Total Functional Software Engineering
Overview Paper (Extended Abstract)

Baltasar Trancón Widemann

Programming Languages and Compilers
Ilmenau University of Technology
baltasar.trancon@tu-ilmenau.de

Abstract. Methods for mathematically basic and precise description of system behavior at discrete interfaces have been developed by David Parnas and his groups and collaborators over many years. Total functions can play a crucial role as constructive and effectively executable semantics for various levels of these descriptions. Straightforward analysis and transformation techniques for functional programs, particulary effective for total functions, can be used as significant steps towards automated generation of implementations. Theoretical claims are supported by practical examples. The focus is on insight into applications from the functional perspective rather than on innovations in functional programming itself.

1 Introduction

The software engineer David Parnas has been influential, besides many other areas, in the development of a particular, mathematically sound methodology for the description and specification of system behavior. The general style of the work of his groups in the Software Quality Research Laboratories at McMaster University, Ontario and the University of Limerick, and many collaborators, can be summarized in the following two maxims:

1. The essential complexity of real-world systems must be acknowledged, and met with appropriately scalable methods, but
2. the mathematics underlying these methods must be as simple and rigorous as possible.

In recent years, it has emerged that many of the proposed methods can be understood in a framework of total functional programming. This paradigm shift, away from the original presentation in elementary set theory, implies multiple illuminating changes in perspective:

1. *Algebraic structure is uncovered.* Method design choices that have been based on practical experience and justified pragmatically by mathematical fitness and economy, can be strengthened theoretically by being mapped to natural algebraic constructs.

2. *Executability is emphasized.* In contrast to descriptive set-theoretic models where effective evaluation procedures are implicit meta-information, the functional programming view puts denotational and operational semantics on equal footing.
3. *Tools are leveraged.* Beyond executability in principle, actual implementations are needed at the end of the day, to serve as test oracles, simulators or prototypes. In the traditional set-theoretic approach, there is a choice between confinement to some theorem prover sandbox, and naive translation to a general-purpose programming language, with all the associated pitfalls. The tried and proven tools of functional programming language implementation support symbolic evaluation and code generation in a way that is both reliable and flexible.

The current state of the art is such that the theoretical basis is established fairly comprehensively. By contrast, the practical side of real tools is basically nonexistent, apart from academic prototypes with limited scope and service lifetime. The purpose of the present paper is to serve as a conceptual reference for future implementations.

The following three sections each discuss one particular method that builds on the preceding. Note that the focus here is on the application of functional technologies outside their principal programming domain. Most of the material discussed in the following is of little novelty from the functional programming perspective proper, but summarizes, condenses, and in places even improves the understanding of the subject matters with respect to their traditional presentation.

2 Predicate Logic

The Parnas approach to algebraical–logical language [22] differs from most other software engineering methodologies by treating algebraic (value-level) expressions as *partial*, but logic (truth-level) expressions as *total*.

2.1 Two Worlds

The algebraic world is one of strict partial functions: An expression of the form $f(e_1, \ldots, e_n)$ is defined if and only if all subexpressions e_i are defined, and the tuple $(v_1, \ldots, v_n)$ of their respective values is in the domain of f. Denotationally, every value type has a bottom element. By contrast, the logic world is one of total predicates: An expression of the form $p(e_1, \ldots, e_n)$ is true if all subexpressions e_i are defined, and the tuple $(v_1, \ldots, v_n)$ of their respective values is in the extension of p, and false otherwise. That is, the type of truth values has no bottom element. Case distinction operators have condition arguments of truth value type, reflexively embedding the logic world in the algebraic world.

This account of partiality is in line with the IEEE 754 standard for floating-point arithmetics, where comparison operators on numbers totalize in the same

way with respect to the bottom value `NaN`. It is, however, distinct from the Z [8] approach, where logic is three-valued with an *undetermined* bottom element, predicates are strict, but logical connectives are non-strict in both arguments. For example, in the expression $0/0 = 1 \vee 1 = 1$, the first clause is false according to [22], but undetermined according to Z, whereas the whole epression is true in either case, albeit for different reasons.

The special role of the bottom element has profound implications on evaluation strategies, which can be seen clearly from basic considerations of denotational semantics of functional programs: A definedness predicate can be formalized within the language, simply as "$\mathrm{defined}\, e = \mathfrak{t}$" which entails that each evaluation strategy must

- either have a solvable halting problem,
- or wrongly assign the value bottom rather than zero to some instances of the expression scheme

$$\mathrm{if\_then\_else}(\mathrm{defined}\, e, 1, 0)$$

2.2 Enter Total Functions

An elegant solution of this dilemma is to restrict the expression language such that it can be interpreted in a strongly normalizing calculus, thus reducing the halting problem to triviality.

