What is a pure functional?

Martin Hofmann[1], Aleksandr Karbyshev[2], and Helmut Seidl[2]

[1] Institut für Informatik, Universität München, Oettingenstrasse 67, 80538 München, Germany,
hofmann@ifi.lmu.de,
[2] Fakultät für Informatik, Technische Universität München

Abstract. Given an ML function `f : (int->int)->int` how can we rigorously specify that `f` is pure, i.e., produces no side-effects other than those arising from calling its functional argument? We show that existing methods based on preservation of invariants and relational parametricity are insufficient for this purpose and thus define a new notion that captures purity in the sense that for any functional F that is pure in this sense there exists a corresponding question-answer strategy. This research is motivated by an attempt to prove algorithms correct that take such supposedly pure functionals as input and apply them to stateful arguments in order to inspect intensional aspects of their behaviour.

1 Introduction

Suppose we are given an unknown SML-function `f : (int->int)->int` as *input*. How can we rigorously specify that `f` does not cause side-effects other than those that might arise by applying `f` to a side-effecting argument? Our motivation for studying this question stems from an attempt to rigorously verify fixpoint solvers that take such a supposedly pure functional as input. Let us explain this application in some more detail.

Generic fixpoint solvers have successfully been used as the core algorithmic engine for program analyser frameworks both for logic programming languages [5,7] and C [16], see also [11]. Such solvers take as input an arbitrary equation system over a set Var of variables and some complete lattice Dom. Each such system is specified as a function from the set Var of unknowns to their respective right-hand sides where each right-hand side is considered as a *function* of type $(\mathsf{Var} \to \mathsf{Dom}) \to \mathsf{Dom}$ which typically is implemented in some specification language. The generic local solver then starts from a set of interesting variables and explores the variable space Var only in so far as their values *contribute* to the values of the interesting variables. In order to evaluate as few right-hand sides as possible, any efficient fixpoint algorithm takes dependencies among variables into account. If right-hand sides, however, are just semantically given as functions, no static preprocessing is possible to identify (a superset of) such dependencies. Therefore, generic solvers such as [3, 6, 9] rely on *self-observation* to identify the variable dependencies when they are encountered during fixpoint iteration. Due to this reflective behaviour, these algorithms are quite involved and thus difficult to be proven correct. While they are formulated as being applicable to systems of equations using *arbitrary* functions as right-hand sides, they clearly can only work properly for right-hand sides which are sufficiently well-behaved.

Other situations where supposedly pure functionals arise as input include Simpson's algorithm for exact integration [17] and the program transformations described in [10]. More distantly, we hope that our results might also contribute to the issue of purity in object-oriented specification [15] and be of interest to the Haskell community.

In this paper, we give an extensional semantic criterion for purity and show that it entails the existence of a strategy tree for the given functional. Thus, when trying to verify one of the aforementioned algorithms one can assume without loss of generality that the functional input is presented in the form of such a strategy tree which allows for a convenient proof by induction. Alternatively, one can use our extensional criterion directly for the verification and indeed we do so when verifying a recursive program that extracts a strategy tree from a given semantically pure functional.

Section 3 and 4 review classical parametricity and explain its weakness in the given context. Section 5 defines a new, stronger, notion of parametricity which by way of universal quantification yields our concept of semantic purity. Section 6 applies our notion to identify snapback (memorising and restoring the initial state) as impure, something that was hitherto impossible in a setting of total functions. Section 7 defines an inductive set of strategy trees which are shown in Sections 8 and 9 to represent pure functionals. Section 10 explains relations to Algol theory and game semantics.

The proofs except the one of Theorem 6 have been formalised in Coq. Moreover, one of us has just completed the formal verification (also in Coq) of a generic fixpoint algorithm using the results reported here. This will be published in a companion paper.

2 Preliminaries

For sets X and Y we denote by $X \times Y$ the Cartesian product and by $X \to Y$ the function space. We denote pairs by (x, y) and projections by $\mathit{fst}(x)$ and $\mathit{snd}(x)$. We use λ and juxtaposition for function abstraction and applications. We use the notations $f : X \to Y$ and $f \in X \to Y$ interchangeably. If $(X_i)_{i\in I}$ is a family of sets then we denote $\prod_{i\in I} X_i$ or simply $\prod_i X_i$ its Cartesian product. If $f \in \prod_i X_i$ then $f_i \in X_i$. We write $\mathbb{B}$ for the set $\{\mathsf{tt}, \mathsf{ff}\}$ of truth values. We use $\Rightarrow$ for logical implication; it binds weaker than the other logical connectives like $\wedge, \vee$. For sets X and S we define the *state monad* by

$$\mathsf{T}_S(X) := S \to S \times X\,.$$

We have the associated operations $\mathsf{val}_S : X \to \mathsf{T}_S(X)$ and $\mathsf{bind}_S : \mathsf{T}_S(X) \times (X \to \mathsf{T}_S(Y)) \to \mathsf{T}_S(Y)$ given by $\mathsf{val}_S(x)(s) = (s, x)$ and $\mathsf{bind}_S(f, g)(s) = g(x)(s_1)$ where $f(s) = (s_1, x)$. We tend to omit the index S whenever sensible.

If S is some set modelling global states, e.g., $S = \mathbb{Z} \times \mathbb{Z}$ in the case of two global variables of integer type, then an element f of $\mathsf{T}_S(X)$ may be viewed as a state-dependent and state-modifying expression of type X.

We let Var and Dom be two fixed sets, for example, $\mathsf{Var} = \mathsf{Dom} = \mathbb{Z}$. We fix elements $x_0 \in \mathsf{Var}, d_0 \in \mathsf{Dom}$. We define

$$\mathsf{Func} = \mathit{\Pi}_S.(\mathsf{Var} \to \mathsf{T}_S(\mathsf{Dom})) \to \mathsf{T}_S(\mathsf{Dom})$$

where the product Π_S ranges over a suitably large universe of sets. We do not intend to have the domain of the product to include Func itself so that we do not need to delve into the somewhat delicate issue of modelling impredicative polymorphism.

We view an element f of the function space $\mathsf{Var} \to \mathsf{T}_S(\mathsf{Dom})$ as a stateful function from Var to Dom: given $x \in \mathsf{Var}$ and a state $s \in S$ then $f(x)(s)$ yields a pair (s_1, d) thought of as final state (s_1) and result (d). The bind-construct models the application of such a function to a stateful expression of type Var, i.e., an element of $\mathsf{T}_S(\mathsf{Var})$.

