that $h \circ \sigma_1 = \sigma_2$ and $h \circ \delta_1 = \delta_2$.[10]

The initial object in the category of φ-factoring state systems is the trivial state system $\big(N(\nu N^C) = C^\infty, \mathrm{id}, (\cdot)\big)$. It qualifies as purely syntactical: Histories are taken at face value, no abstraction occurs, and the state transition merely accumulates results.

[8] From the categorial notion of *epimorphisms*, which coincide with surjective functions in the category of sets.

[9] Note that $\widehat{\varphi}$ is a function of finite tuples of uniform length, and hence the opposite of the "infinitarized" $\overline{\varphi}$ discussed above.

[10] In category jargon, this yields a *double coslice* category.

It is also the most straightforward implementation, and already avoids the exponential blowup incurred by naive evaluation, since common recursive subcomputations are properly shared. But it clearly requires ever-growing amounts of space, since potentially irrelevant historical data is never discarded, and may hence not qualify as a viable implementation for all purposes.

The final object in the category of φ-factoring state systems is the *coimage* system $(\mathrm{Coim}(\varphi), \pi, \tau)$, where $\mathrm{Coim}(\varphi)$ is the partitioning of $N(\nu N^C)$ into the equivalence classes that are identified by φ, π is the mapping of elements to their equivalence class, and τ is a complicated construction that can be shown to exist by the Axiom of Choice. This system qualifies as purely semantical: its states are canonical representants for the relevant historical information, but no hint as to their practical encoding is given. Hence the final system is impractical as an implementation, as the equivalence classes can be hard and inefficient to construct, but it completely eliminates all redundancy in historical information, and it is therefore the ideal, minimal model of behavior real implementations should aspire to.

5 CONCLUSIONS

We have demonstrated that description in the style of the trace function method, which take the form of trace functions of a certain recursive form, are amenable to structured total recursion theory. That is, there is an alternative non-recursive form (a generator) that induces the desired recursive trace function as the unique solution of a certain homomorphism equation. This result should not be misread as meaning TFM *should* be notated with generators. But the mere check that a trace function *could* be generated in this way entails a number of beneficial semantic properties: First of all, generators are necessarily *consistent* and *complete*, in the sense that one and only one solution exists. The solution is also totally defined and computable, making a TFM specification *executable*, in the sense that straightforward effective implementations exist, in the form of primitively recursive total functional programs, or equivalently a certain kind of loops, for the more imperative-minded.

The particular recursion scheme we have identified as adequate for TFM, namely course-of-value iteration, elucidates the role of historical dependencies in trace functions at a level of abstraction and with a precision syntactic and ad-hoc algebraic arguments cannot. On the one hand, the apparent dilemma discussed in section 1.1 can be resolved: References to recursive function results and to stored outputs in traces are redundant, but not in a logical conflict. The cov iteration approach separates the two cleanly; stored values are retrieved in the generator view on a trace function, and map consistently to recursive results in the induced trace function. Counterfactual events are guaranteed to be functionally irrelevant. On the other hand, TFM can be related precisely to established history-aware modeling paradigms, for instance the ARMA approach, considered state of the art for data-driven modeling in diverse empirical fields such as economics and hydrology.

Calculating implementations for TFM description that are practically and immediately usable as simulators, test oracles or prototypes has so far posed difficulties. We have demonstrated that most of the complexity is due to the recursion scheme. Dealing with that complexity in the appropriate theoretical framework is certainly going to help. Finding adequate iterative implementations can be understood theoretically as movement along the initial–final axis of factoring state systems, driven by a tradeoff between cheap state encoding (initial extreme) and strong compression of information (final extreme). From a more tool-oriented viewpoint, the classification of recursive access patterns in TFM, with respect to the costs of the associated state system features, could lead to a layered definition of TFM basic vocabulary, enabling a conscious tradeoff between power of expression and performance of automatically derived correct implementations.