Such calculi define total rather than partial functions; the partiality of algebraic expressions is emulated adequately by the monad $M = (-) + 1$, known as *Maybe* in Haskell. It comes with the natural transformations $\eta_X : X \to M(X)$ (unit, *return* in Haskell) and $\mu_X : M^2(X) \to M(X)$ (multiplication), as well as the family of constants $\bot_X \in M(X)$. Partial functions of type $A \rightharpoonup B$ are encoded as $A \to M(B)$. Values of type A are encoded as $M(A)$. Strict, checked application is given by the operator

$$\frac{e : M(A) \quad f : A \to M(B)}{e \triangleright f : M(B)} \qquad e \triangleright f = \mu_B\big(M(f)(e)\big)$$

known as *bind* in Haskell.

An emulation of the intended partial language can then be given by induction over the syntactic structure:

- Constants and variables are taken to be always defined.

$$\frac{c : A}{c^\dagger : M(A)} \qquad c^\dagger = \eta_A(c)$$

- The pseudo-constant $\star$ denotes an atomic undefined expression.

$$\frac{\star_A : A}{\star_A^\dagger : M(A)} \qquad \star_A^\dagger = \bot_A$$

– Tupling (only binary shown for simplicity) is strict.

$$\frac{(a_1, a_2) : A_1 \times A_n}{(a_1, a_2)^\dagger : M(A_1 \times A_2)} \quad (a_1, a_2)^\dagger = a_1^\dagger \triangleright \left(\lambda\, x_1.\, a_2^\dagger \triangleright \left(\lambda\, x_2.\, \eta_{A_1 \times A_2}(x_1, x_2)\right)\right)$$

– References to partial functions are Kleisli-extended.

$$\frac{f : A \rightharpoonup B}{f^\dagger : M(A) \to M(B)} \qquad f^\dagger = \_ \triangleright f$$

– References to predicates are totalized sending bottom to false.

$$\frac{p : A \to \mathbb{B}}{p^\dagger : M(A) \to \mathbb{B}} \qquad p^\dagger = [p, \mathrm{const}\, \mathsf{f}]$$

– Emulation distributes over application.

$$\big(s(e_1, \ldots, e_n)\big)^\dagger = s^\dagger\big((e_1, \ldots, e_n)^\dagger\big)$$

It is easy to see that this emulation preserves well-typing, and extends to function and predicate definitions.

2.3 Simplification

The administrative operations inserted by the emulation complicate the structure of expressions considerably at first sight. But fortunately, a substantial part can be eliminated by straightforward partial evaluation and simplifications using the monad laws. In particular we have:

$$f^\dagger\big(\eta(x)\big) = \eta(x) \triangleright f = f(x) \qquad p^\dagger\big(\eta(x)\big) = [p, \mathrm{const}\, \mathsf{f}]\big(\eta(x)\big) = p(x)$$

Note that such program transformations are rather easier, and can be applied more aggressively, in a strongly normalizing setting.

For instance, consider a definition of partial function composition:

$$\begin{aligned} \mathit{compose}(g, f)(x) &= (g(f(x)))^\dagger = g^\dagger(f^\dagger(x^\dagger)) \\ &= \eta(x) \triangleright f \triangleright g \\ &= f(x) \triangleright g \end{aligned}$$

Here the inner application is reduced to unchecked form because x is necessarily defined, seeing that applications to undefined arguments are handled at the call site. The outer application needs to remain checked for spontaneous undefinedness of the subexpression $f(x)$.

If additionally f is total by construction, that is $f = \eta \circ f'$ (not an uncommon case, see for example the definition of partial tupling), then we can reduce this further and eliminate another check:

$$\begin{aligned} \mathit{compose}(g, \eta \circ f')(x) &= \eta\big(f'(x)\big) \triangleright g \\ &= g\big(f'(x)\big) \end{aligned}$$

In summary, a reflexive combination of partial algebraical and total logical language can be represented faithfully in a calculus of total functions, by using a well-known monadic lifting. Locally total subexpressions are reduced to their natural form by straightforward simplification of the resulting monadic expressions. This is of course a fairly banal insight in a functional programming context. But it is nowhere nearly as automatic and self-evident in contexts of set-theoretic proof systems or ad-hoc code generators, the standard tools of system engineers, where partiality is an implicit side condition rather than integral part of data flow.

2.4 Discussion: Expressive Power

Of course the proposed framework, based on a calculus of total functions, has a significant limitation: The language that can be interpreted in it has to be quite restricted, compared with the full power of first-order predicate logic that would be available (albeit incompletely operationalized) in a theorem prover environment. Expressing an arbitrary nontrivial algorithm in terms of constructively total functions is known to be a difficult task.

Fortunately, the existing method base definitions and the examples suggest that very simple data structures and algorithms go a long way. Simple algebraic datatypes and primitive recursive access functions, whose representation in strongly normalizing calculi is well-understood, make up most of the framework. By contrast, particularly troublesome features, notably infinite set comprehensions and general recursive function definitions, are apparently not required.