3 Purity at first order

A stateful function $f : \mathsf{Var} \to \mathsf{T}_S(\mathsf{Dom})$ may be considered "pure" (side-effect-free) if there exists a function $g : \mathsf{Var} \to \mathsf{Dom}$ such that $f(x)(s) = (s, g(x))$, i.e., f may be factored through $\mathsf{val}_S : \mathsf{Dom} \to \mathsf{T}_S(\mathsf{Dom})$. This intensional viewpoint can in this case be underpinned by a more extensional yet equivalent definition as follows:

Theorem 1. *Let $f : \mathsf{Var} \to \mathsf{T}_S(\mathsf{Dom})$ be given. The following are equivalent:*

1. *f factors through $\mathsf{val}_S : \mathsf{Dom} \to \mathsf{T}_S(\mathsf{Dom})$.*
2. *For all relations $R \subseteq S \times S$ and $x \in \mathsf{Var}$ and sRs' one has $v = v'$ and $s_1 R s_1$ where $(s_1, v) = f(x)(s)$ and $(s_1', v') = f(x)(s')$.*

Proof. The direction 1$\Rightarrow$ 2 is obvious. For the converse, pick $s_0 \in S$ (the boundary case $S = \emptyset$ is obvious) and define $g : \mathsf{Var} \to \mathsf{Dom}$ by $g(x) = \mathit{snd}(f(x)(s_0))$. We claim that $f = \mathsf{val} \circ g$. To see this, fix $x \in \mathsf{Var}$ and $s \in S$ and define $R = \{(s_0, s)\}$. If $(s_1, v) = f(x)(s_0)$ and $(s_1', v') = f(x)(s)$ then, since $s_0 R s$ we get $v = v'$ and $s_1' = s$. □

A functional $F : (\mathsf{Var} \to \mathsf{T}_S(\mathsf{Dom})) \to \mathsf{T}_S(\mathsf{Dom})$ can be applied to stateful functions. Intuitively, it should be called pure if when applied to a stateful function f then the only side-effects that $F(f) \in \mathsf{T}_S(\mathsf{Dom})$ will have are those caused by calls to f within F. In particular, if f is pure as described above, then $F(f)$ should be pure, too, i.e., of the form $\mathsf{val}_S(d)$ for some $d \in \mathsf{Dom}$.

It is tempting to conjecture that such "pure" F would stem from a functional $G : (\mathsf{Var} \to \mathsf{Dom}) \to \mathsf{Dom}$. However, there is no way of applying such a G to a stateful $f : \mathsf{Var} \to \mathsf{T}_S(\mathsf{Dom})$ and, indeed, such a G does not contain enough information to tell how to transport the state changes and dependencies caused by calls to the argument f.

4 Relational parametricity

Let us therefore try to make progress with the relational approach. The following result may be encouraging.

Theorem 2. *Suppose that $F : (\{\star\} \to \mathsf{T}_S(\{\star\})) \to \mathsf{T}_S(\{\star\})$ is such that for all relations $R \subseteq S \times S$ the following is true: For all $k, k' : \{\star\} \to \mathsf{T}_S(\{\star\})$ such that for all $s, s' \in S$ sRs' implies $\mathit{fst}(k(\star)(s))\, R\, \mathit{fst}(k'(\star)(s'))$, one has that sRs' implies $\mathit{fst}(F(k)(s))\, R\, \mathit{fst}(F(k')(s'))$, for every $s, s' \in S$.*

Then there exists a natural number n such that $F = \mathit{it}_n$ where $\mathit{it}_0(k)(s) = (s, \star)$ and $\mathit{it}_{n+1}(k)(s) = k(\star)(\mathit{fst}(\mathit{it}_n(k)(s)))$.

Proof. We only show the case where $S = \mathbb{N}$, the general case is similar. We define $k_0(\star)(s) = (s+1, \star)$ and $n_0 = \mathit{fst}(F(k_0)(0))$. Intuitively, we assume that the state contains an integer variable which is incremented upon each call to k_0.

Now pick any k and s_0 and define $R = \{(n, \mathit{fst}(\mathit{it}_n(k)(s_0))) \mid n \in \mathbb{N}\}$. We have $(0, s_0)$ and whenever sRs' then $\mathit{fst}(k_0(\star)(s))\, R\, \mathit{fst}(k(\star)(s'))$. Therefore, by assumption $n_0 = \mathit{fst}(F(k_0))\, R\, \mathit{fst}(F(k))$. The claim follows from the definition of R. □

We remark that this result can also be obtained as a consequence of a Theorem in [18].

It is therefore tempting to generalise this approach to the type of our functionals (and sneaking in polymorphic quantification over state types) as follows:

Definition 1. *A functional* $F : \prod_S(\mathsf{Var} \to \mathsf{T}_S(\mathsf{Dom})) \to \mathsf{T}_S(\mathsf{Dom})$ *is* relationally parametric *if the following is true for all* S, S' *and relations* $R \subseteq S \times S'$.

For all $k : \mathsf{Var} \to \mathsf{T}_S(\mathsf{Dom})$ *and* $k' : \mathsf{Var} \to \mathsf{T}_{S'}(\mathsf{Dom})$ *such that for all* s, s'

$$sRs' \Rightarrow \mathit{fst}(k(x)(s))\, R\, \mathit{fst}(k'(x)(s')) \wedge \mathit{snd}(k(x)(s)) = \mathit{snd}(k'(x)(s'))$$

holds, one has that the following holds for all s, s'*:*

$$sRs' \Rightarrow \mathit{fst}(F_S(k)(s))\, R\, \mathit{fst}(F_{S'}(k')(s')) \wedge \mathit{snd}(F_S(k)(s)) = \mathit{snd}(F_{S'}(k')(s'))\,.$$

Definition 2 (Snapback). *Define* $F_{snap} : \prod_S(\mathsf{Var} \to \mathsf{T}_S(\mathsf{Dom})) \to \mathsf{T}_S(\mathsf{Dom})$ *by* $(F_{snap})_S(k)(s) = (s, d)$ *where* $(s_1, d) = k(x_0)(s)$*. Thus,* F_{snap} *invokes* k *but discards the resulting state and only keeps the resulting value in* d*. Instead, the initial state is restored.*

The following is direct.

