On monadic parametricity of second-order functionals

Andrej Bauer[1], Martin Hofmann[2], and Aleksandr Karbyshev[3]

[1] University of Ljubljana, `andrej.bauer@andrej.com`
[2] Universität München, `hofmann@ifi.lmu.de`
[3] Technische Universität München, `aleksandr.karbyshev@in.tum.de`

Abstract. How can one rigorously specify that a given ML functional f : (`int` $\to$ `int`) $\to$ `int` is *pure*, i.e., f produces no computational effects except those produced by evaluation of its functional argument? In this paper, we introduce a semantic notion of *monadic parametricity* for second-order functionals which is a form of purity. We show that every monadically parametric f admits a question-answer strategy tree representation. We discuss possible applications of this notion, e.g., to the verification of generic fixpoint algorithms. The results are presented in two settings: a total set-theoretic setting and a partial domain-theoretic one. All proofs are formalized by means of the proof assistant Coq.

1 Introduction

The problem under consideration is: how do we rigorously specify that a given ML functional f : (`int` $\to$ `int`) $\to$ `int` is pure, i.e., f only produces computational effects (changes store, raises an exceptions, produces output, consumes input, etc.) through calls of its functional argument? Second-order functionals of this type may appear as inputs in various third-order algorithms, such as generic fixpoint solvers [4,5] and algorithms for exact integration [15,20]. The algorithms often apply a presumably pure input f to an effectful argument in order to observe the intentional behaviour of f, and to control the computation process.

In a previous paper [8] we addressed the question with regard to functionals that were polymorphic in the state monad and had the type $\forall S.(A \to \mathit{State}_S B) \to \mathit{State}_S C$. The motivation there was rigorous verification of a generic fixpoint algorithm RLD [7] that used state. As it turns out [8], we could not use the standard notion of relational parametricity [17,18] because it is too weak to exclude the snapback functional $f_{\mathrm{snap}} : \forall S.(A \to \mathit{State}_S B) \to \mathit{State}_S B$, defined by

$$f_{\mathrm{snap}}\, S\, k\, s = \mathbf{let}\ (b, \_) = k\, a_0\, s\ \mathbf{in}\ (b, s)\ \ .$$

The functional invokes its argument k to compute a result b but then discards the new state and restores the initial one instead. We can show that every functional which is pure in the sense of [8] is represented by a question-answer strategy tree that computes the result by calling its function argument and letting through any effects generated by the calls. The functional f_{snap} is not pure in that sense.

The strategy tree reflects only the "skeleton" of a computation and is not specific to the kind of effects that may be raised. Obviously, we should look for a more general representation theorem that applies to other kinds of effectful computations. Indeed, in this paper we remove the limitation of [8] to state and prove a corresponding theorem for the class of second-order functionals polymorphic with respect to monads from an arbitrary fixed collection *Monad*, i.e., those of type

$$\mathrm{Func} = \prod_{T \in Monad} (A \to TB) \to TC \ ,$$

so long as continuation monads $Cont_R$ are included, for all R. We may think of *Monad* as the class of monads present in a programming language. Every monad can be expressed in terms of continuation and state [6], but we do *not* use this fact and do not require $State \in Monad$. Thus our representation result and that of Filinski [6] are different and do not imply each other directly. An interesting corollary is that a functional F which is pure for the state monads in the sense of [8] has an equivalent implementation which makes no use of state, and is moreover polymorphic in all monads from the given collection *Monad*. Such an implementation is defined by a strategy tree for F.

One possible application of the representation result is formal verification of the above mentioned algorithms. For example, when trying to prove correctness of the local fixpoint solver RLD we assumed without loss of generality that the input constraint system is given in the form of strategy trees. That allowed us to formulate sufficient pre- and post-conditions for the algorithm and complete the proof by induction. The fundamental lemma then allows us to argue that the functional input is indeed pure if it can be defined in some restricted programming language (with recursion) which is often the case in real-life program analysis.

The outline of the paper is as follows. After a preliminary Section 2, we define purity in Section 3 as a semantical notion of monadic parametricity. We also formulate a fundamental lemma for the call-by-value lambda calculus with monadic semantics. In section 4, we define a notion of a strategy tree and show they represent pure functionals of type Func in the total setting. Section 5 provides a similar result in the partial setting. In section 6, we discuss generalizations of purity to other types. In section 7, we discuss some application of purity.

All the proofs have been formalized by means of Coq theorem prover [21] and are available for download at [11]. We used the development of constructive ω-cpos and inverse-limit construction for solution of recursive domain equations by Benton et al. [3]. Our contribution takes around 1500 lines of Coq code.

Acknowledgements. We thank Alex Simpson, University of Edinburgh, for raising an interesting question and fruitful discussions on the topic. We thank Helmut Seidl and anonymous reviewers for valuable comments on the paper. The third author was supported by GRK 1480.

2 Preliminaries

We study purity in both the total and the partial setting. For the former we interprets types as sets and the latter as continuous posets (cpos), and thus use the notations $a : X$ and $a \in X$ interchangeably. We write $X \times Y$ and $X \to Y$ for Cartesian product and exponential, respectively. We denote pairs by (x, y), and projections by *fst* and *snd*. We use λ, $\circ$ and juxtaposition for function abstraction, composition and applications, correspondingly. For a family of sets or cpos $(X_i)_{i\in I}$ we write $\prod_{i\in I} X_i$ for its Cartesian product.

Definition 1. A *monad* is a triple $(T, \mathsf{val}_T, \mathsf{bind}_T)$ where T is the *monad constructor* assigning to a type X the type TX of computations over X, and

$$\begin{aligned} &\mathsf{val}_T^A &&: A \to TA \\ &\mathsf{bind}_T^{A,B} &&: TA \to (A \to TB) \to TB \end{aligned}$$

are the *monadic operators*, satisfying for all a, f, g and t of suitable types

$$\begin{aligned} \mathsf{bind}_T^{A,B}(\mathsf{val}_T^A\, a)(f) &= f\, a \\ \mathsf{bind}_T^{A,A}(t)(\mathsf{val}_T^A) &= t \\ \mathsf{bind}_T^{B,C}(\mathsf{bind}_T^{A,B}(t)(f))(g) &= \mathsf{bind}_T^{A,C}(t)(\lambda x.\, \mathsf{bind}_T^{B,C}(f\, x)(g)) \ . \end{aligned}$$

For the partial case, we require that TX is a pointed cpo (cppo) and that T is *strict*,

$$\mathsf{bind}_T^{A,B}\ \bot_{TA}\, f = \bot_{TB} \ .$$

We omit the indices A, B, C when they can be deduced from the context.

