TECHNISCHE UNIVERSITÄT MÜNCHEN
Lehrstuhl für Informatik II

Monadic Parametricity of Second-Order Functionals

Aleksandr Karbyshev

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Die Dissertation wurde am 06.05.2013 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 02.09.2013 angenommen.

This thesis consists of two parts. In the first part, the following main problem is considered: given a (effectful) second-order function implemented in some programming language, say F: (`int` $\to$ `int`) $\to$ `int`, how can one rigorously specify that F is *pure*, i.e., it has no side-effects other than those produced by its (effectful) functional argument. We provide an extensional semantic criterion of *monadic parametricity (purity)* of second-order functionals. The approach in use extends the relational parametricity introduced by Reynolds [Rey83]. Moreover, we show that the criterion implies the existence of a strategy tree for a given functional F which represents a strategy for "playing the game" of computation of F. The results are presented in two settings: a total set-theoretic setting and a partial domain-theoretic one. Additionally, in the total case we consider a problem of parametricity in *state* monads and argue that extraction of a corresponding strategy is possible. The relation of our notion of purity to continuity is discussed.

Purity of higher-order functionals is not only an interesting theoretical question, but is of certain practical importance. We demonstrate applications of the notion to the extraction of intentional information from pure functionals, like modulus of continuity, as well as to design of certified algorithms for exact integration and local generic fixpoint solvers.

The second part of the thesis is concerned with a rigorous verification of partial correctness of the local generic fixpoint solver **RLD**. The assumption that an input of the solver is pure and, therefore, has a strategy tree representation makes it possible to provide sufficiently strong invariants and allows for an inductive proof of correctness. Additionally, we provide a modification of the solver which is exact (**RLDE**) and formulate sufficient conditions for termination of both of the solvers. Finally, we demonstrate how to extract executable OCaml programs of solvers from the formal development.

All the formalized proofs are carried out by means of the proof assistant Coq.

Acknowledgements

This thesis would not have been possible without the help and support, invaluable advices and patience of my supervisors, Prof. Dr. Helmut Seidl and Prof. Dr. Martin Hofmann, to whom I owe my deepest gratitude.

Also I thank sincerely Prof. Dr. Andrej Bauer, University of Ljubljana, and Prof. Dr. Alex Simpson, University of Edinburgh, for inviting me to visit them and for numerous fruitful discussions from which my work has profited a lot.

Special thanks goes to Andreas Reuß for chess lessons and proofreading the draft of the thesis.

Contents

List of Figures

List of Original Publications

1. 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

2. Andrej Bauer, Martin Hofmann, and Aleksandr Karbyshev. On monadic parametricity of second-order functionals. In Frank Pfenning, editor, *FoSSaCS*, volume 7794 of *Lecture Notes in Computer Science*, pages 225–240. Springer, 2013

3. 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

4. Martin Hofmann, Aleksandr Karbyshev, and Helmut Seidl. On the verification of local generic solvers. Technical report, Technische Universität München, 2013

Online Resources

1. Aleksandr Karbyshev. Purity: the accompanying Coq implementation. `https://github.com/karbyshev/purity/`, 2013

2. Aleksandr Karbyshev and Kalmer Apinis. Solvers: verified fixpoint algorithms. `https://github.com/karbyshev/solvers/`, 2013

Introduction

Suppose that we want to find a minimal solution of a system of constraints of two variables (unknowns) $\mathbf{x}$ and $\mathbf{y}$ in natural numbers

$$\begin{aligned} \mathbf{x} &\sqsupseteq \mathbf{y} \sqcup 5 \\ \mathbf{y} &\sqsupseteq \mathbf{x} \sqcap 3 \end{aligned} \tag{*}$$

where $\sqsubseteq$ is interpreted as the usual ordering $\leq$ on natural numbers, $\sqcup$ and $\sqcap$ are maximum and minimum functions, respectively. The system is easy to solve in one's head, and the minimal solution is $\{\mathbf{x} = 5, \mathbf{y} = 3\}$.

The matter gets more complicated when one has to deal with thousands or even millions of variables. In practice, efficient *fixpoint solvers* allow to tackle constraint systems of large size. A fixpoint solver is an algorithm that takes a constraint system as input and returns back a solution (a partial one, if a full solution is not needed) to the constraint system when it terminates. It usually starts with least "bottom" values for variables and tries iteratively to satisfy required constraints increasing values of variables when needed using some kind of strategy for picking a next constraint. The strategy may rely on extra information, e.g., dependencies between unknowns, which the algorithm keeps in internal data structures. The required information may be precomputed statically if right-hand sides are given explicitly in a simple format, like in (*).

Suppose, however, that the right-hand sides are given as *black boxes* which one cannot look into, or semantically as second-order functions of type $(V \to D) \to D$ implemented in some programming language, with the set of unknowns V. In order to efficiently deal with such constraint systems, the solver must rely on *self-observation* and track the variable dependencies on-the-fly storing them by means of side effects. However, then the algorithm works correct only if functions representing right-hand sides are "good" in certain sense: if they do not interfere with the solver by altering, for instance, a global state or producing any other side-effects by themselves. In the thesis, we introduce and study a semantical notion of *monadic parametricity* (which is a form of *purity*) that characterizes this "good" operational behaviour of functions. The monadic parametricity is a subject of Chapter 1.

As our experience confirms, a design of efficient fixpoint solvers is error-prone. For example, a draft of the textbook [SWH10] has presented a version of the local solver which has been in use for quite a lot of time and worked well for constraint systems arising in a real-life program analysis. However, there was no formal proof of correctness for the solver. Indeed, later we have discovered a counterexample and thus have shown that the solver was wrong. As we will see further, the counterexample to this erroneous solver is quite intricate and is based on a special case of circular dependencies between unknowns that never appeared in practice.

Testing as a verification method is not reliable for safety-critical systems. In order to be completely sure in correctness of the software, one needs to apply more rigorous methods like theorem proving. In this thesis, we follow the latter approach and present a formal development for a local generic solver **RLD** in the proof assistant Coq [Coq12] (see Chapter 2). We tried to formalize as many results as possible and verified not only the solver itself, but also a prerequisite theory of monadic parametricity.

Other examples of algorithms that take second-order functions as input are algorithms for exact real integration [Sim98, Lon99] which are briefly discussed in Chapter 1.

All results presented in the thesis are verified in Coq except for that it is explicitly indicated. The Coq code is available for download at [Kar13] and [KA13].

Structure of the Thesis

Chapter 1, Monadic Parametricity, is based on the following two publications:

– The paper [HKS10b] presented the notion of purity for second-order functionals parametric in state monads and characterization of pure functionals of type $\prod_S.(A \to \mathit{State}_S B) \to \mathit{State}_S C$ with *one* functional argument. In the thesis, we generalize the characterization of pure functionals for an arbitrary second-order type. Additionally, we provide a formalization of Theorem 1.3.35 about existence of strategy trees in Coq (although, not fully) which was missing in loc.cit.

– The paper [BHK13] generalized the notion of purity to arbitrary monadically parametric second-order functionals and provided their characterization in the total and the partial settings.

Chapter 2, Verified Generic Fixpoint Algorithms, is based on two other publications:

– The paper [HKS10a] presented a formal development of **RLD** fixpoint solver.

– The technical report [HKS13] which introduced the exact version **RLDE** of the solver. The version of **RLDE** presented in the paper is novel and differs from the one proposed in loc.cit. Moreover, in the thesis, we provide sufficient conditions for termination of both of the solvers. Additionally, we demonstrate how to extract a certified solver in ML from the Coq code.

1. Monadic Parametricity

1.1. Introduction

The main problem considered in this chapter can be formulated as follows. How can one rigorously specify that a given second-order ML functional F : (`int` $\to$ `int`) $\to$ `int` is *pure*, i.e., F only produces computational effects (changes a store, raises an exception, produces output, consumes input, etc.) only through calls of its functional argument?

Our motivation to study this kind of purity stems from a practical problem — an attempt to rigorously formalize and verify a local generic solver. Let us explain the problem in more detail.

Suppose we have a system of constraints over a bounded semi-lattice $\mathbb{D} = (D, \sqsubseteq, \bot)$

$$\mathbf{x}_i \sqsupseteq F_i(\mathbf{x}_1, \ldots, \mathbf{x}_n), \quad i = 1, \ldots, n \tag{*}$$

where each $\mathbf{x}_i \in V$ is a *variable* (or an *unknown*). A *solution* to a system (*) is a *variable assignment* $\sigma : V \to D$ such that $\sigma\,\mathbf{x}_i \sqsupseteq F_i(\sigma\,\mathbf{x}_1, \ldots, \sigma\,\mathbf{x}_n)$. More generally, we can think of the constraint system (*) as given by a second-order functional

$$F : V \to (V \to D) \to D$$

that for every variable $\mathbf{x}$ returns a respective right-hand side $F_{\mathbf{x}}$. A solution σ satisfies then $\sigma\,\mathbf{x} \sqsupseteq F_{\mathbf{x}}\,\sigma$, for all $\mathbf{x} \in V$. Generic fixpoint solver is an algorithm that takes as input an arbitrary constraint system F with a set of variables V over an arbitrary lattice of abstract values $\mathbb{D}$ and returns a solution in the above sense. The function F is usually implemented in some specification language.

However, in practice, a full solution to the constraint system is often not needed, or even not feasible if V is infinite. The user often wants to know a value of only one specific variables (or only few of them) which, for instance, may correspond to information in a particular program point if the given system of constraints is derived by an analysis of the program. *Local* generic solvers additionally take as input a finite list of *interesting* variables $X \subseteq V$ and try to solve only them and those unknowns that are needed for evaluation of right-hand sides $F_{\mathbf{x}}$, for $\mathbf{x} \in X$. An efficient local solver tries to determine relevant variables and thus, to evaluate as few unknowns as possible. For that, the solver must take into account the information about dependencies between variables. However, if the constraint system F is given as a *black box* and a static preprocessing of variable dependencies is not possible, the local solver must rely on *self-observation* in order to identify the dependencies while they are discovered during the iteration process. The bookkeeping of variable dependencies can be achieved by means of *side-effects*. That is, when re-evaluating a right-hand side $F_{\mathbf{x}}$, the latter is applied to a special instrumented function σ' that additionally keeps record of dependencies between unknowns in a dedicated store.

Notice that the algorithm is formulated as being applicable to arbitrary constraint functions F. However, it is clear that the solver does work properly only for sufficiently well-behaved right-hand sides F. As an examples of a "bad" F, one can take any functional that spoils structures maintained by the algorithm or a *snapback*-like functional which, when called, makes a snapshot of a state of the virtual machine, performs some computation and recovers the initial state of the system prior to returning a result. Clearly, the snapback functional cannot be considered as a pure one. We conclude that a "good" functional F should *not* produce any side-effects on its own, but only those that arise from its stateful argument σ'.

In order to make a formal reasoning about such a local solver possible and to implement it in a pure functional language like the one of Coq, one has to make side-effects explicit by means of a state monad *State*. For that, a second-order functional representing a constraint system must by lifted to a monadic computation of the type

$$F^\sharp : \textstyle\prod_S . V \to (V \to \mathit{State}_S D) \to \mathit{State}_S D.$$

It is intuitively clear that the behaviour of $F^\sharp$ should *not* depend on a kind of the state set S and thus, $F^\sharp$ is parametric in S. Two questions naturally arise. How can we characterize this type of parametricity? For which functionals F does a monadic lifting $F^\sharp$ exist?

One can generalize the posed problems to other types of effectful computations. Let *Monad* be a collection of all the effects presented in the programming language, given in terms of *monads* [Wad95]. How can one characterize pure second-order functionals of type

$$F : \textstyle\prod_{T \in \mathit{Monad}} . (A \to TB) \to TC$$

polymorphic with respect to monads? For a given $F : (A \to B) \to C$, is it always possible to construct a monadically parametric lifting $F^\sharp : \prod_{T \in \mathit{Monad}} . (A \to TB) \to TC$ such that $F = F^\sharp_{\mathit{Id}}$ where *Id* is the identity monad? What are the necessary conditions F must satisfy? Is F necessarily continuous?

The outline of the chapter is as follows. After the introductory section, we study a notion of purity in Section 1.2 as a semantical notion of monadic parametricity. In Section 1.3, we study a notion of parametricity for state monads. In Section 1.4, we discuss a relation of the notion of monadic parametricity to continuity. In Section 1.5, we discuss some application of purity to design of certified algorithms and verification of programs. Finally, we conclude and discuss a related work in Section 1.6.

Formal verification We present a formal verification in Coq [Coq12] for all the novel results except Theorem 1.3.35 for which only partial machine proof is available and Theorem 1.2.43 in the domain-theoretic setting. Formal proofs in the domain-theoretic setting are based on the constructive domain-theoretic framework developed in [BKV10] and use SSReflect library [GMT08]. The formal development is available for download at [Kar13].

However, familiarity with Coq is not necessary for understanding of results presented in this chapter. We say little or nothing about details and pitfalls of formalization in Coq though there are some.

The short introduction to Coq is given in Chapter 2.

Preliminaries

In what follows, we study the notion of monadic parametricity in the total set-theoretic setting and the partial domain-theoretic one. For the former we interpret types as sets and for the latter as cpos, *continuous posets* (precise definitions follow below), and thus we use the notations $a : X$ and $a \in X$ interchangeably. For sets (cpos) X and Y we write $X \times Y$ for a Cartesian product $X + Y$ for a disjoint sum and $X \to Y$ for a function space of total functions (a cpo of continuous functions). The symbol $\to$ is right-associative, i.e., $X \to Y \to Z$ means $X \to (Y \to Z)$. We denote pairs by (x, y), and projections by *fst* and *snd*. We use λ, $\circ$ and juxtaposition for function abstraction, composition and application, correspondingly. id denotes the identity function. We write *curry* and *uncurry* for currying and uncurrying functions defined by

$$\begin{aligned} \mathit{curry} &: (X \times Y \to Z) \to X \to Y \to Z = \lambda f.\lambda x.\lambda y.f\,(x, y) \\ \mathit{uncurry} &: (X \to Y \to Z) \to X \times Y \to Z = \lambda f.\lambda p.f\,(\mathit{fst}\ p)\,(\mathit{snd}\ p). \end{aligned}$$

For a family of sets or cpos $(X_i)_{i\in I}$ we write $\prod_{i\in I} X_i$ for its Cartesian product. If $F \in \prod_{i\in I} X_i$ then $F_i \in X_i$ denotes the i-th projection. $\sum_{i\in I} X_i$ stands for a disjoint union of $(X_i)_{i\in I}$. We may think of elements of $\sum_{i\in I} X_i$ as pairs (i, a), for $i \in I$, $x \in X_i$.

If A is a set then A^* denotes the set of all finite sequences of elements of A. The empty sequence is denoted by ε. We put a vector sign above a symbol, like $\vec{a}$, in order to stress that $\vec{a}$ is a sequence. If $\vec{a}$ is a finite sequence, $|\vec{a}|$ denotes its length. For sequences $\vec{a}$ and $\vec{b}$, $\vec{a}\vec{b}$ denotes their concatenation. We may consider an element $a \in A$ as a sequence of length 1. The notation $b \in \vec{a}$ means that b occurs in the sequence $\vec{a}$.

In logical formulas, conjunction $\wedge$ binds stronger than disjunction $\vee$ which in turn binds stronger than implication $\implies$, and quantifiers $\exists, \forall$ bind weakest.

Cpos and continuous functions The following standard definitions and propositions can be found in any textbook on domain theory, e.g., in [AJ94]. We remind them here however for the sake of completeness.

Definition 1.1.1. A partial order $\mathbb{D} = (D, \sqsubseteq_D)$ is called an *ω-cpo* (*ω-complete partial order*) if every ω-chain has a supremum in D, i.e., for every $c : \mathbb{N} \to D$ such that $c_i \sqsubseteq c_{i+1}$, for all i, there exists a least upper bound $\bigsqcup_{i\in\omega} c_i \in D$ for c, for which we will also write $\bigsqcup c$. From now on, we write simply *cpo* to refer to ω-cpo. A cpo $\mathbb{D}$ is *pointed* (shortly, *cppo*) if there is a least element in D denoted by $\bot_D$ (a *bottom* element). We will omit index D if it is clear from context. $\mathbb{D}$ is called *discrete* if $x \sqsubseteq y$ implies $x = y$.

Definition 1.1.2. For cpos $\mathbb{D}$ and $\mathbb{E}$, the function $f : D \to E$ is *monotonic* if $x \sqsubseteq y$ implies $f\,x \sqsubseteq f\,y$, for all x, y. Monotonicity of f implies $\bigsqcup(f \circ c) \sqsubseteq f(\bigsqcup c)$, for any chain c in D. We say that f is *continuous* if it is monotonic and $\bigsqcup(f \circ c) \sqsupseteq f(\bigsqcup c)$.

Definition 1.1.3. For cpos $\mathbb{D}$ and $\mathbb{E}$, the set $\mathbb{D} \to \mathbb{E}$ of all continuous functions when ordered pointwise forms a cpo. Indeed, consider a chain $c : \mathbb{N} \to (\mathbb{D} \to \mathbb{E})$ of continuous functions. Consider a function f defined by $f\,x = \bigsqcup(\lambda n.c_n\,x)$. It is easy to see that $f = \bigsqcup c$. If furthermore $\mathbb{E}$ is pointed then $\mathbb{D} \to \mathbb{E}$ is a cppo with the least element $\bot_{D\to E} = \lambda x.\bot_E$, a constant bottom function.

Definition 1.1.4. For a cpo $\mathbb{D}$, we define a *lifted* cppo $\mathbb{D}_\bot$ with the domain $D_\bot = D \uplus \{\bot\}$ and a partial ordering $\sqsubseteq_{D_\bot}$ defined by

$$x \sqsubseteq_{D_\bot} y \iff \begin{cases} x = \bot & \text{or} \\ x \sqsubseteq_D y, & \text{if } x, y \in D. \end{cases}$$

Let $\eta_D : D \to D_\bot$ and $\mathit{kleisli}_{D,E} : (D \to E_\bot) \to (D_\bot \to E_\bot)$ be defined by

$$\eta_D\,x = x \quad \text{ and } \quad \mathit{kleisli}_{D,E}\,f\,x = \begin{cases} \bot & \text{if } x = \bot \\ f\,x & \text{otherwise} \end{cases}$$

which we call *lifting* function and *Kleisli* function, correspondingly.

Remind some other standard ways of constructing cpos.

Definition 1.1.5. The *Cartesian product* of cpos $\mathbb{D}$ and $\mathbb{E}$ is defined by the following: it's domain $D \times E = \{(d, e) \mid d \in D, e \in E\}$, and ordering is componentwise, $(d_1, e_1) \sqsubseteq (d_1, e_1)$ iff $d_1 \sqsubseteq d_2$ and $e_1 \sqsubseteq e_2$.

Definition 1.1.6. The *direct sum* of cpos $\mathbb{D}$ and $\mathbb{E}$ is defined as follows. The domain $D + E = \{\mathit{inl}\,d \mid d \in D\} \cup \{\mathit{inr}\,e \mid e \in E\}$ is a disjoint union of D and E. The ordering is defined by

$$x \sqsubseteq_{D+E} y \iff \begin{cases} d_1 \sqsubseteq d_2 & \text{if } x = \mathit{inl}\,d_1 \text{ and } y = \mathit{inl}\,d_2 \\ e_1 \sqsubseteq e_2 & \text{if } x = \mathit{inr}\,e_1 \text{ and } y = \mathit{inr}\,e_2 \end{cases}$$

The next theorem allows to define a fixpoint operator for functions on cppos.

Theorem 1.1.7. *Given a cppo $\mathbb{D}$ and a continuous function $f : D \to D$, there exists a fixed point $\mathit{fix}\,f \in D$ of f, that is $f\,(\mathit{fix}\,f) = \mathit{fix}\,f$.*

Proof. Since the function $\mathtt{iter}\,f = \lambda i.f^i\,\bot_D$ forms a chain in $\mathbb{D}$, there exists $\mathit{fix}\,f = \bigsqcup(\mathtt{iter}\,f)$. By continuity of f, we infer $f\,(\mathit{fix}\,f) = f\,(\bigsqcup(\mathtt{iter}\,f)) = \bigsqcup(f \circ (\mathtt{iter}\,f)) = \bigsqcup(\mathtt{iter}\,f) = \mathit{fix}\,f$. $\square$

We introduce the polymorphic *fixpoint* operator for cppos $\mathit{fix}_D : (D \to D) \to D$. It is not difficult to show that fix_D is continuous, for any cppo $\mathbb{D}$.

Definition 1.1.8. For a cpo $\mathbb{D}$, a predicate $P \subseteq D$ is called *admissible* if it closed under taking lubs of chains, i.e., for any chain c in D such that $c_i \in P$, for all i, $\bigsqcup c \in P$ holds.

The following *fixpoint induction* principle can be easily established.

Lemma 1.1.9. *For a cppo $\mathbb{D}$, a continuous function $f : D \to D$ and an admissible relation $R \subseteq D$, the following is true.*

$$\textit{If } \bot_D \in R \textit{ and } (\forall x.\, x \in R \implies (f\,x) \in R) \textit{ then } (\mathit{fix}\, f) \in R.$$

Proof. Since $\mathit{fix}\ f = \bigsqcup_{i\in\omega} f^i \bot_D$ and R is admissible, it is sufficient to prove $f^n \bot_D \in R$, $n \in \mathbb{N}$. The latter can be seen by induction. For $n = 0$, $f^0 \bot_D = \bot_D \in R$ by the assumption. For the inductive step, assume $f^n \bot_D \in R$. Then $f^{n+1} \bot_D = f\,(f^n \bot_D) \in R$ by the assumption and the induction hypothesis. □

Monads

Definition 1.1.10. A *monad* is a triple $(T, \mathsf{val}_T, \mathsf{bind}_T)$ where T is the *monad type constructor* that for every X assigns the type TX of computations over X (TX is a *cppo* in the partial setting), and

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

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

– *left unit*: $\mathsf{bind}_T^{A,B}(\mathsf{val}_T^A\ a)\ f = f\,a$, for every $a \in A$

– *right unit*: $\mathsf{bind}_T^{A,A}\,t\ (\mathsf{val}_T^A) = t$

– *associativity*: $\mathsf{bind}_T^{B,C}(\mathsf{bind}_T^{A,B}\ t\ f)\ g = \mathsf{bind}_T^{A,C}\ t\ (\lambda x.\, \mathsf{bind}_T^{B,C}(f\ x)\ g)$.

For the partial setting, we also require that TX is a cppo and that T is *strict*, i.e.,

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

We omit indices A, B, C when they are clear from context.

Below are some examples of monads that we will use further. A trivial one is the *identity* monad *Id* which is defined by $\mathit{Id}\,X = X$ and trivial monadic operators

$$\mathsf{val}_{\mathit{Id}}\, x = x \quad \text{and} \quad \mathsf{bind}_{\mathit{Id}}\, t\, f = f\, t.$$

In the domain setting, the type constructor of *partiality* (or *lift*) monad is $T_\bot X = X_\bot$ and monadic operators are

$$\mathsf{val}_{T_\bot}^X = \eta_X \quad \text{and} \quad \mathsf{bind}_{T_\bot}^{X,Y}\ t\, f = \mathit{kleisli}_{X,Y}\ f\, t.$$

$T_\bot$ is obviously pointed and strict. It's total counterpart is the *exception* monad *Maybe* (also denoted as *Error*) with type constructor $\mathit{Maybe}\,X = \texttt{option}\,X$ — an inductive type with constructors $\mathsf{None} : \mathtt{option}\, X$ and $\mathsf{Some} : X \to \mathtt{option}\, X$. We will also use $\mathtt{error}$ and $\mathtt{value}$ as synonyms for None and Some, respectively. The monadic operators are given by

$$\mathsf{val}_{\mathit{Maybe}}\, x = \mathsf{Some}\, x \quad \text{and} \quad \mathsf{bind}_{\mathit{Maybe}}\, t\, f = \begin{cases} \mathsf{None} & \text{if } t = \mathsf{None} \\ f\, x & \text{if } t = \mathsf{Some}\, x \end{cases}$$

The *continuation* monad Cont_R with result type R (for the partial case, we require R to be a cppo) is defined by $\mathit{Cont}_R X = (X \to R) \to R$ and

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

In the partial case, Cont_R is pointed and strict since $\mathsf{bind}_{\mathit{Cont}_R} \perp_{\mathit{Cont}_R} f = \lambda c. \perp_R = \perp_{\mathit{Cont}_R}$.
Given a type of states S, we denote by State_S the *state monad* over S with $\mathit{State}_S X = S \to X \times S$, and monadic operations defined by

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

In the partial case, for a cpo S we define $\mathit{State}_S X = S \to (X \times S)_\perp$ and

$$\begin{aligned} \mathsf{val}_{\mathit{State}_S}\, x &= \lambda s. \eta\, (x, s) \\ \mathsf{bind}_{\mathit{State}_S}\, t\, f &= \lambda s. \mathit{kleisli}\, (\mathit{uncurry}\, f)\, (t\, s). \end{aligned}$$

A *monad transformer* is a type constructor that takes a monad as an argument and returns a monad as a result modifying the behaviour of an argument monad. Given a type of states S, the *state monad transformer* StateT_S maps a monad $(T, \mathsf{val}_T, \mathsf{bind}_T)$ to a monad defined by the monad constructor

$$\mathit{StateT}_S\, T\, X = S \to T(X \times S)$$

and monadic operations

$$\begin{aligned} \mathsf{val}_{\mathit{StateT}_S\, T}\, x &= \lambda s.\, \mathsf{val}_T (x, s) \\ \mathsf{bind}_{\mathit{StateT}_S\, T}\, t\, f &= \lambda s.\, \mathsf{bind}_T (t\, s)\, (\mathit{uncurry}\, f). \end{aligned}$$

The *exception monad transformer* ErrorT maps a monad $(T, \mathsf{val}_T, \mathsf{bind}_T)$ to a monad defined by the monad constructor

$$\mathit{ErrorT}\, T\, X = T(\mathtt{option}\, X)$$

with monadic operations

$$\begin{aligned} \mathsf{val}_{\mathit{ErrorT}\, T}\, x &= \mathsf{val}_T\, (\mathtt{value}\, x) \\ \mathsf{bind}_{\mathit{ErrorT}\, T}\, t\, f &= \mathsf{bind}_T\ t\ \left(\lambda x. \begin{cases} \mathsf{val}_T\ \mathtt{error} & x = \mathtt{error} \\ f\, a & x = \mathtt{value}\, a \end{cases} \right). \end{aligned}$$

1.2. Monadic Parametricity

In what follows, we solve one of problems posed in the introduction and provide a semantical characterization of *purity*.

In Subsection 1.2.1, we introduce the notion of an *acceptable monadic relation*. In Subsection 1.2.2, we 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 acceptable monadic relation. After that, in three subsequent subsections 1.2.3–1.2.5, we study the notion of *monadic parametricity* which is a special form of purity for first-order and second-order functionals. We consider both total and partial cases. We prove Representation Theorems stating that pure in that sense functionals can be represented as certain kinds of strategy trees. Finally, in Subsection 1.2.6, we generalize the notion to arbitrary second-order types.

For the rest of the chapter, 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. We think of *Monad* as a class of effects presented in a programming language.

1.2.1. Acceptable Monadic Relations

To define the notion of monadic parametricity we first introduce several notions and notations.

Definition 1.2.1. 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 $R \dot{\to} S \in \mathrm{Rel}(X \to Y, X' \to Y')$ is given by
$$f\,(R \dot{\to} S)\,f' \quad \text{iff} \quad \forall x\,x'.\, x\,R\,x' \implies (f\,x)\,S\,(f'\,x');$$
  As usually, the arrow symbol associates to the right, i.e., $Q \dot{\to} R \dot{\to} S$ should be read as $Q \dot{\to} (R \dot{\to} S)$.
- if $R \in \mathrm{Rel}(X, X')$ and $S \in \mathrm{Rel}(Y, Y')$ then $R \dot{\times} S \in \mathrm{Rel}(X \times Y, X' \times Y')$ is given by
$$p\,(R \dot{\times} S)\,p' \quad \text{iff} \quad (\mathit{fst}\ p)\,R\,(\mathit{fst}\ p') \wedge (\mathit{snd}\ p)\,S\,(\mathit{snd}\ p').$$

Definition 1.2.2. For cpos X, X' and $R \in \mathrm{Rel}(X, X')$, R is *admissible* if it is admissible as a relation in the product cpo $X \times X'$.