This finding suggests an interesting, open philosophical question: is the computational simplicity just a happy coincidence, or are mathematically more involved constucts pragmatically ill-suited to the task of behavioral description of systems, simply because they are harder to understand for the human engineer?

3 Tabular Expressions

Classical mathematics are heavily biased in terms of algebraically simple functions which have a homogenous representation as a simple expression with one or more variables over their whole domain. Mildly heterogeneous definitions such as piecewise definitions for a domain partitioned into intervals are admissible, but more general case distinctions are generally avoided. By contrast, the theory of computation in computer science is discrete by nature, and case distinctions feature pervasively and nestedly in function definitions.

Where case distinctions are made in logically rigorous descriptive formalisms, it is important to ascertain that cases are non-contradictory and complete. Functional programming provides a notation for case distinction that is powerful, theoretically elegant and easy to implement, namely by pattern matching on algebraic datatypes. Unfortunately this approach has little acceptance in system engineering contexts. Engineers traditionally favour a different form, namely tabular expressions, where alternative cases are laid out spatially as columns or rows of a table. The tabular notation has attractive pragmatic advantages [20]:

1. It scales from simple two- and three-way distinctions to extremely complex expressions where many-way and/or hierachical case distinctions along multiple, more or less orthogonal criteria are combined.
2. It is fairly easy and intuitive to read for domain experts without formal training in symbolic programming, and can be used, edited and archived effectively in paper form.
3. It supports manual and machine-supported inspection, validation and verification of descriptions by systematic coverage of rows, columns or cells; applications include soundness and completeness proofs as well as compliance checks and test suite design [4–6].

3.1 Simple Example

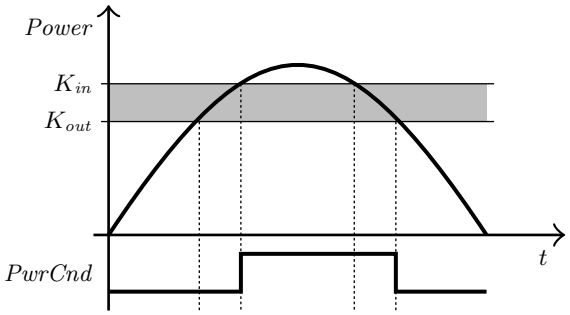
Fig. 1 shows a simple real-world tabular expression. It appeared in inspection documents of the Darlington Nuclear Power Generation Station [14, 15]. In the shutdown system, parts of the monitoring logic only apply near maximum power. The pertaining sensors are "conditioned in" (activated) above a certain power level and "conditioned out" (deactivated) below. To avoid a *jitter* effect (high-frequency switching events), the respective threshold levels K_{in} and K_{out} are set to slightly different values, thus introducing artificial hysteresis. Between the two, the previous value is maintained. See the top half of Fig. 1 for an illustration of the behavior, and the bottom half for the tabular description as a function of the theshold parameters, the current power level and the previous value.

The function definition is organized as a one-dimensional decision table. The top (*header*) (1×3)-grid of truth-level expressions specifies a three-way case distinction. The bottom (*main*) (1×3)-grid of value-level expressions contains the respective function results. Note that values are Boolean by accident, but the main grid is three-valued $(\mathit{true}, \mathit{false}, \bot)$ whereas the header grid is two-valued.

The function is simple in structure and does not appear to merit the tabular form at first sight. But closer inspection reveals a subtlety that illustrates why symmetric case distinctions, as expressed by a header grid, are sometimes superior to if-then-else cascades or first-fit pattern matching: The cases in the upper grid are only consistent and complete under the implicit constraint $K_{out} < K_{in}$. Hence a thorough inspection of the table, either by a human expert or by a theorem prover (such as PVS, which found the error in actual fact [14]), will reveal that the description is formally deficient. By contrast, a cascading definition where each case implies the negation of the preceding, would just silently go wrong.

3.2 Complex Example

Fig. 2 shows a complex tabular expression. It specifies a test procedure for computer keyboards [27]. The general idea is fairly simple: all keys are to be pressed in order, and if all registered keycodes correspond to the expected sequence, the keyboard passes the test. This homogeneous principle is then complicated by



$PwrCnd(Prev : bool; Power, K_{in}, K_{out} : real) : bool \equiv$

$Power \leq K_{out}$	$K_{out} < Power < K_{in}$	$Power \geq K_{in}$
false	*Prev*	*true*