The *continuation monad* Cont_R with result type R is defined by $\mathit{Cont}_R X = (X \to R) \to R$ and

$$\begin{aligned} &\mathsf{val}_{\mathit{Cont}_R}\ x &&= \lambda c.c\, x \\ &\mathsf{bind}_{\mathit{Cont}_R}\ t\, f &&= \lambda c.t(\lambda x.f\, x\, c) \ . \end{aligned}$$

The *state monad* State_S with the type of states S is defined by $\mathit{State}_S X = S \to X \times S$ and

$$\begin{aligned} &\mathsf{val}_{\mathit{State}_S}\ x &&= \lambda s.(x, s) \\ &\mathsf{bind}_{\mathit{State}_S}\ t\, f &&= \lambda s.\mathbf{let}\ (x_1, s_1) \leftarrow t\, s\ \mathbf{in}\ f x_1 s_1 \ . \end{aligned}$$

In the following, we assume that A, B, C, A_i, B_i are sets or cpos, as appropriate. Let *Monad* be a fixed collection of monads such that $\mathit{Cont}_R \in \mathit{Monad}$, for all R, and denote

$$\mathrm{Func} = \prod_{T \in \mathit{Monad}} (A \to TB) \to TC \ .$$

3 Purity

To define purity in our sense we first introduce several notions and notations. We then provide a relational interpretation of types and terms of call-by-value λ-calculus with monadic semantics, and establish a fundamental lemma of logical relations stating that every well-typed program respects any monadic relation, similar to [8].

Definition 2. If X, X' are types then $\mathrm{Rel}(X, X')$ denotes the type of binary relations between X and X'. Furthermore:

– if X is a type then $\Delta_X \in \mathrm{Rel}(X, X)$ denotes the equality on X;
– if $R \in \mathrm{Rel}(X, X')$ and $S \in \mathrm{Rel}(Y, Y')$ then $R \to S \in \mathrm{Rel}(X \to Y, X' \to Y')$ is given by

$$f\,(R \to S)\,f' \quad \text{iff} \quad \forall x\,x'.\, xRx' \implies (f\,x)\,S\,(f'\,x') \ ;$$

– if $R \in \mathrm{Rel}(X, X')$ and $S \in \mathrm{Rel}(Y, Y')$ then $R \times S \in \mathrm{Rel}(X \times Y, X' \times Y')$ is given by

$$p\,(R \times S)\,p' \quad \text{iff} \quad \mathit{fst}(p)\,R\,\mathit{fst}(p') \wedge \mathit{snd}(p)\,S\,\mathit{snd}(p') \ .$$

Definition 3. For cpos X, X' and $R \in \mathrm{Rel}(X, X')$, R is *admissible* if for any chains $\{c_i\}_{i \in \mathbb{N}}$, $\{c'_i\}_{i \in \mathbb{N}}$ such that $c_i\,R\,c'_i$, for all i, $(\bigsqcup c_i)\,R\,(\bigsqcup c'_i)$ holds.

Definition 4. Fix $T, T' \in \mathit{Monad}$. For every X, X' and $Q \in \mathrm{Rel}(X, X')$ fix a relation $T^{\mathsf{rel}}(Q) \in \mathrm{Rel}(TX, T'X')$. We say that the mapping $(X, X', Q) \mapsto T^{\mathsf{rel}}(Q)$ is an *acceptable monadic relation* if

– for all $X, X', Q \in \mathrm{Rel}(X, X')$, $x \in X$, $x' \in X'$,

$$xQx' \implies (\mathsf{val}_T\ x)\,T^{\mathsf{rel}}(Q)\,(\mathsf{val}_{T'}\ x') \ ;$$

– for all $X, X', Q \in \mathrm{Rel}(X, X')$, $Y, Y', R \in \mathrm{Rel}(Y, Y')$, $t \in TX$, $t' \in T'X'$, $f : X \to TY$, $f' : X' \to T'Y'$,

$$t\,T^{\mathsf{rel}}(Q)\,t' \wedge f(Q \to T^{\mathsf{rel}}(R))f' \implies (\mathsf{bind}_T\ t\ f)\,T^{\mathsf{rel}}(R)\,(\mathsf{bind}_{T'}\ t'\ f') \ .$$

In the domain-theoretic setting, we additionally assume that the monadic relation T^{rel} is

– *admissible*, i.e., $T^{\mathsf{rel}}(Q)$ is admissible for every admissible $Q \in \mathrm{Rel}(X, X')$,
– *strict*, i.e., $(\bot_{TX}, \bot_{T'X'}) \in T^{\mathsf{rel}}(Q)$.

Definition 5. A functional $F \in \mathrm{Func}$ is *pure (monadically parametric)* for the collection *Monad* of monads iff

$$(F_T, F_{T'}) \in (\Delta_A \to T^{\mathsf{rel}}(\Delta_B)) \to T^{\mathsf{rel}}(\Delta_C)$$

holds for all $T, T' \in \mathit{Monad}$ and acceptable monadic relations T^{rel} for T, T'.

Define simple types over a set of base types, ranged over by o, by the grammar

$$\tau ::= o \mid \tau_1 \times \tau_2 \mid \tau_1 \to \tau_2 \, .$$

Fix an assignment of a set or a cpo, as the case may be, $[\![o]\!]_T$ for each base type o and monad $T \in \mathit{Monad}$. We extend $[\![-]\!]_T$ to all types by putting

$$[\![\tau_1 \times \tau_2]\!]_T = [\![\tau_1]\!]_T \times [\![\tau_2]\!]_T \, , \quad [\![\tau_1 \to \tau_2]\!]_T = [\![\tau_1]\!]_T \to T[\![\tau_2]\!]_T \, .$$

Given a set of constants ranged over by c, with corresponding types τ^c, and variables ranged over by x, we define the λ-terms by

$$\begin{aligned} e ::= {} & x \mid c \mid \lambda x.e \mid e_1\, e_2 \mid e.1 \mid e.2 \mid \langle e_1, e_2\rangle \mid \\ & \text{let } x \leftarrow e_1 \text{ in } e_2 \mid \text{let rec } f(x) = e \end{aligned}$$

with the last rule for recursive definitions in the partial case only. A typing context Γ is a finite map from variables to types. The typing judgement $\Gamma \vdash e : \tau$ is defined by the usual rules:

$$\frac{x \in \mathrm{dom}(\Gamma)}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{}{\Gamma \vdash c : \tau^c}$$

$$\frac{\Gamma, x:\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\, e_2 : \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2\rangle : \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash e.1 : \tau_1} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash e.2 : \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x:\tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \text{let } x \leftarrow e_1 \text{ in } e_2 : \tau_2} \qquad \frac{\Gamma, f:\tau_1 \to \tau_2, x:\tau_1 \vdash e : \tau_2}{\Gamma \vdash \text{let rec } f(x) = e : \tau_1 \to \tau_2}$$

The term $e : \tau$ is *closed* if $\emptyset \vdash e : \tau$.