Proposition 1. F_{snap} *is relationally parametric.* □

Therefore, relational parametricity is not strong enough to ensure purity in the intuitive sense because snapback cannot be considered pure. Let us introduce the following abbreviations:

Definition 3.
- *If* X, X' *are sets then* $\mathsf{Rel}(X, X')$ *denotes the set of binary relations between* X *and* X'*, i.e.,* $\mathcal{P}(X \times X')$*;*
- *if* X *is a set then* $\Delta_X \in \mathsf{Rel}(X, X)$ *is the equality on set* X*;*
- *if* $R \in \mathsf{Rel}(X, X')$ *and* $S \in \mathsf{Rel}(Y, Y')$ *then* $R{\to}S \in \mathsf{Rel}(X{\to}Y, X'{\to}Y')$ *is given by* $f\ R{\to}S\ f' \iff \forall x\ x'.\, xRx' \Rightarrow f(x) S f'(x')$*;*
- *if* $R \in \mathsf{Rel}(X, X')$ *and* $S \in \mathsf{Rel}(Y, Y')$ *then* $R{\times}S \in \mathsf{Rel}(X{\times}Y, X'{\times}Y')$ *is given by* $f\ R{\times}S\ f' \iff \mathit{fst}(f)\, R\, \mathit{fst}(f') \wedge \mathit{snd}(f)\, S\, \mathit{snd}(f')$*;*
- *if* $R \in \mathsf{Rel}(S, S')$ *and* $Q \in \mathsf{Rel}(X, X')$ *then* $\mathsf{T}_R^{\mathsf{param}}(Q) \in \mathsf{Rel}(\mathsf{T}_S(X), \mathsf{T}_{S'}(X'))$ *is given by* $\mathsf{T}_R^{\mathsf{param}}(Q) := R{\to}R{\times}Q$*.*

Now, $F \in \mathsf{Func}$ is relationally parametric if for all S, S' and $R \in \mathsf{Rel}(S, S')$ one has

$$(F_S, F_{S'}) \in (\Delta_{\mathsf{Var}} \to \mathsf{T}_R^{\mathsf{param}}(\Delta_{\mathsf{Dom}})) \to \mathsf{T}_R^{\mathsf{param}}(\Delta_{\mathsf{Dom}})\,.$$

5 A new notion of parametricity

We view the problem with snapback as a deficiency of the definition $\mathsf{T}_R^{\mathsf{param}}(Q)$. A stronger way of lifting a relation $Q \in \mathsf{Rel}(X, X')$ to $\mathsf{Rel}(\mathsf{T}_S(X), \mathsf{T}_{S'}(X'))$ is needed. Rather than tinkering with specific formats (of which we see examples later on), we jump to the most permissive notion of relation on sets of the form $\mathsf{T}_S(X)$.

Definition 4. *Fix sets S, S'. For each X, X' and $Q \in \mathsf{Rel}(X, X')$ fix a relation $\mathsf{T}^{\mathsf{rel}}(Q) \in \mathsf{Rel}(\mathsf{T}_S(X), \mathsf{T}_{S'}(X'))$. The family $(X, X', Q) \mapsto \mathsf{T}^{\mathsf{rel}}(Q)$ is an* acceptable monadic relation *if*

- *for all $X, X', Q \in \mathsf{Rel}(X, X'), x \in X, x' \in X'$:*

$$xQx' \Rightarrow \mathit{val}_S(x)\ \mathsf{T}^{\mathsf{rel}}(Q)\ \mathit{val}_{S'}(x')\,;$$

- *for all $X, X', Q \in \mathsf{Rel}(X, X')$, $Y, Y', P \in \mathsf{Rel}(Y, Y')$, $x \in \mathsf{T}_S(X)$, $x' \in \mathsf{T}_{S'}(X')$, $f : X \to \mathsf{T}_S(Y)$, $f' : X' \to \mathsf{T}_{S'}(Y')$:*

$$x\mathsf{T}^{\mathsf{rel}}(Q)x' \wedge f(Q{\to}\mathsf{T}^{\mathsf{rel}}(P))f' \Rightarrow \mathsf{bind}_S(x, f)\ \mathsf{T}^{\mathsf{rel}}(P)\ \mathsf{bind}_{S'}(x', f')\,.$$

The lifting of state relations known from relational parametricity forms an example of an acceptable monadic relation as stated in the next proposition. We will later see examples of acceptable monadic relations that are not of this form.

Proposition 2. *If $R \in \mathsf{Rel}(S, S')$ then $Q \mapsto \mathsf{T}_R^{\mathsf{param}}(Q)$ is an acceptable monadic relation.* □

It is now possible to state and prove a parametricity theorem to the effect that all functions definable from lambda calculus, bind, and val respect any acceptable monadic relation. The precise formulation and proof sketch is elided here for lack of space and may be found in the full paper.

Let us return to the specific example set Func. We can use the new parametricity notion to single out the pure elements of Func as follows.

Definition 5. *A functional $F \in \mathsf{Func}$ is* pure *if*

$$(F_S, F_{S'}) \in (\Delta_{\mathsf{Var}} \to \mathsf{T}^{\mathsf{rel}}(\Delta_{\mathsf{Dom}})) \to \mathsf{T}^{\mathsf{rel}}(\Delta_{\mathsf{Dom}})$$

holds for all S, S' and for all acceptable monadic relations $\mathsf{T}^{\mathsf{rel}}$ for S, S'.

Notice that functionals arising as denotations of lambda terms involving "parametric" constants (i.e., those for which the parametricity theorem holds) are pure in this sense.

6 Ruling out snapback

Our aim in this section is to prove that the snapback functional from Def. 2 cannot be pure in the following positive sense:

Theorem 3. *Let $F \in \mathsf{Func}$ be pure. Put $\mathsf{Test} := \mathbb{B}$ and define $k_{\mathsf{test}} : \mathsf{Var} \to \mathsf{T}_{\mathsf{Test}}(\mathsf{Dom})$ by $k_{\mathsf{test}}(x)(s) = (\mathsf{tt}, d_0)$. If $F_{\mathsf{Test}}(k_{\mathsf{test}})(\mathsf{ff}) = (\mathsf{ff}, d)$ then $F_S(k)(s) = (s, d)$, for all S, $s \in S$ and $k : \mathsf{Var} \to \mathsf{T}_S(\mathsf{Dom})$.*

We apply F to a stateful argument k_{test} which — when called — sets a global boolean variable. If this variable remains unset after the evaluation of $F_{\mathsf{Test}}(k_{\mathsf{test}})$ then F did not call its argument and must therefore be constant.

In order to prove the theorem we construct a specific monadic relation.

Definition 6. *Let S be a set and $\mathsf{Test} = \mathbb{B}$. For each X, X' and $Q \in \mathsf{Rel}(X, X')$ define $\mathsf{T}_1^{\mathsf{rel}}(Q) \in \mathsf{Rel}(\mathsf{T}_{\mathsf{Test}}(X), \mathsf{T}_S(X'))$ by*

$$\begin{aligned}\mathsf{T}_1^{\mathsf{rel}}(Q) = \{(f, f') \mid \forall s\ s'\ s_1\ s_1'\ x\ x'.f(s) = (s_1, x) \wedge f'(s') = (s_1', x') \Rightarrow \\ (\exists x_0'.\, xQx_0') \wedge (\exists x_0.\, x_0Qx') \wedge (s_1 = \mathsf{ff} \Rightarrow xQx' \wedge s' = s_1' \wedge s = \mathsf{ff})\}\,.\end{aligned}$$

Note that the relations $\mathsf{T}_1^{\mathsf{rel}}(Q)$ are not of the usual form "related pre-states yield related post-states and related results". Rather, relatedness of results (x and x') is conditional on the final state having a specific property (here "being equal to ff").