Lemma 1.2.3. *Let $\mathbb{D}, \mathbb{D}'$ be cpos, $\mathbb{E}, \mathbb{E}'$ cppos, and relations $R \in \mathrm{Rel}(D, D')$, $S \in \mathrm{Rel}(E, E')$ with S being admissible. Then $R \dot{\to} S$ is admissible.*

Proof. Assume $(f_i, f'_i)_{i\in\omega} \in R \dot{\to} S$ forms a chain. Then $(\bigsqcup_{i\in\omega} f_i, \bigsqcup_{i\in\omega} f'_i) \in R \dot{\to} S$. Indeed, take $(x, x') \in R$. Then
$$((\textstyle\bigsqcup_{i\in\omega} f_i)\,x, (\bigsqcup_{i\in\omega} f'_i)\,x') = (\bigsqcup_{i\in\omega} f_i\,x, \bigsqcup_{i\in\omega} f'_i\,x') \in S. \qquad \square$$

Definition 1.2.4. 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 $X, X', Q \in \mathrm{Rel}(X, X')$,

$$(\mathsf{val}_T, \mathsf{val}_{T'}) \in Q \dot{\to} T^{\mathsf{rel}}(Q);$$

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

$$(\mathsf{bind}_T, \mathsf{bind}_{T'}) \in T^{\mathsf{rel}}(Q) \dot{\to} (Q \dot{\to} T^{\mathsf{rel}}(R)) \dot{\to} T^{\mathsf{rel}}(R).$$

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., $(\perp_{TX}, \perp_{T'X'}) \in T^{\mathsf{rel}}(Q)$.

Below, we consider two examples of acceptable monadic relations that will be used in proofs of representation theorems. For continuation monads with result sets (cppos) S, S', we define a "simulation" monadic relation as follows.

Definition 1.2.5. Fix an arbitrary $W \in \mathrm{Rel}(S, S')$. For X, X' and $Q \in \mathrm{Rel}(X, X')$, we define $T^{\mathsf{rel}}_{\mathit{Cont}}(Q) \in \mathrm{Rel}(\mathit{Cont}_S X, \mathit{Cont}_{S'} X')$ by

$$T^{\mathsf{rel}}_{\mathit{Cont}}(Q) = (Q \dot{\to} W) \dot{\to} W.$$

Essentially, $T^{\mathsf{rel}}_{\mathit{Cont}}(Q)$ relates two functions H, H' employing continuations iff for any continuations h, h' which map related values to related results, the result values $H\,h$ and $H'\,h'$ are related by W.

Lemma 1.2.6. *Given $W \in \mathrm{Rel}(S, S')$, $Q \mapsto T^{\mathsf{rel}}_{\mathit{Cont}}(Q)$ is an acceptable monadic relation. In the partial case, if W is admissible and $(\perp_S, \perp_{S'}) \in W$ then $T^{\mathsf{rel}}_{\mathit{Cont}}$ is admissible and strict.*

Proof. We check properties from the definition of acceptability.

– In the val-case, for X, X' and $Q \in \mathrm{Rel}(X, X')$, take x, x' such that $x\,Q\,x'$ and $(h, h') \in Q \dot{\to} W$. Then we have $(\mathsf{val}_{\mathit{Cont}_S}\,x\,h, \mathsf{val}_{\mathit{Cont}_{S'}}\,x'\,h') = (h\,x, h'\,x') \in W$.

– In the bind-case, for X, X', $Q \in \mathrm{Rel}(X, X')$, and Y, Y', $R \in \mathrm{Rel}(Y, Y')$, take t, t' such that $t\,T^{\mathsf{rel}}_{\mathit{Cont}}(Q)\,t'$, and f, f' such that $f\,(Q \dot{\to} T^{\mathsf{rel}}_{\mathit{Cont}}(R))\,f'$. We have to show $(\mathsf{bind}_{\mathit{Cont}_S}\,t\,f)\,T^{\mathsf{rel}}_{\mathit{Cont}}(R)\,(\mathsf{bind}_{\mathit{Cont}_{S'}}\,t'\,f')$. Indeed, take $(h, h') \in R \dot{\to} W$. To demonstrate

$$(\mathsf{bind}_{\mathit{Cont}_S}\,t\,f\,h, \mathsf{bind}_{\mathit{Cont}_{S'}}\,t'\,f'\,h') = (t\,(\lambda x.f\,x\,h), t'\,(\lambda x.f'\,x\,h')) \in W$$

by the assumption on t, t' it is sufficient to show

$$(\lambda x. f\, x\, h, \lambda x. f'\, x\, h') \in Q \dot{\to} W$$

which is true since $(f\, x\, h, f'\, x'\, h') \in W$ holds by the assumption on f, f' for all x, x' such that $x\, Q\, x'$.

– For the partial case, a strictness of $T^{\mathsf{rel}}_{\mathit{Cont}}$ follows from $(\bot_S, \bot_{S'}) \in W$. As for admissibility, assume $(H_i, H'_i) \in T^{\mathsf{rel}}_{\mathit{Cont}}(Q)$, and let $(h, h') \in Q \dot{\to} W$. Then $((\bigsqcup_{i\in\omega} H_i)\, h, (\bigsqcup_{i\in\omega} H'_i)\, h') = (\bigsqcup_{i\in\omega} H_i\, h, \bigsqcup_{i\in\omega} H'_i\, h') \in W$ since $(H_i\, h, H'_i\, h') \in W$, for all i, and W is admissible. □

Definition 1.2.7. Fix a monad T, a set D and $W \in \mathrm{Rel}(TD, TD)$. For X, X', $Q \in \mathrm{Rel}(X, X')$, we define $T^{\mathsf{rel}}_{\mathit{Cont},T}(Q) \in \mathrm{Rel}(\mathit{Cont}_{TD}X, TX')$ by putting

$$(H, H') \in T^{\mathsf{rel}}_{\mathit{Cont},T}(Q) \quad \text{iff} \quad \forall h, h'.\, h\, (Q \dot{\to} W)\, h' \implies (H\, h)\, W\, (\mathsf{bind}_T\, H'\, h').$$

Lemma 1.2.8. *Given $T \in$ Monad, a set (or cpo, as the case may be) D and $W \in \mathrm{Rel}(TD, TD)$, $T^{\mathsf{rel}}_{\mathit{Cont},T}$ is an acceptable monadic relation. In the partial case, if W is admissible and $(\bot_{TD}, \bot_{TD}) \in W$ then $T^{\mathsf{rel}}_{\mathit{Cont},T}$ is admissible and strict.*

Proof. The proof is straightforward.

– In the val-case, for X, X' and $Q \in \mathrm{Rel}(X, X')$, let $x\, Q\, x'$ and $(h, h') \in Q \dot{\to} W$. Then $(\mathsf{val}_{\mathit{Cont}_{TD}}\, x\, h, \mathsf{bind}_T(\mathsf{val}_T\, x')\, h') = (h\, x, h'\, x') \in W$.

– In the bind-case, for X, X', $Q \in \mathrm{Rel}(X, X')$, Y, Y', $R \in \mathrm{Rel}(Y, Y')$, take t, t' such that $t\, T^{\mathsf{rel}}_{\mathit{Cont},T}(Q)\, t'$, and f, f' such that $f\, (Q \dot{\to} T^{\mathsf{rel}}_{\mathit{Cont},T}(R))\, f'$. We have to prove $(\mathsf{bind}_{\mathit{Cont}_{TD}}\, t\, f)\, T^{\mathsf{rel}}_{\mathit{Cont},T}(R)\, (\mathsf{bind}_T\, t'\, f')$. Indeed, we take $(h, h') \in R \dot{\to} W$ and show

$$\begin{aligned}(\mathsf{bind}_{\mathit{Cont}_{TD}}\, t\, f\, h, \mathsf{bind}_T(\mathsf{bind}_T\, t'\, f')\, h') ={}& \\ &(t\, (\lambda x. f\, x\, h), \mathsf{bind}_T\, t'\, (\lambda x.\, \mathsf{bind}_T(f'\, x)\, h')) \in W.\end{aligned}$$

In the last equality, we used the associativity of bind operator. Since $t\, T^{\mathsf{rel}}_{\mathit{Cont},T}(Q)\, t'$ it is sufficient to prove

$$(\lambda x. f\, x\, h, \lambda x.\, \mathsf{bind}_T(f'\, x)\, h') \in Q \dot{\to} W$$

which is true since $(f\, x\, h, \mathsf{bind}_T(f'\, x')\, h') \in W$ holds by the assumption on f, f', for all x, x' such that $x\, Q\, x'$.

– For the partial case, the strictness of $T^{\mathsf{rel}}_{\mathit{Cont},T}$ follows from $(\bot_{TD}, \bot_{TD}) \in W$ and the strictness of T. Show that $T^{\mathsf{rel}}_{\mathit{Cont},T}$ is admissible. For that, take $(H_i, H'_i) \in T^{\mathsf{rel}}_{\mathit{Cont},T}(Q)$, and let $(h, h') \in Q \dot{\to} W$. Then $((\bigsqcup_{i\in\omega} H_i)\, h, \mathsf{bind}_T(\bigsqcup_{i\in\omega} H'_i)\, h') = (\bigsqcup_{i\in\omega} H_i\, h, \bigsqcup_{i\in\omega} \mathsf{bind}_T\, H'_i\, h') \in W$ since $(H_i\, h, \mathsf{bind}_T\, H'_i\, h') \in W$, for all i, and W is admissible. □

Notice that both of relations $T^{\mathsf{rel}}_{\mathit{Cont}}$ and $T^{\mathsf{rel}}_{\mathit{Cont},T}$ may be considered as instances of the $\top\top$-lifting construction as described by Katsumata [Kat05].

1.2.2. Parametricity Theorem

In the following, we introduce a call-by-value λ-calculus ($\lambda_{\rightarrow}$ for short) with monadic semantics similar to Moggi's computational metalanguage [Mog91]. We provide a relational interpretation of types and terms of $\lambda_{\rightarrow}$. Finally, we establish the fundamental lemma of logical relations stating that every well-typed program respects any acceptable monadic relation.

Call-by-value lambda calculus $\lambda_{\rightarrow}$ We define simple types over sets of base types, ranged over by o, by the grammar

$$\tau ::= o \mid \tau_1 \times \tau_2 \mid \tau_1 \rightarrow \tau_2.$$

For each base type o and monad $T \in \mathit{Monad}$ we fix a set in the total case or a cpo in the partial case $[\![o]\!]_T$. Then we extend the monadic interpretation $[\![-]\!]_T$ to all types by

$$\begin{aligned}[\![\tau_1 \times \tau_2]\!]_T &= [\![\tau_1]\!]_T \times [\![\tau_2]\!]_T\\ [\![\tau_1 \rightarrow \tau_2]\!]_T &= [\![\tau_1]\!]_T \rightarrow T[\![\tau_2]\!]_T.\end{aligned}$$

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 the grammar

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

with the last rule for recursive definitions in the partial case only. A *typing context* Γ is a partial function mapping variables to their types. We write $\Gamma, x : \tau$ to designate the typing context that extends Γ by associating variable x with type τ. The typing judgement $\Gamma \vdash e : \tau$ is defined inductively by the usual rules as in Figure 1.1. 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 an environment for Γ and $T \in \mathit{Monad}$, we define a *monadic semantics* $[\![e]\!]_T(\eta) \in T[\![\tau]\!]_T$ by the clauses in Figure 1.2.

Parametricity theorem

Definition 1.2.9. Fix monads $T, T' \in \mathit{Monad}$ and an acceptable monadic relation T^{rel} for T, T'. Given a binary relation $[\![o]\!]^{\mathsf{rel}} \in \mathrm{Rel}([\![o]\!]_T, [\![o]\!]_{T'})$ (admissible, in the partial case) for each base type o, we associate a relation $[\![\tau]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}} \in \mathrm{Rel}([\![\tau]\!]_T, [\![\tau]\!]_{T'})$ with each type τ by the clauses

$$\begin{aligned}[\![o]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}} &= [\![o]\!]^{\mathsf{rel}}\\ [\![\tau_1 \times \tau_2]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}} &= [\![\tau_1]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}} \mathbin{\dot{\times}} [\![\tau_2]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}}\\ [\![\tau_1 \rightarrow \tau_2]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}} &= [\![\tau_1]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}} \mathbin{\dot{\rightarrow}} T^{\mathsf{rel}}([\![\tau_2]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}})\end{aligned}$$

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

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}\ (\textsc{Abs}) \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\, e_2 : \tau_2}\ (\textsc{App})$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash e.1 : \tau_1}\ (\textsc{Fst}) \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash e.2 : \tau_2}\ (\textsc{Snd}) \qquad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2}\ (\textsc{Prod})$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \text{let } x \leftarrow e_1 \text{ in } e_2 : \tau_2}\ (\textsc{Let})$$

$$\frac{\Gamma, f : \tau_1 \to \tau_2, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \text{let rec } f(x) = e : \tau_1 \to \tau_2}\ (\textsc{Rec})$$

Figure 1.1.: Typing rules for expressions in $\lambda_\to$

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

Figure 1.2.: Monadic semantics of expressions in $\lambda_\to$

Lemma 1.2.10. *In the partial case, $[\![\tau]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}}$ is admissible for every type τ.*

Proof. By induction on type definition. The base case, $\tau = o$, is trivial. The case $\tau = \tau_1 \times \tau_2$ is easy. For the case $\tau = \tau_1 \to \tau_2$, the statement follows from Lemma 1.2.3 and admissibility of T^{rel}. $\square$

The following *parametricity theorem* states that any function definable in $\lambda_{\to}$ respects any acceptable monadic relation. It is proven similarly for the total and the partial settings.

Theorem 1.2.11. *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*

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

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

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

See Appendix A.1 for the full proof. $\square$

Remark. Note that every well-typed program $\emptyset \vdash e : \tau$ of $\lambda_{\to}$ defines a (parametrically) polymorphic function of type $\prod_T . [\![\tau]\!]_T$ by taking a product over the type of monads. $\square$

Structural recursion and parametricity In the total case, the general recursion is not adapted. However, one can include in $\lambda_{\to}$ as a basic type any inductive data type and introduce corresponding constructors and a recursive scheme as constants. For example, we can introduce a type of natural numbers, finite lists or well-founded trees into the language. For the type of natural numbers `nat`, we include zero `0` : `nat` and the successor function `succ` : `nat` $\to$ `nat` constructors along with a primitive recursion operator

$$\mathcal{R}^{\tau}_{\mathtt{nat}} : \tau \times (\mathtt{nat} \times \tau \to \tau) \times \mathtt{nat} \to \tau$$

as in Gödel's system **T**. The relational semantics is defined by $[\![\mathtt{nat}]\!]_T = \mathbb{N}$ and $[\![\mathtt{nat}]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}} = \Delta_{\mathbb{N}}$ for any monad T and monadic relation T^{rel}.

Lemma 1.2.12. *The recursion operator $\mathcal{R}^{\tau}_{\mathit{nat}}$ is monadically parametric.*

Proof. By induction over `nat`. $\square$

For the type of finite lists over τ, we introduce two constructors `[]` : $\mathtt{list}\,\tau$ and $::$: $\tau \to \mathtt{list}\,\tau \to \mathtt{list}\,\tau$. The semantic relation $[\![\mathtt{list}\,\tau]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}}$ is defined by structural induction:

– `[]` $[\![\mathtt{list}\,\tau]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}}$ `[]`;

– if $a\, [\![\tau]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}}\, a'$ and $l\, [\![\mathtt{list}\,\tau]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}}\, l'$ then $(a :: l)\, [\![\mathtt{list}\,\tau]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}}\, (a' :: l')$.

Thus, $[\![\mathtt{list}\,\tau]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}}$ asserts structural equality of related lists and relatedness of their elements. Similarly to the above-mentioned $\mathcal{R}^{\tau}_{\mathtt{nat}}$, we can introduce a corresponding recursion operator $\mathcal{R}^{\sigma}_{\mathtt{list}\,\tau}$ and establish its parametricity by structural induction.

1.2.3. Purity

Purity at the first-order First, we discuss the monadic parametricity of first-order functions. We show that every pure in this sense first-order functional factorizes through the operator val.

Let *Monad* be a collection of monads. Intuitively, the function

$$f : \textstyle\prod_{T \in \mathit{Monad}}.A \to TB$$

can be considered as "pure" if there exists $g : A \to B$ such that $f = \Lambda T.\, \mathsf{val}_T \circ g$, i.e., the behaviour of f is fully determined by a total g having no effects. We show that our intuition is captured by the following precise extensional notion of parametricity.

Definition 1.2.13. The function $f : \prod_{T \in \mathit{Monad}}.A \to TB$ is *monadically parametric (pure)* if

$$(f_T, f_{T'}) \in \Delta_A \dot{\to} T^{\mathsf{rel}}(\Delta_B)$$

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

As a corollary from Parametricity Theorem 1.2.11 we obtain that such functionals exist.

Corollary 1.2.14 (of Theorem 1.2.11)**.** *Every first-order function F implemented in $\lambda_{\to}$ with monadic interpretation is monadically parametric in T.* □

First, we characterize monadically parametric functionals in the total case.

Theorem 1.2.15. *For a total $f : \prod_{T \in \mathit{Monad}}.A \to TB$ such that $\mathit{Id} \in \mathit{Monad}$, the following are equivalent*

1. *there exists $g : A \to B$ such that $f_T = \mathsf{val}_T \circ g$, for all $T \in \mathit{Monad}$.*
2. *f is monadically parametric.*

Proof. The direction 1 ⇒ 2 is easy. Indeed, given an acceptable monadic relation T^{rel} for T, T' and $a \in A$, $(\mathsf{val}_T(g\,a), \mathsf{val}_{T'}(g\,a)) \in T^{\mathsf{rel}}(\Delta_B)$ by definition. The reverse is more complicated.

Let f be monadically parametric. Put $g = f_{\mathit{Id}}$. To prove the required we construct an appropriate monadic relation using a ⊤⊤-construction similar to the one of Katsumata [Kat05]. Given X, X', $Q \in \mathrm{Rel}(X, X')$, define $Q^{\top} \in \mathrm{Rel}(X \to TB, X' \to B)$ by

$$h\, Q^{\top}\, h' \quad \text{iff} \quad \forall x, x'.x\, Q\, x' \implies h\,x = \mathsf{val}_T(h'\,x').$$

Define $Q^{\top\top} \in \mathrm{Rel}(TX, X')$ by

$$t\, Q^{\top\top}\, t' \quad \text{iff} \quad \forall h, h'.h\, Q^{\top}\, h' \implies \mathsf{bind}_T\, t\, h = \mathsf{val}_T(h'\,t').$$

It is not difficult to see that $Q \mapsto Q^{\top\top}$ is an acceptable monadic relation for T and *Id*. The val-case is easy. Indeed, let X, X', $Q \in \mathrm{Rel}(X, X')$ and take $(x, x') \in Q$, and let $(h, h') \in Q^{\top}$. Then $\mathsf{bind}_T(\mathsf{val}_T\ x)\, h = \mathsf{val}_T(h'\,(\mathsf{val}_{\mathit{Id}}\, x'))$ simplifies to $h\,x = \mathsf{val}_T(h'\,x')$

which holds by the assumption on h, h'. For the bind-case, let X, X', $Q \in \mathrm{Rel}(X, X')$, Y, Y', $R \in \mathrm{Rel}(Y, Y')$, and take $(t, t') \in Q^{\top\top}$ and $(g, g') \in Q \to R^{\top\top}$. We have to prove that for all h, h' such that $h\, R^{\top}\, h'$

$$\mathsf{bind}_T(\mathsf{bind}_T\, t\, g)\, h = \mathsf{val}_T(h'\,(\mathsf{bind}_{Id}\, t'\, g')).$$

We rewrite the last equation as

$$\mathsf{bind}_T\, t\, (\lambda x.\, \mathsf{bind}_T(g\, x)\, h) = \mathsf{val}_T((h' \circ g')\, t')$$

and apply the fact $(t, t') \in Q^{\top\top}$. The new goal is $(\lambda x.\, \mathsf{bind}_T(g\, x)\, h)\, Q^{\top}\, (h' \circ g')$. Finally, take $(x, x') \in Q$. The required equality $\mathsf{bind}_T(g\, x)\, h = \mathsf{val}_T(h'\,(g'\, x'))$ follows from the fact $(g\, x, g'\, x') \in R^{\top\top}$.

Since f is pure, we have $(f_T, f_{Id}) \in \Delta_A \dot{\to} \Delta_B^{\top\top}$. Therefore, for every $a \in A$ and $(h, h') \in \Delta_B^{\top}$, $\mathsf{bind}_T(f_T\, a)\, h = \mathsf{val}_T(h'\,(f\, a))$ holds. Put $h = \mathsf{val}_T$ and $h' = \mathsf{id}$. The required follows. □

Since $A_1 \to \cdots \to A_k \to TB \simeq A_1 \times \cdots \times A_k \to TB$, the theorem is valid for an arbitrary first-order type. Namely, any monadically parametric function

$$f : \textstyle\prod_{T \in Monad}. A_1 \to \cdots \to A_k \to TB$$

admits a factorization $f = \Lambda T.\, \mathsf{val}_T \circ g$ with $g = f_{Id}$.

Notice that the statement of Theorem 1.2.15 cannot be applied directly for the partial case. Consider, for example, the function $f_T\, x = \bot_{TB}$. It can be seen easily that f is monadically parametric, but $f_T\, x = \bot_{TB} \neq \mathsf{val}_T(g\, x)$ for any $g : A \to B$ and a non-trivial monad T. However, the claim turns out true if one takes functions of type $A \to B_\bot$ as representatives of pure first-order computations as shown by the next theorem.

Theorem 1.2.16. *For $f : \prod_{T \in Monad}. A \to TB$ such that the partiality monad $T_\bot \in$ Monad, the following are equivalent*

1. *there exists $g : A \to B_\bot$ such that $f_T = (kleisli\ \mathsf{val}_T) \circ g$, for all $T \in Monad$.*
2. *f is monadically parametric.*

Proof. The proof is similar to the one of Theorem 1.2.15. Towards $1 \Rightarrow 2$, take an acceptable monadic relation T^{rel} for T, T' and $a \in A$. We show

$$(kleisli\ \mathsf{val}_T\,(g\, a), kleisli\ \mathsf{val}_{T'}\,(g\, a)) \in T^{\mathsf{rel}}(\Delta_B)$$

by case distinction on $g\, a$. Thus, the proof is not constructive and uses the excluded middle axiom. If $g\, a = \bot$ then the assertion follows from strictness of T^{rel}. Otherwise, $g\, a = b : B$, and the goal simplifies to $(\mathsf{val}_T\ b, \mathsf{val}_{T'}\ b) \in T^{\mathsf{rel}}(\Delta_B)$ which is true by val-acceptability of T^{rel}.

For the reverse, we put $g = f_{T_\bot}$. Define a monadic relation $Q \mapsto Q^{\top\top}$ for monads T, $T_\bot$ as follows. Given X, X', $Q \in \mathrm{Rel}(X, X')$, define $Q^{\top} \in \mathrm{Rel}(X \to TB, X' \to B_\bot)$ by

$$h\, Q^{\top}\, h' \quad \text{iff} \quad \forall x, x'. x\, Q\, x' \implies h\, x = kleisli\ \mathsf{val}_T\,(h'\, x').$$

Define $Q^{\top\top} \in \mathrm{Rel}(TX, X')$ by

$$t\, Q^{\top\top}\, t' \quad \text{iff} \quad \forall h, h'. h\, Q^{\top}\, h' \implies \mathsf{bind}_T\, t\, h = \mathit{kleisli}\ \mathsf{val}_T\, (\mathit{kleisli}\, h'\, t').$$

The constructed monadic relation is acceptable for T and $T_\perp$. We omit the proof. The required is then derived from parametricity of f and the fact $(\mathsf{val}_T^B, \eta_B) \in \Delta_B^\top$. □

Conclusion. We have shown that there is a one-to-one correspondence between monadically parametric first-order functions of type $\prod_T.A \to TB$ and effect-free functions of type $A \to B_{(\perp)}$. In particular, this means that there is only one way of "lifting" of a function $f : A \to B_{(\perp)}$ to a monadically parametric first-order computation. Namely, one must compose f with the (lifted) monadic operator val.

Purity at the second order As we have shown, at the first-order any parametric functional $f : \prod_T.A \to TB$ is fully determined by an effect-free "strategy" $g : A \to B_{(\perp)}$ which may be considered as a forest that for every $a : A$ returns a sole answer of type B ($B_\perp$, in the partial case).

Not surprisingly, the matter is more complicated in the second-order case. Intuitively, the second-order functional $F : (A \to TB) \to TC$ is "pure" if F causes no effects on its own — the only effects produced by $F\,f$ are those that arise from calls to an argument function f within F. However, the hypothesis that every such a pure F arises from some effect-free $G : (A \to B) \to C$, similar to the first-order case, is wrong in general. Indeed, such a G is unable to tell how to treat and propagate effects from it's effectful argument f, and it is not even clear how one could apply such G to an effectful f at all.

For now, let us formalize what are pure second-order functionals we are interested in. Denote by

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

a product type taken over some fixed collection of monads *Monad*. We might again think of *Monad* as a collection of effects present in the given programming language.

Definition 1.2.17. A functional $F \in \mathrm{Func}$ is *monadically parametric (pure)* for the collection *Monad* of monads iff

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

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

The Parametricity Theorem provides us with an existence of pure second-order functionals.

Corollary 1.2.18 (of Theorem 1.2.11)**.** *Every second-order function F :* Func *implemented in $\lambda_\to$ with monadic interpretation is monadically parametric.* □

In the next two subsections we prove that such pure functionals admit question-answer strategy tree representations of certain kind both in a total set-theoretic setting and a partial domain-theoretic one. Moreover, the correspondence between functionals and strategies trees is one-to-one.

1.2.4. Second Order: the Total Case

First, we study our notion of purity in the set-theoretic setting in which every functional is total and a general recursion is not adapted. We assume that the functionals under consideration are defined for all continuation monads, i.e., $\mathit{Cont}_R \in \mathit{Monad}$ for all resulting sets R.

Define inductively a set of strategy trees — "skeletons" of pure second order functions.

Definition 1.2.19. The set of *strategy trees* $\mathit{Tree}_{A,B,C}$ is a minimal set generated by the constructors

- $\mathsf{Ans} : C \to \mathit{Tree}_{A,B,C}$
- $\mathsf{Que} : A \to (B \to \mathit{Tree}_{A,B,C}) \to \mathit{Tree}_{A,B,C}$

Thus, a strategy tree is either an *answer* leaf $\mathsf{Ans}\ c$ with the answer value $c : C$ or a *question* node $\mathsf{Que}\ a\, f$ with the query $a : A$ (an argument for which a question is asked) and the branching (continuation) function $f : B \to \mathit{Tree}$ that for every possible answer of type B returns another tree. Since types A, B, C are fixed, we omit these indices.

Example 1.2.20. For $A = B = C = \mathbb{N}$, $\mathsf{Ans}\ 1$ and $\mathsf{Que}\ 0\, (\lambda x.\mathsf{Ans}\, x)$ are strategy trees. □

For a given monad $T \in \mathit{Monad}$, every strategy tree defines a functional of type $(A \to TB) \to TC$ as in the following definition.