For each $T \in \mathit{Monad}$ and constant c fix an interpretation $[\![c]\!]_T \in [\![\tau^c]\!]_T$. An *environment* for a context Γ and $T \in \mathit{Monad}$ is a mapping η such that $x \in \mathrm{dom}(\Gamma)$ implies $\eta(x) \in [\![\Gamma(x)]\!]_T$. If $\Gamma \vdash e : \tau$ and η is such an environment then we define $[\![e]\!]_T(\eta) \in T[\![\tau]\!]_T$ by the following clauses:

$$\begin{aligned}
[\![x]\!]_T(\eta) &= \mathsf{val}_T(\eta(x)) \\
[\![c]\!]_T(\eta) &= \mathsf{val}_T([\![c]\!]_T) \\
[\![\lambda x.e]\!]_T(\eta) &= \mathsf{val}_T(\lambda v.[\![e]\!]_T(\eta[x \mapsto v])) \\
[\![e_1\, e_2]\!]_T(\eta) &= \mathsf{bind}_T([\![e_1]\!]_T(\eta))\,(\mathsf{bind}_T([\![e_2]\!]_T(\eta))) \\
[\![e.i]\!]_T(\eta) &= \mathsf{bind}_T([\![e]\!]_T(\eta))\,(\mathsf{val}_T \circ \pi_i),\ i = 1, 2 \\
[\![\langle e_1, e_2\rangle]\!]_T(\eta) &= \mathsf{bind}_T([\![e_1]\!]_T(\eta))(\mathsf{bind}_T([\![e_2]\!]_T(\eta)) \circ \mathit{curry}(\mathsf{val}_T)) \\
[\![\text{let } x \leftarrow e_1 \text{ in } e_2]\!]_T(\eta) &= \mathsf{bind}_T([\![e_1]\!]_T(\eta))(\lambda v.[\![e_2]\!]_T(\eta[x \mapsto v])) \\
[\![\text{let rec } f(x) = e]\!]_T(\eta) &= \mathsf{val}_T(\mathit{fixp}(\lambda h.\lambda v.[\![e]\!]_T(\eta[f \mapsto h][x \mapsto v])))
\end{aligned}$$

where $\mathit{fixp} : \forall D.(D \to D) \to D$ is the least fixpoint operator for cppos, and *curry* is the currying function.

Definition 6. Fix monads $T, T' \in \mathit{Monad}$ and an acceptable monadic relation T^{rel} for T, T'. Given a binary relation $[\![o]\!]^{\mathsf{rel}} \in \mathrm{Rel}([\![o]\!]_T, [\![o]\!]_{T'})$ for each base type o, we can associate a relation $[\![\tau]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}} \in \mathrm{Rel}([\![\tau]\!]_T, [\![\tau]\!]_{T'})$ with each type τ by the following clauses:

$$\begin{aligned} & [\![o]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}} = [\![o]\!]^{\mathsf{rel}}, \quad [\![\tau_1 \times \tau_2]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}} = [\![\tau_1]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}} \times [\![\tau_2]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}}, \\ & [\![\tau_1 \to \tau_2]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}} = [\![\tau_1]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}} \to T^{\mathsf{rel}}([\![\tau_2]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}}). \end{aligned}$$

The following *parametricity theorem* is immediate from the definition of acceptable monadic relation and the previous one.

Theorem 7. *Fix $T, T' \in Monad$, and an acceptable monadic relation T^{rel} for T, T'. Suppose that $[\![c]\!]_T \; [\![\tau^c]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}} \; [\![c]\!]_{T'}$ holds for all constants c. If $\emptyset \vdash e : \tau$ then*

$$[\![e]\!]_T \; T^{\mathsf{rel}}([\![\tau]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}}) \; [\![e]\!]_{T'} \enspace .$$

Proof. One proves the following stronger statement by induction on typing derivations. Given $\Gamma \vdash e : \tau$ and environments η for Γ and T and η' for Γ and T' then

$$\forall x.\, \eta(x)\, [\![\Gamma(x)]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}}\, \eta'(x) \quad \text{implies} \quad [\![e]\!]_T(\eta)\, T^{\mathsf{rel}}([\![\tau]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}})\, [\![e]\!]_{T'}(\eta') \enspace .$$

The assertion of the theorem follows. □

Every well-typed program $\emptyset \vdash e : \tau$ defines a truly polymorphic function of type $\forall T. [\![\tau]\!]_T$ by taking a product over *Monad*. From theorem 7, we obtain

Corollary 8. *Every truly polymorphic $F \in$ Func implemented in the calculus is monadically parametric.* □

We remark that we could incorporate Theorem 7 into the definition of Func after the definition 1, which would provide "higher-kinded type polymorphism" at that level. The theorem would then turn into a well-definedness assertions to go with the interpretation of type formers. We find the chosen presentation more convenient because it allows for a priori impure functionals whose purity can then be established a posteriori.

4 The total case

We first consider the set-theoretic semantics in which all functions are total and there is no general recursion, but we can use structural recursion on inductively defined sets.

Let the set of *strategy trees Tree* be inductively generated by the constructors $\mathsf{Ans} : C \to \mathit{Tree}$ and $\mathsf{Que} : A \to (B \to \mathit{Tree}) \to \mathit{Tree}$. Thus, a strategy tree is either an *answer* leaf $\mathsf{Ans}\; c$ with an answer value $c : C$, or a *question* node $\mathsf{Que}\; a\; f$ with a query $a : A$ and a branching (continuation) function $f : B \to \mathit{Tree}$ that returns a tree for every possible answer of type B.

For a given monad $T \in Monad$, every strategy tree defines a functional. The conversion from trees to functionals is performed by the function $\mathit{tree2fun}_T : \mathit{Tree} \to (A \to TB) \to TC$ defined by structural recursion as

$$\begin{aligned} \mathit{tree2fun}_T(\mathsf{Ans}\, c) &= \lambda k.\, \mathsf{val}_T\; c \\ \mathit{tree2fun}_T(\mathsf{Que}\, a\, f) &= \lambda k.\, \mathsf{bind}_T(k\, a)(\lambda b.\, \mathit{tree2fun}_T\, (f\, b)\, k). \end{aligned}$$

The functional queries and answers its argument k according to the strategy tree, and passes through any effects produces by k. The definition is parametric in the monad T, so we can define the polymorphic version $\mathit{tree2fun}\, t = \Lambda T : Monad.\, \mathit{tree2fun}_T\, t$ whose type is $\mathit{Tree} \to$ Func.

Example 9. For $A = B = C = \mathbb{N}$ and the tree $t = \mathsf{Que}\,0\,(\lambda x.\mathsf{Ans}\,42)$ we have

$$\begin{aligned}\mathit{tree2fun}_T\ t &= \mathit{tree2fun}_T\,(\mathsf{Que}\,0\,(\lambda x.\mathsf{Ans}\,42)) = \\ &= \lambda k.\,\mathsf{bind}_T\,(k\,0)\,(\lambda b.\,\mathit{tree2fun}_T\,(\mathsf{Ans}\,42)\,k) \\ &= \lambda k.\,\mathsf{bind}_T\,(k\,0)\,(\lambda b.\,\mathsf{val}_T\ 42).\end{aligned}$$

Thus, the tree $t = \mathsf{Que}\,0\,(\lambda x.\mathsf{Ans}\,42)$ corresponds to a second-order function that queries its argument k at 0 and returns 42. Any effect produced by k is propagated, and no other effects are produced.