Lemma 1. *The relations $\mathsf{T}_1^{\mathsf{rel}}(Q)$ form an acceptable monadic relation.*

Proof (Sketch). Let us abbreviate

$$\begin{aligned}Z(Q, s, s_1, s', s_1', x, x') \equiv \\ (\exists x_0'.\, xQx_0') \wedge (\exists x_0.\, x_0Qx') \wedge (s_1 = \mathsf{ff} \Rightarrow xQx' \wedge s' = s_1' \wedge s = \mathsf{ff})\,.\end{aligned}$$

In the val-case we have $s = s_1$ and $s' = s_1'$ and xQx' by assumption. The claim $Z(Q, s, s_1, s', s_1', x, x')$ is then trivial.

For the bind-case assume $Z(\check{Q}, s, \check{s}, s', \check{s}', \check{x}, \check{x}')$ and $g\,(\check{Q} \to \mathsf{T}_1^{\mathsf{rel}}(Q))\,g'$. We put $(s_1, x) = g(\check{x})(\check{s})$ and $(s_1', x') = g'(\check{x}')(\check{s}')$. We should prove $Z(Q, s, s_1, s', s_1', x, x')$. Choose $\check{x}_0'$ such that $\check{x}\check{Q}x_0'$. The assumption on g yields $Z(Q, \check{s}, s_1, \check{s}', ?, x, ?)$ thus in particular the existence of x_0' such that xQx_0'. Similarly, we show $\exists x_0.\, x_0Qx'$.

Now assume $s_1 = \mathsf{ff}$. Applying $g\,(\check{Q} \to \mathsf{T}_1^{\mathsf{rel}}(Q))\,g'$ to $\check{x}\check{Q}x_0'$ yields $\check{s} = \mathsf{ff}$ (this step is the reason why we carry these $\exists$-clauses around). From $Z(\check{Q}, s, \check{s}, s', \check{s}', \check{x}, \check{x}')$ and $\check{s} = \mathsf{ff}$ we then conclude $\check{x}\check{Q}\check{x}'$ and also $s = \mathsf{ff}$. Using the assumption on g, g' *again* we then obtain the remaining bit x_1Qx_1'. □

Lemma 2. *Let S be a set and $k : \mathsf{Var} \to \mathsf{T}_S(\mathsf{Dom})$. We have $(k_{\mathsf{test}}, k) \in \Delta_{\mathsf{Var}} \to \mathsf{T}_1^{\mathsf{rel}}(\Delta_{\mathsf{Dom}})$.*

Proof. Suppose that $(s_1, d) = k_{\mathsf{test}}(x)(s)$ and $(s_1', d') = k(x)(s')$. Since $s_1 = \mathsf{tt}$ all we have to prove is $\exists x_0'.\, d = x_0'$ and $\exists x_0.\, x_0 = d'$ which is obvious. □

Note that the only relation R such that $k_{\mathsf{test}}\,(\Delta_{\mathsf{Var}} \to R \to R \times \Delta_{\mathsf{Dom}})\,k$ holds for all k is the empty relation but that is useless since it does not relate initial states to each other.

Proof (of Theorem 3). We prove $F_{\mathsf{Test}}(k_{\mathsf{test}})\,\mathsf{T}_1^{\mathsf{rel}}(\Delta_{\mathsf{Dom}})\,F_S(k)$ using purity of F together with Lemmas 1 and 2. This directly gives the desired result. □

7 Strategy trees

In this section we show that pure elements of Func are in fact first-order objects, i.e., define a question-answer dialogue. We first define those dialogues that can be seen as strategies in a game leading to the computation of $F(k)$ for any given k. We associate with each such strategy t a pure functional *tree2fun*(t) in the obvious way. We then define a functional program *fun2tree* (see Appendix for ML code) that can extract a strategy from *any* functional whether pure or not.

However, the program might in general fail to terminate and produce "strategies" whose continuation functions do not terminate. We will first prove that *if* the program returns a proper strategy and the input functional is pure then the computed strategy corresponds to the input functional. To do this, we axiomatise the graph *Fun2tree* of the functional program restricted to proper strategies as a well-founded relation. Later in Section 9 we show that for pure input functional the program does indeed return a proper strategy, i.e., the well-founded relation defines a total function on pure functionals.

We focus on the set Func here since it comes from the intended applications to fixpoint solvers.

Definition 7 (Strategies). *The set* Tree *is inductively defined by the following clauses.*

- *If* $d \in \mathsf{Dom}$ *then* $\mathsf{answ}(d) \in \mathsf{Tree}$.
- *If* $x \in \mathsf{Var}$ *and* $f : \mathsf{Dom} \to \mathsf{Tree}$ *then* $\mathsf{que}(x, f) \in \mathsf{Tree}$.

The function tree2fun $: \mathsf{Tree} \to \mathsf{Func}$ *is (well-founded) recursively defined by:*

- *tree2fun*$(\mathsf{answ}(d))(k)(s) = (s, d)$*;*
- *tree2fun*$(\mathsf{que}(x, f))(k)(s) =$ *tree2fun*$(f(d))(k)(\check{s})$ *where* $(\check{s}, d) = k(x)(s)$.

In order to extract an element of Tree from a given functional we define the state set

$$\mathsf{Test} = \mathsf{Dom}^* \times \mathsf{Var}^* \times \mathsf{Var} \times \mathbb{B}\,.$$

As usual, $(-)^*$ is Kleene star. We refer to the components of $s = (\vec{d}, \vec{x}, x, b)$ by $\vec{d} = s.\mathsf{ans}$, $\vec{x} = s.\mathsf{qns}$, $x = s.\mathsf{arg}$, $b = s.\mathsf{cal}$.

We write $s[\mathsf{qns} := \vec{x}']$ for $(\vec{d}, \vec{x}', a, b)$ and use similar notation for the other components. For $\vec{d} \in \mathsf{Dom}^*$ the initial state $r_{\vec{d}}$ is given by $(\vec{d}, \varepsilon, x_0, \mathsf{ff})$ (recall that x_0 and d_0 are the default elements of Var, Dom).