Definition 1.2.21. Given $T \in \mathit{Monad}$, we define the function $\mathit{tree2fun}_T : \mathit{Tree} \to (A \to TB) \to TC$ by structural recursion as

- $\mathit{tree2fun}_T(\mathsf{Ans}\ c) = \lambda k.\, \mathsf{val}_T\, c$
- $\mathit{tree2fun}_T(\mathsf{Que}\ a\, f) = \lambda k.\, \mathsf{bind}_T(k\, a)\, (\lambda b.\mathit{tree2fun}_T\, (f\, b)\, k)$.

Since $\mathit{tree2fun}_T$ is parametric in the monad T, we define a polymorphic version

$$\mathit{tree2fun}\, t = \textstyle\prod_{T \in \mathit{Monad}}.\mathit{tree2fun}_T\, t$$

of type $\mathit{Tree} \to \mathrm{Func}$.

Intuitively, the computation defined by a strategy tree can by considered as a sequence of queries to the first-order argument function k applied to elements from A followed by an answer in C. For every question node $\mathsf{Que}\ a\, f$ the result of $k\, a$ determines the followed branch defined by the continuation function f. The definition ensures that all the monadic effects come only from the argument function.

Example 1.2.22. Let $A = B = C = \mathbb{N}$ and a strategy tree $t = \mathsf{Que}\ 0\, (\lambda x.\mathsf{Ans}\ x)$. Then for a monad T

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

Thus, the tree $t = \mathsf{Que}\ 0\,(\lambda x.\mathsf{Ans}\ x)$ corresponds to a second-order function that queries its argument k at 0 and returns a produced result, in addition an effect of 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 1.2.23. *For any $t \in \mathit{Tree}$, $\mathit{tree2fun}\, t$ is monadically parametric.*

Proof. Take monads T, T' and an acceptable monadic relation T^{rel} for T, T'. We prove the statement by induction on t.

- $t = \mathsf{Ans}\ c$. It suffices to show that $(\mathsf{val}_T\, c, \mathsf{val}_{T'}\, c) \in T^{\mathsf{rel}}(\Delta_C)$. Indeed, it holds by the definition of acceptability of T^{rel}.
- $t = \mathsf{Que}\ a\, f$. Assume that

$$\mathit{tree2fun}\,(f\, b) \text{ is pure for every } b \in B \tag{IH}$$

Take $(k, k') \in \Delta_A \dot{\to} T^{\mathsf{rel}}(\Delta_B)$ and thus, $(k\, a, k'\, a) \in T^{\mathsf{rel}}(\Delta_B)$ for all $a : A$. Since T^{rel} is acceptable, it suffices to show that

$$(\lambda b.\mathit{tree2fun}_T\,(f\, b)\, k, \lambda b.\mathit{tree2fun}_{T'}\,(f\, b)\, k') \in \Delta_B \dot{\to} T^{\mathsf{rel}}(\Delta_C)$$

which holds by (IH). □

Perhaps somewhat surprisingly, a strategy tree can be extracted by means of the continuation monad.

Definition 1.2.24. We define function $\mathit{fun2tree} : \mathrm{Func} \to \mathit{Tree}$ as follows:

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

Representation theorem In what follows, we prove that functions *fun2tree* and *tree2fun* are mutually inverse in the total case. One direction is quite easy.

Lemma 1.2.25. *For any $t \in \mathit{Tree}$, $\mathit{fun2tree}\,(\mathit{tree2fun}\, t) = t$.*

Proof. By structural induction on t.

- $t = \mathsf{Ans}\ c$. Indeed,

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

- $t = \mathsf{Que}\ a\, f$. Assume the induction hypothesis

$$\mathit{fun2tree}\ (\mathit{tree2fun}\ (f\ b)) = f\ b \quad \text{for all } b \in B.$$

We have $\mathit{tree2fun}_T\,(\mathsf{Que}\,a\,f) = \lambda k.\,\mathsf{bind}_T(k\,a)\,(\lambda b.\mathit{tree2fun}_T\,(f\,b)\,k)$, for any T, and thus,

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

We used the induction hypothesis and extensionality in the last step. □

For the reverse statement we use purity of functionals.

Theorem 1.2.26. *Given a pure $F \in \mathrm{Func}$,*

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

holds (extensionally) for an arbitrary monad $T \in \mathit{Monad}$.

We first verify the statement for an arbitrary continuation monad.

Lemma 1.2.27. *Given a pure $F \in \mathrm{Func}$, $\mathit{tree2fun}_{\mathit{Cont}_S}\,(\mathit{fun2tree}\,F) = F_{\mathit{Cont}_S}$ holds for every S.*

Proof. Given a set 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

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

where $\langle f\rangle$ is a graph of f, i.e., $(x,y) \in \langle f\rangle$ iff $f\,x = y$.

We prove the last proposition by constructing an appropriate monadic relation for $\mathit{Cont}_{\mathit{Tree}}$ and Cont_S and utilizing the purity of F. Fix some q and a. For X, X' and $Q \in \mathrm{Rel}(X, X')$, define a monadic relation $T^{\mathsf{rel}}_{\mathit{conv}_{q,a}}(Q) \in \mathrm{Rel}(\mathit{Cont}_{\mathit{Tree}}X, \mathit{Cont}_S X')$ as an instantiation of $T^{\mathsf{rel}}_{\mathit{Cont}}$ from Definition 1.2.5 with $W = \langle \mathit{conv}_{q,a} \rangle \in \mathrm{Rel}(\mathit{Tree}, S)$. By Lemma 1.2.6, $T^{\mathsf{rel}}_{\mathit{conv}_{q,a}}$ is an acceptable monadic relation. Since F is pure,

$$(F_{\mathit{Cont}_{\mathit{Tree}}}, F_{\mathit{Cont}_S}) \in (\Delta_A \dot{\to} T^{\mathsf{rel}}_{\mathit{conv}_{q,a}}(\Delta_B)) \dot{\to} T^{\mathsf{rel}}_{\mathit{conv}_{q,a}}(\Delta_C)$$

which by definition of $T^{\mathsf{rel}}_{\mathit{Cont}}$ means that for all $k : A \to \mathit{Cont}_{\mathit{Tree}}B$, $k' : A \to \mathit{Cont}_S B$ such that $(k, k') \in \Delta_A \dot{\to} T^{\mathsf{rel}}_{\mathit{conv}_{q,a}}(\Delta_B)$ and for all $h : C \to \mathit{Tree}$, $h' : C \to S$ such that $(h, h') \in \Delta_C \dot{\to} \langle \mathit{conv}_{q,a} \rangle$,

$$(F_{\mathit{Cont}_{\mathit{Tree}}}\, k\, h, F_{\mathit{Cont}_S}\, k'\, h') \in \langle \mathit{conv}_{q,a} \rangle.$$

Thus, to prove the goal it suffices to check that $(\mathsf{Que}, q) \in \Delta_A \dot{\to} T^{\mathsf{rel}}_{\mathit{conv}_{q,a}}(\Delta_B)$ and $(\mathsf{Ans}, a) \in \Delta_C \dot{\to} \langle \mathit{conv}_{q,a} \rangle$. Indeed, take $c \in C$. Then $\mathit{conv}_{q,a}(\mathsf{Ans}\ c) = a\, c$ and the latter holds. Take $a_1 \in A$ and $f : B \to \mathit{Tree}, f' : B' \to S$ such that $(f, f') \in \Delta_B \dot{\to} \langle \mathit{conv}_{q,a} \rangle$. 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 Lemma 1.2.27, we have

$$\mathit{tree2fun}_{\mathit{Cont}_{TC}}\ (F_{\mathit{Cont}_{\mathit{Tree}}}\ \mathsf{Que}\ \mathsf{Ans}) = F_{\mathit{Cont}_{TC}}.$$

Using functions

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

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

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

which translates a functional for *Cont* to a functional for T. We prove

Lemma 1.2.28. *For any pure $F \in \mathrm{Func}$ and $T \in \mathit{Monad}$, $\Phi_T\, F_{\mathit{Cont}_{TC}} = F_T$ holds.*

Proof. Again, the idea is to construct a suitable acceptable monadic relation and to exploit the purity of F. For X, X', $Q \in \mathrm{Rel}(X, X')$, we define $T^{\mathsf{rel}}_\Phi(Q) \in \mathrm{Rel}(\mathit{Cont}_{TC}X, TX')$ as $T^{\mathsf{rel}}_{\mathit{Cont},T}$ (Definition 1.2.7) instantiated with $D = C$ and $W = \Delta_{TC}$. By Lemma 1.2.8, T^{rel}_Φ is an acceptable monadic relation. Since F is pure, we have

$$(F_{\mathit{Cont}_{TC}}, F_T) \in (\Delta_A \dot{\to} T^{\mathsf{rel}}_\Phi(\Delta_B)) \dot{\to} T^{\mathsf{rel}}_\Phi(\Delta_C)$$

which by definition of $T^{\mathsf{rel}}_{\mathit{Cont},T}$ means that for all $k : A \to \mathit{Cont}_{TC}B$, $k' : A \to TB$ such that $(k, k') \in \Delta_A \dot{\to} T^{\mathsf{rel}}_{\Phi}(\Delta_B)$ and for all $h : C \to TC$, $h' : C \to TC$ such that $(h, h') \in \Delta_C \dot{\to} \Delta_{TC}$ (and thus, $h = h'$), $(F_{\mathit{Cont}_{TC}}\, k\, h, \mathsf{bind}_T(F_T\, k')\, h') \in \Delta_{TC}$ holds, i.e.,

$$F_{\mathit{Cont}_{TC}}\, k\, h = \mathsf{bind}_T(F_T\, k')\, h. \tag{1.1}$$

Note that for any $g : A \to TB$,

$$\Phi_T\, F_{\mathit{Cont}_{TC}}\, g = F_{\mathit{Cont}_{TC}}\, (\mathsf{bind}^{B,C}_T \circ g)\ \mathsf{val}^C_T$$

and

$$F_T\, g = \mathsf{bind}^{C,C}_T\, (F_T\, g)\ \mathsf{val}^C_T\,.$$

Thus, using (1.1), it is sufficient to prove $(\mathsf{bind}^{B,C}_T \circ g, g) \in \Delta_A \dot{\to} T^{\mathsf{rel}}_{\Phi}(\Delta_B)$. Indeed, take $a \in A$ and h, h' such that $(h, h') \in \Delta_B \dot{\to} \Delta_{TC}$ (and thus, $h = h'$). Then $(\mathsf{bind}^{B,C}_T \circ g)\, a\, h = \mathsf{bind}^{B,C}_T\, (g\, a)\ h'$ holds trivially. □

Proof (of Theorem 1.2.26). Finally, we derive

$$\begin{aligned} F_T &= \Phi_T F_{\mathit{Cont}_{TC}} && \text{(by Lemma 1.2.28)}\\ &= \Phi_T(\mathit{tree2fun}_{\mathit{Cont}_{TC}}\, (\mathit{fun2tree}\, F)) && \text{(by Lemma 1.2.27)}\\ &= \mathit{tree2fun}_T\, (\mathit{fun2tree}\, F) && \text{(by Lemmas 1.2.23, 1.2.28)} \end{aligned}$$

This proves the theorem. □

The alternative proof As an anonymous reviewer of the paper [BHK13] has pointed out, one can provide an alternative proof on the assumption that a syntactical strategy tree monad belongs to the collection *Monad* for which F is defined.

The *strategy tree monad* T_{Tree} is defined by abstracting $\mathit{Tree}_{A,B,C}$ over the return type C, i.e., the type constructor is given by $T_{\mathit{Tree}}X = \mathit{Tree}\, A\, B\, X$ and monadic operators are

$$\mathsf{val}_{T_{\mathit{Tree}}} = \mathsf{Ans} \quad \text{and} \quad \mathsf{bind}_{T_{\mathit{Tree}}} = \mathtt{subst}$$

where $\mathtt{subst} : \Lambda XY.\, \mathit{Tree}_{A,B,X} \to (X \to \mathit{Tree}_{A,B,Y}) \to \mathit{Tree}_{A,B,Y}$ is a tree substitution function. When applied to t and g, it substitutes all the leaves $\mathsf{Ans}\, x$ in t with trees $g\, x$. `subst` is recursively defined by

$$\begin{aligned} \mathtt{subst}\, (\mathsf{Ans}\ x)\, g &= g\, x\\ \mathtt{subst}\, (\mathsf{Que}\ a\, f)\, g &= \mathsf{Que}\ a\, (\lambda b.\mathtt{subst}\, (f\, b)\, g). \end{aligned}$$

On the assumption $T_{\mathit{Tree}} \in \mathit{Monad}$, $\mathit{fun2tree} : \text{Func} \to \mathit{Tree}$ can be defined as

$$\mathit{fun2tree}\, F = F_{T_{\mathit{Tree}}}\, (\lambda a.\mathsf{Que}\ a\, \mathsf{Ans})$$

Theorem 1.2.29. *For a strategy tree $t \in \mathit{Tree}$,*

$$\mathit{fun2tree}\, (\mathit{tree2fun}\, t) = t.$$

Proof. By induction on t. The case $t = \mathsf{Ans}\ c$ is easy. We have

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

In the case $t = \mathsf{Que}\ a\,f$, we compute

$$\begin{aligned}&\mathit{fun2tree}\,(\mathit{tree2fun}\,(\mathsf{Que}\ a\,f)) = \\ &\quad = \mathit{fun2tree}\,(\Lambda T.\lambda k.\,\mathsf{bind}_T(k\,a)\,(\lambda b.\mathit{tree2fun}_T\,(f\,b)\,k)) \\ &\quad = (\lambda k.\,\mathsf{bind}_{T_{\mathit{Tree}}}(k\,a)\,(\lambda b.\mathit{tree2fun}_{T_{\mathit{Tree}}}\,(f\,b)\,k))\,(\lambda a.\mathsf{Que}\ a\,\mathsf{Ans}) \\ &\quad = \mathsf{bind}_{T_{\mathit{Tree}}}(\mathsf{Que}\ a\,\mathsf{Ans})\,(\lambda b.\mathit{tree2fun}_{T_{\mathit{Tree}}}\,(f\,b)\,(\mathsf{Que}\ a\,\mathsf{Ans})) \\ &\quad = \mathsf{Que}\ a\,(\lambda b.\mathtt{subst}\,(\mathsf{Ans}\,b)\,(\lambda b'.\mathit{tree2fun}_{T_{\mathit{Tree}}}\,(f\,b')\,\mathsf{Que}\ \mathsf{Ans})) \\ &\quad = \mathsf{Que}\ a\,(\lambda b.\mathit{tree2fun}_{T_{\mathit{Tree}}}\,(f\,b)\,\mathsf{Que}\ \mathsf{Ans}) \\ &\quad = \mathsf{Que}\ a\,(\lambda b.\mathit{fun2tree}\,(\mathit{tree2fun}\,(f\,b))) \\ &\quad = \mathsf{Que}\ a\,f.\end{aligned}$$

In the last step, the induction hypothesis and extensionality are used . □

Theorem 1.2.30. *For a pure $F \in \mathrm{Func}$ and a monad $T \in \mathit{Monad}$,*

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

Proof. Take $f : A \to TB$ and show $\mathit{tree2fun}_T\,(F_{T_{\mathit{Tree}}}\,(\lambda a.\mathsf{Que}\ a\,\mathsf{Ans}))\,f = F_T\,f$. For that, we exploit the purity of F with an appropriate acceptable monadic relation. Given X, X', $Q \in \mathrm{Rel}(X, X')$, define $Q^\top \in \mathrm{Rel}(X \to \mathit{Tree}_{A,B,C}, X' \to TC)$ by

$$h\,Q^\top\,h' \quad \text{iff} \quad \forall x, x'.x\,Q\,x' \implies \mathit{tree2fun}_T\,(h\,x)\,f = h'\,x'.$$

Define $Q^{\top\top} \in \mathrm{Rel}(T_{\mathit{Tree}}X, TX')$ by

$$t\,Q^{\top\top}\,t' \quad \text{iff} \quad \forall h, h'.h\,Q^\top\,h' \implies \mathit{tree2fun}_T\,(\mathsf{bind}_{T_{\mathit{Tree}}}\,t\,h)\,f = \mathsf{bind}_T\,t'\,h'.$$

It is not difficult to show that $Q \mapsto Q^{\top\top}$ is an acceptable monadic relation for T_{Tree} and T. The val-case is easy. Let X, X', $Q \in \mathrm{Rel}(X, X')$ and take $(x, x') \in Q$. Take $(h, h') \in Q^\top$. Then the goal $\mathit{tree2fun}_T\,(\mathsf{bind}_{T_{\mathit{Tree}}}(\mathsf{val}_{T_{\mathit{Tree}}}\,x)\,h)\,f = \mathsf{bind}_T(\mathsf{val}_T\,x')\,h'$ simplifies to $\mathit{tree2fun}_T\,(h\,x)\,f = h'\,x'$ which holds by the assumption on h, h'. For the bind-case, let X, X', $Q \in \mathrm{Rel}(X, X')$, Y, Y', $R \in \mathrm{Rel}(Y, Y')$, and take $(t, t') \in Q^{\top\top}$ and $(g, g') \in Q \dot{\to} R^{\top\top}$. We have to prove that for all h, h' such that $h\,R^\top\,h'$

$$\mathit{tree2fun}_T\,(\mathsf{bind}_{T_{\mathit{Tree}}}(\mathsf{bind}_{T_{\mathit{Tree}}}\,t\,g)\,h)\,f = \mathsf{bind}_T(\mathsf{bind}_T\,t'\,g')\,h'.$$

Using properties of monads, we rewrite the last equation to

$$\mathit{tree2fun}_T\,(\mathsf{bind}_{T_{\mathit{Tree}}}\,t\,(\lambda x.\,\mathsf{bind}_{T_{\mathit{Tree}}}(g\,x)\,h))\,f = \mathsf{bind}_T\,t'\,(\lambda x.\,\mathsf{bind}_T(g'\,x)\,h')$$

and apply the assumption $(t, t') \in Q^{\top\top}$. What left to show is

$$(\lambda x.\, \mathsf{bind}_{T_{\mathit{Tree}}}(g\, x)\, h)\; Q^{\top} (\lambda x.\, \mathsf{bind}_T(g'\, x)\, h').$$

Take $(x, x') \in Q$. Then the required equality

$$\mathit{tree2fun}\,(\mathsf{bind}_{T_{\mathit{Tree}}}(g\, x)\, h)\, f = \mathsf{bind}_T(g'\, x')\, h'$$

follows from the fact $(g\, x, g'\, x') \in R^{\top\top}$. From purity of F, we have $(F_{T_{\mathit{Tree}}}, F_T) \in (\Delta_A \dot{\to} \Delta_B^{\top\top}) \dot{\to} \Delta_C^{\top\top}$. We rewrite the goal as

$$\mathit{tree2fun}_T\,(\mathsf{bind}_{T_{\mathit{Tree}}}(F_{T_{\mathit{Tree}}}\,(\lambda a.\mathsf{Que}\; a\,\mathsf{Ans}))\; \mathsf{val}_{T_{\mathit{Tree}}})\, f = \mathsf{bind}_T(F_T\, f)\; \mathsf{val}_T\,.$$

It is sufficient to show $(\lambda a.\mathsf{Que}\; a\,\mathsf{Ans}, f) \in \Delta_A \dot{\to} \Delta_B^{\top\top}$ and $(\mathsf{val}_{T_{\mathit{Tree}}}, \mathsf{val}_T) \in \Delta_C^{\top}$. The latter is obvious since $\mathit{tree2fun}_T\,(\mathsf{Ans}\; c)\, f = \mathsf{val}_T\, c$, for any $c : C$. Towards the former, take $a : A$ and $(h, h') \in \Delta_B^{\top}$ and show

$$\mathit{tree2fun}_T\,(\mathsf{bind}_{T_{\mathit{Tree}}}(\mathsf{Que}\; a\,\mathsf{Ans})\, h)\, f = \mathsf{bind}_T(f\; a)\, h'$$

which simplifies to $\mathsf{bind}_T(f\; a)\,(\lambda b.\mathit{tree2fun}_T\,(h\, b)\, f) = \mathsf{bind}_T(f\; a)\, h'$. The last equality follows from the assumption on h, h'. □

An example of non-parametric functional Consider the functional $\mathit{Min} : (\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$ that returns a minimal value of its argument function, i.e., $\mathit{Min}\, f = \min\{f\, n \mid n \in \mathbb{N}\}$.

Proposition 1.2.31. *The functional Min* cannot *be represented by means of a strategy tree and thus, cannot be implemented as a monadically parametric function.*

Proof (sketch). To the contrary, suppose there exists a strategy tree $t \in \mathit{Tree}$ such that $\mathit{tree2fun}_{\mathit{Id}}\, t = \mathit{Min}$, and consider the constant function $f\, x = 1$. Obviously, $\mathit{Min}\, f = 1$. Then $\mathit{tree2fun}_{\mathit{Id}}\, t\, f = \mathit{Min}\, f = 1$ which means that when traversing the tree t with f the leaf $\mathsf{Ans}\, 1$ is met. Note that the list of asked questions can be explicitly extracted by means of the instrumented function $f^{\mathsf{instr}} : \mathbb{N} \to \mathit{State}_{\mathbb{N}^*}\mathbb{N}$ defined as $f^{\mathsf{instr}}\; x\; \vec{s} = (f\, x, \vec{s}x)$ which additionally keeps record of nodes visited in t. Let $\vec{q} = \mathit{snd}\,(\mathit{tree2fun}_{\mathit{State}_{\mathbb{N}^*}}\, t\, f^{\mathsf{instr}})$, and define

$$f_{\vec{q}}\, x = \begin{cases} 1 & \text{if } x \in \vec{q} \\ 0 & \text{otherwise.} \end{cases}$$

It is intuitively clear that $\mathit{tree2fun}_{\mathit{IdM}}\, t\, f_{\vec{q}} = 1$ as the computation must go through the very same sequence of questions and thus, meet the very same leaf $\mathsf{Ans}\ 1$ (for the formal proof see the discussion on extraction of a *modulus of continuity* in Section 1.5). This contradicts $\mathit{Min}\, f_{\vec{q}} = 0$. □

Actually, it is not a surprise that *Min* cannot be represented by means of a strategy tree since *Min* is not continuous. Intuitively, *Min* cannot decide what is a minimal value of its functional argument by asking only finitely many questions to it. *Min* might need to look arbitrarily deep to find out a minimum of its argument. This computation cannot be determined by any well-founded strategy. See also the discussion on purity and continuity in Section 1.4.

1.2.5. Second Order: the Partial Case

In this subsection, we provide a characterisation of monadically parametric second-order functionals for partial semantics 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 1.2.4. As in the total case, results of this subsections are valid on the assumption $Cont_R \in Monad$, for all cppos R.

Solution of the domain equation We adapt a formal development of solution of the domain equations from [BKV10] which in turn follows the approach by Freyd [Fre90, Fre91] and Pitts [Pit96].

The idea is to construct a cppo of "strategy trees" as a solution of a recursive domain equation $X \simeq \mathcal{F}(X)$ for an appropriate locally continuous functor $\mathcal{F} : \mathcal{C} \to \mathcal{C}$ in a suitable category $\mathcal{C}$ of cpos. Let $\mathcal{F}(X) = C + A \times (B \to X_\perp)$ be such a functor for the Kleisli category for $T_\perp$ over the category **Cpo** (category of cpos with continuous functions). Let *Tree* be a cpo such that $\mathit{Tree} \simeq \mathcal{F}(\mathit{Tree})$, together with two (continuous) isomorphism functions

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

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 $g : Y \to X_\perp$ such 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

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

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

$$\mathit{fix}\ \delta = \eta \tag{1.2}$$

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 [BKV10].

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 $\varphi \circ F(h) = h \circ \mathsf{fold}$.

We notice that since the Kleisli category for the lift monad is isomorphic to the category $\mathbf{Cppo}_\perp$ (of cppos with strict continuous functions), $\mathit{Tree}_\perp$ can also be considered as a solution of the domain equation

$$X \simeq C_\perp \oplus A_\perp \otimes (B_\perp \multimap X)_\perp$$

in $\mathbf{Cppo}_\bot$, where $\oplus$ and $\otimes$ are the smash sum and the smash product operations respectively.

Definition 1.2.32. We refer to elements of $\mathit{Tree}_\bot$ as *strategy trees*. Define continuous "constructor" functions $\mathsf{Ans} : C \to \mathit{Tree}_\bot$ and $\mathsf{Que} : A \to (B \to \mathit{Tree}_\bot) \to \mathit{Tree}_\bot$ by

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

Definition 1.2.33. We define the function $\mathit{fun2tree} : \mathrm{Func} \to \mathit{Tree}_\bot$ by

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

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

Definition 1.2.34. Given $T \in \mathit{Monad}$, we construct

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

where

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

and the polymorphic version $\mathit{tree2fun}\; t = \Lambda T.\mathit{tree2fun}_T\; t$.

$\mathit{tree2fun}_T$ is correctly defined (since Func_T is pointed) and is continuous and strict for every strict $T \in \mathit{Monad}$.

As in the total case, strategy trees define pure computations.

Lemma 1.2.35. *For any $t \in \mathit{Tree}_\bot$, $\mathit{tree2fun}\; t$ is pure.*

Proof. Fix pointed $T, T' \in \mathit{Monad}$ and an acceptable monadic relation T^{rel} for T, T'. From admissibility of Δ_C and T^{rel} and Lemma 1.2.3, we conclude that the relation $(\Delta_A \dot{\to} T^{\mathsf{rel}}(\Delta_B)) \dot{\to} T^{\mathsf{rel}}(\Delta_C)$ is admissible. Define a relation $P \in \mathrm{Rel}(\mathrm{Func}_T, \mathrm{Func}_{T'})$ by

$$f P f' \quad \text{iff} \quad \forall t.(f\, t, f'\, t) \in (\Delta_A \dot{\to} T^{\mathsf{rel}}(\Delta_B)) \dot{\to} T^{\mathsf{rel}}(\Delta_C).$$

Clearly, P is admissible. Then the required statement is equivalent to $(\mathit{fix}\; G_T)\, P\, (\mathit{fix}\; G_{T'})$ which we prove by the fixpoint induction principle of Lemma 1.1.9. It is sufficient to show

- $\bot\, P\, \bot$ and
- for all g, g', $g\, P\, g'$ implies $(G_T\, g)\, P\, (G_{T'}\, g')$.