Fig. 1. Simple tabular expression: power conditioning for sensors

$N(T) \equiv$

$keyOK(T)$			$T = \_$	$T \neq \_$		
				$N(p(T)) = 1$	$1 < N(p(T)) < L$	$N(p(T)) = L$
$\neg keyOK(T)$	$\neg keyEsc(T)$	$pkeyOK(T)$		$N(p(T)) + 1$		*Pass*
		$\neg pkeyOK(T) \wedge$ $pkeyEsc(T) \wedge$ $ppkeyOK(T)$			$N(p(T)) - 1$	
		$\neg pkeyOK(T) \wedge$ $pkeyEsc(T) \wedge$ $\neg ppkeyOK(T)$				
		$\neg pkeyOK(T) \wedge$ $\neg pkeyEsc(T)$	1		$N(p(T))$	
	$keyEsc(T)$	$\neg pkeyEsc(T)$				
		$pekeyEsc(T)$				
		$pkeyEsc(T) \wedge$ $\neg pekeyEsc(T)$			*Fail*	

where $keyOK(T) \equiv r(T) = N(p(T))$ $\quad$ $keyEsc(T) \equiv r(T) = Esc$

$pkeyOK(T) \equiv keyOK(p(T))$ $\quad$ $pkeyEsc(T) \equiv keyEsc(p(T))$

$ppkeyOK(T) \equiv keyOK(p^2(T))$ $\quad$ $pekeyEsc(T) \equiv N(p^2(T)) = Esc$

Fig. 2. Complex tabular expression: computer keyboard checking procedure

provisions for correcting errors both of the hardware and of the human tester by pressing the escape key, without exempting the escape key from the test sequence. The tabular expression is the result of a formal analysis of several pages of prose, eliminating several ambiguities, inconsistencies and loopholes.

The function N specifies the number of the next key to be pressed or a verdict (*Pass* or *Fail*), depending on the sequence T of keys pressed so far. The constant _ denotes the empty sequence. Nonempty sequences T can be deconstructed into a most *recent* event $r(T)$ and a *previous* sequence $p(T)$.

The tabular form highlights several features that are hard to emulate in functional programming style or any other textual format:

1. The table is two-dimensional: the chosen variant depends on two more or less orthogonal case distinctions. They are specified by the top and left header grids of truth-level expressions, respectively. Together they select a value-level cell of the bottom-right main grid at the spatial intersection of their axes. The choice which of these to evaluate first is completely arbitrary.
2. Each header is hierarchical, having a binary decision tree structure ending in flat two-or-more-way distinctions. The choice of decision criteria and order of flat distinctions is logically arbitrary, but pragmatically highly relevant for good readability of the resulting table: Related cases should end up close together, ideally in adjacent cells of the main grid.
3. Homogeneous regions of cases in the main grid, or "modes" of the system, are indicated by invisible cell boundaries. Missing expressions indicate unsatisfiable conditions; those are an artifact of the headers not being fully independent.
4. Case distinctions make effective use of the treatment of undefined values, discussed in the previous section, for keeping things simple. For instance, all auxiliary predicates involve equations whose parts are defined for nonempty sequences T only; hence the only row that is satisfiable for the column corresponding to T being empty is the fifth, where all predicates occur in negative form. The header expression $T \neq \_$ is redundant and included only for greater clarity. More generally, nested distinctions can always be read as conjunctions and simplified accordingly, which would be difficult in logically three-valued frameworks, because one clause may govern the definedness of another.

3.3 Table Combinators

A connection between tabular expressions and functional programming has already been noted by [13]. There a combinator approach is followed: a collection of compositional table contruction operators is given, with executable Haskell implementation for operational semantics, and proof support in the Isabelle system for denotational semantics. While both theoretically elegant and techically effective, the approach suffers from severe limitations regarding the shapes of tables that can be constructed; a drawback shared with both other practical tools such as [25] and theoretical formalizations such as [9–11].

$$
\begin{aligned}
\mathit{Content}[I,J,X,Y] &= \mathit{Map}[I,\mathit{Map}[J,\mathit{Expr}[X,Y]]] \\
\mathit{TType}[I,J,X,Y] &= \begin{pmatrix} \mathit{wellf} : \mathit{Content}[I,J,X,Y] \times X \to \mathit{bool}; \\ \mathit{eval} \ : \mathit{Content}[I,J,X,Y] \times X \nrightarrow Y \end{pmatrix} \\
\mathit{Table}[I,J,X,Y] &= \begin{pmatrix} \mathit{content} : \mathit{Content}[I,J,X,Y]; \\ \mathit{type} \quad : \mathit{TType}[I,J,X,Y] \end{pmatrix}
\end{aligned}
$$

Fig. 3. Table model as functional datatype

3.4 General Table Model

The examples have already shown a weakness, or rather looseness, of the tabular notation: Considerable amounts of semantic detail are implicit in the graphical layout or the conventions of users. This is typically adequate for a team of experts, but not for broader communication, formal verification or automated evaluation. Support for certain ad-hoc tabular formats has been built into many engineering tools. These may be pragmatically useful, but there is no theoretical boundary of what should be included; the examples already show both plain one- and two-dimensional forms, and several advanced features, such as branching headers [7, 28], and shared and empty main grid cells. Various generalizations (for instance extension to n dimensions, grids with circular or otherwise fancy topology, and specialized case distinctions such as C-style *switch*) are mathematically straightforward, but defeat the capabilities of implemented, hard-wired table models.