The following lemma states that every $t \in \mathit{Tree}$ defines a monadically parametric computation.

Lemma 10. *For any $t \in \mathit{Tree}$, $\mathit{tree2fun}\,t$ is pure.*

Proof. By induction on t, see Appendix. □

It may be a bit surprising that *tree2fun* has an inverse *fun2tree* which is defined with the help of the continuation monad simply as

$$\mathit{fun2tree}\,F = F_{\mathit{Cont}_{\mathit{Tree}}}\ \mathsf{Que}\ \mathsf{Ans}.$$

Let us show that *fun2tree* and *tree2fun* are inverses of each other. As is to be expected, one direction is easier than the other, so we first dispose of the easy one:

Lemma 11. *For any $t \in \mathit{Tree}$, $\mathit{fun2tree}(\mathit{tree2fun}\,t) = t$.*

Proof. We proceed by structural induction on t. The case $t = \mathsf{Ans}\,c$ is easy:

$$\begin{aligned}&\mathit{fun2tree}(\mathit{tree2fun}(\mathsf{Ans}\,c)) = \\ &\qquad \mathit{fun2tree}(\Lambda T.\lambda k.\,\mathsf{val}_T\ c) = (\lambda k.\,\mathsf{val}_{\mathit{Cont}_{\mathit{Tree}}}\ c)\,\mathsf{Que}\,\mathsf{Ans} = \mathsf{Ans}\,c.\end{aligned}$$

To check the case $t = \mathsf{Que}\,a\,f$, assume the induction hypothesis, for all $b \in B$ $\mathit{fun2tree}(\mathit{tree2fun}(f\,b)) = f\,b$, and compute:

$$\begin{aligned}&\mathit{fun2tree}(\mathit{tree2fun}(\mathsf{Que}\,a\,f)) = \\ &\quad = \mathit{fun2tree}(\Lambda T.\lambda k.\,\mathsf{bind}_T(k\,a)(\lambda b.\,\mathit{tree2fun}_T(f\,b)\,k)) \\ &\quad = (\lambda k.\,\mathsf{bind}_{\mathit{Cont}_{\mathit{Tree}}}(k\,a)(\lambda b.\,\mathit{tree2fun}_{\mathit{Cont}_{\mathit{Tree}}}(f\,b)\,k))\,\mathsf{Que}\,\mathsf{Ans} \\ &\quad = (\mathsf{bind}_{\mathit{Cont}_{\mathit{Tree}}}(\mathsf{Que}\,a)(\lambda b.\,\mathit{tree2fun}_{\mathit{Cont}_{\mathit{Tree}}}(f\,b)\,\mathsf{Que}))\,\mathsf{Ans} \\ &\quad = (\mathsf{Que}\,a)(\lambda b.\,\mathit{tree2fun}_{\mathit{Cont}_{\mathit{Tree}}}(f\,b)\,\mathsf{Que}\,\mathsf{Ans}) \\ &\quad = (\mathsf{Que}\,a)(\lambda b.\,\mathit{fun2tree}(\mathit{tree2fun}(f\,b))) \\ &\quad = \mathsf{Que}\,a\,f.\end{aligned}$$

We used the induction hypothesis in the last step. □

Of course, for the other inverse we have to use purity of functionals:

Theorem 12. *For a* pure $F \in \mathrm{Func}$ *and* $T \in \mathit{Monad}$,

$$\mathit{tree2fun}_T(\mathit{fun2tree}\, F) = F_T.$$

We first verify the theorem for the continuation monad.

Lemma 13. *Given a pure* $F \in \mathrm{Func}$, $\mathit{tree2fun}_{\mathit{Cont}_S}(\mathit{fun2tree}\, F) = F_{\mathit{Cont}_S}$ *holds for all* S.

Proof. Given S and functions $q : A \to (B \to S) \to S$ and $a : C \to S$, we define the *conversion* function $\mathit{conv}_{q,a} : \mathit{Tree} \to S$ by $\mathit{conv}_{q,a} = \lambda t.\, \mathit{tree2fun}_{\mathit{Cont}_S}\, t\, q\, a$. We have:

$$\begin{aligned}&\mathit{tree2fun}_{\mathit{Cont}_S}(\mathit{fun2tree}\, F) = F_{\mathit{Cont}_S}\\ &\qquad\qquad \iff \forall q, a.(F_{\mathit{Cont}_{\mathit{Tree}}}(\mathsf{Que})(\mathsf{Ans}), F_{\mathit{Cont}_S}\, q\, a) \in \mathcal{G}_{\mathit{conv}_{q,a}}\end{aligned}$$

where $\mathcal{G}_f$ is a *graph* of f, i.e., $(x, y) \in \mathcal{G}_f$ iff $y = f\, x$. We prove the last proposition by constructing an appropriate monadic relation for $\mathit{Cont}_{\mathit{Tree}}$ and Cont_S and utilizing purity of F. Fix some q and a. For X, X' and $R \in \mathrm{Rel}(X, X')$, we define $T_1^{\mathsf{rel}}(R) \in \mathrm{Rel}(\mathit{Cont}_{\mathit{Tree}}X, \mathit{Cont}_S X')$ by

$$(H, H') \in T_1^{\mathsf{rel}}(R) \quad \text{iff} \quad \forall h, h'.(h, h') \in R \to \mathcal{G}_{\mathit{conv}_{q,a}} \implies (Hh, H'h') \in \mathcal{G}_{\mathit{conv}_{q,a}}$$

It is straightforward to show T_1^{rel} is an acceptable monadic relation. Since F is pure, $(F_{\mathit{Cont}_{\mathit{Tree}}}, F_{\mathit{Cont}_S}) \in (\Delta_A \to T_1^{\mathsf{rel}}(\Delta_B)) \to T_1^{\mathsf{rel}}(\Delta_C)$. Thus, it suffices to check that $(\mathsf{Que}, q) \in \Delta_A \to T_1^{\mathsf{rel}}(\Delta_B)$ and $(\mathsf{Ans}, a) \in \Delta_C \to \mathcal{G}_{\mathit{conv}_{q,a}}$. The latter is obvious. For the former, take $a_1 \in A$ and $f : B \to \mathit{Tree}$, $f' : B \to S$ such that $(f, f') \in \Delta_B \to \mathcal{G}_{\mathit{conv}_{q,a}}$. Then

$$\begin{aligned}\mathit{conv}_{q,a}(\mathsf{Que}\, a_1\, f) &= \mathit{tree2fun}_{\mathit{Cont}_S}(\mathsf{Que}\, a_1\, f)\, q\, a\\ &= \mathsf{bind}_{\mathit{Cont}_S}(q\, a_1)(\lambda b.\, \mathit{tree2fun}_{\mathit{Cont}_S}(f\, b)\, q)\, a\\ &= (q\, a_1)(\lambda b.\, \mathit{tree2fun}_{\mathit{Cont}_S}(f\, b)\, q\, a)\\ &= (q\, a_1)(\lambda b. \mathit{conv}_{q,a}(f\, b))\\ &= q\, a_1\, f'\end{aligned}$$