Definition 8. *The function* $k_{\mathsf{test}} : \mathsf{Var} \to \mathsf{T}_{\mathsf{Test}}(\mathsf{Dom})$ *is given by:*

- $k_{\mathsf{test}}(x)(s) = (s, d_0)$*, if* $s.\mathsf{cal} = \mathsf{tt}$*;*
- $k_{\mathsf{test}}(x)(s) = (s[\mathsf{arg}{:=}x, \mathsf{cal}{:=}\mathsf{tt}], d_0)$*, if* $s.\mathsf{cal} = \mathsf{ff}$ *and* $s.\mathsf{ans} = \varepsilon$*;*
- $k_{\mathsf{test}}(x)(s) = (s[\mathsf{ans}{:=}\vec{d}, \mathsf{qns}{:=}\vec{x}x], d)$*, if* $s.\mathsf{cal} = \mathsf{ff}$*,* $s.\mathsf{ans} = d\vec{d}$ *and* $s.\mathsf{qns} = \vec{x}$*;*

where $d_0 \in \mathsf{Dom}$ *is the default element.*

Intuitively, so long as cal is not set, k_{test} reproduces the prerecorded answers from ans and stores the questions asked in qns. Once ans is empty the next question is stored in arg and cal is set preventing any further state modifications.

Definition 9. *The relation*

$$\mathit{Fun2treeAux} \subseteq ((\mathsf{Var}\to\mathsf{T}_{\mathsf{Test}}(\mathsf{Dom}))\to\mathsf{T}_{\mathsf{Test}}(\mathsf{Dom})) \times \mathsf{Dom}^* \times \mathsf{Tree}$$

is inductively defined by the following clauses.

- *If* $F(k_{\mathsf{test}})(r_{\vec{d}}) = (r_1, d)$ *and* $r_1.\mathsf{cal} = \mathsf{ff}$ *then* $\mathit{Fun2treeAux}(F, \vec{d}, \mathsf{answ}(d))$.
- *If* $F(k_{\mathsf{test}})(r_{\vec{d}}) = (r_1, d)$ *and* $r_1.\mathsf{cal} = \mathsf{tt}$ *and* $r_1.\mathsf{arg} = x$ *and* $f : \mathsf{Dom} \to \mathsf{Tree}$ *is such that* $\mathit{Fun2treeAux}(F, \vec{d}b, f(b))$, $b \in \mathsf{Dom}$, *holds then* $\mathit{Fun2treeAux}(F, \vec{d}, \mathsf{que}(x, f))$.

We also define

$$\mathit{Fun2tree}(F, t) \iff \mathit{Fun2treeAux}(F, \varepsilon, t)\,.$$

8 Strategy trees for pure functionals

We will argue later in section 9 that for any pure F there always exists t such that $\mathit{Fun2tree}(F, t)$. Here, we merely show that *if* $\mathit{Fun2tree}(F, t)$ then $F = \mathit{tree2fun}(t)$, thus F is induced by a strategy tree.

Theorem 4. *Suppose that* $F \in \mathsf{Func}$ *is pure and that* $\mathit{Fun2tree}(F_{\mathsf{Test}}, t)$ *holds. Then* $F = \mathit{tree2fun}(t)$.

We prove a more general statement involving the auxiliary relation *Fun2treeAux*. For that, we relate sequences of questions to sequences of answers w.r.t. a given $k : \mathsf{Var} \to \mathsf{T}_S(\mathsf{Dom})$.

Definition 10. *Suppose* S *is a set and* $k : \mathsf{Var} \to \mathsf{T}_S(\mathsf{Dom})$. *We define* $\mathsf{Mat}_S(k) \subseteq \mathsf{Var}^* \times \mathsf{Dom}^* \times S \times S$ *inductively by:*

- $\mathsf{Mat}_S(k)(\varepsilon, \varepsilon, s, s)$ *for all* $s \in S$.
- *If* $\mathsf{Mat}_S(k)(\vec{x}, \vec{d}, s, \check{s})$ *and* $(s_1, d) = k(x)(\check{s})$ *then* $\mathsf{Mat}_S(k)(\vec{x}x, \vec{d}d, s, s_1)$.

Basically, $\mathsf{Mat}_S(k)(\vec{x}, \vec{d}, s, s_1)$ asserts that if we apply k successively to the arguments in $\vec{x}$ beginning in state s then (threading intermediate states through) we end up in state s_1 and the results we obtain along the way are recorded in $\vec{d}$.

Theorem 4 is a direct consequence of the following characterisation of *Fun2treeAux*.

Theorem 5. *Suppose that* $F \in \mathsf{Func}$ *is pure and that* $\mathit{Fun2treeAux}(F_{\mathsf{Test}}, \vec{d}, t)$ *holds. Suppose furthermore that* $F_{\mathsf{Test}}(k_{\mathsf{test}})(r_{\vec{d}}) = (r, \_)$ *and* $\mathsf{Mat}_S(k)(r.\mathsf{qns}, \vec{d}, s, \check{s})$ *holds. If* $F_S(k)(s) = (s_1, d_1)$ *and* $\mathit{tree2fun}(t, k, \check{s}) = (s_2, d_2)$ *then* $s_1 = s_2$ *and* $d_1 = d_2$.

The proof of Theorem 4 is by induction on *Fun2treeAux* and breaks down into the following two lemmas covering base case and inductive case.

Lemma 3 (Base case). *Let* F *be a pure functional. If* $F_{\mathsf{Test}}(k_{\mathsf{test}})(r_{\vec{d}}) = (r, v)$ *and* $\mathsf{Mat}_S(k)(r.\mathsf{qns}, \vec{d}, s, s_1)$ *and* $r.\mathsf{cal} = \mathsf{ff}$ *then* $F_S(k)(s) = (s_1, v)$.

This lemma is similar to Theorem 3 but is complicated by the fact that k_{test} only sets cal to $\mathtt{tt}$ after having worked off the pre-recorded answers $\vec{d}$. Accordingly, the Lemma requires that k match these prerecorded answers w.r.t. the questions asked on the way ($r.\mathsf{qns}$). The proof uses an acceptable monadic relation in the following general format.

Definition 11. *Let S, S' be sets. Let $Tr \in \mathsf{Rel}(S,S)$ and $Re, Gu \in \mathsf{Rel}(S\times S', S\times S')$ and $Q \in \mathsf{Rel}(X,X')$. The relation $\mathsf{T}^{\mathsf{rel}}_{Tr,Re,Gu}(Q) \in \mathsf{Rel}(\mathsf{T}_S(X), \mathsf{T}_{S'}(X'))$ is defined by*

$$\begin{array}{l} f\ \mathsf{T}^{\mathsf{rel}}_{Tr,Re,Gu}(Q)\ f' \iff \forall s\ s'\ s_1\ s_1'\ x\ x'.\\ \quad f(s) = (s_1,x) \wedge f'(s') = (s_1',x') \Rightarrow (\exists x_0'.\, xQx_0') \wedge (\exists x_0.\, x_0 Q x') \wedge \\ \qquad \mathit{Tr}(s,s_1) \wedge (\mathit{Re}((s,s'),(s_1,s_1')) \Rightarrow xQx' \wedge \mathit{Gu}((s,s'),(s_1,s_1'))\,. \end{array}$$

Lemma 4. *If Tr, Gu are reflexive and transitive and furthermore*

$$\begin{array}{l} \mathit{Re}((s,s'),(s_1,s_1')) \wedge \mathit{Tr}(s,\check{s}) \wedge \mathit{Tr}(\check{s},s_1) \Rightarrow \\ \quad \mathit{Re}((s,s'),(\check{s},\check{s}')) \wedge (\mathit{Gu}((s,s'),(\check{s},\check{s}')) \Rightarrow \mathit{Re}((\check{s},\check{s}'),(s_1,s_1'))) \end{array}$$

holds then $Q \mapsto \mathsf{T}^{\mathsf{rel}}_{Tr,Re,Gu}(Q)$ is an acceptable monadic relation. □

We could have a more general format that also maintains a transition relation corresponding to Tr on the S'-component, but this is not needed for our present purpose.