A unified theoretical approach [12] defines tabular expressions abstractly as a formal structure of three components:

1. The *content* of the table as an indexed set of indexed grids containing cell expressions. Both grid and cell indices are abstract; no layout geometry is implied. This is the only component particular to a concrete table.
2. A *well-formedness* predicate that decides whether the table content conforms to given shape and consistency constraints. This component is shared among tabular expressions of a common *type*.
3. One or more *evaluation* functions that interpret the cell content, conditional on its well-formedness, as a function of its argument variables. This component is also shared among tabular expressions of a common type.

3.5 Functional Table Model

Fig. 3 shows a datatype definition for table models. Type parameters are I, J for grid and cell indices, and X, Y for domain and range, respectively.

The function argument may occur in each table cell subexpression. Whereas first-order tools default to symbolic representations, in higher-order functional programming the obvious encoding technique is *lambda lifting*: every cell contains an individual function of the global arguments. The transformation is trivial

because cell contents are independent (neither individually not mutually recursive). Consequently we simply have $Expr[X, Y] = (X \nrightarrow Y)$.

The well-formedness predicate may contain both static and dynamic constraints. These could be separated by binding time analysis, in order to be checked as early as possible. For n-dimensional regular function tables, which subsume the two given examples with $n = 2$, the types parameters are chosen such that:

1. The cell indices of each header grid are sets of finite paths closed under prefixes.
2. The cell indices of the main grid are the Cartesian product of the cell indices of the header grids.

The well-formedness constraints are:

1. There are $n+1$ grids. The first n grids are headers and contain truth-valued expressions. The last grid is the main grid and contains Y-valued expressions.
2. In each header, the respective conjunctions of formulas along maximal maths partition the function domain.
3. For each main index $(j_1, \ldots, j_n)$, if the formulas indexed by j_i in the i-th header, respectively, are jointly satisfiable for all i, there is a corresponding cell; other main cells may be omitted.

We treat hierarchical header structure as a syntactic abbreviation for simplicity. The corresponding evaluation function is:

1. For each i-th header grid, find the maximal path j_i such that the conjunction of all formulas in cells along the path is satisfied.
2. Evaluate the main grid cell at $(j_1, \ldots, j_n)$.

The table type effectively defines a semantic checker and interpreter for the table content. Obviously when both type and content are provided, these algorithms can be simplified drastically by partial evaluation of the pair. Assume well-formedness is split by binding time analysis into a static and a dynamic part

$$\begin{aligned} &\mathit{swellf} : \mathit{Content}[I, J, X, Y] \to \mathit{bool}; \\ &\mathit{dwellf} : \mathit{Content}[I, J, X, Y] \times X \to \mathit{bool} \end{aligned}$$

Then we can partially evaluate the table type components applied to the concrete content, obtaining the following record of table operations, which hides the defining content completely:

$$\begin{pmatrix} \mathit{pswellf} & : \mathit{bool}; \\ \mathit{pdwellf} & : X \to \mathit{bool}; \\ \mathit{peval} & : X \nrightarrow Y \end{pmatrix}$$

Possibly more static safety is required, such as a guarantee that the "compiled" evaluation function *peval* is defined whenever the dynamic well-formedness

check *pdwellf* holds. The strong connection between total function calculi and constructive proof systems, exemplified in tools such as Coq or Agda, could be used to resolve these issues statically. No practical attempts to perform this manually or automatically have been documented so far.

4 Trace Function Method

The trace function method is a formalism for black-box description of observable system or component behavior at an interface. It is intended as a mathematically simple and direct replacement for algebraic and automata-theoretic approaches, as well as the earlier trace assertion [2, 19] and trace rewriting [33, 34] methods.

A *trace* is a sequence of relevant events at some interface, where each event is a discrete point in time at which interface variables may change their values. Input and output variables under control of the environment and the system, respectively, are unified. Valid system behavior is specified by giving, for each output variable, a trace function or relation, which maps traces to possible output values for the final event. The set of valid traces is then defined inductively:

- The empty trace is valid.
- A valid trace can be extended by a following event if and only if all output variables of that event conform to the respective trace functions.

Having the trace, that is the sequence of preceding interface events, instead of internal state as the causal determinant of future behavior makes this approach mathematically and epistemologically very abstract and elegant. However, there are a couple of logical, philosophical and technical issues:

1. Trace functions specify part of an event, namely the value of one output variable, in terms of a trace ending in that event; how can we avoid circularity?
2. The proposed way to avoid circularity is to include only the input part of the most recent event in the argument to trace functions; is that a natural solution, and if so, the only natural one?
3. Trace functions take syntactically well-formed, as opposed to valid, traces as their arguments (necessarily to avoid meta-circularity); how do counterfactual values in invalid traces affect the specification of behavior?
4. A trace function has two distinct ways of depending recursively on its own value for trace prefixes, namely by retrieval from events and by recursive self-application; are they exchangeable, and if not, which one is logically preferred?
5. Trace functions can depend on variables in past events in many ways: on the last event only, on a fixed sliding window, on the most recent event matching a certain pattern, etc.; which complexity classes are there with respect to implementation as a state system, and can efficient iterative representations be derived automatically?