and the former holds. ☐

Now by the lemma $\mathit{tree2fun}_{\mathit{Cont}_{TC}}(F_{\mathit{Cont}_{\mathit{Tree}}}\, \mathsf{Que}\, \mathsf{Ans}) = F_{\mathit{Cont}_{TC}}$. Let

$$\begin{aligned}\varphi_1 &= \mathsf{bind}_T^{B,C} : TB \to \mathit{Cont}_{TC}B,\\ \varphi_2 &= \lambda g. g\, (\mathsf{val}_T^C) \colon \mathit{Cont}_{TC}C \to TC\end{aligned}$$

and define $\Phi_T : ((A \to \mathit{Cont}_{TC}B) \to \mathit{Cont}_{TC}C) \to (A \to TB) \to TC$ as

$$\Phi_T F = \lambda h. \varphi_2(F(\varphi_1 \circ h)) = \lambda h. F(\mathsf{bind}_T^{B,C} \circ h)(\mathsf{val}_T^C) \ .$$

Lemma 14. *For any pure* $F \in \mathrm{Func}$ *and with* Φ_T *as above,* $\Phi_T(F_{\mathit{Cont}_{TC}}) = F_T$.

Proof. The idea is to construct a suitable acceptable monadic relation and exploit the purity of F. For X, X', $R \in \mathrm{Rel}(X, X')$, we define $T_2^{\mathsf{rel}}(R)$ as an element of $\mathrm{Rel}(\mathit{Cont}_{TC} X, TX')$ by letting $(H, H') \in T_2^{\mathsf{rel}}(R)$ iff

$$\forall h, h'.(h, h') \in R \to \Delta_{TC} \implies (Hh)\, \Delta_{TC}\, (\mathsf{bind}_T\, H'\, h') \enspace .$$

It is straightforward to show that T_2^{rel} is an acceptable monadic relation, so we omit the proof. Since F is pure, we have $(F_{\mathit{Cont}_{TC}}, F_T) \in (\Delta_A \to T_2^{\mathsf{rel}}(\Delta_B)) \to T_2^{\mathsf{rel}}(\Delta_C)$. Note that for any $g\colon A \to TB$,

$$\begin{aligned} \Phi_T(F_{\mathit{Cont}_{TC}})\, g &= F_{\mathit{Cont}_{TC}}(\mathsf{bind}_T^{B,C} \circ g)(\mathsf{val}_T^C) \qquad \text{and} \\ F_T\, g &= \mathsf{bind}_T^{C,C}(F_T\, g)(\mathsf{val}_T^C) \enspace . \end{aligned}$$

First, we show that $(\mathsf{bind}_T^{B,C} \circ g, g) \in \Delta_A \to T_2^{\mathsf{rel}}(\Delta_B)$. Indeed, for any $a \in A$ and h, h' such that $(h, h') \in \Delta_B \to \Delta_{TC}$ (and thus, $h = h'$) we have $(\mathsf{bind}_T^{B,C} \circ g)\, a\, h = \mathsf{bind}_T^{B,C}(g\, a)\, h'$. Therefore, we conclude

$$(F_{\mathit{Cont}_{TC}}(\mathsf{bind}_T^{B,C} \circ g), \mathsf{bind}_T^{C,C}(F_T\, g)) \in T_2^{\mathsf{rel}}(\Delta_C) \enspace .$$

Since $(\mathsf{val}_T^C, \mathsf{val}_T^C) \in \Delta_C \to \Delta_{TC}$, the lemma is proved. □

Proof (of Theorem 12). With the help of lemmas we see that

$$\begin{aligned} F_T &= \Phi_T(F_{\mathit{Cont}_{TC}}) && \text{(by lemma 14)} \\ &= \Phi_T(\mathit{tree2fun}_{\mathit{Cont}_{TC}}(\mathit{fun2tree}\, F)) && \text{(by lemma 13)} \\ &= \mathit{tree2fun}_T(\mathit{fun2tree}\, F) && \text{(by lemmas 10, 14)} \end{aligned}$$

and the other inverse is established. □

We link the present result with that of [8]:

Corollary 15. *Any functional*

$$F : \forall S.(A \to \mathit{State}_S B) \to \mathit{State}_S C$$

which is pure in the sense of [8] may be implemented generically without using state, i.e., there exists a monadically parametric functional $G \in \mathrm{Func}$ *such that* $F_S = G_{\mathit{State}_S}$ *for all* S.

Proof. Take $G = \mathit{tree2fun}\, t_F$ where t_F is the tree representation of F.

5 The partial case

In this section, we generalize the characterisation of monadically parametric second-order functionals for the partial case in the domain-theoretic setting. In what follows, we will use the term *acceptable monadic relation* to refer to acceptable monadic relations which are strict and admissible as formulated in Definition 4.

5.1 Domain of strategy trees

We construct a cppo of "strategy trees" as a solution of a recursive domain equation $X \simeq \mathcal{F}(X)$ with a locally continuous functor $\mathcal{F} : \mathcal{C} \to \mathcal{C}$ for a suitable category $\mathcal{C}$ of domains.

Let $\eta_X : X \to X_\perp$ and $\mathit{kleisli}_X : (X \to X_\perp) \to (X_\perp \to X_\perp)$ be defined by

$$\eta_X\, x = x \qquad \mathit{kleisli}_X\, f\, x = \begin{cases} \perp & \text{if } x = \perp \\ f\, x & \text{otherwise} \end{cases} .$$

Define the lift monad $T_\perp$ over **Cpo** (category of cpos with continuous functions) by

$$T_\perp X = X_\perp, \quad \mathsf{val}_{T_\perp}^{X} = \eta_X, \quad \mathsf{bind}_{T_\perp}^{X,Y}\, t\, f = \mathit{kleisli}_X\, f\, t \ .$$