The former follows from the strictness of T^{rel}. For the latter, take g, g' such that $g\,P\,g'$ and a strategy tree t. In the case $t = \bot_{\mathit{Tree}_\bot}$,

$$(G_T\,g\,\bot_{\mathit{Tree}_\bot}, G_{T'}\,g'\,\bot_{\mathit{Tree}_\bot}) = (\bot, \bot) \in (\Delta_A \dot{\to} T^{\mathsf{rel}}(\Delta_B)) \dot{\to} T^{\mathsf{rel}}(\Delta_C).$$

In the case $t = \mathsf{Ans}\;c$, we have

$$(G_T\,g\,(\mathsf{Ans}\;c), G_{T'}\,g'\,(\mathsf{Ans}\;c)) = \\ (\lambda h.\,\mathsf{val}_T\,c, \lambda h.\,\mathsf{val}_T\,c) \in (\Delta_A \dot{\to} T^{\mathsf{rel}}(\Delta_B)) \dot{\to} T^{\mathsf{rel}}(\Delta_C)$$

by val-acceptability of T^{rel}. Finally, if $t = \mathsf{Que}\;a\,f$ then

$$(G_T\,g\,(\mathsf{Que}\;a\,f), G_{T'}\,g'\,(\mathsf{Que}\;a\,f)) = \\ (\lambda h.\,\mathsf{bind}_T(h\,a)\,(\lambda b.g\,(f\,b)\,h), \lambda h.\,\mathsf{bind}_{T'}(h\,a)\,(\lambda b.g'\,(f\,b)\,h)).$$

Take $(h, h') \in \Delta_A \dot{\to} T^{\mathsf{rel}}(\Delta_B)$. Using bind-acceptability of T^{rel}, it suffices to check $(\lambda b.g\,(f\,b)\,h, \lambda b.g'\,(f\,b)\,h) \in \Delta_B \dot{\to} T^{\mathsf{rel}}(\Delta_C)$. It follows directly from the assumption $g\,P\,g'$. Lemma proved. Note that the proof is not constructive since the case distinction in use is classical and relies on the excluded middle axiom. □

Representation theorem

Lemma 1.2.36. *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. Let us give a direct formal proof using the minimal invariance property. We verify $(\mathit{tree2fun}_{\mathit{Cont}_{\mathit{Tree}_\bot}}\,t)\,\mathsf{Que}\,\mathsf{Ans} = t$. For that, it is sufficient to show

$$\lambda t.(\mathit{tree2fun}_{\mathit{Cont}_{\mathit{Tree}_\bot}}\,t)\,\mathsf{Que}\;\mathsf{Ans} = \mathit{kleisli}\,\eta_{\mathit{Tree}}$$

or unfolding the definition of $\mathit{tree2fun}$ and using the minimal invariance (1.2),

$$\lambda t.(\mathit{fix}\;G_{\mathit{Cont}_{\mathit{Tree}_\bot}})\,t\;\mathsf{Que}\;\mathsf{Ans} = \mathit{kleisli}\,(\mathit{fix}\;\delta).$$

Define the relation $Q \in \mathrm{Rel}(\mathit{Tree}_\bot \to (A \to TB) \to TC, \mathit{Tree} \to \mathit{Tree}_\bot)$ by

$$f\,Q\,f' \quad \text{iff} \quad \lambda t.f\,t\,\mathsf{Que}\;\mathsf{Ans} = \mathit{kleisli}\,f'.$$

Then the goal can be equivalently expressed as $(\mathit{fix}\;G_{\mathit{Cont}_{\mathit{Tree}_\bot}})\,Q\,(\mathit{fix}\;\delta)$. It is not difficult to see that Q is admissible and $\bot\,Q\,\bot$. Thus, by fixpoint induction, it is sufficient to prove $(G_{\mathit{Cont}_{\mathit{Tree}_\bot}}\,g)\,Q\,(\delta\,g')$, for $g : \mathit{Tree}_\bot \to (A \to TB) \to TC$ and $g' : \mathit{Tree} \to \mathit{Tree}_\bot$ such that $g\,Q\,g'$. Unfolding the definitions of G and δ, the goal is to show for every $t \in \mathit{Tree}_\bot$

$$\mathit{kleisli}\,([\phi_{\mathit{Cont}_{\mathit{Tree}_\bot}}, \psi^g_{\mathit{Cont}_{\mathit{Tree}_\bot}}] \circ \mathsf{unroll})\,t\;\mathsf{Que}\;\mathsf{Ans} = \mathit{kleisli}\,(\mathsf{fold} \circ \mathcal{F}(g') \circ \mathsf{unfold})\,t.$$

The case $t = \bot_{\mathit{Tree}_\bot}$ is trivial. The case $t = \mathsf{Ans}\ c$, for $c \in C$, is easy. After simplifications, we get a trivial equality with $\mathsf{Ans}\ c$ on both sides of it. In the case $t = \mathsf{Que}\ a\, f$, on the left-hand side we compute

$$\begin{aligned}
&\mathit{kleisli}\,([\phi_{\mathit{Cont}_{\mathit{Tree}_\bot}}, \psi^{g}_{\mathit{Cont}_{\mathit{Tree}_\bot}}] \circ \mathsf{unroll})\,(\mathsf{Que}\ a\, f)\ \mathsf{Que}\ \mathsf{Ans} = \\
&\quad = (\lambda h.\, \mathsf{bind}_{\mathit{Cont}_{\mathit{Tree}_\bot}}\,(h\, a)\,(\lambda b.(g \circ f)\, b\, h))\ \mathsf{Que}\ \mathsf{Ans} \\
&\quad = (\mathsf{bind}_{\mathit{Cont}_{\mathit{Tree}_\bot}}\,(\mathsf{Que}\ a)\,(\lambda b.g\,(f\, b)\ \mathsf{Que}))\ \mathsf{Ans} \\
&\quad = \mathsf{Que}\ a\,(\lambda b.g\,(f\, b)\ \mathsf{Que}\ \mathsf{Ans}).
\end{aligned}$$

The right-hand side simplifies as

$$\mathit{kleisli}\,(\mathsf{fold} \circ \mathcal{F}(g') \circ \mathsf{unfold})\,(\mathsf{Que}\ a\, f) = \mathsf{Que}\ a\,((\mathit{kleisli}\, g') \circ f).$$

Thus, it is sufficient to show $\lambda b.g\,(f\, b)\ \mathsf{Que}\ \mathsf{Ans} = (\mathit{kleisli}\, g') \circ f$ which holds by extensionality and the assumption on g, g'. ☐

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

Theorem 1.2.37. *For any pure $F \in \mathrm{Func}$ and $T \in \mathit{Monad}$,*

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

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

Lemma 1.2.38. *Given pure F, $\mathit{tree2fun}_{\mathit{Cont}_S}\,(\mathit{fun2tree}\ F) = F_{\mathit{Cont}_S}$ holds for any cppo S.*

Proof. The proof is similar to the one of Lemma 1.2.27. First, we define a continuous and strict function $\mathit{conv}_{q,a} : \mathit{Tree}_\bot \to S$ for given $q : A \to (B \to S) \to S$ and $a : C \to S$ as

$$\mathit{conv}_{q,a} = \lambda t.\mathit{tree2fun}_{\mathit{Cont}_S}\, t\, q\, a.$$

Then we construct an acceptable, strict and admissible monadic relation $T^{\mathsf{rel}}_{\mathit{conv}_{q,a}}$ for monads $\mathit{Cont}_{\mathit{Tree}_\bot}$ and Cont_S as an instantiation of $T^{\mathsf{rel}}_{\mathit{Cont}}$ with $W = \langle \mathit{conv}_{q,a} \rangle$ and utilize the purity of F. By Lemma 1.2.6, to prove that $T^{\mathsf{rel}}_{\mathit{conv}_{q,a}}$ is strict and acceptable it sufficient to show $(\bot_{\mathit{Tree}_\bot}, \bot_S) \in \langle \mathit{conv}_{q,a} \rangle$ and that $\langle \mathit{conv}_{q,a} \rangle$ is acceptable which both hold. ☐

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 1.2.39. *For any 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 1.2.28. We only have to check that T^{rel}_{Φ} defined as in Lemma 1.2.28 is strict and admissible which is guaranteed by Lemma 1.2.8 since Δ_{TC} is obviously strict and admissible. ☐

The alternative proof Apparently, like in the total case, the alternative proof using tree monads is possible, but we did not formalize it in Coq.

1.2.6. Generalizations

In this subsection, we argue that it is possible to extend the notion of monadic parametricity to arbitrary second-order types. The question of possible generalizations to types of order higher than two is discussed in Conclusion.

Consider a general type n-Func of second-order functionals with n functional arguments

$$\begin{aligned} n\text{-Func} &= \forall T.(A_1 \to TB_1) \to \cdots \to (A_n \to TB_n) \to TC \\ &\simeq \forall T.(A_1 \to TB_1) \times \cdots \times (A_n \to TB_n) \to TC. \end{aligned}$$

Definition 1.2.40. We say that $F \in n$-Func is *monadically parametric* if

$$(F_T, F_{T'}) \in (\Delta_{A_1} \dot{\to} T^{\mathsf{rel}}(\Delta_{B_1})) \dot{\to} \ldots \dot{\to} (\Delta_{A_n} \dot{\to} T^{\mathsf{rel}}(\Delta_{B_n})) \dot{\to} T^{\mathsf{rel}}(\Delta_C)$$

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

Parametricity Theorem guarantees the existence of monadically parametric functionals of type n-Func.

Corollary 1.2.41 (of Theorem 1.2.11). *Every second-order functional $F : n$-Func implemented in $\lambda_\to$ with monadic interpretation is monadically parametric.* □

Definition 1.2.42. In the total case, one defines a set of *strategy trees n-Tree* as the minimal set generated inductively by constructors

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

In the partial case, a cpo of strategy trees is obtained as a solution of the domain equation

$$X \simeq C + B_1 \times (A_1 \to X_\perp) + \cdots + B_n \times (A_n \to X_\perp).$$

Similar to the case of one functional argument, we define functions

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

$\mathit{tree2fun} = \Lambda T.\mathit{tree2fun}_T$ with $\mathit{tree2fun}_T$ defined by structural recursion as

- $\mathit{tree2fun}_T(\mathsf{Ans}\ c) = \lambda k_1 \ldots \lambda k_n.\, \mathsf{val}_T\, c$
- $\mathit{tree2fun}_T(\mathsf{Que}_i\, a\, f) = \lambda k_1 \ldots \lambda k_n.\, \mathsf{bind}_T(k_i\, a)\, (\lambda b.\mathit{tree2fun}_T\, (f\, b)\, k_1 \ldots k_n)$, for $i = 1, \ldots, n$,

and $\mathit{fun2tree}\,F = F_{\mathit{Cont}_{n\text{-}\mathit{Tree}}}\mathsf{Que}_1 \dots \mathsf{Que}_n\,\mathsf{Ans}$. The results of representation theorems 1.2.26 and 1.2.37 are generalized for n-Func.

Theorem 1.2.43. *Given a pure $F \in n\text{-Func}$, $\mathit{tree2fun}_T\,(\mathit{fun2tree}\;F) = F_T$ holds (extensionally) for any $T \in \mathit{Monad}$.*

Proof. In both settings, proofs are similar to corresponding proofs of the aforementioned theorems. The author provides a formal Coq proof of the theorem in the total setting which uses dependent types. □

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

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

is similar, with parameterized strategies of type

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

A parameterized strategy is a forest that for any input value $d : D$ returns a corresponding strategy for computing the functional.

1.3. Monadic Parametricity for State Monads

In the previous section, we demonstrated that one can effectively extract a strategy tree for a parametric functional on one of two conditions, either $\mathit{Cont}_R \in \mathit{Monad}$ for all result sets (cpos, as the case may be) R, or if $T_{\mathit{Tree}} \in \mathit{Monad}$. Perhaps surprisingly, it turns out that parametricity in the state monad *State* alone allows for extraction of such strategies. In this section, we consider the type of functionals

$$\mathrm{Func}_{\mathit{State}} = \textstyle\prod_S.(A \to \mathit{State}_S B) \to \mathit{State}_S C$$

and show that in the total case it is possible to extract a unique strategy tree representation. We show that, as in the general case, there exists a bijection between strategy trees and parametric computations of type $\mathrm{Func}_{\mathit{State}}$. However, there is no such bijection in the partial case. For brevity, we write val_S and bind_S for $\mathsf{val}_{\mathit{State}_S}$ and $\mathsf{bind}_{\mathit{State}_S}$, correspondingly. It is an interesting question, what monads else imply the existence of such strategy representation.

1.3.1. Relational Parametricity

We show that while purity at the first order can be captured by the classical notion of relational parametricity [Rey83], it is too weak to distinguish pure second-order functionals.

The first order First, let us discuss purity at the first-order. Intuitively, a stateful function $f : A \to \mathit{State}_S B$ may be considered as "pure" if there exists a function $g : A \to B$ such that $f = \mathsf{val}_S \circ g$, i.e., f is side-effect free. In this case, purity can be extensionally captured by relational parametricity as shown by the theorem below. First, we introduce the following notation.

Definition 1.3.1. For sets S, S' and X, X' and relations $R \in \mathrm{Rel}(S, S')$ and $Q \in \mathrm{Rel}(X, X')$, we define $T_R^{\mathsf{param}}(Q) \in \mathrm{Rel}(\mathit{State}_S X, \mathit{State}_{S'} X')$ by

$$T_R^{\mathsf{param}}(Q) = R \dot{\to} Q \times R.$$

Theorem 1.3.2. *For $f : \prod_S . A \to \mathit{State}_S B$, the following are equivalent.*

1. *there exists $g : A \to B$ such that $f_S = \mathsf{val}_S \circ g$ for all S.*
2. *for all S, S' and relations $R \in \mathrm{Rel}(S, S')$, $(f_S, f_{S'}) \in \Delta_A \dot{\to} T_R^{\mathsf{param}}(\Delta_B)$.*

Proof. Assume $S \neq \emptyset$ (the case $S = \emptyset$ is obvious). The direction $1 \Rightarrow 2$ is easy. Indeed, for S, S' take $R \in \mathrm{Rel}(S, S)$, $a \in A$ and $s \, R \, s'$. Then $f \, a \, s = (g \, a, s)$ and $f \, a \, s' = (g \, a, s')$, and the assertion follows.

Towards $1 \Rightarrow 2$, given S, pick $s_0 \in S$ and define $g : A \to B$ by $g \, a = \mathit{fst}(f_S \, a \, s_0)$. We claim that $f_{S'} = \mathsf{val}_{S'} \circ g$ for all S'. Indeed, take $a \in A$ and an arbitrary $s' \in S'$ and define $R = \{(s_0, s')\} \in \mathrm{Rel}(S, S')$. If $f_S \, a \, s_0 = (b, s_1)$ and $f_{S'} \, a \, s' = (b', s'_1)$ then since $s_0 \, R \, s'$ the assumption on f yields $b = b'$ and $s'_1 = s'$ (since $s_1 \, R \, s'_1$). Thus, $(\mathsf{val}_{S'} \circ g) \, a \, s' = (g \, a, s') = (\mathit{fst}(f_S \, a \, s_0), s') = f_{S'} \, a \, s'$. □

Since $\prod_S . A_1 \to \cdots \to A_k \to \mathit{State}_S B \simeq \prod_S . A_1 \times \cdots \times A_k \to \mathit{State}_S B$, the theorem is valid for an arbitrary first-order type.

The second order Let us first try to adapt the relational parametricity approach for a characterization of pure second-order computations $F : \mathrm{Func}_{\mathit{State}}$, i.e., those F that produce side-effects only through calls to their functional argument.

Definition 1.3.3. We say that a functional $F : \prod_S . (A \to \mathit{State}_S B) \to \mathit{State}_S C$ is *relationally parametric* if for all S, S' and relations $R \in \mathrm{Rel}(S, S')$

$$(F_S, F_{S'}) \in (\Delta_A \dot{\to} T_R^{\mathsf{param}}(\Delta_B)) \dot{\to} T_R^{\mathsf{param}}(\Delta_C).$$

However, it turns out we can provide an example of a functional parametric in that sense but *not* pure.

Definition 1.3.4. The *snapback* functional

$$F_{\mathrm{snap}} : \prod_S . (A \to \mathit{State}_S B) \to \mathit{State}_S B$$

is defined by $(F_{\mathrm{snap}})_S \, k \, s = (b, s)$ where $k \, a_0 \, s = (b, s_1)$.

It can be understood as a functional that takes a snapshot of the current state of a computational device, invokes its argument function k to compute a result b but discards a new state and instead restores the initial one. Intuitively, F_{snap} is not pure since it does not propagate the state changes incurred by k, but reveals it's own effect of memorization and further recovery of the state.

Proposition 1.3.5. *F_{snap} is relationally parametric.*

Proof. The assertion is easy to show. Take $k : A \to \mathit{State}_S B$ and $k' : A \to \mathit{State}_S B$ such that $(k, k') \in \Delta_A \dot{\to} T_R^{\mathsf{param}}(\Delta_B)$ and $s\,R\,s'$, and let $(b, s_1) = k\,a_0\,s$ and $(b', s_1') = k'\,a_0\,s'$. The assumption on k, k' implies $b = b'$ which together with $s\,R\,s'$ proves $((F_{\text{snap}})_S\,k, (F_{\text{snap}})_{S'}\,k') \in T_R^{\mathsf{param}}(\Delta_B)$. □

Thus, the attempt to employ relational parametricity for $\text{Func}_{\mathit{State}}$ fails. The relational parametricity proved to be not strong enough to exclude the snapback functional from a class of parametric functionals in the sense of Definition 1.3.3. The problem of the approach is that T^{rel} does not impose any relation on input and output states for neither of components S, S' individually. This allows to do the trick of replacing the actual pair of states by an initial one, like in F_{snap}.

1.3.2. Monadic Parametricity

Since the relational parametricity is unsuitable to reason about propagation of side-effects componentwise and thus, to capture the intuitive notion of purity, in our next attempt we proceed with an approach similar to the general case of monadic parametricity. First, we define a notion of an acceptable monadic relation.

Definition 1.3.6. Given sets S, S', for each X, X' and $Q \in \text{Rel}(X, X')$ fix a relation $T^{\mathsf{rel}}(Q) \in \text{Rel}(\mathit{State}_S X, \mathit{State}_{S'} X')$. We say that a mapping $(X, X', Q) \mapsto T^{\mathsf{rel}}(Q)$ is an *acceptable monadic relation (for S, S')* if

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

$$(\mathsf{val}_S, \mathsf{val}_{S'}) \in Q \dot{\to} T^{\mathsf{rel}}(Q);$$

– for all $X, X', Q \in \text{Rel}(X, X')$, $Y, Y', R \in \text{Rel}(Y, Y')$,

$$(\mathsf{bind}_S, \mathsf{bind}_{S'}) \in T^{\mathsf{rel}}(Q) \dot{\to} (Q \dot{\to} T^{\mathsf{rel}}(R)) \dot{\to} T^{\mathsf{rel}}(R).$$

Consider some examples.

Lemma 1.3.7. *For any S, S', $R \in \text{Rel}(S, S')$, $Q \mapsto T_R^{\mathsf{param}}(Q)$ is an acceptable monadic relation.*

Proof. We check the two properties of acceptability. Given $R \in \text{Rel}(S, S')$ and $Q \in \text{Rel}(X, X')$, take $s\,R\,s'$ and $c\,Q\,c'$ then $(\mathsf{val}_S\,c\,s, \mathsf{val}_{S'}\,c'\,s') = ((c, s), (c', s')) \in Q \times R$, and the val-acceptability holds. To prove the bind-case, we take $t\, T_R^{\mathsf{param}}(P)\, t'$ and $f\,(P \dot{\to} T_R^{\mathsf{param}}(Q))\, f'$ with $P \in \mathrm{Rel}(X, X')$, $Q \in \mathrm{Rel}(Y, Y')$ and show

$$(\mathsf{bind}_{\mathit{States}_S}\, t\, f)\, T_R^{\mathsf{param}}(Q)\, (\mathsf{bind}_{\mathit{States}_S}\, t'\, f').$$

Indeed, for $s\,R\,s'$, let $(x, s_1) = t\,s$ and $(x', s_1') = t'\,s'$. Then $x\,P\,x'$ and $s_1\,R\,s_1'$ hold. Therefore, if $(y, s_2) = f\,x\,s_1$ and $(y', s_2') = f\,x'\,s_1'$ then $y\,Q\,y'$ and $s_2\,R\,s_2'$. □

Definition 1.3.8. Given S, S', X, X' and a relation $Q \in \mathrm{Rel}(X, X')$, define $T_0^{\mathsf{rel}}(Q) \in \mathrm{Rel}(\mathit{States}_S X, \mathit{States}_{S'} X')$ by

$$\begin{aligned} f\, T_0^{\mathsf{rel}}(Q)\, f' \equiv\ & \forall s\, s_1\, s'\, s_1'\, x\, x'.(x, s_1) = f\,s \wedge (x', s_1') = f'\,s' \implies \\ & (\exists u'.x\,Q\,u') \wedge (\exists u.u\,Q\,x') \wedge \\ & (\mathit{Inv}\, s_1 \implies x\,Q\,x' \wedge \mathit{Inv}\, s \wedge \mathit{Tran}\,(s, s')\,(s_1, s_1')) \end{aligned}$$

with $\mathit{Tran} \in \mathrm{Rel}(S \times S', S \times S')$ and $\mathit{Inv} \subseteq S$.

Lemma 1.3.9. *For S, S', if $\mathit{Tran} \in \mathrm{Rel}(S \times S', S \times S')$ is reflexive and transitive then $Q \mapsto T_0^{\mathsf{rel}}(Q)$ is an acceptable monadic relation.*

Proof. Let us abbreviate

$$\begin{aligned} \Phi(Q, s, s_1, s', s_1', x, x') \equiv\ & (\exists u'.x\,Q\,u') \wedge (\exists u.u\,Q\,x') \wedge \\ & (\mathit{Inv}\, s_1 \implies x\,Q\,x' \wedge \mathit{Inv}\, s \wedge \mathit{Tran}\,(s, s')\,(s_1, s_1')). \end{aligned}$$

For the val-case, given $c\,Q\,c'$, we have $x = c$, $x' = c'$, $s_1 = s$ and $s_1' = s'$ that yield $x\,Q\,x'$ by the assumption, and the first two conjuncts of Φ follow trivially. The third conjunct follows from reflexivity of *Tran*.

For the bind-case, take $t\,T_0^{\mathsf{rel}}(Q)\,t'$ and $f\,(Q \dot{\to} T_0^{\mathsf{rel}}(R))\,f'$, for some $Q \in \mathrm{Rel}(X, X')$, $R \in \mathrm{Rel}(Y, Y')$, and assume $\Phi(Q, s, s_1, s', s_1', x, x')$ where $(x, s_1) = t\,s$ and $(x', s_1') = t'\,s'$. Assume $(y, s_2) = f\,x\,s_1$ and $(y', s_2') = f'\,x'\,s_1'$. We have to prove $\Phi(R, s, s_2, s', s_2', y, y')$. Take u' such that $x\,Q\,u'$ and let $(z', q') = f'\,u'\,s_1'$. From the assumption on f, f', we get $\Phi(R, s_1, s_2, s_1', q', y, z')$ that yields the existence of v' such that $y\,R\,v'$. Analogously, one shows $\exists v.v\,R\,y'$.

Assume now $\mathit{Inv}\ s_2$. Application of the assumption $f\,(Q \dot{\to} T_0^{\mathsf{rel}}(R))\,f'$ to inferred $x\,Q\,u'$ yields

$$\mathit{Inv}\, s_2 \implies y\,R\,v' \wedge \mathit{Tran}\,(s_1, s_1')\,(s_2, q') \wedge \mathit{Inv}\, s_1$$

and thus, $\mathit{Inv}\ s_1$. Then from $\Phi(Q, s, s_1, s', s_1', x, x')$ we conclude $x\,Q\,x'$, $\mathit{Inv}\ s$ and $\mathit{Tran}\,(s, s')\,(s_1, s_1')$. Using again the assumption on f, f' with $x\,Q\,x'$, we get

$$\mathit{Inv}\, s_2 \implies y\,R\,y' \wedge \mathit{Tran}\,(s_1, s_1')\,(s_2, s_2') \wedge \mathit{Inv}\, s_1$$

from which we easily deduce $y\,R\,y'$ and $\mathit{Tran}\,(s, s')\,(s_2, s_2')$ by transitivity of *Tran*. □

Definition 1.3.10. A functional $F \in \mathrm{Func}_{State}$ is *monadically parametric in state monads* (*pure for state monads*, or simply *pure*) if

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

holds for all S, S' and for all acceptable monadic relations (for *State*) T^{rel} for S, S'.

The latter definition excludes the snapback functional from the class of pure functionals.

Theorem 1.3.11. *Let $F \in \mathrm{Func}_{State}$ be pure for state monads. Let $\mathsf{Test} = \mathtt{bool}$ and define $k_{\mathtt{test}} : A \to State_{\mathsf{Test}} B$ by $k_{\mathtt{test}}\, a\, s = (b_0, \mathsf{true})$. If $F_{\mathsf{Test}}\, k_{\mathtt{test}}\, \mathsf{false} = (c, \mathsf{false})$ then $F_S\, k\, s = (c, s)$, for all S, $s \in S$ and $k : A \to State_S B$.*

Test set is intended to track if the functional argument k of F is called. When we apply the functional F_{Test} to $k_{\mathtt{test}}$, the latter — when called — raises a global boolean flag to true. If the flag remains unset after the evaluation of $F_{\mathsf{Test}}\, k_{\mathtt{test}}$, we can conclude (relying on purity of F) that F does not call its argument and thus, must be constant.

Proof (of Theorem 1.3.11). We define an acceptable monadic relation as T_0^{rel} instantiated with relations $Inv\, s \equiv s = \mathsf{false}$, $Inv \subseteq \mathsf{Test}$, and $Tran\, p\, p_1 \equiv snd\, p = snd\, p_1$, $Tran \in \mathrm{Rel}(\mathsf{Test} \times S, \mathsf{Test} \times S)$. Obviously, $Tran$ is reflexive and transitive. Let us now show that

$$k_{\mathtt{test}}\, (\Delta_A \to T_0^{\mathsf{rel}}(\Delta_B))\, k$$

holds. For $a : A$, $s : S$, let $k\, a\, s = (b_1, s_1)$. Since for all $s' : \mathsf{Test}$, $k_{\mathtt{test}}\, a\, s' = (b_0, \mathsf{true})$, the implication part of $(k_{\mathtt{test}}\, a)\, T_0^{\mathsf{rel}}(\Delta_B)\, (k\, a)$ has the form $\mathsf{true} = \mathsf{false} \Rightarrow \_$ and thus, trivially holds. It is left to show $\exists u'.\mathsf{false} = u'$ and $\exists u.u = b_1$ which are also trivial.

Since F is pure, we obtain $(F_{\mathsf{Test}}\, k_{\mathtt{test}}, F_S\, k) \in T_0^{\mathsf{rel}}(\Delta_C)$, i.e., $\mathsf{false} = \mathsf{false} \Rightarrow b_0 = b_1 \wedge \mathsf{false} = \mathsf{false} \wedge s = s_1$ which proves the required. $\square$

There is no contradiction in that T^{param} is an acceptable monadic relation and the fact that T^{param} is too weak to exclude F_{snap} from the class of pure functionals, since Definition 1.3.10 of purity universally quantifies over *all* acceptable monadic relations while T^{param} is just a particular instance of the class of acceptable monadic relations.

1.3.3. Strategy Trees

In Definition 1.2.19, we introduced the notion of strategy trees that define question-answer dialogues. For a set S, we will denote $tree2fun_S = tree2fun_{State_S}$ and define $tree2fun : Tree \to \mathrm{Func}_{State}$ by

$$tree2fun = \Lambda S.tree2fun_S.$$

In order to extract an element of $Tree$ from $F \in \mathrm{Func}_{State}$, we construct a specific set Test as

$$\mathsf{Test} = (\mathtt{option}\, A) \times A^* \times B^*.$$

We refer to the components of $s : \mathsf{Test} = (x, \vec{a}, \vec{b})$ using a notation for records, by $s.\mathsf{arg} = x$, $s.\mathsf{que} = \vec{a}$, $s.\mathsf{ans} = \vec{b}$. For $s = (x, \vec{a}, \vec{b})$, we write $s[\mathsf{que} := \vec{b}']$ for $(x, \vec{a}, \vec{b}')$, and use similar notation to denote updates of the other components. For $\vec{b} : B^*$, $r_{\vec{b}}$ denotes the state $(\mathsf{None}, \varepsilon, \vec{b})$. We assume that B is not empty, and write b_0 for a default element of B.

The following function serves for extraction of intentional information from a given second-order functionals and helps to construct a strategy tree for it.

Definition 1.3.12. The function $k_{\mathsf{test}} : A \to \mathit{State}_{\mathsf{Test}} B$ is given by

- $k_{\mathsf{test}}\, a\, s = (b_0, s)$, if $s.\mathsf{arg} = \mathsf{Some}\ \_$
- $k_{\mathsf{test}}\, a\, s = (b_0, s[\mathsf{arg} := \mathsf{Some}\ a])$, if $s.\mathsf{arg} = \mathsf{None}$ and $s.\mathsf{ans} = \varepsilon$
- $k_{\mathsf{test}}\, a\, s = (b, s[\mathsf{ans} := \vec{b}, \mathsf{que} := \vec{a}a])$, if $s.\mathsf{arg} = \mathsf{None}$, $s.\mathsf{ans} = b\vec{b}$ and $s.\mathsf{que} = \vec{a}$

Intuitively, k_{test} continues to replay prerecorded answers while arg-flag is not set to $\mathsf{Some}\ \_$ and records asked questions in the que component. As soon as all the answers from ans are consumed, k_{test} stores the next asked question in arg. After that, no more changes to the state are made.