All of these can be addressed by rephrasing trace functions, with their peculiar recursive structure, in a recursion scheme for total functions, namely *course-of-values* iteration, in category-theoretic presentation [32].

The formal scheme, in a nutshell, is as follows: Consider an endofunctor F whose initial algebra $(\mu F, \mathrm{in}_F : F\mu F \to \mu F)$ is a datatype of interest. The simplest total recursion scheme over F is iteration: For every F-algebra $(C, \varphi : FC \to C)$ there is a unique function $(\!|\varphi|\!) : \mu F \to C$ such that

$$(\!|\varphi|\!) = \varphi \circ F(\!|\varphi|\!) \circ \mathrm{in}_F^{-1}$$

The canonical example is the functor $N = 1 + \_$ with the initial algebra $(\mathbb{N}, [0, \mathit{succ}])$. Every N-algebra $(C, \varphi = [z, s])$ gives rise to a function $(\!|\varphi|\!) : \mathbb{N} \to C$ with $(\!|\varphi|\!)(n) = s^n(z)$, hence the name *iteration* for the scheme. Written as an explicitly self-referential definition, this gives

$$(\!|\varphi|\!)(n) = \begin{cases} z & n = 0 \\ s\big((\!|\varphi|\!)(n-1)\big) & n > 0 \end{cases}$$

That is, the function may depend recursively on its own value for the *immediate* predecessor(s) of the current argument value. The recursive tabular expression depicted in Fig. 2 is easily seen to be of this form.

The scheme of course-of-value iteration generalizes ordinary iteration to functions that depend on their own value for all *transitive* predecessors of the current argument value. Consider the composite functor $F^C = C \times F(\_)$ and a final F^C-coalgebra $(\nu F^C, \mathrm{out}_{F^C} : \nu F^C \to F^C \nu F^C)$ as a (co)datatype. Then for every map $\psi : F\nu F^C \to C$ there is a unique function $\{\!|\psi|\!\} : \mu F \to C$ whose precise definition and characterizing universal property are out of scope here.

Coming back to the previous example, we find that $\nu N^C \cong C^+ \cup C^\omega$ (the set of non-empty finite *and* infinite sequences over C) and $N\nu N^C \cong C^* \cup C^\omega$. Then we have specifically

$$\{\!|\psi|\!\}(n) = \psi\big(\langle \{\!|\psi|\!\}(n-1), \ldots \{\!|\psi|\!\}(0)\rangle\big) \tag{1}$$

The canonical example is $C = \mathbb{N}$ and

$$\psi(s) = \begin{cases} 0 & |s| = 0 \\ 1 & |s| = 1 \\ s_0 + s_1 & |s| > 1 \end{cases}$$

which generates the *Fibonacci* function $\{\!|\psi|\!\}$.

Let I, O be the record type of inputs/outputs of an interface, respectively, and write $IO = I \times O$. Defining a functor $T = N^I = I \times (1 + \_)$ gives $\mu T \cong I^+$. Consequently $\nu T^O \cong IO^+ \cup IO^\omega$ and $T\nu T^O \cong I \times (IO^+ \cup IO^\omega)$. In words, $T\nu T^O$ differs from $IO^* \cup IO^\omega$ only in the fact that the first O is missing. A map of the form $\psi : T\nu T^O \to O$ maps such a partial trace to the (missing) output, and hence defines a trace function $\{\!|\psi|\!\}$ with

$$\{\!|\psi|\!\}(\langle i_0, \ldots, i_n\rangle) = \psi\Big(i_0, \big\langle \big(i_1, \{\!|\psi|\!\}(\langle i_1, \ldots, i_n\rangle)\big), \ldots, \big(i_n, \{\!|\psi|\!\}(\langle i_n\rangle)\big)\big\rangle\Big) \tag{2}$$

$$bag(T) \equiv \begin{array}{|c|c|c|}\hline T = \_ & & 0 \\ \hline r(T).op = & \mathsf{clr} & \\ & \mathsf{cnt} & n \\ & \mathsf{inc} & min(n+1, B) \\ & \mathsf{dec} & max(n-1, 0) \\ \hline \end{array}$$

where $\quad n \equiv bag\big(p(T) \triangleright r(T).arg\big)$

$$T \triangleright x \equiv \begin{array}{|c|c|}\hline T = \_ & \_ \\ \hline r(T).op = \mathsf{clr} \vee r(T).arg = x & T \\ \hline r(T).op \neq \mathsf{clr} \wedge r(T).arg \neq x & p(T) \triangleright x \\ \hline \end{array}$$

Fig. 4. Trace function specification of multiset data structure

It can be seen, analogously to equation (1), that values for either infinite and illegal traces are irrelevant for the result.