Let $\mathcal{F}(X) = C + B \times (A \to X_\perp)$ be such a functor for the Kleisli category for $T_\perp$ over **Cpo**. Let *Tree* be a cpo such that $\mathit{Tree} \simeq \mathcal{F}(\mathit{Tree})$, together with two (continuous) isomorphism functions

$$\begin{aligned} \mathsf{fold} &: C + B \times (A \to \mathit{Tree}_\perp) \to \mathit{Tree}_\perp \quad \text{and} \\ \mathsf{unfold} &: \mathit{Tree} \to (C + B \times (A \to \mathit{Tree}_\perp))_\perp \ , \end{aligned}$$

i.e., $\mathit{kleisli}(\mathsf{fold}) \circ \mathsf{unfold} = \eta_{\mathit{Tree}}$ and $\mathit{kleisli}(\mathsf{unfold}) \circ \mathsf{fold} = \eta_{\mathcal{F}(\mathit{Tree})}$ hold. For all isomorphisms in the Kleisli category for $T_\perp$, say, $f : X \to Y_\perp$ and $g : Y \to X_\perp$ that $\mathit{kleisli}(f) \circ g = \eta$ and $\mathit{kleisli}(g) \circ f = \eta$, f and g are total functions. Therefore, we can define total

$$\begin{aligned} \mathsf{roll} &: C + B \times (A \to \mathit{Tree}_\perp) \to \mathit{Tree} \quad \text{and} \\ \mathsf{unroll} &: \mathit{Tree} \to C + B \times (A \to \mathit{Tree}_\perp) \end{aligned}$$

using their "partial" counterparts fold and unfold. Moreover, the *minimal invariance* property takes place

$$\mathit{fixp}\, \delta = \eta$$

for $\delta : (\mathit{Tree} \to \mathit{Tree}_\perp) \to (\mathit{Tree} \to \mathit{Tree}_\perp)$ defined by $\delta\, e = \mathsf{fold} \circ F(e) \circ \mathsf{unfold}$. For details on a Coq development of the reverse-limit construction and a formal proof of the minimal invariance, refer to [3].

It is well known that the morphism fold forms an initial F-algebra in the Kleisli category, i.e., for any other F-algebra $\varphi : F(D) \to D$ there exists the *unique* homomorphism $h : \mathit{Tree} \to D_\perp$ such that the $\varphi \circ F(h) = h \circ \mathsf{fold}$.

Definition 16. We call elements of $\mathit{Tree}_\perp$ *strategy trees.* Define continuous "constructor" functions $\mathsf{Ans} : C \to \mathit{Tree}_\perp$ and $\mathsf{Que} : A \to (B \to \mathit{Tree}_\perp) \to \mathit{Tree}_\perp$ by

$$\mathsf{Ans} = \mathsf{fold} \circ \mathit{inl} \quad \text{and} \quad \mathsf{Que} = \mathsf{fold} \circ \mathit{inr} \ .$$

As in the total case, a strategy tree can be extracted by means of the continuation monad $\mathit{Cont}_{\mathit{Tree}_\perp}$. We define the extracting function $\mathit{fun2tree} : \mathrm{Func} \to \mathit{Tree}_\perp$ by

$$\mathit{fun2tree}\, F = F_{\mathit{Cont}_{\mathit{Tree}_\perp}}\, \mathsf{Que}\, \mathsf{Ans} \ .$$

The definition is correct since $\mathit{Cont}_{\mathit{Tree}_\bot}$ is a strict monad. The function *fun2tree* is strict and continuous.

The reverse translation mapping $\mathit{Tree}_\bot$ into Func_T is defined by means of the fixpoint operator $\mathit{fixp} : \forall D.(D \to D) \to D$ for cppos as follows. Given $T \in \mathit{Monad}$, we construct

$$\mathit{tree2fun}_T : \mathit{Tree}_\bot \to \mathrm{Func}_T = \mathit{fixp}\, G_T$$

where

$$\begin{aligned}
G_T &: (\mathit{Tree}_\bot \to \mathrm{Func}_T) \to \mathit{Tree}_\bot \to \mathrm{Func}_T = \lambda f.\, \mathit{kleisli}([\phi_T, \psi_T^f] \circ \mathsf{unroll})\\
\phi_T &: C \to \mathrm{Func}_T = \lambda c.\lambda h.\, \mathsf{val}_T\, c\\
\psi_T^f &: A \times (B \to \mathit{Tree}_\bot) \to \mathrm{Func}_T = \lambda p.\lambda h.\, \mathsf{bind}_T(h(\pi_1\, p))(\lambda b.(f \circ \pi_2\, p)\, b\, h)\ .
\end{aligned}$$

For every pointed $T \in \mathit{Monad}$, $\mathit{tree2fun}_T$ is correctly defined (since Func_T is pointed) and is continuous and strict. The parametric version is defined by $\mathit{tree2fun}\, t = \Lambda T.\, \mathit{tree2fun}_T\, t$. The following result is proved in Appendix.

Lemma 17. *For any $t \in \mathit{Tree}_\bot$, $\mathit{tree2fun}\, t$ is pure.* □

5.2 Representation theorem

Lemma 18. *For any $t \in \mathit{Tree}_\bot$, $\mathit{fun2tree}(\mathit{tree2fun}\, t) = t$.*

Proof. We note that $\mathit{fun2tree} \circ \mathit{tree2fun}$ is a homomorphism for *Tree*. Thus, the statement follows from initiality of fold. We give a direct formal proof using the minimal invariance property. □

Proofs of the following results are similar to the proofs in the total case.

Theorem 19. *For a* pure $F \in \mathrm{Func}$,

$$\mathit{tree2fun}_T(\mathit{fun2tree}\, F) = F_T$$

holds (extensionally) for any $T \in \mathit{Monad}$.

We first prove that the statement holds for an arbitrary continuation monad with a pointed result domain. See Appendix for the proof.

Lemma 20. *Given pure F, $\mathit{tree2fun}_{\mathit{Cont}_S}(\mathit{fun2tree}\, F) = F_{\mathit{Cont}_S}$ holds for any cppo S.* □

As in the total case, for $T \in \mathit{Monad}$ we define

$$\Phi_T : ((A \to \mathit{Cont}_{TC} B) \to \mathit{Cont}_{TC} C) \to (A \to TB) \to TC$$

and prove

Lemma 21. *For a pure $F \in \mathrm{Func}$ and $T \in \mathit{Monad}$, $\Phi_T(F_{\mathit{Cont}_{TC}}) = F_T$.*

Proof. The proof repeats the one of lemma 14. We only have to check that T_2^{rel} defined as in lemma 14 is a strict, admissible and acceptable monadic relation, which does hold. □

6 Generalizations

In this section, we argue that it is possible to extend the notion of purity to an arbitrary second-order type. Consider a general type n-Func of second-order functionals with n functional arguments

$$n\text{-Func} = \forall T.(A_1 \to TB_1) \to \cdots \to (A_n \to TB_n) \to TC \enspace .$$

Definition 22. A functional $F \in n$-Func is *pure (monadically parametric)* iff

$$(F_T, F_{T'}) \in (\varDelta_{A_1} \to T^{\mathsf{rel}}(\varDelta_{B_1})) \to \cdots \to (\varDelta_{A_n} \to T^{\mathsf{rel}}(\varDelta_{B_n})) \to T^{\mathsf{rel}}(\varDelta_C)$$

holds for all $T, T' \in \mathit{Monad}$ and acceptable monadic relations T^{rel} for T, T'.

By theorem 7, any well-typed program of type n-Func is pure in this sense.