The que-component of Test serves merely for a logging purpose and does not affect the computation. All the obtained questions are appended to an initial list as shown by the following technical lemma.

Lemma 1.3.13. *For any $F : \mathrm{Func}_{\mathit{State}}$ pure in state monads, if $F_{\mathsf{Test}}\, k_{\mathsf{test}}\, s = (c, s_1)$ then for all $\vec{b} : B^*$, $F_{\mathsf{Test}}\, k_{\mathsf{test}}\, (s[\mathsf{que} := \vec{b}s.\mathsf{que}]) = (c, s_1[\mathsf{que} := \vec{b}s_1.\mathsf{que}])$.*

Proof. For $\vec{b} : B^*$, define $R \in \mathrm{Rel}(\mathsf{Test}, \mathsf{Test})$ by $R\, s\, s_1 \equiv s_1 = s[\mathsf{que} := \vec{b}s.\mathsf{que}]$. By Lemma 1.3.7, T_R^{param} is an acceptable monadic relation. Clearly, $k_{\mathsf{test}}\, (\Delta_A \dot{\to} T_R^{\mathsf{param}}(\Delta_B))\, k_{\mathsf{test}}$. Therefore, $(F_{\mathsf{Test}}\, k_{\mathsf{test}}, F_{\mathsf{Test}}\, k_{\mathsf{test}}) \in T_R^{\mathsf{param}}(\Delta_C)$ by purity of F which proves the required. $\square$

Definition 1.3.14. For sets S, S', let $\mathit{Tran} \in \mathrm{Rel}(S, S)$ and $\mathit{Inv} \in \mathrm{Rel}(S, S')$. For $Q \in \mathrm{Rel}(X, X')$, define a relation $T^{\mathsf{rel}}_{\mathit{Tran},\mathit{Inv}}(Q) \in \mathrm{Rel}(\mathit{State}_S X, \mathit{State}_{S'} X')$ by

$$\begin{aligned} f\, T^{\mathsf{rel}}_{\mathit{Tran},\mathit{Inv}}(Q)\, f' \equiv \forall s\, s'\, s_1\, s_1'\, x\, x'.\, f\ s = (x, s_1) \wedge f'\ s' = (x', s_1') \implies \\ (\exists u'.x\, Q\, u') \wedge (\exists u.u\, Q\, x') \wedge \mathit{Tran}\, s\, s_1 \wedge \\ (\mathit{Inv}\, s\, s' \implies x\, Q\, x' \wedge \mathit{Inv}\, s_1\, s_1'). \end{aligned}$$

One can additionally introduce a Tran' transition relation for a primed component if needed. Similar to Lemma 1.3.9, we prove

Lemma 1.3.15. *If the relation Tran is reflexive and transitive then the monadic relation $Q \mapsto T^{\mathsf{rel}}_{\mathit{Tran},\mathit{Inv}}(Q)$ is acceptable.*

Proof. Let us abbreviate $T^{\mathsf{rel}} = T^{\mathsf{rel}}_{\mathit{Tran},\mathit{Inv}}$ and

$$\begin{aligned}\Phi(Q, s, s_1, s', s'_1, x, x') \equiv\ & (\exists u'.x\, Q\, u') \wedge (\exists u.u\, Q\, x') \wedge \mathit{Tran}\, s\, s_1 \wedge \\ & (\mathit{Inv}\, s\, s' \implies x\, Q\, x' \wedge \mathit{Inv}\, s_1\, s'_1).\end{aligned}$$

The val-case easily follows from reflexivity of *Tran*. For the bind-case, take $t\, T^{\mathsf{rel}}(Q)\, t'$ and $f\,(Q \dot{\to} T^{\mathsf{rel}}(R))\, f'$ and assume that $\Phi(Q, s, s_1, s', s'_1, x, x')$ holds where $(x, s_1) = t\, s$ and $(x', s'_1) = t'\, s'$. Assume $(y, s_2) = f\, x\, s_1$ and $(y', s'_2) = f'\, x'\, s'_1$. We have to prove $\Phi(R, s, s_2, s', s'_2, y, y')$. Take u' such that $x\, Q\, u'$ and let $(z', q') = f'\, u'\, s'_1$. From the assumption on f, f', we get $\Phi(R, s_1, s_2, s'_1, q', y, z')$ that yields the existence of v' such that $y\, R\, v'$ and $\mathit{Tran}\, s_1\, s_2$ from which we deduce $\mathit{Tran}\, s\, s_2$ by transitivity. Analogously, one shows $\exists v.v\, R\, y'$.

Assume now $\mathit{Inv}\, s\, s'$. From $\Phi(Q, s, s_1, s', s'_1, x, x')$, we deduce $x\, Q\, x'$ and $\mathit{Inv}\, s_1\, s'_1$. The former when used with $f\,(Q \dot{\to} T^{\mathsf{rel}}(R))\, f'$ yields $\mathit{Inv}\, s_1\, s'_1 \Rightarrow y\, R\, y' \wedge \mathit{Inv}\, s_2\, s'_2$. The rest is easy. □

The next three lemmas show that F_{Test} indeed cannot modify the state by its own, but all the effects come from the argument function k_{test} which for each asked question consumes at most one prerecorded answer. Moreover, if in the final state s the flag $s.\mathsf{arg}$ is raised, i.e., $s.\mathsf{arg} = \mathsf{Some\_}$, then all the initially prerecorded answers must be consumed during the computation and a number of asked questions equals a number of prerecorded answers.

Lemma 1.3.16. *If $F_{\mathit{Test}}\, k_{\mathsf{test}}\, r_{\vec{b}} = (c, r_1)$ and $r_1.\mathit{arg} = \mathit{Some\_}$ then $r_1.\mathit{ans} = \varepsilon$.*

Proof. We instantiate Lemma 1.3.15 with $S = S' = \mathsf{Test}$ and define

$$\begin{aligned}\mathit{Tran}\, r\, r_1 \equiv\ & (r.\mathsf{arg} = \mathsf{None} \wedge r_1.\mathsf{arg} = \mathsf{Some\_} \implies r_1.\mathsf{ans} = \varepsilon) \wedge \\ & (r.\mathsf{ans} = \varepsilon \implies r_1.\mathsf{ans} = \varepsilon) \\ \mathit{Inv}\, r\, r' \equiv\ & r = r'.\end{aligned}$$

Tran is obviously reflexive. Let us show transitivity. Take r, r_1, r_2 such that $\mathit{Tran}\, r\, r_1$ and $\mathit{Tran}\, r_1\, r_2$. The second conjunct $r.\mathsf{ans} = \varepsilon \Rightarrow r_2.\mathsf{ans} = \varepsilon$ is obvious. For the first conjunct, let $r.\mathsf{arg} = \mathsf{None}$ and $r_2.\mathsf{arg} = \mathsf{Some\_}$. Consider the two cases: $r_1.\mathsf{arg} = \mathsf{None}$ and $r_1.\mathsf{arg} = \mathsf{Some\_}$. In the first case, we have $r_1.\mathsf{arg} = \mathsf{None}$ and $r_2.\mathsf{arg} = \mathsf{Some\_}$ and by $\mathit{Tran}\, r_1\, r_2$ we conclude $r_2.\mathsf{ans} = \varepsilon$. In the second case, we have $r.\mathsf{arg} = \mathsf{None}$ and $r_1.\mathsf{arg} = \mathsf{Some\ \_}$, and thus $r_1.\mathsf{ans} = \varepsilon$ by $\mathit{Tran}\, r\, r_1$, and hence $r_2.\mathsf{ans} = \varepsilon$ by $\mathit{Tran}\, r_1\, r_2$.

The monadic relation $T^{\mathsf{rel}} = T^{\mathsf{rel}}_{\mathit{Tran},\mathit{Inv}}$ is acceptable by Lemma 1.3.15. Moreover, $k_{\mathsf{test}}\,(\Delta_A \dot{\to} T^{\mathsf{rel}}(\Delta_B))\, k_{\mathsf{test}}$ holds. Indeed, for $a \in A$, let $(c, r_1) = k_{\mathsf{test}}\, a\, r$ and $(c', r'_1) = k_{\mathsf{test}}\, a\, r'$. Then $\mathit{Tran}\, r\, r_1$ holds by definition of k_{test}. If $r = r'$ then obviously $c = c'$ and $r_1 = r'_1$. By purity of F, we obtain

$$(F_{\mathsf{Test}}\, k_{\mathsf{test}})\, T^{\mathsf{rel}}(\Delta_C)\, (F_{\mathsf{Test}}\, k_{\mathsf{test}})$$

that yields $r.\mathsf{arg} = \mathsf{None} \wedge r_1.\mathsf{arg} = \mathsf{Some\_} \Rightarrow r_1.\mathsf{ans} = \varepsilon$. The required directly follows. □

The next lemma states that for pure F the number of consumed answers equals the number of questions asked during a computation of $F_{\mathsf{Test}}\, k_{\mathsf{test}}$.

Lemma 1.3.17. *If $F_{\mathsf{Test}}\, k_{\mathsf{test}}\, r_{\vec{b}} = (c, r_1)$ then there exists $\vec{d} \in B^*$ such that $\vec{b} = \vec{d} r_1.\mathsf{ans}$ and $|r_1.\mathsf{que}| = |\vec{d}|$.*

Proof. We instantiate the format of Lemma 1.3.15 with $S = S' = \mathsf{Test}$ and

$$\begin{aligned} \mathit{Tran}\, r\, r_1 &\equiv \exists \vec{a}\, \vec{b}.\, r_1.\mathsf{que} = r.\mathsf{que}\, \vec{a} \wedge r.\mathsf{ans} = \vec{b} r_1.\mathsf{ans} \wedge |\vec{a}| = |\vec{b}| \\ \mathit{Inv}\, r\, r' &\equiv r = r'. \end{aligned}$$

Certainly, the relation *Tran* is reflexive and transitive. The result then follows from purity of F. □

The next lemma states that if the flag $r_1.\mathsf{arg}$ is raised in the final state r_1 of a computation $F_{\mathsf{Test}}\, k_{\mathsf{test}}\, r_{\vec{b}}$ then all the prerecorded answers are worked off.

Lemma 1.3.18. *For any pure $F : \mathrm{Func}_{\mathit{State}}$ and $\vec{b} \in B^*$, if $F_{\mathsf{Test}}\, k_{\mathsf{test}}\, r_{\vec{b}} = (c, r_1)$ and $r_1.\mathsf{arg} = \mathsf{Some\_}$ then $|r_1.\mathsf{que}| = |\vec{b}|$.*

Proof. Lemma 1.3.17 yields existence of $\vec{d} \in B^*$ such that $\vec{b} = \vec{d} r_1.\mathsf{ans}$ and $|r_1.\mathsf{que}| = |\vec{d}|$. Since $r_1.\mathsf{ans} = \varepsilon$ by Lemma 1.3.16, the required follows. □

Figure 1.3 presents a functional implementation of *fun2tree* : $\mathrm{Func}_{\mathit{State}} \to \mathit{Tree}$ in OCaml. Notice that the function is applicable to *any* $F : \mathrm{Func}_{\mathit{State}}$ whether it is pure or not, and extracts a strategy tree t even for non-pure total functionals. However, as we show further, t corresponds to F only on the assumption of purity of F, and this correspondence is one-to-one.

The program for *fun2tree* may in general fail to terminate and produce any strategy. We first show that *if fun2tree* returns t and F is pure then t is a valid representation of F. We argue later in Subsection 1.3.4 that *fun2tree* always terminates on pure inputs. In order to reason about *fun2tree* formally, we formalize a graph *Fun2tree* of the function as a well-founded relation.

Definition 1.3.19. The relation

$$\mathit{Fun2treeAux} \subseteq \mathrm{Func}_{\mathit{State}} \times B^* \times \mathit{Tree}$$

is inductively defined by the following clauses:

– if $F\, k_{\mathsf{test}}\, r_{\vec{b}} = (c, r_1)$ and $r_1.\mathsf{arg} = \mathsf{None}$ then $\mathit{Fun2treeAux}(F, \vec{b}, \mathsf{Ans}\; c)$;

– if $F\, k_{\mathsf{test}}\, r_{\vec{b}} = (c, r_1)$ and $r_1.\mathsf{arg} = \mathsf{Some}\; a$, and let $f : B \to \mathit{Tree}$ be such that $\mathit{Fun2treeAux}(F, \vec{b}b, f\, b)$ holds, for all $b : B$, then $\mathit{Fun2treeAux}(F, \vec{b}, \mathsf{Que}\; a\, f)$.

We define

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

Indeed, one can show by induction that *fun2tree* is a graph of a function, i.e., for all $t_1, t_2 \in \mathit{Tree}$, $\mathit{Fun2tree}(F, t_1)$ and $\mathit{Fun2tree}(F, t_2)$ imply $t_1 = t_2$.

```
type ('a,'b,'c) tree =
  | Ans of 'c
  | Que of 'a * ('b → ('a,'b,'c) tree)

let rec tree2fun = function
  | Ans c → fun k →
      fun s → (c,s)
  | Que (a,f) → fun k →
      fun s → let (b,s1) = k a s in tree2fun (f b) k s1

type ('a,'b,'c) test =
  { arg : 'a option; que : 'a list; ans : 'b list }

let initTest bs = {arg = None; que = []; ans = bs}

let kTest b0 = fun a s →
  match s.arg with
    | Some _ → (b0, s)
    | None →
        match s.ans with
          | [] → (b0, {arg = Some a; que = s.que; ans = s.ans})
          | b :: bs →
              (b, {arg = s.arg; que = s.que @ [a]; ans = bs})

let fun2tree b0 ff =
  let rec fun2tree_aux ff bs =
    let (c,s) = ff (kTest b0) (initTest bs) in
      match s.arg with
        | None → Ans c
        | Some a →
            Que (a, fun b → fun2tree_aux ff (bs @ [b]))
  in
  fun2tree_aux ff []
```

Figure 1.3.: Functional implementation of *tree2fun* and *fun2tree* in OCaml

First, we show the easy part: *fun2tree* is an inverse of *tree2fun*.

Theorem 1.3.20. *For all $t \in$ Tree, Fun2tree$\,(tree2fun\; t, t)$.*

Proof. By induction on t we show $\mathit{Fun2treeAux}\,(\mathit{tree2fun}\; t, \varepsilon, t)$. Let $t = \mathsf{Ans}\; c$. Then $\mathit{tree2fun}\,(\mathsf{Ans}\; c)\; k_{\mathsf{test}}\, r_\varepsilon = (c, r_\varepsilon)$ and $r_\varepsilon.\mathsf{arg} = \mathsf{None}$ hold, and the assertion follows. Let $t = \mathsf{Que}\; a\, f$, and assume the induction hypothesis

$$\mathit{Fun2treeAux}\,(\mathit{tree2fun}\,(f\, b), \varepsilon, f\, b) \tag{IH}$$

holds for all $b : B$. We have $k_{\mathsf{test}}\, a\, r_\varepsilon = (b_0, (\mathsf{Some}\; a, \varepsilon, \varepsilon))$, and let

$$\mathit{tree2fun}_{\mathsf{Test}}\,(\mathsf{Que}\; a\, f)\; k_{\mathsf{test}}\, r_\varepsilon = \mathit{tree2fun}_{\mathsf{Test}}\,(f\, b_0)\; k_{\mathsf{test}}\,(\mathsf{Some}\; a, \varepsilon, \varepsilon) = (c, r_1)$$

for some $c : C$, $r_1 : \mathsf{Test}$. We need to show that $r_1.\mathsf{arg} = \mathsf{Some}\; a$ and that for every $b : B$, $\mathit{Fun2treeAux}\,(\mathit{tree2fun}\,(\mathsf{Que}\; a\, f), b, f\, b)$ holds. The former is a consequence of the following lemma which can be easily verified by induction on t.

Lemma 1.3.21. *If $\mathit{tree2fun}_{\mathsf{Test}}\, t\, k_{\mathsf{test}}\, s = (c, s_1)$ and $s.\mathsf{arg} = \mathsf{Some}\_$ then $s_1 = s$, for all $t :$ Tree, $c : C$, $s, s_1 : \mathsf{Test}$.* □

Towards $\mathit{Fun2treeAux}\,(\mathit{tree2fun}\,(\mathsf{Que}\; a\, f), b, f\, b)$, we prove another lemma.

Lemma 1.3.22. *For all $f : B \to$ Tree, a, b, $\vec{b}$ and t,*

$$\mathit{Fun2treeAux}\,(\mathit{tree2fun}\,(f\, b), \vec{b}, t) \implies \mathit{Fun2treeAux}\,(\mathit{tree2fun}\,(\mathsf{Que}\; a\, f), b\vec{b}, t).$$

Proof. By induction on t.

For the base case $t = \mathsf{Ans}\, c$, assume $\mathit{Fun2treeAux}\,(\mathit{tree2fun}\,(f\, b), \vec{b}, \mathsf{Ans}\, c)$. Then by definition of *Fun2treeAux*, $\mathit{tree2fun}_{\mathsf{Test}}\,(f\, b)\, k_{\mathsf{test}}\, r_{\vec{b}} = (c, s)$ and $s.\mathsf{arg} = \mathsf{None}$. Denote $s_1 = s[\mathsf{que} := a s.\mathsf{que}]$. Since $s_1.\mathsf{arg} = \mathsf{None}$, it is sufficient to prove

$$\mathit{tree2fun}_{\mathsf{Test}}\,(\mathsf{Que}\; a\, f)\; k_{\mathsf{test}}\, r_{b\vec{b}} = (c, s_1)$$

or after simplification, $\mathit{tree2fun}_{\mathsf{Test}}\,(f\, b)\, k_{\mathsf{test}}\,(\mathsf{None}, a, \vec{b}) = (c, s_1)$. The latter follows in turn from Lemma 1.3.13.

In the case $t = \mathsf{Que}\; a\, f$, assume the induction hypothesis

$$\begin{aligned}&\forall a : A, b, b' : B, \vec{b} : B^*, g : B \to \mathit{Tree}.\\ &\qquad \mathit{Fun2treeAux}\,(\mathit{tree2fun}\,(g\, b'), \vec{b}, f\, b) \implies\\ &\qquad\qquad \mathit{Fun2treeAux}\,(\mathit{tree2fun}\,(\mathsf{Que}\; a\, g), b'\vec{b}, f\, b)\end{aligned} \tag{1.3}$$

and assume $\mathit{Fun2treeAux}\,(\mathit{tree2fun}\,(g\, b), \vec{b}, \mathsf{Que}\; a\, f)$. The latter implies by definition of *Fun2treeAux* that

$$\mathit{tree2fun}_{\mathsf{Test}}\,(g\, b)\, k_{\mathsf{test}}\, r_{\vec{b}} = (c, s) \tag{1.4}$$

for some c and s such that $s.\mathsf{arg} = \mathsf{Some}\; a$ and

$$\forall d : B. \mathit{Fun2treeAux}\,(\mathit{tree2fun}\,(g\, b), \vec{b}d, f\, d). \tag{1.5}$$

We need to show $\mathit{Fun2treeAux}\,(\mathit{tree2fun}\,(\mathsf{Que}\; a_1\, g), b\vec{b}, \mathsf{Que}\; a\, f)$, for an arbitrary $a_1 : A$. For that, we define $s_1 = s[\mathsf{que} := a_1 s.\mathsf{que}]$ and prove

1) $\mathit{tree2fun}_{\mathsf{Test}}\,(\mathsf{Que}\ a_1\ g)\ k_{\mathtt{test}}\ r_{b\vec{b}} = (c, s_1)$,

2) $s_1.\mathsf{arg} = \mathsf{Some}\ a$, and

3) $\forall d : B.\ \mathit{Fun2treeAux}\,(\mathit{tree2fun}\,(\mathsf{Que}\ a_1\ g), b\vec{b}d, f\ d)$.

Claim 1) simplifies to $\mathit{tree2fun}_{\mathsf{Test}}(g\,b)\ k_{\mathtt{test}}\,(\mathsf{None}, a_1, \vec{b}) = (c, s_1)$ which in turn follows from (1.4) and Lemma 1.3.13. Since $s_1.\mathsf{arg} = s.\mathsf{arg} = \mathsf{Some}\ a$, 2) holds. Towards 3), fix some $d : B$ and apply the induction hypothesis (1.3). Then it is only left to show $\mathit{Fun2treeAux}\,(\mathit{tree2fun}\,(g\,b), \vec{b}d, f\ d)$ which in turn is implied by (1.5). □

Lemma 1.3.22 together with (IH) finish the proof of Theorem 1.3.20. □

Next we show that whenever *fun2tree* returns a strategy $t \in \mathit{Tree}$ for a pure F, F is determined by t.

Theorem 1.3.23. *Suppose that $F \in \mathrm{Func}_{\mathit{State}}$ is pure for state monads and that $\mathit{Fun2tree}\,(F, t)$ holds for some $t \in \mathit{Tree}$. Then $F = \mathit{tree2fun}\ t$.*

The assertion follows from a more general statement involving the auxiliary relation *Fun2treeAux* that we prove below. First, let us introduce the following definition.

Definition 1.3.24. Given a set S and a function $k : A \to \mathit{State}_S B$, we define a relation $\mathit{Mat}_S\,k \subseteq A^* \times B^* \times S \times S$ inductively by

– $\mathit{Mat}_S\,k\,(\varepsilon, \varepsilon, s, s)$, for all $s : S$;

– if $\mathit{Mat}_S\,k\,(\vec{a}, \vec{b}, s, \check{s})$ and $k\,a\,\check{s} = (b, s')$ then $\mathit{Mat}_S\,k\,(\vec{a}a, \vec{b}b, s, s')$, for all $\vec{a}$,$\vec{b}$,s,$\check{s}$,s',a,b.

Given $k : A \to \mathit{State}_S B$, $\mathit{Mat}_S\,k$ relates a sequence of questions (asked to k) to a sequence of answers (received from k). Intuitively, $\mathit{Mat}_S\,k\,(\vec{a}, \vec{b}, s, s')$ asserts that $\vec{a}$ and $\vec{b}$ *match* each other relative to k and s, s' taken as initial and final states, correspondingly. Namely, starting from a state s, if we successively apply k to the arguments in $\vec{a}$, threading intermediate states through, we finish in the state s' and $\vec{b}$ contains a list of answers obtained along the way.

Lemma 1.3.25. *The following statements are true.*

1. *For all $\vec{a}, \vec{b}$, and s, s_1, s_2, $\mathit{Mat}_S\,k\,(\vec{a}, \vec{b}, s, s_1)$ implies $|\vec{a}| = |\vec{b}|$.*

2. *$\mathit{Mat}_S\,k$ is injective, i.e., for all $\vec{a}, \vec{b}$, and s, s_1, s_2,*

$$\mathit{Mat}_S\,k\,(\vec{a}, \vec{b}, s, s_1) \wedge \mathit{Mat}_S\,k\,(\vec{a}, \vec{b}, s, s_2) \implies s_1 = s_2.$$

3. *$\mathit{Mat}_S\,k$ is transitive, i.e., for all $\vec{a}_1, \vec{b}_1$, $\vec{a}_2, \vec{b}_2$ and s, s_1, s_2,*

$$\mathit{Mat}_S\,k\,(\vec{a}_1, \vec{b}_1, s, s_1) \wedge \mathit{Mat}_S\,k\,(\vec{a}_2, \vec{b}_2, s_1, s_2) \implies \mathit{Mat}_S\,k\,(\vec{a}_1\vec{a}_2, \vec{b}_1\vec{b}_2, s, s_2).$$

4. *For all $\vec{a}_1, \vec{b}_1$, $\vec{a}_2, \vec{b}_2$ and s, s_2, if $\mathit{Mat}_S\,k\,(\vec{a}_1\vec{a}_2, \vec{b}_1\vec{b}_2, s, s_2)$ and $|\vec{a}_1| = |\vec{b}_1|$ then there exists s_1 such that $\mathit{Mat}_S\,k\,(\vec{a}_1, \vec{b}_1, s, s_1)$.*

5. *For all $\vec{a}_1, \vec{b}_1$, $\vec{a}_2, \vec{b}_2$ and s, s_1, s_2,*

$$\mathit{Mat}_S\, k\, (\vec{a}_1\vec{a}_2, \vec{b}_1\vec{b}_2, s, s_2) \wedge \mathit{Mat}_S\, k\, (\vec{a}_1, \vec{b}_1, s, s_1) \implies \mathit{Mat}_S\, k\, (\vec{a}_2, \vec{b}_2, s_1, s_2).$$

Proof. Claim 1 is easily seen by induction on structure of *Mat*.

Claim 2 is proved by induction on *Mat* in $\mathit{Mat}_S\, k\, (\vec{a}, \vec{b}, s, s_1)$. Consider two cases. In the first case, $\vec{a} = \varepsilon$, $\vec{b} = \varepsilon$, $s = s_1$ and $\mathit{Mat}_S\, k\, (\varepsilon, \varepsilon, s, s)$. Hence $s_1 = s_2 = s$, and the claim is true. For the inductive case, assume $\mathit{Mat}_S\, k\, (\vec{a}, \vec{b}, s, \check{s}_1)$ and $k\, a\, \check{s}_1 = (b, s_1)$ and the induction hypothesis

$$\forall \check{s}_2. \mathit{Mat}_S\, k\, (\vec{a}, \vec{b}, s, \check{s}_2) \implies \check{s}_1 = \check{s}_2. \tag{IH}$$

From $\mathit{Mat}_S\, k\, (\vec{a}a, \vec{b}b, s, s_2)$ we conclude that for some $\check{s}_2$, $\mathit{Mat}_S\, k\, (\vec{a}, \vec{b}, s, \check{s}_2)$ and $k\, a\, \check{s}_2 = (b, s_2)$ hold. From (IH), we derive $\check{s}_1 = \check{s}_2$, and hence $s_1 = s_2$.

Claim 3 is shown similarly by induction on *Mat* in $\mathit{Mat}_S\, k\, (\vec{a}_2, \vec{b}_2, s_1, s_2)$.

Claims 4,5 are proved by tail-induction on $\vec{b}_2$. □

Theorem 1.3.23 is a direct consequence of the following characterisation of *Fun2treeAux*.

Theorem 1.3.26. *Suppose that $F \in \mathrm{Func}_{\mathit{State}}$ is pure for a state monad and that $\mathit{Fun2treeAux}\,(F, \vec{b}, t)$ holds, for given $t \in \mathit{Tree}$. Suppose furthermore that $F_{\mathsf{Test}}\, k_{\mathsf{test}}\, r_{\vec{b}} = (\_, r)$ and $\mathit{Mat}_S\, k\, (r.\mathsf{que}, \vec{b}, s, \check{s})$ hold. If $F_S\, k\, s = (c_1, s_1)$ and $\mathit{tree2fun}\, t\, k\, \check{s} = (c_2, s_2)$ then $c_1 = c_2$ and $s_1 = s_2$.*

The proof of Theorem 1.3.26 is by induction on $t : \mathit{Tree}$. It reduces to the next two lemmas covering the base case and the inductive case.

Lemma 1.3.27 (Base case)**.** *Let $F : \mathrm{Func}_{\mathit{State}}$ be pure. If $F_{\mathsf{Test}}\, k_{\mathsf{test}}\, r_{\vec{b}} = (c, r)$ and $\mathit{Mat}_S\, k\, (r.\mathsf{que}, \vec{b}, s, s_1)$ and $r.\mathsf{arg} = \mathsf{None}$ hold then $F_S\, k\, s = (c, s_1)$.*

The proof uses an acceptable monadic relation in the following general format.