The following simple but nontrivial example specifies the behavior of a mutable multiset component that can hold elements of some type E. Its interface supports four operations $op \in \{\mathsf{clr}, \mathsf{cnt}, \mathsf{inc}, \mathsf{dec}\}$ which completely remove a given element arg, count its multiplicity, and add or remove one instance, respectively. Each operation returns the resulting multiplicity of the given element. Real-world constraints are added with requirements for robust behavior: No element may exceed a fixed multiplicity B, and removal of non-existent elements is ignored. These simple but natural constraints break naive attempts at algebraic specification: the algebraic structure may be correct, but is too complicated and irregular to be considered adequate.

Fig. 4 depicts a trace function specification of the multiset component. It is explicitly recursive via the auxiliary term n, but not in ordinary iteration form: the recursive argument is *some* predecessor of the current one, determined dynamically by the auxiliary function $T \triangleright x$ which implements the (ordinarily iterative) search "subtrace up to the most recent event that affects the multiplicity of element x". This definition is easily transformed to a course-of-value generator map ψ, by replacing recursive calls with retrievals of recorded output values from the trace.

The functor for the recursion scheme of trace functions can be simplified from T to N, at the price of complicating the range type from O to $O^{I^+ \cup I^\omega}$, thus making trace function recursion a higher-order iteration [31]. The functor N is distinguished because for each of its course-of-value iterations, there is a whole category of simulating deterministic transition systems with more or less elaborate state space [30], which form a semantic framework for both prototypic and production-quality implementations.

The selection of a particular implementation is a trade-off between ease of derivation and efficiency of execution, and cannot be automated straightfor-

wardly. The initial implementation, which simply accumulates inputs and outputs, has unbounded space requirements, and may not be acceptable for any but the most prototypic uses. But practical hints can be gained from the analysis of access patterns to recursive predecessors: For instance, if there is a horizon such that each output depends only on the preceding k, then a ring buffer of size k is a fairly good state space for canonical implementation. In the Fibonacci example, we have $k = 2$ for the standard imperative implementation. Other, more dynamic access patterns such as in the multiset example could possibly be classified according to their complexity and associated implementation techniques.

5 Conclusion

The three levels of Parnas-style mathematical description of system behavior form a stack of methods, where each layer benefits from a (total) function viewpoint in a particular way. Predicate logic with partial value level and total truth level requires an implementation in terms of total (strongly terminating) functions because of the mutual dependencies between the levels. Tabular expression scale inhomogeneous function definitions up to very complex case distinctions. Their generic definition requires datatypes to contain explicit functions for checking and evaluation, and implicit functions for cell expressions with free variables. The generic parts can be fused with concrete contents to form specific table semantics by means of standard specialization techniques such as bindign time analysis and partial evaluation. Finally, the trace function method employs tabular expressions with a particular recursion scheme over traces represented as sequences of hypothetical past events, for the description of component behavior without explicit reference to internal state. The precise form and meaning of this recursion scheme is given by categorical course-of-value iteration, for which rough but general implementation guidelines can be given, depending on the pattern of access to the past.

References

[1] V. Balat and O. Danvy. “Strong Normalization by Type-Directed Partial Evaluation and Run-Time Code Generation”. In: *Proc. WTC ’97*. LNCS 1473. Springer, 1997, pp. 240–252.

[2] W. Bartoussek and D. L. Parnas. “Using Assertions about Traces to Write Abstract Specifications for Software Modules”. In: *Proc. Second Conf. European Cooperation in Informatics and Information Systems Methodology*. 1978, pp. 211–236.

[3] J. Desharnais, R. Khédry, and A. Mili. “Interpretation of tabular expressions using arrays of relations”. In: *Relational methods for computer science applications*. Physica Verlga Rudolf Liebing KG, 2001, pp. 3–13.

[4] X. Feng, D. L. Parnas, and T. H. Tse. “Tabular expression-based testing strategies: A comparison”. In: *Proc. MUTATION ’07*. IEEE Computer Society, 2007.