Definition 23. The set of *strategy trees* n*-Tree* is a minimal set generated by constructors

- $\mathsf{Ans} : C \to n\text{-}\mathit{Tree}$
- $\mathsf{Que}_i : A_i \to (B_i \to n\text{-}\mathit{Tree}) \to n\text{-}\mathit{Tree}$, $i = 1, \ldots, n$

Similar to the case of one functional argument, one defines functions

$$\mathit{tree2fun} : n\text{-}\mathit{Tree} \to n\text{-Func} \quad \text{and} \quad \mathit{fun2tree} : n\text{-Func} \to n\text{-}\mathit{Tree} \enspace .$$

Now, the result of Theorem 12 can be generalized for n-Func.

Theorem 24. *Given a pure $F \in n$-Func, $\mathit{tree2fun}_T(\mathit{fun2tree}\, F) = F_T$ holds (extensionally) for any $T \in \mathit{Monad}$.* □

The formal Coq proof of the theorem is provided in the total setting and uses dependent types.

Characterization for the type n-Func with a parameter type D (equivalently, with finitely many parameter types $D_1, \ldots, D_k$)

$$n\text{-Func}_D = \forall T.D \to (A_1 \to TB_1) \to \cdots \to (A_n \to TB_n) \to TC$$

is similar, with parameterized strategies of type

$$n\text{-}\mathit{Tree}_D = D \to n\text{-}\mathit{Tree} \enspace .$$

For types of order higher than two it is not that clear yet what corresponding strategies should be let alone how one could characterise their existence by parametricity. It could be, however, that strategies in the sense of game semantics, like in [1, 2, 9], are the right generalization. The another possible approach is in using of Kripke relations of varying arity as in [10]. This might be an interesting question for further investigation.

```
let rec solve (n:int) x $s$ : $Maybe\, S$ =
  match n with
  | 0 → None
  | _ →
    if is_stable x $s$ then
      Some $s$
    else
      let $s_0$ = add_stable x $s$ in
      do p ← $F$ (eval (n−1) x) $s_0$;
      let (d, $s_1$) = p in
      let cur = getval $s_1$ x in
      if d ⊑ cur then
        Some $s_1$
      else
        let $s_2$ = setval x (cur ⊔ d) $s_1$ in
        let (w, $s_3$) = extract_work $s_2$ in
          solve_all (n−1) w $s_3$
```

```
and solve_all (n:int) w $s$ : $Maybe\, S$ =
  match n with
  | 0 → None
  | _ →
    match w with
    | [] → Some $s$
    | x :: xs →
      (solve (n−1) x $s$) ≫=
        solve_all (n−1) xs

and eval n x y : $State T_S\ Maybe\ \mathbb{D}$ =
  match n with
  | 0 → fun $s$ → None
  | _ → fun $s$ →
      let $s_0$ = add_infl y x $s$ in
      do $s_1$ ← solve (n−1) y $s_0$;
      Some (getval $s_1$ y, $s_1$)
```

Fig. 1. The pure functional implementation of totalized RLD

7 Applications

Modulus of continuity Recall that a functional $F : \mathbb{B} \to \mathbb{N}$ defined on the Baire space $\mathbb{B} = \mathbb{N} \to \mathbb{N}$ is *continuous at* $f \in \mathbb{B}$ *if* $F\ f$ depends only on finitely many elements of f. A *modulus* for F at f is a number n such that $F\ f$ depends only on the first n terms of f. Suppose F is *pure* functional, i.e., it is given by means of a monadically parametric function $\overline{F} : \prod_T .(\mathbb{N} \to T\mathbb{N}) \to T\mathbb{N}$ such that $F = \overline{F}_{Id}$, where Id is the identity monad. Then we can effectively extract a modulus for F at f by means of the functional

$$\texttt{Mod}\ F\ f = \max\left(snd\ (\overline{F}_{State_{list\,\mathbb{N}}}(\texttt{instr}\ f)\ [\,])\right)$$

where $\texttt{instr}\ f : \mathbb{N} \to State_{list\,\mathbb{N}} = \lambda a.\lambda l.(f\,a, l{+}{+}[a])$ instruments f by means of recording of a list of visited indices. That Mod computes what it is supposed to is shown by the following proposition.

Proposition 25. *Let $F : \mathbb{B} \to \mathbb{N}$ be pure, $f : \mathbb{B}$ and $m = \texttt{Mod}F\,f$. Then for every $g : \mathbb{B}$ if $f\,i = g\,i$ holds for all $i \le m$, then $F\,f = F\,g$.* □

Proof (sketch). Given $l_f = snd\ (\overline{F}_{State_{list\,\mathbb{N}}}(\texttt{instr}\ f)\ [\,])$, one can use l_f to traverse the strategy tree for $\overline{F}$ using values from l_f as corresponding answers at Que-nodes. We show that by so using l_f one reaches a leaf Ans$\,n$ with $n = F\,f$. By assumption, for all the questions queried when traversing with l_f, f and g must deliver identical answers. We conclude, $l_f = l_g$, and hence $F\,f = F\,g$. □

Verified fixpoint algorithms The provided characterization of pure functionals of type Func can be used for verification of generic off-the-shelf fixpoint algorithms which are used to compute a (local) solution of a constraint system $\mathtt{x} \sqsupseteq F_{\mathtt{x}}$, $\mathtt{x} \in V$, defined over a bounded join-semilattice $\mathbb{D}$ of abstract values and a set of variables V. The local solver RLD, which relies on *self-observation*, applies F to a special stateful function to discover variable dependencies and perform demand-driven evaluations [7]. In order to reason about the algorithm formally, we implement RLD in purely functional manner and model side-effects by means of the state monad. Thus, the pure right-hand side F is assumed to be of type

$$F : \forall S. V \to (V \to \mathit{State}_S \mathbb{D}) \to \mathit{State}_S \mathbb{D} \ .$$

Assuming that all right-hand sides F are pure and hence representable by strategy trees, one can formulate sufficient pre- and post-conditions to verify partial correctness of the algorithm.

Notice that RLD may diverge since we pose no extra restrictions on $\mathbb{D}$ (e.g., ascending chain condition) in general. However, we can define a totalized version of RLD by passing an extra natural parameter to every main function of the algorithm which limits a maximum depth of recursion. Once the limit is reached, the solver terminates with None. Figure 1 gives a pure functional implementation of the totalized version of RLD. Since F is pure, by Corollary 15, a corresponding strategy tree provides a monadically parametric implementation, which can be used as

$$F : V \to (V \to \mathit{StateT}_S\, \mathit{Maybe}\, \mathbb{D}) \to \mathit{StateT}_S\, \mathit{Maybe}\, \mathbb{D}$$

where *Maybe* is an option monad, *StateT* is a state monad transformer, and S is a state structure managed by the solver. The total version can be implemented and proven correct in Coq with the certified code extracted in ML.

The characterization of 2-Func can be applied to verification of local fixpoint algorithms for *side-effecting* constraint systems [19] used for interprocedural analysis and analysis of multithreaded code. The main idea here is that in each constraint $x \sqsupseteq F_{\mathtt{x}}$ the right-hand side $F_{\mathtt{x}}$ is a pure function representable by a strategy tree with two kind of question nodes: QueR for which values of variables are queried using a stateful function `get` and QueW which, when accessed, update current values of some variables by means of a stateful function `set`. Thus, the strategy tree specifies a sequence of *reading* and *writing* accesses to some constraint variables. One version of such a solver although not verified presently is implemented in the program analyzer Goblint [23].