Definition 1.3.28. Let S, S' be sets. Let $\mathit{Tran} \in \mathrm{Rel}(S, S)$, $\mathit{Tran}' \in \mathrm{Rel}(S', S')$, and $\mathit{Re}, \mathit{Gu} \in \mathrm{Rel}(S \times S', S \times S')$, and $Q \in \mathrm{Rel}(X, X')$. Define a relation $T^{\mathsf{rel}}_{\mathit{Tran}, \mathit{Tran}', \mathit{Re}, \mathit{Gu}}(Q) \in \mathrm{Rel}(\mathit{State}_S X, \mathit{State}_{S'} X')$ by

$$\begin{aligned} f\, T^{\mathsf{rel}}_{\mathit{Tran}, \mathit{Tran}', \mathit{Re}, \mathit{Gu}}(Q)\, f' \equiv\ & \forall s\, s'\, s_1\, s_1'\, x\, x'.\, f\, s = (x, s_1) \wedge f'\, s' = (x', s_1') \implies \\ & (\exists u'. x\, Q\, u') \wedge (\exists u. u\, Q\, x') \wedge \mathit{Tran}\, s\, s_1 \wedge \mathit{Tran}'\, s'\, s_1' \wedge \\ & (\mathit{Re}\,(s, s')\,(s_1, s_1') \implies x\, Q\, x' \wedge \mathit{Gu}\,(s, s')\,(s_1, s_1')). \end{aligned}$$

The relation asserts that if changes in states are constrained by the *rely*-condition $\mathit{Re}\,(s, s')\,(s_1, s_1')$ then also the *guarantee*-condition $\mathit{Gu}\,(s, s')\,(s_1, s_1')$ is satisfied. The rely-guarantee method was originally developed in [Jon83] for verification of parallel programs. The essential requirement for rely-guarantee conditions is that for two consecutive constrained transitions, a guarantee-predicate of the first one implies a rely-predicate of the second one. We prove

Lemma 1.3.29. *If Tran, Tran′ and Gu are reflexive and transitive, and*

$$\begin{aligned}\mathit{Re}\,(s,s')\,(s_2,s_2') \wedge \mathit{Tran}\,s\,s_1 \wedge \mathit{Tran}\,s_1\,s_2 \wedge \mathit{Tran}'\,s'\,s_1' \wedge \mathit{Tran}'\,s_1'\,s_2' \implies \\ \mathit{Re}\,(s,s')\,(s_1,s_1') \wedge (\mathit{Gu}\,(s,s')\,(s_1,s_1') \implies \mathit{Re}\,(s_1,s_1')\,(s_2,s_2'))\end{aligned} \tag{1.6}$$

holds then $Q \mapsto T^{\mathsf{rel}}_{\mathit{Tran},\mathit{Tran}',\mathit{Re},\mathit{Gu}}(Q)$ is an acceptable monadic relation.

Proof. Let us abbreviate $T^{\mathsf{rel}} = T^{\mathsf{rel}}_{\mathit{Tran},\mathit{Tran}',\mathit{Re},\mathit{Gu}}$ and

$$\begin{aligned}\Phi(Q,s,s_1,s',s_1',x,x') \equiv\ & (\exists u'.x\,Q\,u') \wedge (\exists u.u\,Q\,x') \wedge \mathit{Tran}\,s\,s_1 \wedge \mathit{Tran}'\,s'\,s_1' \wedge \\ & (\mathit{Re}\,(s,s')\,(s_1,s_1') \implies x\,Q\,x' \wedge \mathit{Gu}\,(s,s')\,(s_1,s_1')).\end{aligned}$$

For the val-case, given $c\,Q\,c'$, we have $x = c$, $x' = c'$, $s_1 = s$ and $s_1' = s'$ that yield $x\,Q\,x'$ by the assumption, and the first two conjuncts of Φ follow trivially. The third and fourth conjuncts follow from reflexivity of *Tran*, and the last one is implied by reflexivity of *Gu*.

For the bind-case, take $t\,T^{\mathsf{rel}}(Q)\,t'$ and $f\,(Q \dot{\to} T^{\mathsf{rel}}(R))\,f'$ with $Q \in \mathrm{Rel}(X,X')$, $R \in \mathrm{Rel}(Y,Y')$, and assume that $\Phi(Q,s,s_1,s',s_1',x,x')$ holds where $(x,s_1) = t\,s$ and $(x',s_1') = t'\,s'$. Assume $(y,s_2) = f\,x\,s_1$ and $(y',s_2') = f'\,x'\,s_1'$. We have to prove $\Phi(R,s,s_2,s',s_2',y,y')$. Take u' such that $x\,Q\,u'$ and let $(z',q') = f'\,u'\,s_1'$. From the assumption on f, f', we obtain $\Phi(R,s_1,s_2,s_1',q',y,z')$ that yields existence of v' such that $y\,R\,v'$. Analogously, one shows $\exists v.v\,R\,y'$. From $\Phi(R,s_1,s_2,s_1',q',y,z')$ we also get $\mathit{Tran}\,s_1\,s_2$ and thus, $\mathit{Tran}\,s\,s_2$ holds by transitivity of *Tran*. Similarly, one shows $\mathit{Tran}'\,s'\,s_2'$.

Towards the implication part, assume $\mathit{Re}\,(s,s')\,(s_2,s_2')$. Since all the premises of (1.6) are true, we obtain

$$\mathit{Re}\,(s,s')\,(s_1,s_1') \tag{1.7}$$

and

$$\mathit{Gu}\,(s,s')\,(s_1,s_1') \implies \mathit{Re}\,(s_1,s_1')\,(s_2,s_2'). \tag{1.8}$$

From (1.7) and $\Phi(Q,s,s_1,s',s_1',x,x')$ we obtain $x\,Q\,x'$ and $\mathit{Gu}\,(s,s')\,(s_1,s_1')$. The latter together with (1.8) yields

$$\mathit{Re}\,(s_1,s_1')\,(s_2,s_2'). \tag{1.9}$$

Again, we use the assumption $f\,(Q \dot{\to} T^{\mathsf{rel}}(R))\,f'$ together with $x\,Q\,x'$ and conclude $\Phi(R,s_1,s_2,s_1',s_2',y,y')$ which applied to (1.9) yields $\mathit{Gu}\,(s_1,s_1')\,(s_2,s_2')$. Therefore, by transitivity of *Gu* we conclude $\mathit{Gu}\,(s,s')\,(s_2,s_2')$. □

Proof of Lemma 1.3.27. We instantiate Lemma 1.3.29 w.r.t. state sets Test and S and define the following relations.

$$\begin{aligned}
\mathit{Tran}\, r\, r_1 &\equiv \exists \vec{a}\, \vec{b}.\ \mathit{TranP}(r, r_1, \vec{a}, \vec{b}) \\
\mathit{TranP}(r, r_1, \vec{a}, \vec{b}) &\equiv r_1.\mathsf{arg} = \mathsf{None} \implies r.\mathsf{arg} = \mathsf{None} \wedge |\vec{a}| = |\vec{b}| \wedge \\
&\qquad r_1.\mathsf{que} = r.\mathsf{que}\, \vec{a} \wedge r.\mathsf{ans} = \vec{b}\, r_1.\mathsf{ans} \\
\mathit{Tran}'\, r\, r_1 &\equiv \mathsf{True} \\
\mathit{Re}\,(r, s)\,(r_1, s_1) &\equiv r_1.\mathsf{arg} = \mathsf{None} \wedge \\
&\qquad (\forall \vec{x}\, \vec{y}.\ \mathit{TranP}(r, r_1, \vec{x}, \vec{y}) \implies \exists \check{s}.\ \mathit{Mat}_S\, k\, (\vec{x}, \vec{y}, s, \check{s})) \\
\mathit{Gu}\,(r, s)\,(r_1, s_1) &\equiv \exists \vec{a}\, \vec{b}.\ \mathit{TranP}(r, r_1, \vec{a}, \vec{b}) \wedge \mathit{Mat}_S\, k\, (\vec{a}, \vec{b}, s, s_1)
\end{aligned}$$

Note that *TranP* is injective in the sense, if $\mathit{TranP}\, r_1\, r_2\, \vec{a}_1\, \vec{b}_1$ and $\mathit{TranP}\, r_1\, r_2\, \vec{a}'_1\, \vec{b}'_1$ then $\vec{a}_1 = \vec{a}'_1$ and $\vec{b}_1 = \vec{b}'_1$.

We first show that the definitions above indeed satisfy the conditions of Lemma 1.3.29. *Tran* is reflexive. For $r : \mathsf{Test}$, take $\vec{a} = \vec{b} = \varepsilon$, and $\mathit{TranP}\, r\, r\, \varepsilon\, \varepsilon$ is clearly true. *Tran* is transitive. Indeed, let $\mathit{Tran}\, r_1\, r_2$ and $\mathit{Tran}\, r_2\, r_3$. Then there exist $\vec{a}_1, \vec{b}_1$ and $\vec{a}_2, \vec{b}_2$ such that $\mathit{TranP}\, r_1\, r_2\, \vec{a}_1\, \vec{b}_1$ and $\mathit{TranP}\, r_2\, r_3\, \vec{a}_2\, \vec{b}_2$. Put $\vec{a} = \vec{a}_1 \vec{a}_2$ and $\vec{b} = \vec{b}_1 \vec{b}_2$ and show $\mathit{TranP}(r_1, r_3, \vec{a}, \vec{b})$. Suppose $r_3.\mathsf{arg} = \mathsf{None}$. Then $r_2.\mathsf{arg} = \mathsf{None}$ and thus, $r_1.\mathsf{arg} = \mathsf{None}$. Also, $|\vec{a}_1 \vec{a}_2| = |\vec{b}_1 \vec{b}_2|$ as well as $r_3.\mathsf{que} = r_2.\mathsf{que}\, \vec{a}_2 = r_1.\mathsf{que}\, \vec{a}_1 \vec{a}_2$, and $r_3.\mathsf{ans} = \vec{b}_2\, r_2.\mathsf{ans} = \vec{b}_1 \vec{b}_2\, r_1.\mathsf{ans}$ hold clearly. Tran' is obviously reflexive and transitive.

Check the condition (1.6). Let $\vec{a}_1, \vec{b}_1$ and $\vec{a}_2, \vec{b}_2$ be such that $\mathit{TranP}\, r\, r_1\, \vec{a}_1\, \vec{b}_1$ and $\mathit{TranP}\, r_1\, r_2\, \vec{a}_2\, \vec{b}_2$, and let $\mathit{Re}\,(r, s)\,(r_2, s_2)$, i.e., $r_2.\mathsf{arg} = \mathsf{None}$ and

$$\forall \vec{x}\, \vec{y}.\ \mathit{TranP}(r, r_2, \vec{x}, \vec{y}) \implies \exists \check{s}.\ \mathit{Mat}_S\, k\, (\vec{x}, \vec{y}, s, \check{s}). \tag{1.10}$$

First, we show $\mathit{Re}\,(r, s)\,(r_1, s_1)$. The bit $r_1.\mathsf{arg} = \mathsf{None}$ directly follows from $r_2.\mathsf{arg} = \mathsf{None}$ and $\mathit{TranP}\, r_1\, r_2\, \vec{a}_2\, \vec{b}_2$. As for the implication part, let $\vec{x}, \vec{y}$ such that $\mathit{TranP}\, r\, r_1\, \vec{x}\, \vec{y}$. Since $\mathit{TranP}\, r\, r_1\, \vec{a}_1\, \vec{b}_1$, we conclude from injectivity of *TranP* that $\vec{x} = \vec{a}_1$ and $\vec{y} = \vec{b}_1$. Thus, it is left to show $\exists \check{s}.\ \mathit{Mat}_S\, k\, (\vec{a}_1, \vec{b}_1, s, \check{s})$. Indeed, by transitivity of *TranP*, we conclude $\mathit{TranP}\, r\, r_2\, (\vec{a}_1 \vec{a}_2)\, (\vec{b}_1 \vec{b}_2)$, hence by (1.10) there exists s' such that $\mathit{Mat}_S\, k\, (\vec{a}_1 \vec{a}_2, \vec{b}_1 \vec{b}_2, s, s')$, and thus by Lemma 1.3.25, claim 4, since $|\vec{a}_1| = |\vec{b}_1|$, we obtain $\exists \check{s}.\ \mathit{Mat}_S\, k\, (\vec{a}_1, \vec{b}_1, s, \check{s})$.

We now show the $\mathit{Gu}\,(r, s)\,(r_1, s_1) \implies \mathit{Re}\,(r_1, s_1)\,(r_2, s_2)$ part of (1.6). Assume $\mathit{Gu}\,(r, s)\,(r_1, s_1)$. Then by definition of *Gu* there exist $\vec{a}, \vec{b}$ such that $\mathit{TranP}(r, r_1, \vec{a}, \vec{b})$ and $\mathit{Mat}_S\, k\, (\vec{a}, \vec{b}, s, s_1)$. By injectivity of *TranP*, $\vec{a} = \vec{a}_1$ and $\vec{b} = \vec{b}_1$ necessarily, hence

$$\mathit{Mat}_S\, k\, (\vec{a}_1, \vec{b}_1, s, s_1). \tag{1.11}$$

Since $r_2.\mathsf{arg} = \mathsf{None}$ holds, it is only left to show

$$\forall \vec{x}\, \vec{y}.\ \mathit{TranP}(r_1, r_2, \vec{x}, \vec{y}) \implies \exists \check{s}.\ \mathit{Mat}_S\, k\, (\vec{x}, \vec{y}, s_1, \check{s}).$$

Take $\vec{x}, \vec{y}$ such that $\mathit{TranP}(r_1, r_2, \vec{x}, \vec{y})$. From injectivity of *TranP*, we conclude $\vec{x} = \vec{a}_2$ and $\vec{y} = \vec{b}_2$. Therefore, the goal is to show $\exists \check{s}.\ \mathit{Mat}_S\, k\, (\vec{a}_2, \vec{b}_2, s_1, \check{s})$. Indeed, by transitivity of *TranP*, $\mathit{TranP}(r, r_2, \vec{a}_1 \vec{a}_2, \vec{b}_1 \vec{b}_2)$, hence by (1.10), there exists s'_2 such that

$\mathit{Mat}_S\, k\,(\vec{a}_1\vec{a}_2, \vec{b}_1\vec{b}_2, s, s_2')$. Put $\check{s} = s_2'$. Then by Lemma 1.3.25, claim 5, we obtain $\mathit{Mat}_S\, k\,(\vec{a}_2, \vec{b}_2, s_1, s_2')$ using (1.11).

We abbreviate $T^{\mathsf{rel}} = T^{\mathsf{rel}}_{\mathit{Tran},\mathit{Tran}',\mathit{Re},\mathit{Gu}}$ which is an acceptable monadic relation by Lemma 1.3.29. Let us now show that $T^{\mathsf{rel}}(\Delta_B)$ relates $k_{\mathsf{test}}\, a$ and $k\, a$, for all $a : A$. Let $(x, r_1) = k_{\mathsf{test}}\, a\, r$ and $(b, s_1) = k\, a\, s$. The bits $\exists u'.x = u'$, $\exists u.u = b$ and $\mathit{Tran}'\, s\, s_1$ are trivial. To show $\mathit{Tran}\, r\, r_1$, i.e., $\exists \vec{a}\, \vec{b}.\ \mathit{TranP}(r, r_1, \vec{a}, \vec{b})$, put $\vec{a} = a$, $\vec{b} = x$. If $r_1.\mathsf{arg} = \mathsf{None}$ then $r.\mathsf{arg} = \mathsf{None}$, $r_1.\mathsf{que} = r.\mathsf{que}\, a$ and $r.\mathsf{ans} = x r_1.\mathsf{ans}$ necessarily, and the goal is proved. Now assume $\mathit{Re}\,(r, s)\,(r_1, s_1)$, thus $r_1.\mathsf{arg} = \mathsf{None}$ and

$$\forall \vec{x}\, \vec{y}.\ \mathit{TranP}(r, r_1, \vec{x}, \vec{y}) \implies \exists \check{s}.\ \mathit{Mat}_S\, k\,(\vec{x}, \vec{y}, s, \check{s}). \tag{1.12}$$

From the fact $\mathit{TranP}(r, r_1, a, x)$ proved above and (1.12), we conclude that there exists $\check{s}$ such that $\mathit{Mat}_S\, k\,(a, x, s, \check{s})$ which is only possible if $\check{s} = s_1$, by definition of Mat. This proves $\mathit{Gu}\,(r, s)\,(r_1, s_1)$.

Now we prove the main statement of the lemma. Since F is pure for state monads and $k_{\mathsf{test}}\,(\Delta_A \dot{\to} T^{\mathsf{rel}}(\Delta_B))\, k$ holds, we have $(F_{\mathsf{Test}}\, k_{\mathsf{test}})\, T^{\mathsf{rel}}(\Delta_C)\,(F_S\, k)$. Let $(c, r_1) = F_{\mathsf{Test}}\, k_{\mathsf{test}}\, r_{\vec{b}}$ and $(c', s_1') = F_S\, k\, s$. There exist $\vec{a}_1, \vec{b}_1$ such that $\mathit{TranP}(r_{\vec{b}}, r_1, \vec{a}_1, \vec{b}_1)$. Since $r_1.\mathsf{arg} = \mathsf{None}$ holds by the assumption of the lemma, we infer $\vec{a}_1 = r_1.\mathsf{que}$ and $\vec{b}_1 = \vec{b}$. Indeed, we compute $r_1.\mathsf{que} = r_{\vec{b}}.\mathsf{que}\, \vec{a}_1 = \varepsilon \vec{a}_1 = \vec{a}_1$, and the former holds. For the latter, we have $\vec{b} = r_{\vec{b}}.\mathsf{ans} = \vec{b}_1 r_1.\mathsf{ans}$. By the assumption of the lemma, $r_1.\mathsf{que}$ and $\vec{b}$ are matched with $\mathit{Mat}_S\, k$ and hence, $|r_1.\mathsf{que}| = |\vec{b}|$ by Lemma 1.3.25, claim 1. Hence $|\vec{a}_1| = |r_1.\mathsf{que}| = |\vec{b}| = |\vec{b}_1| + |r_1.\mathsf{ans}| = |\vec{a}_1| + |r_1.\mathsf{ans}|$, hence $|r_1.\mathsf{ans}| = 0$ and thus, $r_1.\mathsf{ans} = \varepsilon$.

Let us prove $\mathit{Re}\,(r_{\vec{b}}, s)\,(r_1, s_1')$. $r_1.\mathsf{arg} = \mathsf{None}$ holds by the assumption. Let $\vec{a}', \vec{b}'$ be such that $\mathit{TranP}(r_{\vec{b}}, r_1, \vec{a}', \vec{b}')$. By injectivity of TranP, $\vec{a}' = \vec{a}_1$ and $\vec{b}' = \vec{b}_1$ for $\vec{a}_1, \vec{b}_1$ defined in the previous paragraph. Hence, it is sufficient to show $\exists \check{s}.\ \mathit{Mat}_S\, k\,(\vec{b}, r_1.\mathsf{que}, s, \check{s})$ which is true by assumption of the lemma.

From $\mathit{Re}\,(r_{\vec{b}}, s)\,(r_1, s_1')$, we obtain $c = c'$ and $\mathit{Gu}\,(r_{\vec{b}}, s)\,(r_1, s_1')$. From the latter, reasoning in a similar way as above we derive $\mathit{Mat}_S\, k\,(r_1.\mathsf{que}, \vec{b}, s, s_1')$. Finally, using the latter and the assumption $\mathit{Mat}_S\, k\,(r_1.\mathsf{que}, \vec{b}, s, s_1)$ we infer $s_1' = s_1$ by Lemma 1.3.25, claim 2. ☐

Lemma 1.3.30 (Inductive case)**.** *Let $F : \mathrm{Func}_{\mathit{State}}$ be pure. If $F_{\mathsf{Test}}\, k_{\mathsf{test}}\, r_{\vec{b}} = (c, r)$ and $F_{\mathsf{Test}}\, k_{\mathsf{test}}\, r_{\vec{b}b} = (c', r')$ and $r.\mathsf{arg} = \mathsf{Some}\, a$ then $r'.\mathsf{que} = r.\mathsf{que}\, a$.*

Notice that in contrast to the base case, the inductive case no longer involves the state set S and an argument k but operates entirely on the specific state set Test. To prove the lemma, we use an acceptable monadic relation in the following generic format.

Definition 1.3.31. Let S, S' be sets. Let $\mathit{Tran} \in \mathrm{Rel}(S, S)$, $\mathit{Tran}' \in \mathrm{Rel}(S', S')$ and $\mathit{Inv}_1, \mathit{Inv}_2 \in \mathrm{Rel}(S, S')$. For X, X' and $Q \in \mathrm{Rel}(X, X')$, define a monadic relation

$T^{\mathsf{rel}}_{\mathit{Tran},\mathit{Tran}',\mathit{Inv}_1,\mathit{Inv}_2}(Q) \in \mathrm{Rel}(\mathit{State}_S X, \mathit{State}_{S'} X')$ by

$$\begin{aligned} f\, T^{\mathsf{rel}}_{\mathit{Tran},\mathit{Tran}',\mathit{Inv}_1,\mathit{Inv}_2}(Q)\, f' \equiv{} & \forall s\, s'\, s_1\, s_1'\, x\, x'.\, f\, s = (x, s_1) \wedge f'\, s' = (x', s_1') \implies \\ & (\exists u'.\, x\, Q\, u') \wedge (\exists u.\, u\, Q\, x') \wedge \mathit{Tran}\, s\, s_1 \wedge \mathit{Tran}'\, s'\, s_1' \wedge \\ & (\mathit{Inv}_1\, s\, s' \implies \mathit{Inv}_1\, s_1\, s_1' \wedge x\, Q\, x' \vee \mathit{Inv}_2\, s_1\, s_1'). \end{aligned}$$

$T^{\mathsf{rel}}_{\mathit{Tran},\mathit{Tran}',\mathit{Inv}_1,\mathit{Inv}_2}$ generalizes $T^{\mathsf{rel}}_{\mathit{Tran},\mathit{Inv}}$ from Definition 1.3.14, and does not seem to be an instance of any relation defined previously.

Lemma 1.3.32. *If Tran, Tran' are reflexive and transitive and furthermore,*

$$\mathit{Inv}_2\, s\, s' \wedge \mathit{Tran}\, s\, s_1 \wedge \mathit{Tran}'\, s'\, s_1' \implies \mathit{Inv}_2\, s_1\, s_1' \tag{1.13}$$

holds then $Q \mapsto T^{\mathsf{rel}}_{\mathit{Tran},\mathit{Tran}',\mathit{Inv}_1,\mathit{Inv}_2}(Q)$ is an acceptable monadic relation.

Proof. Let us abbreviate $T^{\mathsf{rel}} = T^{\mathsf{rel}}_{\mathit{Tran},\mathit{Tran}',\mathit{Inv}_1,\mathit{Inv}_2}$ and

$$\begin{aligned} \Phi(Q, s, s_1, s', s_1', x, x') \equiv{} & (\exists u'.x\, Q\, u') \wedge (\exists u.u\, Q\, x') \wedge \mathit{Tran}\, s\, s_1 \wedge \mathit{Tran}'\, s'\, s_1' \wedge \\ & (\mathit{Inv}_1\, s\, s' \implies \mathit{Inv}_1\, s_1\, s_1' \wedge x\, Q\, x' \vee \mathit{Inv}_2\, s_1\, s_1'). \end{aligned}$$

For the val-case, given $c\, Q\, c'$, we have $x = c$, $x' = c'$, $s_1 = s$ and $s_1' = s'$ that yield $x\, Q\, x'$ by the assumption, and the first two conjuncts of Φ follow trivially. The third and fourth conjuncts follow from reflexivity of Tran, and the last one is trivial.

For the bind-case, take $t\, T^{\mathsf{rel}}(Q)\, t'$ and $f\, (Q \dot{\to} T^{\mathsf{rel}}(R))\, f'$ for $Q \in \mathrm{Rel}(X, X')$, $R \in \mathrm{Rel}(Y, Y')$, and assume that $\Phi(Q, s, s_1, s', s_1', x, x')$ holds where $(x, s_1) = t\, s$ and $(x', s_1') = t'\, s'$. Assume $(y, s_2) = f\, x\, s_1$ and $(y', s_2') = f'\, x'\, s_1'$. We now prove $\Phi(R, s, s_2, s', s_2', y, y')$. Take u' such that $x\, Q\, u'$ and let $(z', q') = f'\, u'\, s_1'$. From the assumption on f, f', we get $\Phi(R, s_1, s_2, s_1', q', y, z')$ that yields the existence of v' such that $y\, R\, v'$. Analogously, one shows $\exists v.v\, R\, y'$. From $\Phi(R, s_1, s_2, s_1', q', y, z')$, we also obtain $\mathit{Tran}\, s_1\, s_2$, and thus $\mathit{Tran}\, s\, s_2$ by transitivity of Tran. Similarly, one shows $\mathit{Tran}'\, s'\, s_2'$.

Assume $\mathit{Inv}_1\, s\, s'$. Then $\mathit{Inv}_1\, s_1\, s_1' \wedge x\, Q\, x' \vee \mathit{Inv}_2\, s_1\, s_1'$ holds and thus, two cases are possible. In the first case, $\mathit{Inv}_1\, s_1\, s_1'$ and $x\, Q\, x'$, we apply the assumption $f\, (Q \dot{\to} T^{\mathsf{rel}}(R))\, f'$ to $x\, Q\, x'$ and get $\mathit{Inv}_1\, s_1\, s_1' \implies \mathit{Inv}_1\, s_2\, s_2' \wedge y\, R\, y' \vee \mathit{Inv}_2\, s_2\, s_2'$ which proves the goal. In the second case, $\mathit{Inv}_2\, s_1\, s_1'$, from (1.13) we obtain $\mathit{Inv}_2\, s_2\, s_2'$. □

Proof (of Lemma 1.3.30). We instantiate Lemma 1.3.32 with $S = S' = \mathsf{Test}$ and define the following relations.

$$\begin{aligned} \mathit{Tran}\, r\, r_1 \equiv{} & \mathit{Tran}'\, r\, r_1 \equiv \\ & (r_1.\mathsf{arg} = \mathsf{None} \implies r.\mathsf{arg} = \mathsf{None}) \wedge \\ & (r.\mathsf{arg} = \mathsf{Some}\ \_ \implies r = r_1) \wedge \\ & (r.\mathsf{ans} = \varepsilon \implies \\ & \quad r = r_1 \vee r_1.\mathsf{ans} = \varepsilon \wedge r_1.\mathsf{arg} = \mathsf{Some}\ \_ \wedge r_1.\mathsf{que} = r.\mathsf{que}) \\ \mathit{Inv}_1\, r\, r' \equiv{} & r.\mathsf{arg} = \mathsf{None} \wedge r'.\mathsf{arg} = \mathsf{None} \wedge r'.\mathsf{ans} = r.\mathsf{ans}\, b \wedge r'.\mathsf{que} = r.\mathsf{que} \\ \mathit{Inv}_2\, r\, r' \equiv{} & \exists a.\, r.\mathsf{arg} = \mathsf{Some}\ a \wedge r.\mathsf{ans} = \varepsilon \wedge r'.\mathsf{ans} = \varepsilon \wedge r'.\mathsf{que} = r.\mathsf{que}\, a \end{aligned}$$

Note that Inv_1 depends on $b : B$ assumed by the lemma. The defined relations satisfy the conditions of Lemma 1.3.32. It is clear that Tran and Tran' are reflexive and transitive. Let us check the condition (1.13). Assume $\mathit{Tran}\, r\, r_1$, $\mathit{Tran}'\, r'\, r_1'$ and $\mathit{Inv}_2\, r\, r'$ hold. From the latter, there exists a such that $r.\mathsf{arg} = \mathsf{Some}\ a$ and $r.\mathsf{ans} = \varepsilon \wedge r'.\mathsf{ans} = \varepsilon \wedge r'.\mathsf{que} = r.\mathsf{que}\, a$. Since $r.\mathsf{arg} = \mathsf{Some}\ a$, from $\mathit{Tran}\, r\, r_1$ we obtain $r_1 = r$. Therefore, the first two conjuncts of $\mathit{Inv}_2\, r_1\, r_1'$, $r_1.\mathsf{arg} = \mathsf{Some}\ a$ and $r_1.\mathsf{ans} = \varepsilon$ hold. Since $r'.\mathsf{ans} = \varepsilon$, by $\mathit{Tran}'\, r'\, r_1'$ two cases are possible. In the first case, $r' = r_1'$, the bit $r_1'.\mathsf{ans} = \varepsilon$ is true, and $r_1'.\mathsf{que} = r'.\mathsf{que} = r.\mathsf{que}\, a = r_1.\mathsf{que}\, a$ holds. In the second case, we have $r_1'.\mathsf{ans} = \varepsilon$ and $r_1'.\mathsf{que} = r'.\mathsf{que}$. From the latter, we deduce $r_1'.\mathsf{que} = r'.\mathsf{que} = r.\mathsf{que}\, a = r_1.\mathsf{que}\, a$, and (1.13) is proved. By Lemma 1.3.32, the monadic relation $T^{\mathsf{rel}}_{\mathit{Tran},\mathit{Tran}',\mathit{Inv}_1,\mathit{Inv}_2}$ is acceptable. Let us abbreviate it as T^{rel}.