Proof (of Lemma 3, Sketch). We instantiate Lemma 4 w.r.t. the state sets Test and S:

$$\begin{array}{ll} \mathit{Tr}(r,r_1) & \equiv \exists \vec{x}\ \vec{d}.\, \mathit{TrP}(r,r_1,\vec{x},\vec{d})\,,\\ \mathit{Re}((r,s),(r_1,s_1)) & \equiv r_1.\mathsf{cal} = \mathtt{ff} \wedge \forall \vec{x}\ \vec{d}.\, \mathit{TrP}(r,r_1,\vec{x},\vec{d}) \Rightarrow \exists \check{s}.\, \mathsf{Mat}_S(k)(\vec{x},\vec{d},s,\check{s})\,,\\ \mathit{Gu}((r,s),(r_1,s_1)) & \equiv \exists \vec{x}\ \vec{d}.\, \mathit{TrP}(r,r_1,\vec{x},\vec{d}) \wedge \mathsf{Mat}_S(k)(\vec{x},\vec{d},s,s_1)\,,\\ \mathit{TrP}(r,r_1,\vec{x},\vec{d}) & \equiv r_1.\mathsf{cal} = \mathtt{ff} \Rightarrow r.\mathsf{cal} = \mathtt{ff} \wedge r_1.\mathsf{arg} = r.\mathsf{arg} \wedge |\vec{x}| = |\vec{d}| \wedge \\ & \qquad r_1.\mathsf{qns} = r.\mathsf{qns}\,\vec{x} \wedge r.\mathsf{ans} = \vec{d}\, r_1.\mathsf{ans}\,. \end{array}$$

One must now show that these definitions meet the conditions of Lemma 4 and that the resulting monadic relation relates $k_{\mathsf{test}}(x)$ to $k(x)$ for all x. Note that via $\mathsf{Mat}_S(k)$ the definition of the monadic relation is dependent on the k in question. This was not the case in the proof of Theorem 3. The result then follows. □

Lemma 5 (Inductive Case). *Let F be a pure functional. If $F_{\mathsf{Test}}(k_{\mathsf{test}})(r_{\vec{d}}) = (r,v)$ and $F_{\mathsf{Test}}(k_{\mathsf{test}})(r_{\vec{d}\,d}) = (t',v')$ and $r.\mathsf{cal} = \mathtt{tt}$ then $r'.\mathsf{qns} = r.\mathsf{qns}\ r.\mathsf{arg}$.*

Notice that the inductive case no longer involves the state set S and k but operates entirely on the specific state set Test.

We use an acceptable monadic relation obeying the following generic format that does not seem to be an instance of the previous one used for the base case.

Definition 12. *Let S, S' be sets. Let $Tr \in \mathsf{Rel}(S,S)$, $Tr' \in \mathsf{Rel}(S',S')$ and $St1, St2 \in \mathsf{Rel}(S,S')$. The relation $\mathsf{T}^{\mathsf{rel}}_{Tr,Tr',St1,St2}(Q) \in \mathsf{Rel}(\mathsf{T}_S(X), \mathsf{T}_{S'}(X'))$ is defined by*

$$\begin{array}{l} f\ \mathsf{T}^{\mathsf{rel}}_{Tr,Tr',St1,St2}(Q)\ f' \iff \forall s\ s'\ s_1\ s_1'\ x\ x'.\\ \quad f(s) = (s_1,x) \wedge f'(s') = (s_1',x') \Rightarrow (\exists x_0'.\, xQx_0') \wedge (\exists x_0.\, x_0 Q x') \wedge \\ \qquad \mathit{Tr}(s,s_1) \wedge \mathit{Tr}'(s',s_1') \wedge (\mathit{St1}(s,s') \Rightarrow \mathit{St1}(s_1,s_1') \wedge xQx' \vee \mathit{St2}(s_1,s_1'))\,. \end{array}$$

Lemma 6. *If Tr, Tr' are reflexive and transitive and furthermore*

$$St2(s, s') \wedge Tr(s, s_1) \wedge Tr'(s', s_1') \Rightarrow St2(s_1, s_1')$$

then $Q \mapsto \mathsf{T}^{\mathsf{rel}}_{Tr, Tr', St1, St2}(Q)$ is an acceptable monadic relation. □

Proof (of Lemma 5 (Sketch)). We instantiate the framework with state sets $S := \mathsf{Test}$ and $S' := \mathsf{Test}$ and

$$\begin{aligned}
Tr(r, r_1) \;&\equiv Tr'(r, r_1) \equiv \\
&\quad (r_1.\mathsf{cal} = \mathsf{ff} \Rightarrow r.\mathsf{cal} = \mathsf{ff}) \wedge (r.\mathsf{cal} = \mathsf{tt} \Rightarrow r = r_1) \wedge \\
&\quad (r.\mathsf{ans} = \varepsilon \Rightarrow r = r_1 \vee r_1.\mathsf{ans} = \varepsilon \wedge r_1.\mathsf{cal} = \mathsf{tt} \wedge r_1.\mathsf{qns} = r.\mathsf{qns})\,, \\
St1(r, r') &\equiv r.\mathsf{cal} = \mathsf{ff} \wedge r'.\mathsf{cal} = \mathsf{ff} \wedge r'.\mathsf{ans} = r.\mathsf{ans}\; d \wedge r'.\mathsf{qns} = r.\mathsf{qns}\,, \\
St2(r, r') &\equiv r.\mathsf{cal} = \mathsf{tt} \wedge r.\mathsf{ans} = \varepsilon \wedge r'.\mathsf{ans} = \varepsilon \wedge r'.\mathsf{qns} = r.\mathsf{qns}\; r.\mathsf{arg}\,.
\end{aligned}$$

The main result is then a fairly direct consequence. □

Theorem 4 is proved by induction on *Fun2treeAux* employing Lemmas 3 and 5. □

9 Existence of strategy trees

We will now show that for any pure functional one can indeed find a corresponding strategy tree in the sense of *Fun2tree*. By the results of the previous section this then implies that any pure functional can be represented by or seen as a strategy tree.