8 Conclusion

We have provided two equivalent characterisations of pure second-order functionals in the presence of nontermination; an extensional one based on preservation of relations and an intensional one based on strategy trees. All verifications have been formalized in Coq.

Our results can be applied to the verification of algorithms that take pure second-order functionals as input. Among these are generic fixpoint algorithms and algorithms for exact real arithmetic. It is generally easier to verify the correctness of such an algorithm assuming the intensional characterisation of purity for its input. On the other hand, for a concretely given input, e.g. in the form of a program in some restricted language it will be easier to establish the extensional characterisation. The techniques developed in this paper were extended to impure higher-order functions enabling modular reasoning about monadic mixin components [13].

We note that a closely related characterisation albeit in a rather different guise has already been given in O'Hearn and Reynolds landmark paper [16]. Our strategy trees appear there as an intensional characterisation of first-order Algol procedures which due to the call-by-name policy are in fact second-order functionals. New aspects of the present work are in particular the monadic formulation, the generalisation of the extensional characterisation to monads other than the state monad, and the complete formalisation in Coq.

Interestingly, our acceptable monadic relations in the total case (Definition 4), also appear in [22] where they are used to derive free theorems in the sense of Wadler [24] for Haskell programs in monadic style. However, the application to the characterisation of pure second-order functionals and the subsequent characterisation with strategy trees do not appear in loc.cit. It is, however, fair to say that the method of [22], being essentially the same as ours, could be used to derive our main result (Representation Theorems), assuming that one adapts it to the partial case which was left open in loc.cit.

As pointed out by an anonymous reviewer, a proof of our results can be given using Katsumata's $\top\top$-lifting construction [12] if one considers stategy trees as free monads. This approach would require $\mathit{Tree} \in \mathit{Monad}$ for all possible result sets C. However, from the practical point of view, we would prefer that F is defined for continuation monads rather than for syntactical monads Tree.

A natural question, albeit of mostly academic interest, is the extension of this work to higher than second order. Given that the strategy trees resemble winning strategies in game semantics it would seem natural to attempt to find extensional characterisations of the existence of a winning strategy. Care would have to be taken so as to sidestep the undecidability of lambda definability [14], thus the extensional property would have to be undecidable even if basic types receive a finite interpretation.

References

1. Samson Abramsky, Pasquale Malacaria, and Radha Jagadeesan. Full abstraction for pcf. In Masami Hagiya and John C. Mitchell, editors, *TACS*, volume 789 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 1994.
2. Samson Abramsky and Guy McCusker. Linearity, sharing and state: a fully abstract game semantics for idealized algol with active expressions. *Electr. Notes Theor. Comput. Sci.*, 3, 1996.
3. Nick Benton, Andrew Kennedy, and Carsten Varming. Formalizing domains, ultrametric spaces and semantics of programming languages, 2010. Submitted to Math. Struct. in Comp. Science.

4. Baudouin Le Charlier and Pascal Van Hentenryck. A universal top-down fixpoint algorithm. Technical Report CS-92-25, Brown University, Providence, RI 02912, 1992.
5. Christian Fecht and Helmut Seidl. A faster solver for general systems of equations. *Sci. Comput. Program.*, 35(2):137–161, 1999.
6. Andrzej Filinski. Representing monads. In Hans-Juergen Boehm, Bernard Lang, and Daniel M. Yellin, editors, *POPL*, pages 446–457. ACM Press, 1994.
7. Martin Hofmann, Aleksandr Karbyshev, and Helmut Seidl. Verifying a local generic solver in Coq. In Radhia Cousot and Matthieu Martel, editors, *SAS*, volume 6337 of *Lecture Notes in Computer Science*, pages 340–355. Springer, 2010.
8. Martin Hofmann, Aleksandr Karbyshev, and Helmut Seidl. What is a pure functional? In Samson Abramsky, Cyril Gavoille, Claude Kirchner, Friedhelm Meyer auf der Heide, and Paul G. Spirakis, editors, *ICALP (2)*, volume 6199 of *Lecture Notes in Computer Science*, pages 199–210. Springer, 2010.
9. J. M. E. Hyland and C.-H. Luke Ong. On full abstraction for pcf: I, ii, and iii. *Inf. Comput.*, 163(2):285–408, 2000.
10. Achim Jung and Jerzy Tiuryn. A new characterization of lambda definability. In Marc Bezem and Jan Friso Groote, editors, *TLCA*, volume 664 of *Lecture Notes in Computer Science*, pages 245–257. Springer, 1993.
11. Aleksandr Karbyshev. The accompanying Coq implementation. `https://github.com/karbyshev/purity/`, 2013.
12. Shin-ya Katsumata. A semantic formulation of $\top\top$-lifting and logical predicates for computational metalanguage. In C.-H. Luke Ong, editor, *CSL*, volume 3634 of *Lecture Notes in Computer Science*, pages 87–102. Springer, 2005.
13. Steven Keuchel and Tom Schrijvers. Modular monadic reasoning, a (co-)routine. In *IFL'12, pre-proceedings, RR-12-06*, August 2012.
14. Ralph Loader. The undecidability of *lambda*-definability.
15. John Longley. When is a functional program not a functional program? In *ICFP*, pages 1–7, 1999.
16. Peter W. O'Hearn and John C. Reynolds. From algol to polymorphic linear lambda-calculus. *J. ACM*, 47(1):167–223, 2000.
17. John C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, pages 513–523, 1983.
18. John C. Reynolds and Gordon D. Plotkin. On functors expressible in the polymorphic typed lambda calculus. *Inf. Comput.*, 105(1):1–29, 1993.
19. Helmut Seidl, Varmo Vene, and Markus Müller-Olm. Global invariants for analyzing multithreaded applications. *Proc. of the Estonian Academy of Sciences: Phys., Math.*, 52(4):413–436, 2003.
20. Alex K. Simpson. Lazy functional algorithms for exact real functionals. In Lubos Brim, Jozef Gruska, and Jirí Zlatuska, editors, *Proc. MFCS, LNCS 1450*, volume 1450 of *Lecture Notes in Computer Science*, pages 456–464. Springer, 1998.
21. The Coq Development Team. *The Coq proof assistant reference manual.* TypiCal Project (formerly LogiCal), 2012. Version 8.4.
22. Janis Voigtländer. Free theorems involving type constructor classes: functional pearl. In Graham Hutton and Andrew P. Tolmach, editors, *ICFP*, pages 173–184. ACM, 2009.
23. Vesal Vojdani and Varmo Vene. Goblint: Path-sensitive data race analysis. In *Annales Univ. Sci. Budapest., Sect. Comp*, volume 30, pages 141–155, 2009.
24. Philip Wadler. Theorems for free! In *FPCA*, pages 347–359, 1989.