It is straightforward to show $k_{\mathsf{test}}\, (\Delta_A \dot{\to} T^{\mathsf{rel}}(\Delta_B))\, k_{\mathsf{test}}$. For $a \in A$, let $(b, r_1) = k_{\mathsf{test}}\, a\, r$ and $(b', r_1') = k_{\mathsf{test}}\, a\, r'$. The bits $\exists u'.b = u'$, $\exists u.u = b'$ are trivial. Let us show $\mathit{Tran}\, r\, r_1$. Indeed, by definition of k_{test}, $r_1.\mathsf{arg} = \mathsf{None} \Rightarrow r.\mathsf{arg} = \mathsf{None}$ and $r.\mathsf{arg} = \mathsf{Some}\_ \Rightarrow r = r_1$ are true. If $r.\mathsf{ans} = \varepsilon$ then either $r.\mathsf{arg} = \mathsf{Some}\_$ and thus, $r = r_1$, or $r.\mathsf{arg} = \mathsf{None}$, and hence $r_1.\mathsf{arg} = \mathsf{Some}\_$, $r_1.\mathsf{ans} = \varepsilon$ and $r_1.\mathsf{que} = r.\mathsf{que}$ hold. $\mathit{Tran}'\, r'\, r_1'$ can be shown analogously. Let us prove the remaining implication part. Assume $\mathit{Inv}_1\, r\, r'$, hence $r.\mathsf{arg} = \mathsf{None}$ and $r'.\mathsf{arg} = \mathsf{None}$. We consider two possible cases: $r.\mathsf{ans} = \varepsilon$ and $r.\mathsf{ans} = d\vec{d}$, for some d, $\vec{d}$. In the former case, $r'.\mathsf{ans} = b$ and $\mathit{Inv}_2\, r_1\, r_1'$ can be easily seen. In the latter case, we deduce $c = c'$ and $\mathit{Inv}_1\, r_1\, r_1'$.

Since F is pure for state monads and $k_{\mathsf{test}}\, (\Delta_A \dot{\to} T^{\mathsf{rel}}(\Delta_B))\, k_{\mathsf{test}}$ holds, we have $(F_{\mathsf{Test}}\, k_{\mathsf{test}})\, T^{\mathsf{rel}}(\Delta_C)\, (F_{\mathsf{Test}}\, k_{\mathsf{test}})$. Let $(c, r_1) = F_{\mathsf{Test}}\, k_{\mathsf{test}}\, r_{\vec{b}}$ and $(c', r_1') = F_{\mathsf{Test}}\, k_{\mathsf{test}}\, r_{\vec{b}b}$ and $r_1.\mathsf{arg} = \mathsf{Some}\ a$. Since $\mathit{Inv}_1\, r_{\vec{b}}\, r_{\vec{b}b}$ and not $\mathit{Inv}_1\, r_1\, r_1'$, we have $\mathit{Inv}_2\, r_1\, r_1'$, hence $r_1'.\mathsf{que} = r_1.\mathsf{que}\, a$ which proves the lemma. □

Proof of Theorem 1.3.26. By induction on $t \in \mathit{Tree}$.

Consider the case $t = \mathsf{Ans}\ c$. Assume $\mathit{Fun2treeAux}(F, \vec{b}, \mathsf{Ans}\ c)$ holds. Then by definition of $\mathit{Fun2treeAux}$, there exists r such that $(c, r) = F_{\mathsf{Test}}\, k_{\mathsf{test}}\, r_{\vec{b}}$ and $r.\mathsf{arg} = \mathsf{None}$. Assume $\mathit{Mat}_S\, k\, (r.\mathsf{que}, \vec{b}, s, \check{s})$, and let $(c_1, s_1) = F_S\, k\, s$ and $(c_2, s_2) = \mathit{tree2fun}\, (\mathsf{Ans}\ c)\, k\, \check{s}$. From the latter we get $c_2 = c$ and $s_2 = \check{s}$. In view of these equalities, the goal rewrites to $c_1 = c$ and $s_1 = \check{s}$ that directly follow from Lemma 1.3.27 covering the base case.

For the case $t = \mathsf{Que}\ a\, f$, assume the induction hypothesis

$$\begin{aligned}\forall b, \vec{b}, s, \check{s}, s_1, s_2, r, c_1, c_2.\ &\mathit{Fun2treeAux}(F, \vec{b}, f\, b) \wedge \\ &F_{\mathsf{Test}}\, k_{\mathsf{test}}\, r_{\vec{b}} = (\_, r) \wedge \mathit{Mat}_S\, k\, (r.\mathsf{que}, \vec{b}, s, \check{s}) \wedge \\ &F_S\, k\, s = (c_1, s_1) \wedge \mathit{tree2fun}\, (f\, b)\, k\, \check{s} = (c_2, s_2) \implies c_1 = c_2 \wedge s_1 = s_2 \end{aligned} \tag{IH}$$

Assume $\mathit{Fun2treeAux}(F, \vec{b}, \mathsf{Que}\, a\, f)$ holds. By definition of $\mathit{Fun2treeAux}$, there exist c, r such that $(c, r) = F_{\mathsf{Test}}\, k_{\mathsf{test}}\, r_{\vec{b}}$ and $r.\mathsf{arg} = \mathsf{Some}\ a$, and $\mathit{Fun2treeAux}(F, \vec{b}b, f\, b)$ holds, for all $b : B$. Assume $\mathit{Mat}_S\, k\, (r.\mathsf{que}, \vec{b}, s, \check{s})$, and let $(c_1, s_1) = F_S\, k\, s$ and $(c_2, s_2) = \mathit{tree2fun}\, (\mathsf{Que}\, a\, f)\, k\, \check{s}$. From the latter, putting $(b, s_2') = k\, a\, \check{s}$, we get $\mathit{tree2fun}\, (f\, b)\, f\, s_2' = (c_2, s_2)$ and also, by definition of Mat,

$$\mathit{Mat}_S\, k\, (r.\mathsf{que}\, a, \vec{b}b, s, s_2'). \tag{1.14}$$

Our goal is to prove $c_1 = c_2$ and $s_1 = s_2$. For that, we apply (IH) together with (1.14), and the only thing left to show is that for all r' such that $F_{\mathsf{Test}}\, k_{\mathtt{test}}\, r_{\vec{b}b} = (\_, r')$, $r'.\mathsf{que} = r.\mathsf{que}\, a$ holds. The latter directly follows from Lemma 1.3.30. Theorem 1.3.26 proved. □

Thus, we have shown that if *fun2tree* terminates for a given functional $F : \mathrm{Func}_{\mathit{State}}$ and returns $t \in \mathit{Tree}$ then t is a valid strategy for F.

1.3.4. Existence of Strategy Trees

Below we show that for any pure functional $F : \mathrm{Func}_{\mathit{State}}$ there indeed exists a strategy tree t such that $F = \mathit{tree2fun}\ t$. Theorem 1.3.20 implies that such a representation (if it exists) is unique. Indeed, suppose $F = \mathit{tree2fun}\ t_1 = \mathit{tree2fun}\ t_2$. Then by Theorem 1.3.20, $\mathit{Fun2tree}\,(F, t_1)$ and $\mathit{Fun2tree}\,(F, t_2)$ hold, hence $t_1 = t_2$ by functionality of *Fun2tree*.

To prove existence of a tree strategy for any pure $F : \mathrm{Func}_{\mathit{State}}$, we introduce the following definitions, though not formalized formally in Coq.

Definition 1.3.33. For a set X, consider a set X^* of all finite sequences of elements of X. A *tree* over a set X is an arbitrary prefix-closed subset $\mathcal{T} \subseteq X^*$. We write $\vec{x} \preceq \vec{y}$ for $\vec{x}$ is a prefix of $\vec{y}$. We call X^* a *full (infinite) tree* over X. A *branch* (or *infinite branch*) through $\mathcal{T}$ is an infinite sequence $\vec{x} = x_0 x_1 \ldots$ such that for all $n \in \mathbb{N}$, $\vec{x} \restriction_n = x_0 x_1 \ldots x_n \in \mathcal{T}$. We say that $\mathcal{T}$ is a *well-founded* tree if there are no branches through $\mathcal{T}$.

We define a set of *inductive trees* over X as a minimal set $\mathit{Tree}_{\mathrm{ind}}(X)$ such that

- $\{\varepsilon\} \in \mathit{Tree}_{\mathrm{ind}}(X)$;
- for any $f : X \to \mathit{Tree}_{\mathrm{ind}}(X)$ and $x \in X$, $\{\varepsilon\} \cup \bigcup_{x \in X} \{x(f\, x)\} \in \mathit{Tree}_{\mathrm{ind}}(X)$ holds, with xU denoting a set of sequences $\{xu \mid u \in U\}$, for $U \subseteq X^*$.

In other words, there is a one-to-one correspondence between inductive trees from $\mathit{Tree}_{\mathrm{ind}}(X)$ and elements from the inductive set Tree'_X generated by two constructors

- $\mathsf{Leaf} : \mathit{Tree}'_X$ and
- $\mathsf{Node} : (X \to \mathit{Tree}'_X) \to \mathit{Tree}'_X$.

Formally, every $t \in \mathit{Tree}'_X$ represents a tree $\{\vec{x} \mid \mathit{Legal}\, t\, \vec{x}\}$ where the predicate $\mathit{Legal} \subseteq \mathit{Tree}'_X \times X^*$ is defined by

- $\mathit{Legal}\, t\, \varepsilon$, for all $t \in \mathit{Tree}'_X$;
- if $\mathit{Legal}\,(f\, x)\, \vec{x}$ then $\mathit{Legal}\,(\mathsf{Node}\, f)\,(x\vec{x})$, for all $x \in X$.

We give the following characterization of inductive trees.

Lemma 1.3.34. *The tree $\mathcal{T} \subseteq X^*$ is inductive if and only if it is well-founded and*

$$\vec{x}x \in \mathcal{T} \implies \vec{x}x' \in \mathcal{T}, \text{ for all } \vec{x} \in X^*, x, x' \in X. \tag{1.15}$$

Proof. The "if" direction is easy by induction on tree structure. For the "only if", we give a classical proof using the axiom of choice. Let $\mathcal{T}$ be well-founded, and assume for contradiction that $\mathcal{T}$ is not inductive. Then $\mathcal{T} \neq \{\varepsilon\}$, and for all $x \in X$, $x \in \mathcal{T}$. For every $x \in \mathcal{T}$, define $\mathcal{T}_x^0 = \{\vec{x} \mid x\vec{x} \in \mathcal{T}\}$. Certainly, every $\mathcal{T}_x^0$ is a tree. Since $\mathcal{T}$ is not inductive, there exists x_0 such that $\mathcal{T}_{x_0}^0$ is not inductive. Repeating the argument, we construct recursively an infinite sequence $x_0 x_1 \ldots$ which is a branch through $\mathcal{T}$. A contradiction with well-foundedness of $\mathcal{T}$. □

Theorem 1.3.35. *Let $F \in \mathrm{Func}_{State}$ be pure for a state monad. There exists t : Tree such that Fun2tree(F, t).*

Proof. The theorem is formalized in Coq only partially. The complete mathematical pen-and-paper proof follows. Consider the full tree B^*. Assume for contradiction that there exists a branch $\vec{b} = b_0 b_1 b_2 \ldots$ through B^* such that

$$(snd(F_{\mathsf{Test}}\, k_{\mathsf{test}}\, r_{\vec{b}\restriction_n})).\mathsf{arg} = \mathsf{Some\_} \quad \text{for all } n \in \mathbb{N}. \tag{1.16}$$

Let B^∞ be a coinductive type of finite and infinite lists (streams) over B. Now let the set Test^∞ be defined similar to Test except that the ans-component has the type B^∞. Let $\iota : \mathsf{Test} \to \mathsf{Test}^\infty$ be a naturally defined embedding of Test into Test^∞, and let k^∞_{test} be an extension of k_{test} to Test^∞. In the following, we will omit the coercion ι and assume that $\mathsf{Test} \subseteq \mathsf{Test}^\infty$ if it does not lead to a confusion.

The following four lemmas are formalized in Coq.

Lemma 1.3.36. *$F_{\mathsf{Test}^\infty}\, k^\infty_{\mathsf{test}}\, s = F_{\mathsf{Test}}\, k_{\mathsf{test}}\, s$ whenever $s \in \mathsf{Test} \subseteq \mathsf{Test}^\infty$.*

Proof. We prove the claim by a simulation argument using the following monadic relation. Let $R \in \mathrm{Rel}(\mathsf{Test}, \mathsf{Test}^\infty)$ be defined by $R\, r\, r' \equiv \iota\, r = r'$. The monadic relation T_R^{param} is acceptable by Lemma 1.3.7. It can be easily seen that $k_{\mathsf{test}}\, (\Delta_A \dot{\to} T_R^{\mathsf{param}}(\Delta_B))\, k^\infty_{\mathsf{test}}$.

Since F is pure, we conclude $(F_{\mathsf{Test}}\, k_{\mathsf{test}})\, T^{\mathsf{param}}(\Delta_C)\, (F_{\mathsf{Test}^\infty}\, k^\infty_{\mathsf{test}})$. Now take r : Test, and let $F_{\mathsf{Test}}\, k_{\mathsf{test}}\, r = (c, r_1)$ and $F_{\mathsf{Test}^\infty}\, k^\infty_{\mathsf{test}}\, (\iota\, r) = (c', r_1')$. Then $c = c'$ and $\iota\, r_1 = r_1'$. □

Similar to Lemmas 1.3.16, 1.3.17, we prove the following two lemmas. The first lemma states that if k_{test} raises the flag indicating a called argument then all the prerecorded answers must be consumed (even if the list is infinite).

Lemma 1.3.37. *For a pure F and $\vec{b} \in B^\infty$, if $F_{\mathsf{Test}^\infty}\, k^\infty_{\mathsf{test}}\, r_{\vec{b}} = (r_1, c)$ and $r_1.\mathsf{arg} = \mathsf{Some\_}$ then $r_1.\mathsf{ans} = \varepsilon$.* □

The next lemma states that a pure F when applied to k^∞_{test} can consume only finitely many prerecorded answers to produce a result. This amount equals the number of asked questions.

Lemma 1.3.38. *For a pure F and $\vec{b} \in B^\infty$, if $F_{\mathsf{Test}^\infty}\, k^\infty_{\mathsf{test}}\, r_{\vec{b}} = (r_1, c)$ then there exists $\vec{d} : B^*$ such that $\vec{b} = \vec{d}\, r_1.\mathsf{ans}$ and $|r_1.\mathsf{que}| = |\vec{d}|$.* □

We introduce the following notation. For $r, r' \in \mathsf{Test}^\infty$ and $\vec{b} \in B^\infty$, we write $r \sim_{\vec{b}} r'$ iff $r.\mathsf{ans}\,\vec{b} = r'.\mathsf{ans}$ and all the other components are equal. The following lemma holds.

Lemma 1.3.39. *For $\vec{d} : B^*$ and a non-empty stream $\vec{b}^\infty : B^\infty$, if $F_{\mathsf{Test}}\, k_{\mathsf{test}}\, r_{\vec{d}} = (c, r_1)$ and $F_{\mathsf{Test}^\infty}\, k^\infty_{\mathsf{test}}\, r_{\vec{d}\vec{b}} = (c', r'_1)$ then either $r_1.\mathsf{arg} = \mathsf{None}$ and $r_1 \sim_{\vec{b}} r'_1$, or $r_1.\mathsf{arg} = \mathsf{Some\_}$ and $|r_1.\mathsf{que}| < |r'_1.\mathsf{que}|$ hold.*

Proof. Let $\vec{b} : B^\infty$ be an non-empty stream of answer values. Instantiate Lemma 1.3.32 with $S = \mathsf{Test}$, $S' = \mathsf{Test}^\infty$ and define

$$\begin{aligned} \mathit{Tran}\, r\, r_1 &\equiv r.\mathsf{arg} = \mathsf{Some\_} \implies r = r_1 \\ \mathit{Tran}'\, r\, r_1 &\equiv |r.\mathsf{que}| \le |r_1.\mathsf{que}| \\ \mathit{Inv}_1\, r\, r' &\equiv r.\mathsf{arg} = \mathsf{None} \wedge r \sim_{\vec{b}} r' \\ \mathit{Inv}_2\, r\, r' &\equiv r.\mathsf{arg} = \mathsf{Some\_} \wedge r.\mathsf{ans} = \varepsilon \wedge |r.\mathsf{que}| < |r'.\mathsf{que}|. \end{aligned}$$

It is not difficult to see that Tran and Tran' are reflexive and transitive. Moreover, $\mathit{Inv}_2\, r\, r' \wedge \mathit{Tran}\, r\, r_1 \wedge \mathit{Tran}'\, r'\, r'_1 \implies \mathit{Inv}_2\, r_1\, r'_1$ holds. Indeed, $\mathit{Inv}_2\, r\, r'$ yields $r.\mathsf{arg} = \mathsf{Some\_}$, hence $r = r_1$ by $\mathit{Tran}\, r\, r_1$. Thus, $r_1.\mathsf{arg} = \mathsf{Some\_}$ and $r_1.\mathsf{ans} = \varepsilon$ hold. Also, $|r_1.\mathsf{que}| = |r.\mathsf{que}| < |r'.\mathsf{que}| \le |r_1.\mathsf{que}|$ and thus, $|r_1.\mathsf{que}| < |r_1.\mathsf{que}|$ holds. By Lemma 1.3.32, $T^{\mathsf{rel}} = T^{\mathsf{rel}}_{\mathit{Tran}, \mathit{Tran}', \mathit{Inv}_1, \mathit{Inv}_2}$ is an acceptable monadic relation.

We verify $k_{\mathsf{test}}\, (\Delta_A \dot{\to} T^{\mathsf{rel}}(\Delta_B))\, k^\infty_{\mathsf{test}}$. For $a : A$, let $(b_1, r_1) = k_{\mathsf{test}}\, a\, r$ and $(b'_1, r'_1) = k^\infty_{\mathsf{test}}\, a\, r'$. The goal is to show

$$\begin{aligned} &(\exists u'.\, b_1 = u') \wedge (\exists u.\, u = b'_1) \wedge \mathit{Tran}\, r\, r_1 \wedge \mathit{Tran}'\, r'\, r'_1 \wedge \\ &(\mathit{Inv}_1\, r\, r' \implies \mathit{Inv}_1\, r_1\, r'_1 \wedge b_1 = b'_1 \vee \mathit{Inv}_2\, r_1\, r'_1). \end{aligned}$$

The first two conjuncts are trivial. The bit $\mathit{Tran}\, r\, r_1$ is easy, and $\mathit{Tran}'\, r'\, r'_1$ holds since the que-component may only grow when applying k_{test}. Let us show the last conjunct. Assume $\mathit{Inv}_1\, r\, r'$ that yields $r.\mathsf{arg} = \mathsf{None}$ and $r \sim_{\vec{b}} r'$. We consider two cases. The first case is $r.\mathsf{ans} = \varepsilon$. Then $r_1 = r[\mathsf{arg} := \mathsf{Some}\, a]$, and we can show $\mathit{Inv}_2\, r_1\, r'_1$. Indeed, $r_1.\mathsf{arg} = \mathsf{Some\_}$ and $r_1.\mathsf{ans} = \varepsilon$ are obvious. The bit $|r_1.\mathsf{que}| < |r'_1.\mathsf{que}|$ does hold since $|r_1.\mathsf{que}| = |r.\mathsf{que}| = |r'.\mathsf{que}|$ (by $r \sim_{\vec{b}} r'$) and $|r'_1.\mathsf{que}| = |r'.\mathsf{que}| + 1$ (as $r'.\mathsf{ans}$ is non-empty by the assumption on $\vec{b}$). In the second case, $r.\mathsf{ans} = b_1 \vec{v}$ with $\vec{v} = r_1.\mathsf{ans}$, we verify $\mathit{Inv}_1\, r_1\, r'_1 \wedge b_1 = b'_1$. The bit $b_1 = b'_1$ can be derived from $r \sim_{\vec{b}} r'$ and the fact $r'.\mathsf{ans}$ is non-empty. Moreover, from $r \sim_{\vec{b}} r'$ we derive $r'.\mathsf{ans} = b_1 r'_1.\mathsf{ans}$ and thus, $r_1 \sim_{\vec{b}} r'_1$. Obviously, $r_1.\mathsf{arg} = \mathsf{None}$ holds.

Since the monadic relation $T^{\mathsf{rel}} = T^{\mathsf{rel}}_{\mathit{Tran}, \mathit{Tran}', \mathit{Inv}, \mathit{Inv}'}$ is acceptable and F is pure, we obtain $(F_{\mathsf{Test}}\, k_{\mathsf{test}})\, T^{\mathsf{rel}}(\Delta_C)\, (F_{\mathsf{Test}^\infty}\, k^\infty_{\mathsf{test}})$. The latter and $\mathit{Inv}_1\, r_{\vec{d}} r_{\vec{d}\vec{b}}$ which obviously holds, imply the required. □

We proceed with the proof of the theorem. Let $\vec{b} \in B^\infty$ be an infinite list of b_i satisfying (1.16) and let $(c', r_1') = F_{\mathsf{Test}^\infty}\, k^\infty_{\mathsf{test}}\, r_{\vec{b}}$. By Lemma 1.3.38, there exists $\vec{d} : B^*$ such that $\vec{b} = \vec{d}\, r_1'.\mathsf{ans}$ and $|\vec{d}| = |r_1'.\mathsf{que}|$. Suppose $r_1'.\mathsf{arg} = \mathsf{Some}\,\_$. Then by Lemma 1.3.37 we obtain $r_1'.\mathsf{ans} = \varepsilon$ and thus, $\vec{b} = \vec{d}$, a contradiction. Therefore, we conclude $r_1'.\mathsf{arg} = \mathsf{None}$ and $r_1'.\mathsf{ans}$ is infinite (and thus, non-empty). Denote $r_1'.\mathsf{ans}$ by $\vec{v}$.

Let $(c, r_1) = F_{\mathsf{Test}}\, k_{\mathsf{test}}\, r_{\vec{d}}$. As $r_{\vec{d}} \sim_{\vec{v}} r_{\vec{b}} = r_{\vec{d}\vec{v}}$, two cases are possible by Lemma 1.3.39. 1) $r_1.\mathsf{arg} = \mathsf{None}$ and $r_1 \sim_{\vec{v}} r_1'$ hold. It leads to a contradiction with the assumption (1.16) on $\vec{b}$, namely, that $(\mathit{snd}(F_{\mathsf{Test}}\, k_{\mathsf{test}}\, r_{\vec{b}\restriction_{|\vec{d}|}})).\mathsf{arg} = \mathsf{Some}\,\_$. 2) $r_1.\mathsf{arg} = \mathsf{Some}\,\_$ and $|r_1.\mathsf{que}| < |r_1'.\mathsf{que}|$. Then by Lemma 1.3.18, $|r_1.\mathsf{que}| = |\vec{d}|$, and thus $|\vec{d}| < |\vec{d}|$, a contradiction. In both cases, we arrive to a contradiction which implies classically that for every branch $\vec{b} = b_0 b_1 b_2 \ldots$ through B^* there exists n such that $(\mathit{snd}(F_{\mathsf{Test}}\, k_{\mathsf{test}}\, r_{\vec{b}\restriction_n})).\mathsf{arg} = \mathsf{None}$. For a branch $\vec{b}$, find a minimal $n_{\vec{b}} \in \mathbb{N}$ with the latter property. We refer to the finite prefix $\vec{b} \restriction_{n_{\vec{b}}}$ as a *cut of the branch* $\vec{b}$. Lemma 1.3.39 implies that if $\vec{d}$ is a cut of a branch $\vec{b}$ then $\vec{d}$ is a cut of every $\vec{b}'$ such that $\vec{d} \preceq \vec{b}'$. Define

$$\mathcal{T} = \{\vec{x} \mid \vec{x} \preceq \vec{d}, \vec{d} \text{ is a cut of some branch } \vec{b} \text{ in } B^*\}.$$

Clearly, $\mathcal{T}$ is well-founded. Moreover, $\mathcal{T}$ possesses the property (1.15). Indeed, if $\vec{x}b \in \mathcal{T}$ then $\vec{x}$ cannot be a cut and thus, $\vec{x}b' \in \mathcal{T}$, for all $b' \in B$. Hence by Lemma 1.3.34, $\mathcal{T}$ is inductive, and there exists $t_F \in \mathit{Tree}'_X$ such that $\mathcal{T} = \{\vec{b} \mid \mathit{Legal}\, t_F\, \vec{b}\}$. Let us define the function

$$\mathsf{subtree} : \mathit{Tree}'_X \to X^* \to \mathit{Tree}'_X$$

by induction as follows

- $\mathsf{subtree}\, t\, \varepsilon = t$
- $\mathsf{subtree}\, \mathsf{Leaf}\, \vec{x} = \mathsf{Leaf}$
- $\mathsf{subtree}\, (\mathsf{Node}\, f)\, x\vec{x} = \mathsf{subtree}\, (f\, x)\, \vec{x}$.

Cuts in $\mathcal{T}$ correspond to leaves in t_F, as stated by the next lemma.

Lemma 1.3.40. *The sequence $\vec{b}$ is a cut in t_F iff $\mathit{Legal}\, t_F\, \vec{b}$ and $\mathsf{subtree}\, t_F\, \vec{b} = \mathsf{Leaf}$ hold.* □

We define a translation function from Tree'_B to the set of strategy trees $\mathit{Tree}_{A,B,C}$

$$\mathsf{trans}_F : \mathit{Tree}'_B \to B^* \to \mathit{Tree}$$

inductively by

- $\mathsf{trans}_F\, \mathsf{Leaf}\, \vec{b} = \mathsf{Ans}\, c$, for $c = \mathit{fst}(F_{\mathsf{Test}}\, k_{\mathsf{test}}\, r_{\vec{b}})$;
- for $f : B \to \mathit{Tree}'_B$, $\mathsf{trans}_F\, (\mathsf{Node}\, f)\, \vec{b} = \mathsf{Que}\, a\, g$ with $g = \lambda b.\mathsf{trans}_F\, (f\, b)\, (\vec{b}b)$, $g : B \to \mathit{Tree}$, and $a = \begin{cases} a' & \text{if } (\mathit{snd}(F_{\mathsf{Test}}\, k_{\mathsf{test}}\, r_{\vec{b}})).\mathsf{arg} = \mathsf{Some}\, a' \\ a_0 & \text{otherwise} \end{cases}$ for a default element $a_0 \in A$.