[5] X. Feng, D. L. Parnas, and T. H. Tse. "Fault Propagation in Tabular Expression-Based Specifications". In: *Proc. COMPSAC '08*. IEEE Computer Society, 2008, pp. 180–183.
[6] X. Feng et al. "A Comparison of Tabular Expression-Based Testing Strategies". In: *IEEE Trans. Software Eng.* 37.5 (2011), pp. 616–634.
[7] H. Furusawa and W. Kahl. *Table algebras: Algebraic structures for tabular notation, including nested headers*. Programming Science Technical Report. AIST, 2004.
[8] *Information Technology – Z Formal Specification Notation – Syntax, Type System and Semantics*. Standard 13568. ISO/IEC. 2002.
[9] R. Janicki. "Towards a formal semantics of Parnas tables". In: *Proc. ICSE '95*. ACM, 1995, pp. 231–240.
[10] R. Janicki and R. Khédry. "On a formal semantics of tabular expressions". In: *Sci. Comp. Progr.* 39.2-3 (2001), pp. 189–213.
[11] R. Janicki, D. L. Parnas, and J. Zucker. "Tabular representations in relational documents". In: *Relational methods in computer science*. Springer, 1997, pp. 184–196.
[12] Y. Jin and D. L. Parnas. "Defining the meaning of tabular mathematical expressions". In: *Sci. Comput. Program.* 75.11 (2010), pp. 980–1000.
[13] W. Kahl. *Compositional syntax and semantics of tables*. SQRL Report 15. McMaster University, 2003.
[14] M. Lawford, P. Froebel, and G. Moum. "Application of tabular methods to the specification and verification of a nuclear reactor shutdown system". In: *Formal Methods in System Design* (2001). Accepted for publication 2004.
[15] M. Lawford et al. "Practical Application of Functional and Relational Methods for the Specification and Verification of Safety Critical Software". In: *Proc. AMAST '00*. LNCS 1816. Springer, 2000, pp. 73–88.
[16] Z. Liu, D. L. Parnas, and B. Trancón y Widemann. "Documenting and verifying systems assembled from components". In: *Frontiers of Computer Science in China* 4.2 (2010), pp. 151–161.
[17] V. Pantelic et al. "Inspection of concurrent systems: Combining tables, theorem proving and model checking". In: *Proc. SERP '06*. 2006, pp. 629–635.
[18] V. Pantelic. "Combining tables, theorem proving and model checking". MSc Thesis. McMaster University, 2006.
[19] D. L. Parnas and Y. Wang. *The Trace Assertion Method of Module Interface Specification*. CIS Report 89-261. Queen's University, 1989.
[20] D. L. Parnas. *Tabular representation of relations*. CLR Report 260. McMaster University, 1992.
[21] D. L. Parnas. "Inspection of Safety-Critical Software Using Program-Function Tables". In: *IFIP Congress (3)*. 1994, pp. 270–277.
[22] D. L. Parnas. "Predicate Logic for Software Engineering". In: *IEEE Trans. Software Eng.* 19.9 (1993), pp. 856–862.
[23] D. L. Parnas. "The Tabular Method for Relational Documentation". In: *Electr. Notes Theor. Comput. Sci.* 44.3 (2001), pp. 1–26.
[24] D. L. Parnas, J. Madey, and M. Iglewski. "Precise Documentation of Well-Structured Programs". In: *IEEE Trans. Software Eng.* 20.12 (1994), pp. 948–976.
[25] D. L. Parnas and D. K. Peters. "An Easily Extensible Toolset for Tabular Mathematical Expressions". In: *Proc. TACAS '99*. LNCS 1579. Springer, 1999, pp. 345–359.

[26] D. Peters, M. Lawford, and B. Trancón y Widemann. “An IDE for software development using tabular expressions”. In: *Proc. CASCON ’07*. ACM, 2007, pp. 248–251.

[27] C. Quinn et al. “Specification of Software Component Requirements Using the Trace Function Method”. In: *Proc. ICSEA ’06*. IEEE Computer Society, 2006, p. 50.

[28] S. Sepehr. “Adding Nested Headers and a Proper Gtk-Based GUI to The Haskell Table Tools”. McMaster University, 2010. Open Access Dissertations and Theses: `4412`.

[29] B. Trancón y Widemann and D. L. Parnas. “Tabular expressions and total functional programming”. In: *Proc. IFL ’07, Revised selected papers*. LNCS 5083. Springer, 2008, pp. 219–236.

[30] B. Trancón y Widemann. “State-based Simulation of Linear Course-of-value Iteration”. In: *Proc. CMCS ’12*. Short contribution. Tallinn University of Technology. 2012.

[31] B. Trancón y Widemann. “The Recursion Scheme of the Trace Function Method”. In: *Proc. ENASE ’12*. 2012, pp. 146–155.

[32] T. Uustalu and V. Vene. “Primitive (Co)Recursion and Course-of-Value (Co)Iteration, Categorically”. In: *Informatica, Lith. Acad. Sci.* 10.1 (1999), pp. 5–26.

[33] Y. Wang and D. L. Parnas. “Simulating the Behaviour of Software Modules by Trace Rewriting”. In: *Proc. ICSE ’93*. IEEE Computer Society, 1993, pp. 14–23.

[34] Y. Wang and D. L. Parnas. “Trace Rewriting Systems”. In: *Proc. CTRS ’92*. LNCS 656. Springer, 1993, pp. 343–356.