Admittedly, this result came as a certain surprise to us: we believed for a long time that existence of strategy trees could only be guaranteed under some additional continuity assumptions. For example, the minimum functional $Min : (\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$ given by $Min(f) = \min\{f(n) \mid n\}$ is not continuous and cannot be represented by a strategy tree. However, there is no pure functional (with $\mathsf{Var} = \mathsf{Dom} = \mathbb{N}$) because it would have to make infinitely many calls to its argument which could be tracked by a suitable set S: consider, for instance, the application of such putative pure functional F to a $k : \mathbb{N} \to \mathsf{T}_{\mathbb{N}}(\mathbb{N})$ that increments a global variable upon each call.

Theorem 6. *Let $F \in \mathsf{Func}$ be pure. There exists t such that Fun2tree(F_{Test}, t).*

Proof (Sketch). Assume for a contradiction that no such t exists. By studying an unsuccessful attempt at constructing such a t (formally this involves the use of the axiom of choice) we can construct an infinite sequence $d_1, d_2, d_3, \ldots$ of elements of Dom such that $fst(F_{\mathsf{Test}}(k_{\mathsf{test}})(s_{\vec{d}_n})).\mathsf{cal} = \mathsf{tt}$ for all n where $\vec{d}_n = d_1 d_2 d_3 \ldots d_{n-1}$.

Now let Test^∞ be defined like Test except that the ans-component may contain finite as well as infinite lists over Dom. Let k^∞_{test} be the extension of k_{test} to Test^∞. By a simulation argument using Proposition 2 one finds $F_{\mathsf{Test}^\infty}(k^\infty_{\mathsf{test}})(s) = F_{\mathsf{Test}}(k_{\mathsf{test}})(s)$ whenever $s \in \mathsf{Test} \subseteq \mathsf{Test}^\infty$. The following facts are proved using a mild generalisation of the acceptable monadic relation used in the proof of Lemma 3:

- $F_{\mathsf{Test}^\infty}(k^\infty_{\mathsf{test}})(r_{\vec{d}}) = (r_1, d) \wedge r_1.\mathsf{cal} = \mathsf{tt} \Rightarrow r_1.\mathsf{ans} = \varepsilon$;
- $F_{\mathsf{Test}^\infty}(k^\infty_{\mathsf{test}})(r_{\vec{d}}) = (r_1, d) \Rightarrow \exists ans \in Dom^*.\vec{d} = ans\; r_1.\mathsf{ans}$;

– $F_{\mathsf{Test}^\infty}(k^\infty_{\mathsf{test}})(r_{\vec{d}}) = (r_1, d) \wedge r_1.\mathsf{cal} = \mathsf{ff} \Rightarrow \mathit{fst}(F_{\mathsf{Test}}(k_{\mathsf{test}})(r_{\vec{d}\restriction}))).\mathsf{cal} = \mathsf{ff}$ where $\vec{d}\restriction$ comprises the first $|r_1.\mathsf{qns}|$ elements of $\vec{d}$.

Let $\vec{d}$ be the infinite list of the d_i and write $(r_1, d) = F_{\mathsf{Test}^\infty}(k^\infty_{\mathsf{test}})(r_{\vec{d}})$. By the first and second fact we must have $r_1.\mathsf{cal} = \mathsf{ff}$. Thus, by the third fact, there exists n (namely $|r_1.\mathsf{qns}|$) with $\mathit{fst}(F_{\mathsf{Test}}(k_{\mathsf{test}})(s_{\vec{d}_n}))).\mathsf{cal} = \mathsf{ff}$, a contradiction. □

10 Related work

Relational parametricity has been introduced by Reynolds [13, 14] as a means for restricting the range of polymorphic type quantification. Wadler then popularised Reynolds' results in his famous [18].

Relational parametricity has also been used in order to justify program equivalences in *Idealized Algol* a higher-order call-by-name language with local variables and stateful integer expressions. The equivalences of interest rely on the fact that a procedure cannot modify or read a local variable declared in a disjoint scope. In this context, an alternative extension of relational parametricity has been developed which can also rule out snapback functionals: *strict logical relations* [12]. It works in the setting of monotone continuous functions on Scott domains and relies crucially on the following "built-in" parametricity property of such functions: if $F(\lambda x.\bot) = x \neq \bot$ then $F(k) = x$ for all k. Loc. cit. also relates certain functionals to strategy trees (there called resumptions).

The differences to our work are twofold: (1) we address all types of computational lambda calculus in particular allow to return functions as results; (2) we work in a total, set-theoretic framework whereas strict logical relations can only operate in the presence of $\bot$ and monotonicity.

The strategy trees are reminiscent of game semantics [1,2,8] and can be traced back even further to Berry-Curiens sequential algorithms [4] and even to Kleene's. The new aspect here is the construction of a strategy tree for any set-theoretic functional that is pure in an extensional sense defined by preservation of structure rather than by existence of intensional representations. It would be very interesting to investigate to what extent our notion of purity which makes sense at all types of the computational lambda calculus entails existence of strategies in the sense of game semantics or sequential algorithms also at those higher types.

References

1. S. Abramsky, R. Jagadeesan, and P. Malacaria. Full abstraction for PCF (extended abstract). In M. Hagiya and J. C. Mitchell, editors, *Theoretical Aspects of Computer Software*, pages 1–15. Springer-Verlag, 1994. 10
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. 10
3. Baudouin Le Charlier and Pascal Van Hentenryck. A Universal Top-Down Fixpoint Algorithm. Technical Report CS-92-25, Brown University, Providence, RI 02912, 1992. 1
4. G. Berry and Pierre-Louis Curien. Sequential algorithms on concrete data structures. *Theor. Comput. Sci.*, 20:265–321, 1982. 10