$$
\begin{aligned}
bag_{\mathrm{r}} &: \mathbb{I}^* \to \mathbb{O} & \mathbb{I} &= P \times X \\
(\triangleright) &: \mathbb{I}^* \times X \to \mathbb{I}^* & \mathbb{O} &= \{0, \ldots, B\}
\end{aligned}
$$

$$
\begin{aligned}
bag_{\mathrm{r}}(\varepsilon) &= 0 \\
bag_{\mathrm{r}}\big((\mathsf{cnt}, x) \cdot T\big) &= bag_{\mathrm{r}}(T \triangleright x) \\
bag_{\mathrm{r}}\big((\mathsf{inc}, x) \cdot T\big) &= \min\big(bag_{\mathrm{r}}(T \triangleright x) + 1, B\big) \\
bag_{\mathrm{r}}\big((\mathsf{dec}, x) \cdot T\big) &= \max\big(bag_{\mathrm{r}}(T \triangleright x) - 1, 0\big) \\
bag_{\mathrm{r}}\big((\mathsf{clr}, x) \cdot T\big) &= 0
\end{aligned}
$$

$$
\begin{aligned}
\varepsilon \triangleright x &= \varepsilon \\
\big((p, y) \cdot T\big) \triangleright x &= \begin{cases} (p, y) \cdot T & \text{if } x = y \text{ or } p = \mathsf{clr} \\ T \triangleright x & \text{if } x \neq y \text{ and } p \neq \mathsf{clr} \end{cases}
\end{aligned}
$$

Figure 1: TFM-style multiset, recursively.

$$bag_{\mathrm{i}} : L_{\mathbb{I}}(\nu L_{\mathbb{I}}^{\mathbb{O}}) \to \mathbb{O} \qquad \mathbb{I} = P \times X$$
$$(\blacktriangleright) : \mathbb{I}^* \times X \to \mathbb{O} \qquad \mathbb{O} = \{0, \ldots, B\}$$
$$bag_{\mathrm{i}}(\varepsilon) = 0$$
$$bag_{\mathrm{i}}\big((\mathsf{cnt}, x) \cdot T\big) = T \blacktriangleright x$$
$$bag_{\mathrm{i}}\big((\mathsf{inc}, x) \cdot T\big) = \min\big((T \blacktriangleright x) + 1, B\big)$$
$$bag_{\mathrm{i}}\big((\mathsf{dec}, x) \cdot T\big) = \max\big((T \blacktriangleright x) - 1, 0\big)$$
$$bag_{\mathrm{i}}\big((\mathsf{clr}, x) \cdot T\big) = 0$$
$$\varepsilon \blacktriangleright x = 0$$
$$\Big((n, (p, y)) \cdot T\Big) \blacktriangleright x = \begin{cases} n & \text{if } x = y \text{ or } p = \mathsf{clr} \\ T \blacktriangleright x & \text{if } x \neq y \text{ and } p \neq \mathsf{clr} \end{cases}$$

Figure 2: TFM-style multiset, first-order cov generator.

$$bag_{\mathrm{h}} : \mathbb{I}^{\omega} \times \mathbb{O}^{\omega} \to \mathbb{O} \qquad \mathbb{I} = P \times X$$
$$(\succ) : \mathbb{I}^* \times X \to \mathbb{O} \qquad \mathbb{O} = \{0, \ldots, B\}$$
$$bag_{\mathrm{h}}(\Box \cdot I, O) = 0$$
$$bag_{\mathrm{h}}\big((\mathsf{cnt}, x) \cdot I, O\big) = (I, O) \succ x$$
$$bag_{\mathrm{h}}\big((\mathsf{inc}, x) \cdot I, O\big) = \min\big(((I, O) \succ x) + 1, B\big)$$
$$bag_{\mathrm{h}}\big((\mathsf{dec}, x) \cdot I, O\big) = \max\big(((I, O) \succ x) - 1, 0\big)$$
$$bag_{\mathrm{h}}\big((\mathsf{clr}, x) \cdot I, O\big) = 0$$
$$(\Box \cdot I, O) \succ x = 0 \qquad (I, \Box \cdot O) \succ x = 0$$
$$\begin{pmatrix} (p, y) \cdot I, \\ n \cdot O \end{pmatrix} \succ x = \begin{cases} n & \text{if } x = y \text{ or } p = \mathsf{clr} \\ (I, O) \succ x & \text{if } x \neq y \text{ and } p \neq \mathsf{clr} \end{cases}$$

Figure 3: TFM-style multiset, higher-order cov generator.