5. Christian Fecht. GENA - A Tool for Generating Prolog Analyzers from Specifications. In *2nd Static Analysis Symposium (SAS)*, pages 418–419. LNCS 983, 1995. 1
6. Christian Fecht and Helmut Seidl. A faster solver for general systems of equations. *Sci. Comput. Program.*, 35(2):137–161, 1999. 1
7. Manuel V. Hermenegildo, Germán Puebla, Kim Marriott, and Peter J. Stuckey. Incremental analysis of constraint logic programs. *ACM Trans. Program. Lang. Syst.*, 22(2):187–223, 2000. 1
8. 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
9. Niels Jorgensen. Finding Fixpoints in Finite Function Spaces Using Neededness Analysis and Chaotic Iteration. In *1st Static Analysis Symposium (SAS)*, pages 329–345. LNCS 864, Springer Verlag, 1994. 1
10. John Longley. When is a functional program not a functional program? In *ICFP*, pages 1–7, 1999. 1
11. Hanne Riis Nielson and Flemming Nielson. Flow logics for constraint based analysis. In *Compiler Construction, 7th International Conference (CC)*, pages 109–127. LNCS 1383, springer Verlag, 1998. 1
12. Peter W. O'Hearn and John C. Reynolds. From algol to polymorphic linear lambda-calculus. *J. ACM*, 47(1):167–223, 2000. 10
13. John Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing*. IFIP, North-Holland, 1983. 10
14. John C. Reynolds and Gordon D. Plotkin. On functors expressible in the polymorphic typed lambda calculus. Technical Report ECS-LFCS-88-53, University of Edinburgh, May 1988. 10
15. Arsenii Rudich, Ádám Darvas, and Peter Müller. Checking well-formedness of pure-method specifications. In *Proc. FM, LNCS 5014*, pages 68–83, 2008. 1
16. Helmut Seidl and Vesal Vojdani. Region analysis for race detection. In *Static Analysis, 16th Int. Symposium, (SAS)*, pages 171–187. LNCS 5673, Springer Verlag, 2009. 1
17. 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. 1
18. Philip Wadler. Theorems for free! In *FPCA*, pages 347–359, 1989. 4, 10

A Parametricity theorem

In the following, we introduce the call by value lambda calculus. Define simple types over some set of base types ranged over by o through the grammar

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

Fix an assignment of a set $[\![o]\!]_S$ for each base type o and set S; we extend $[\![-]\!]$ to all types by putting

$$\begin{aligned} [\![\tau_1 \times \tau_2]\!]_S &= [\![\tau_1]\!]_S \times [\![\tau_2]\!]_S \,, \\ [\![\tau_1 \to \tau_2]\!]_S &= [\![\tau_1]\!]_S \to \mathsf{T}_S([\![\tau_2]\!]_S) \,. \end{aligned}$$

Given a set of constants (ranged over by c) with their types τ^c and variables ranged over by x we define the lambda terms by

$$e ::= x \mid c \mid \lambda x.e \mid e_1\, e_2 \mid e.1 \mid e.2 \mid \langle e_1, e_2 \rangle \,.$$

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:

$$\frac{x \in \mathrm{dom}(\Gamma)}{\Gamma \vdash x : \Gamma(x)} \quad \frac{}{\Gamma \vdash c : \tau^c} \quad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2} \quad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \;\; \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\, e_2 : \tau_2}$$

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

For each S and c fix an interpretation $[\![c]\!]_S \in [\![\tau^c]\!]_S$.

An *environment* for a context Γ and S is a mapping η such that $x \in \mathrm{dom}(\Gamma)$ implies $\eta(x) \in [\![\Gamma(x)]\!]_S$. If $\Gamma \vdash e : \tau$ and η is such an environment then we define $[\![e]\!]_S(\eta) \in \mathsf{T}_S([\![\tau]\!]_S)$ by the following clauses:

$$\begin{aligned} [\![x]\!]_S(\eta) &= \mathsf{val}_S(\eta(x)) \,, \\ [\![c]\!]_S(\eta) &= \mathsf{val}_S([\![c]\!]_S) \,, \\ [\![\lambda x.e]\!]_S(\eta) &= \mathsf{val}_S(\lambda v.[\![e]\!]_S(\eta[x \mapsto v])) \,, \\ [\![e_1\, e_2]\!]_S(\eta) &= \mathsf{bind}_S([\![e_1]\!]_S(\eta), \lambda f.\mathsf{bind}_S([\![e_2]\!]_S(\eta), f)) \,. \end{aligned}$$

We omit semantic equations for pairing and projections.

Definition 13. *Fix sets S, S' and an acceptable monadic relation $\mathsf{T}^{\mathsf{rel}}$ for S, S'. Given a binary relation $[\![o]\!]^{\mathsf{rel}} \in \mathsf{Rel}([\![o]\!]_S, [\![o]\!]_{S'})$ for each base type o, we can associate a relation $[\![\tau]\!]^{\mathsf{rel}}_{\mathsf{T}^{\mathsf{rel}}} \in \mathsf{Rel}([\![\tau]\!]_S, [\![\tau]\!]_{S'})$ with each type τ by the following clauses:*

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

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

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

Proof. One proves the following stronger statement by induction on typing derivations. If $\Gamma \vdash e : \tau$ and η is an environment for Γ and S and η' is an environment for Γ and S' then $\forall x.\, \eta(x)\, [\![\Gamma(x)]\!]^{\mathsf{rel}}_{\mathsf{T}^{\mathsf{rel}}}\, \eta'(x)$ implies $[\![e]\!]_S(\eta)\, \mathsf{T}^{\mathsf{rel}}([\![\tau]\!]^{\mathsf{rel}}_{\mathsf{T}^{\mathsf{rel}}})\, [\![e]\!]_{S'}(\eta')$. □

As for base types and constants we can for example include a type of natural numbers N with zero, successor function, and a recursor as in Gödel's system T. One takes $[\![N]\!]_S = \mathbb{N}$ and $[\![N]\!]^{\mathsf{rel}}_{\mathsf{T}^{\mathsf{rel}}} = \Delta_{\mathbb{N}}$. Parametricity of the recursor is easily proved by induction. We can also include other datatypes like lists or trees. The relation on such types would insist on equal shapes and componentwise relatedness of the entries.

Note that since we model computations as total functions (no domains etc) we do not interpret nontermination and hence do not accommodate general recursion.

B OCAML Code

The function *tree2fun* corresponds to the following OCAML-program where $\mathsf{Var}, \mathsf{Dom}$ are represented by `int`:

```
type tree =
    Answer of int
  | Question of int * (int -> tree)
let rec tree2fun t k =
  match t with
    Answer c -> c
  | Question (a,f) -> tree2fun (f (k a)) k
```

The inductive definition *Fun2treeAux* represents the well-founded part of the graph of the following OCAML program.

```
let called = ref false
let arg = ref 0
let questions = ref []
let answers = ref []
let init ds = called:=false; arg:=0; questions:=[]; answers:=ds

let ktest a =
  if !called then 0
  else
    match !answers with
    | [] -> (called:=true; arg:=a; 0)
    | x::r -> questions:=!questions@[a]; answers:=r; x

let rec fun2tree_aux ff ds =
  init ds;
  let result = ff ktest in
    if not (!called) then Answer result
    else let a0 = !arg in
      Question (a0,fun b -> fun2tree_aux ff (ds@[b]))
let fun2tree ff = fun2tree_aux ff []
```

Applying `fun2tree` to `fun k -> 1 + k 0` correctly yields `Question(0,f)` where `f` $n = \mathtt{Answer}(n+1)$.