ACKNOWLEDGEMENTS

Parts of this work have been performed at the Software Quality Research Laboratory, University of Limerick, Ireland, supported by Science Foundation Ireland under Grants 01/P1.2/C009 and 03/CE3/1405.

REFERENCES

Baber, R. L., Parnas, D. L., Vilkomir, S. A., Harrison, P., and O'Connor, T. (2005). Disciplined methods of software specification: A case study. In *ITCC (2)*, pages 428–437. IEEE Computer Society.

Bird, R. and de Moor, O. (1997). *Algebra of Programming*, volume 100 of *International Series in Computing Science*. Prentice Hall.

Bonfante, G. (2011). Course of value distinguishes the intentionality of programming languages. In *Proceedings 2nd Symposium on Information and Communication Technology (SoICT '11)*, pages 189–198. ACM.

Box, G. E. and Jenkins, G. M. (1970). *Time series analysis: Forecasting and control*. Holden–Day, San Francisco.

Erwig, M. (1998). Categorical programming with abstract data types. In Haeberer, A. M., editor, *AMAST*, number 1548 in Lecture Notes in Computer Science, pages 406–421. Springer.

Granger, C. W. J. and Joyeux, R. (1980). An introduction to long-memory time series models and fractional differencing. *Journal of Time Series Analysis*, 1:15–30.

Liu, Z., Parnas, D. L., and Trancón y Widemann, B. (2010). Documenting and verifying systems assembled from components. *Front. Comput. Sci. China*, 4(2):151–161.

Meijer, E., Fokkinga, M. M., and Paterson, R. (1991). Functional programming with bananas, lenses, envelopes and barbed wire. In Hughes, J., editor, *Proceedings 5th International Conference on Functional Programming Languages and Computer Architecture (FPCA 1991)*, number 523 in Lecture Notes in Computer Science, pages 124–144. Springer.

Montanari, A., Rosso, R., and Taqqu, M. S. (1997). Fractionally differenced arima models applied to hydrologic time series: Identification, estimation, and simulation. *Water Resources Research*, 33(5):1035–1044.

Parnas, D. L. (1993). Predicate logic for software engineering. *IEEE Trans. Softw. Eng.*, 19:856–862.

Parnas, D. L. (2009). Document based rational software development. *Knowledge-Based Systems*, 22(3):132–141.

Quinn, C., Vilkomir, S. A., Parnas, D. L., and Kostic, S. (2006). Specification of software component requirements using the trace function method. In *Proceedings International Conference on Software Engineering Advances (ICSEA 2006)*, page 50. IEEE Computer Society.

Reynolds, J. (1972). Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference*, pages 717–740, Boston, Massachusetts.

Trancón y Widemann, B. (2012). State-based simulation of linear course-of-value iteration. In *Proceedings 11th International Workshop on Coalgebraic Methods in Computer Science (CMCS 2012)*. Short contribution.

Trancón y Widemann, B. and Parnas, D. L. (2008). Tabular expressions and total functional programming. In Chitil, O., Horváth, Z., and Zsók, V., editors, *Implementation and Application of Functional Languages (IFL 2007), Revised Selected Papers*, number 5083 in Lecture Notes in Computer Science, pages 219–236. Springer.

Turner, D. A. (2004). Total functional programming. *Journal of Universal Computer Science*, 10(7):751–768.

Uustalu, T. and Vene, V. (1999). Primitive (co)recursion and course-of-value (co)iteration, categorically. *Informatica*, 10(1):5–26.