We show that $\mathsf{trans}_F\, t_F\, \varepsilon$ is indeed a strategy tree for F. For that, we make use of Theorem 1.3.23 and verify $\mathit{Fun2tree}(F, \mathsf{trans}_F\, t_F\, \varepsilon)$. The latter in turn follows from a more general statement

$$\forall t \forall \vec{b}.\ \mathit{Legal}\, t_F\, \vec{b} \wedge t = \mathsf{subtree}\, t_F\, \vec{b} \implies \mathit{Fun2treeAux}(F, \vec{b}, \mathsf{trans}_F\, t\, \vec{b})$$

which we prove by induction on t. For the base case, $t = \mathsf{Leaf}$, from $\mathsf{Leaf} = \mathsf{subtree}\, t_F\, \vec{b}$ and $\mathit{Legal} t_F\, \vec{b}$ we conclude by Lemma 1.3.40 that $\vec{b}$ is a cut. Therefore, $F_{\mathsf{Test}}\, k_{\mathsf{test}}\, r_{\vec{b}} = (c, s)$, with $s.\mathsf{arg} = \mathsf{None}$, and $\mathsf{trans}_F\, \mathsf{Leaf}\vec{b} = \mathsf{Ans}\, c$ take place. Hence, by definition of *Fun2treeAux*, we conclude $\mathit{Fun2treeAux}(F, \vec{b}, \mathsf{Ans}\, c)$. For the inductive case, assume $t = \mathsf{Node}\, f$ and the induction hypothesis

$$\forall b \forall \vec{b}.\ \mathit{Legal}\, t_F\, \vec{b} \wedge f\, b = \mathsf{subtree}\, t_F\, \vec{b} \implies \mathit{Fun2treeAux}(F, \vec{b}, \mathsf{trans}_F\, (f\, b)\, \vec{b}).$$

Since $t = \mathsf{Node}\, f = \mathsf{subtree}\, t_F\, \vec{b}$, $\vec{b}$ is not a cut. Hence for $(c, s) = F_{\mathsf{Test}}\, k_{\mathsf{test}}\, r_{\vec{b}}$, we conclude $s.\mathsf{arg} = \mathsf{Some}\, a$. By definition, we have $\mathsf{trans}_F\, t\, \vec{b} = \mathsf{Que}\, a\, g$ with $g = \lambda b.\mathsf{trans}_F\, (f\, b)\, (\vec{b}b)$. To prove $\mathit{Fun2treeAux}(F, \vec{b}, \mathsf{Que}\, a\, g)$, by definition of *Fun2treeAux*, it is necessary to verify $\mathit{Fun2treeAux}(F, \vec{b}b, g\, b)$ for any b. We take $b \in B$ and apply the induction hypothesis. All is left to show are $\mathit{Legal}\ t_F\, \vec{b}b$ and $f\, b = \mathsf{subtree}\, t_F\, \vec{b}b$ which both clearly hold, since $\vec{b}$ is not a cut. □

From Theorems 1.3.20, 1.3.23 and 1.3.35 we deduce

Corollary 1.3.41. *There is a one-to-one correspondence between total pure second-order functionals of type* $\mathrm{Func}_{\mathit{State}}$ *and strategy trees.* □

1.3.5. Generalizations

Similar to the general case of parametricity (Section 1.2.6), it is possible to characterize purity for second-order functional of type

$$\begin{aligned} n\text{-}\mathrm{Func}_{\mathit{State}} = & \textstyle\prod_S.(A_1 \to \mathit{State}_S B_1) \to \dots \to (A_n \to \mathit{State}_S B_n) \to \mathit{State}_S C \simeq \\ & \textstyle\prod_S.(\prod_{i\in[1,n]}.A_i \to \mathit{State}_S B_i) \to \mathit{State}_S C \end{aligned}$$

in terms of strategy trees of type $n\text{-}\mathit{Tree}$ generated by constructors

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

The proof uses similar technique as in the case of one functional parameter. In the case on n arguments, since consecutive questions may be asked to any of the given functional arguments, we tweak the specific state set Test as

$$\mathsf{Test} = \mathsf{option}\, (\textstyle\sum_{i\in[1,n]}.A_i) \times (\sum_{i\in[1,n]}.A_i)^* \times (\sum_{i\in[1,n]}.B_i)^*.$$

Let b_i^0 be a default element of B_i, for $i = 1, \dots, n$. The test function

$$k_{\mathsf{test}} : \textstyle\prod_{i\in[1,n]}.A_i \to \mathit{State}_S B_i$$

is defined by

- $k_{\mathsf{test}}\, i\, a\, s = (b_i^0, s)$, if $s.\mathsf{arg} = \mathsf{Some}\ \_$
- $k_{\mathsf{test}}\, i\, a\, s = (b_i^0, s[\mathsf{arg} := \mathsf{Some}\,(i, a)])$, if $s.\mathsf{arg} = \mathsf{None}$ and $s.\mathsf{ans} = \varepsilon$
- $k_{\mathsf{test}}\, i\, a\, s = (b', s[\mathsf{ans} := \vec{b}, \mathsf{que} := \vec{a}a])$, if $s.\mathsf{arg} = \mathsf{None}$, $s.\mathsf{ans} = (b, j)\vec{b}$ and $s.\mathsf{que} = \vec{a}$ where $b' = \begin{cases} b & i = j \\ b_i^0 & \text{otherwise} \end{cases}$ and b_i^0 is a default element of type B_i.

In the third case, we analyse whether the next prerecorded answer in $s.\mathsf{ans}$ is of appropriate type. In the positive case, we return this answer, otherwise the default element is returned.

The relation

$$\mathit{Fun2treeAux} \subseteq \mathrm{Func}_{\mathit{State}} \times B^* \times n\text{-}\mathit{Tree}$$

is inductively defined by the following clauses:

- if $F\, k_{\mathsf{test}}\, r_{\vec{b}} = (c, r_1)$ and $r_1.\mathsf{arg} = \mathsf{None}$ then $\mathit{Fun2treeAux}(F, \vec{b}, \mathsf{Ans}\ c)$;
- if $F\, k_{\mathsf{test}}\, r_{\vec{b}} = (c, r_1)$ and $r_1.\mathsf{arg} = \mathsf{Some}\,(i, a)$, and let $f : B_i \to n\text{-}\mathit{Tree}$ be such that $\mathit{Fun2treeAux}(F, \vec{b}(i, b), f\, b)$ holds, for all $b : B_i$, then $\mathit{Fun2treeAux}(F, \vec{b}, \mathsf{Que}_i\, a\, f)$.

Define $\mathit{Fun2tree}(F, t) \equiv \mathit{Fun2treeAux}(F, \varepsilon, t)$. We can prove the following results.

Theorem 1.3.42. *For all $t \in n\text{-}\mathit{Tree}$, $\mathit{Fun2tree}(\mathit{tree2fun}\, t, t)$.* □

Theorem 1.3.43. *Suppose that $F \in n\text{-}\mathrm{Func}_{\mathit{State}}$ is pure for a state monad and that $\mathit{Fun2tree}(F, t)$ holds for some $t \in n\text{-}\mathit{Tree}$. Then $F = \mathit{tree2fun}\, t$.*

Proof. A proof is analogous to the proof of Theorem 1.3.23. We only mention that relation $\mathit{Mat}_S\, k \subseteq (\sum_{i\in[1,n]}.A_i)^* \times (\sum_{i\in[1,n]}.B_i)^* \times S \times S$ defined similarly as in Definition 1.3.24 must additionally ensure that types of questions and answers match componentwise. □

Theorem 1.3.44. *Let $F \in n\text{-}\mathrm{Func}_{\mathit{State}}$ be pure for a state monad. There exists $t \in n\text{-}\mathit{Tree}$ such that $\mathit{Fun2tree}(F, t)$.* □

1.3.6. The Partial Case

Let us consider the type

$$\mathrm{Func}_{\mathit{State}} = \textstyle\prod_S.(A \to \mathit{State}_S B) \to \mathit{State}_S C$$

where A, B, C are cpos and $\to$ denotes a type of continuous functions. Characterization of second-order partial functions of type $\mathrm{Func}_{\mathit{State}}$ pure for state monads, however, meets certain problems. The first difficulty is in the requirement of admissibility of acceptable monadic relations which is not easy to achieve in general. One possible solution is to assume that cpos A, B, C and state cpos S are *discrete*.

First, we try to prove that the snapback functional $F_{\text{snap}} : \prod_S.(A \to \mathit{State}_S B) \to \mathit{State}_S B$ is not pure. In the partial case, F_{snap} is defined by

$$(F_{snap})_S\, k\, s = \begin{cases} \bot & k\, a_0\, s = \bot \\ (b, s) & k\, a_0\, s = (b, s_1) \end{cases}$$

Let us formulate a

Conjecture 1.3.45. *Let $F \in \mathrm{Func}_{\mathit{State}}$ be pure for a state monad. Let $\mathsf{Test} = \mathsf{bool}$ and define $k_{\mathsf{test}} : A \to \mathit{State}_{\mathsf{Test}} B$ by $k_{\mathsf{test}}\, a\, s = (b_0, \mathsf{true})$. If $F_{\mathsf{Test}}\, k_{\mathsf{test}}\, \mathsf{false} = (c, \mathsf{false})$ then $F_S\, k\, s = (c, s)$, for all S, $s \in S$ and $k : A \to \mathit{State}_S B$.*

Intuitively, the conjecture seems valid. If $F_{\mathsf{Test}}\, k_{\mathsf{test}}\, \mathsf{false} = (c, \mathsf{false})$ then F does *not* query its functional argument, and thus computation $F_S\, k\, s$ must return a constant non-bottom value for any k and s.

Remind a definition of the monadic relation T_0^{rel}. For S, S', X, X', $Q \in \mathrm{Rel}(X, X')$, $T_0^{\mathsf{rel}}(Q) \in \mathrm{Rel}(\mathit{State}_S X, \mathit{State}_{S'} X')$ is defined as

$$\begin{aligned} f\, T_0^{\mathsf{rel}}(Q)\, f' \equiv\ & \forall s\, s_1\, s'\, s_1'\, x\, x'.(x, s_1) = f\, s \wedge (x', s_1') = f'\, s' \implies \\ & (\exists u'.x\, Q\, u') \wedge (\exists u.u\, Q\, x') \wedge \\ & (\mathit{Inv}\, s_1 \implies x\, Q\, x' \wedge \mathit{Inv}\, s \wedge \mathit{Tran}\,(s, s')\,(s_1, s_1')) \end{aligned}$$

with $\mathit{Tran} \in \mathrm{Rel}(S \times S', S \times S')$ and $\mathit{Inv} \subseteq S$. By Lemma 1.3.9, T_0^{rel} is acceptable. Moreover, clearly, it is strict. Generally, we can not prove its admissibility since Q may not respect the orderings on X, X': $x_1\, Q\, x_1'$ and $x_2\, Q\, x_2'$ with $x_1 \sqsubseteq x_2$ does not necessarily imply $x_1' \sqsubseteq x_2'$. However, T_0^{rel} is admissible if we restrict S, S' and X, X' to discrete cpos. Indeed, let S, S' and X, X' be discrete. Then any $Q \in \mathrm{Rel}(X, X')$ is admissible. Given chains $(t_i)_{i\in\mathbb{N}}$ and $(t_i')_{i\in\mathbb{N}}$ such that $\bigsqcup_{i\in\mathbb{N}} t_i = t \in \mathit{State}_S X$ and $\bigsqcup_{i\in\mathbb{N}} t_i' = t' \in \mathit{State}_{S'} X'$ and $t_i\, T_0^{\mathsf{rel}}(Q)\, t_i'$, for all $i \in \mathbb{N}$, assume $t\, s = (x, s_1)$ and $t'\, s' = (x', s_1')$. Then there exists i_0 such that $t_{i_0}\, s = (x, s_1)$ and $t_{i_0}'\, s' = (x', s_1')$. Hence $t\, T_0^{\mathsf{rel}}(Q)\, t'$ by assumption on t_{i_0}, t_{i_0}'.

Next, we try to prove the conjecture similarly to the total case. We instantiate T_0^{rel} with $\mathit{Inv}\, s \equiv s = \mathsf{false}$, $\mathit{Inv} \subseteq \mathsf{Test}$, and $\mathit{Tran}\, p\, p_1 \equiv \mathit{snd}\, p = \mathit{snd}\, p_1$, $\mathit{Tran} \in \mathrm{Rel}(\mathsf{Test} \times S, \mathsf{Test} \times S)$. Moreover, $k_{\mathsf{test}}\, (\Delta_A \dot{\to} T_0^{\mathsf{rel}}(\Delta_B))\, k$ holds. Hence, using purity of F, we obtain $F_{\mathsf{Test}}\, k_{\mathsf{test}}\, T_0^{\mathsf{rel}}(\Delta_C)\, F_S\, k$. However, T_0^{rel} contains no information why $F_S\, k\, s$ must be non-bottom. This proof attempt fails.

Certainly, a *weaker* version of the conjecture is true.

Theorem 1.3.46. *Let $F \in \mathrm{Func}_{\mathit{State}}$ be pure for a state monad. For Test and k_{test} defined as above, if $F_{\mathsf{Test}}\, k_{\mathsf{test}}\, \mathsf{false} = (c, \mathsf{false})$ and $F_S\, k\, s = (c_1, s_1)$ then $c = c_1$ and $s_1 = s$, for all S, $s \in S$ and k.* □

It is interesting if the stronger version is still provable. We leave it as an open question.

Interestingly, there is no bijection between strategy trees and partial pure functions of type Func_{State}. Take $A = B = C = \texttt{unit} = \{\star\}$, and consider the following strategy trees.

$$\begin{aligned}
t_0 &= \bot \\
t_1 &= \mathsf{Que}\ \star\ (\lambda\_.\bot) \\
t_2 &= \mathsf{Que}\ \star\ (\lambda\_.\mathsf{Que}\ \star\ (\lambda\_.\bot)) \\
&\ \ \dots \\
t_\infty &= \mathsf{Que}\ \star\ (\lambda\_.\mathsf{Que}\ \star\ (\lambda\_.\mathsf{Que}\ \star\ (\lambda\_.\dots)))
\end{aligned}$$

t_∞ is an "infinite" tree defining a computation that performs infinitely many queries to its functional argument and thus, fails to produce a result. All the t_i, $i \in \mathbb{N} \cup \{\infty\}$, are undistinguishable using state monads. They correspond to the same non-terminating computation, i.e., $\mathit{tree2fun}_S\, t_i\, k = \bot$, for all i. Hence, we conclude that there is no analogue of Theorem 1.3.20 in the partial case. However, the author believes that some version of Theorem 1.3.23 still holds. It is an open question if it really does.

1.4. Monadic Parametricity and Continuity

For the set of all number sequences $\mathbb{B} = \mathbb{N} \to \mathbb{N}$ (also called a *Baire space*), we say that a functional $F : \mathbb{B} \to \mathbb{N}$ defined on the Baire space is *continuous* at $f : \mathbb{B}$ if there exists $m \in \mathbb{N}$ such that for any $g : \mathbb{B}$, $\overline{f}(m) = \overline{g}(m)$ implies $F\,f = F\,g$, where $\overline{f}(n)$ denotes a tuple $\langle f\,0, \dots, f\,(n-1)\rangle$, that is $F\,f$ depends only on a finite prefix $\overline{f}(m)$. It is easy to see that this definition of continuity coincides with a standard definition if one equips $\mathbb{B}$ with Baire metric $\rho(f,g) = 2^{-\min\{k \mid f\,k \neq g\,k\}}$ and $\mathbb{N}$ with the discrete metric $\rho(x,y) = |x-y|$. Generally, we say that a total $F : (A \to B) \to C$ is *continuous* at $f : A \to B$ if $F\,f$ depends only on finitely many terms of f.

Let us introduce the following notations $[\![\cdot]\!] = \mathit{tree2fun}$ and $[\![\cdot]\!]^* = \mathit{tree2fun}_{Id}$.

Definition 1.4.1. We say that $F : (A \to B) \to C$ is *strongly continuous* if there exists a strategy tree $t \in \mathit{Tree}_{A,B,C}$ such that $F = [\![t]\!]^*$. Clearly, strongly continuous F is continuous at every f.

In [Esc12], Escardó recovers a classical result that any functional of type $\mathbb{B} \to \mathbb{N}$ definable in Gödel's system **T** is strongly continuous. However, our Parametricity Theorem 1.2.11 shows more than that. Namely, from it we conclude that every definable functional $F : (A \to B) \to C$ is strongly continuous and can be lifted to a monadically parametric function $F^\sharp : \prod_T.(A \to TB) \to TC$ such that $F = F^\sharp_{Id}$ for the identity monad *Id*.

In the partial case, we use a standard notion of $\sqcup$-continuity and a domain of strategy trees as a solution of the domain equation $X \simeq C + B \times (A \to X_\bot)$ as described in Subsection 1.2.5. Summarizing all above, we formulate

Theorem 1.4.2. *The class of strongly continuous functionals coincides with a class of functionals for which a parametric monadic lifting is defined, in both total and partial settings.* □

The theorem is valid for a general case of n functional arguments.

We notice that for the partial case, the class of strongly continuous functionals is a strict subclass of continuous functionals as shown by the following counterexample. It makes use of a continuous parallel-or function $por : \mathtt{bool}_\bot^2 \to \mathtt{bool}_\bot$ defined by

$$por\,(x, y) = \begin{cases} \mathsf{true} & x = \mathsf{true} \text{ or } y = \mathsf{true} \\ \mathsf{false} & x = \mathsf{false} \text{ and } y = \mathsf{false} \\ \bot & \text{otherwise.} \end{cases}$$

Consider the functional $F : (\mathtt{bool} \to \mathtt{bool}_\bot) \to (\mathtt{bool} \to \mathtt{bool}_\bot) \to \mathtt{bool}_\bot$ defined by

$$F\,f\,g = por\,(f\,\mathsf{true}, g\,\mathsf{true}).$$

Then F is not strongly continuous. Indeed, if t for F existed, it could be neither $\bot$ nor Ans _ since F is not constant. Neither could it be of the form $t = \mathsf{Que}_1$ _ _ since $[\![t]\!]^*\,\bot\,\eta = \bot$ and $F\,\bot\,\eta = \mathsf{true}$. Analogously, t could not start with Que_2. Of course, it is not a surprise that *por* is not strongly continuous since every strategy tree defines a *sequential* computation while *por* is essentially "parallel".

1.5. Applications

In this section, we discuss possible applications of the notion of monadic parametricity.

1.5.1. Modulus of Continuity

Suppose F is a second-order *strongly continuous* functional of type $(\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$, and be $F^\sharp : \prod_T.(\mathbb{N} \to T\mathbb{N}) \to T\mathbb{N}$ its parametric monadic lifting, i.e., $F = F^\sharp_{Id}$ for the identity monad *Id*. Then for each $f : \mathbb{N} \to \mathbb{N}$ we can effectively extract an upper bound on number of arguments needed for computation of $F\,f$ (*modulus of continuity* of F at f) by means of the functional

$$\mathtt{Mod}_\mathbb{N}\,F\,f = \max\,(snd\,(F^\sharp_{State_{\mathbb{N}^*}}\,(\mathtt{instr}\,f)\,\varepsilon)) + 1$$

where $\mathtt{instr}\,f : \mathbb{N} \to State_{\mathbb{N}^*}\mathbb{N} = \lambda a.\lambda\vec{l}.(f\,a, \vec{l}\,a)$ instruments the argument f by means of recording of a list of visited indices; we assume $\max\,\varepsilon = -1$. Below we prove that $\mathtt{Mod}_\mathbb{N}$ computes what it is supposed to. For that, we consider a general modulus of continuity functional applicable to strongly continuous functionals of type $(A \to B) \to C$,

$$\mathtt{Mod}\,F\,f = snd\,(F^\sharp_{State_{(A\times B)^*}}\,(\mathtt{instr}\,f)\,\varepsilon)$$

with $\mathtt{instr}\,f : A \to State_{(A\times B)^*}B = \lambda a.\lambda\vec{l}.(f\,a, \vec{l}\,(a, f\,a))$.

Definition 1.5.1. For a strategy tree $t \in Tree_{A,B,C}$ and $f : A \to B$, we can extract a list of visited nodes when traversing t using f for selection of branches by means of the function

$$\mathtt{deps} : Tree \to (A \to B) \to (A \times B)^*$$

defined recursively by

– $\mathsf{deps}\,(\mathsf{Ans}\,c)\,f = \varepsilon$

– $\mathsf{deps}\,(\mathsf{Que}\,a\,k)\,f = (a, f\,a)(\mathsf{deps}\,(k\,(f\,a))\,f)$

or, alternatively, by means of $\mathsf{deps}'\,t\,k = \mathit{snd}\,(\llbracket t\rrbracket_{\mathit{State}_{(A\times B)^*}}(\mathsf{instr}\,f)\,\varepsilon)$. By induction on t, we can show the two definitions are equivalent, i.e., $\mathsf{deps}\,t\,k = \mathsf{deps}'\,t\,k$ for all t, k.

The next statement is clear.

Lemma 1.5.2. *If $F : (A \to B) \to C$ is strongly continuous with a strategy $t \in \mathit{Tree}$ such that $F = \llbracket t\rrbracket^*$ then $\mathsf{Mod}\,F\,f = \mathsf{deps}\,t\,f$.* □

Definition 1.5.3. We define the function

$$\mathsf{subtree} : \mathit{Tree} \to (A \times B)^* \to \mathit{Tree}$$

recursively by

– $\mathsf{subtree}\,t\,[\,] = t$

– $\mathsf{subtree}\,(\mathsf{Ans}\,c)\,\_ = \mathsf{Ans}\,c$

– $\mathsf{subtree}\,(\mathsf{Que}\,a\,f)\,((\_, b)\,\vec{r}) = \mathsf{subtree}\,(f\,b)\,\vec{r}$

Lemma 1.5.4. *For all t and f, $\mathsf{subtree}\,t\,(\mathsf{deps}\,t\,f) = \mathsf{Ans}\,(\llbracket t\rrbracket^* f)$.*

Proof. By induction on t. □

Lemma 1.5.5. *For all t, f and g, if $l = \mathsf{deps}\,t\,f$ and $\forall (a, b) \in l.\, f\,a = g\,a$ hold then $\mathsf{deps}\,t\,f = \mathsf{deps}\,t\,g$.*

Proof. By induction on t. The case $t = \mathsf{Ans}\,c$ is trivial. If $t = \mathsf{Que}\,a\,k$, using the induction hypothesis we deduce $\mathsf{deps}\,t\,f = (a, f\,a)\,(\mathsf{deps}\,(k\,(f\,a))\,f) = (a, g\,a)\,(\mathsf{deps}\,(k\,(g\,a))\,g) = \mathsf{deps}\,t\,g$. □

Lemma 1.5.6. *Given t and f, if $l = \mathsf{deps}\,t\,f$ and $\forall (a, b) \in l.\, f\,a = g\,a$ then $\llbracket t\rrbracket^* f = \llbracket t\rrbracket^* g$.*

Proof. By Lemma 1.5.5, we have $\mathsf{deps}\,t\,f = \mathsf{deps}\,t\,g = l$. By Lemma 1.5.4, we therefore have $\mathsf{subtree}\,t\,l = \mathsf{Ans}\,(\llbracket t\rrbracket^* f) = \mathsf{Ans}\,(\llbracket t\rrbracket^* g)$ and hence, $\llbracket t\rrbracket^* f = \llbracket t\rrbracket^* g$. □

Now we can prove that the function Mod is correct.

Theorem 1.5.7. *Given strongly continuous $F : (A \to B) \to C$ and $f, g : A \to B$, let $m = \mathsf{Mod}\,F\,f$. If for all $(a, b) \in m$, $f\,a = g\,a$ then $F\,f = F\,g$.*

Proof. Since F is strongly continuous, there exists a strategy tree $t \in \mathit{Tree}$ such that $F = \llbracket t\rrbracket^*$ and thus, it is sufficient to show $\llbracket t\rrbracket^* f = \llbracket t\rrbracket^* g$. The last equality directly follows from Lemmas 1.5.6 and 1.5.2. □

For a continuous function $F : (\mathbb{N} \to B) \to C$, we define

$$\mathsf{Mod}_{\mathbb{N}}\, F\, f = \max\left(\mathit{snd}(\mathtt{split}\,(\mathsf{Mod}\, F\, f))\right) + 1$$

where $\mathtt{split} : (A \times B)^* \to A^* \times B^*$ is a list components splitting function.

As a corollary of Theorem 1.5.7, we obtain

Theorem 1.5.8. *Let $F : (\mathbb{N} \to B) \to C$ be strongly continuous, $f : \mathbb{N} \to B$ and $m = \mathsf{Mod}_{\mathbb{N}}\ F\, f$. Then for every $g : \mathbb{N} \to B$ if $f\, i = g\, i$ holds for all $i < m$, then $F\, f = F\, g$.* $\square$

Alternatively, the modulus of continuity $\mathsf{Mod}_{\mathbb{N}}$ can be defined in terms of *Maybe* monad (which is a kind of exception monad). Consider the functional $\mathsf{Mod}_{\mathit{Maybe}}$ recursively defined by

$$\mathsf{Mod}_{\mathit{Maybe}}\, F\, f\, n = \begin{cases} \mathsf{Mod}_{\mathit{Maybe}}\, F\, f\,(n+1) & \text{if } F^{\sharp}_{\mathit{Maybe}}\,(\mathsf{instr}'\, n\, f) = \mathsf{None} \\ n & \text{otherwise} \end{cases}$$

with

$$\mathsf{instr}'\, n\, f\, x = \mathit{if}\ x < n\ \mathit{then}\ \mathsf{Some}\ (f\, x)\ \ \mathit{else}\ \mathsf{None}$$

and define $\mathsf{Mod}'_{\mathbb{N}}\, F\, f = \mathsf{Mod}_{\mathit{Maybe}}\, F\, f\, 0$. Surely, the implementation of $\mathsf{Mod}'_{\mathbb{N}}$ using exceptions suffers from some inefficiency which makes the implementation with side-effects $\mathsf{Mod}_{\mathbb{N}}$ preferable in practice. Similar to Theorem 1.5.8, we show that $\mathsf{Mod}'_{\mathbb{N}}$ is also correct.

Theorem 1.5.9. *Let $F : (\mathbb{N} \to B) \to C$ be strongly continuous, $f : \mathbb{N} \to B$ and $m = \mathsf{Mod}'_{\mathbb{N}}\, F\, f$. Then for every $g : \mathbb{N} \to \mathbb{N}$ if $f\, i = g\, i$ holds for all $i < m$, then $F\, f = F\, g$.* $\square$

Termination of $\mathsf{Mod}'_{\mathbb{N}}$ follows from the well-foundedness of strategy trees for pure functionals.

Notice that for practical use it is convenient if $\mathsf{Mod}\, F\, f$ returns both the result of $F\, f$ and the modulus of F at f. For instance, for the state monad one may define

$$\mathsf{Mod}_{\mathbb{N}}\, F\, f = \mathit{let}\ (v, m) = F^{\sharp}_{\mathit{State}_{\mathbb{N}^*}}\,(\mathtt{instr}\, f)\, \varepsilon\ \ \mathit{in}\ (v, \max\, m + 1)$$

John Longley in [Lon99] considered two possible applications of Mod, for implementation of general search algorithms and for algorithms for exact real-number computations. In what follows, we discuss the algorithm for exact real-number integration presented by Longley.

Exact integration We represent real numbers in the interval $[-1, 1]$ by infinite "streams" of type $\mathtt{real} = \mathtt{nat} \to \mathtt{bit}$ with $\mathtt{bit} = \{\mathtt{-1}, \mathtt{0}, \mathtt{1}\}$ being a set of signed bits. The stream $r : \mathtt{real}$ represents a real number $\sum_{i=0}^{\infty}(r\, i) \cdot 2^{-(i+1)}$. Representations of real numbers are not unique. For example, the stream of zeros $\mathtt{0}^{\infty}$ and the stream $\mathtt{1}(\mathtt{-1})^{\infty}$ both represent the real value 0; $\mathtt{10}^{\infty}$ and $\mathtt{01}^{\infty}$ both represent the real $\frac{1}{2}$, and so on.

We construct an algorithm that takes as input $n \in \mathbb{N}$ and a total strongly continuous $F : \texttt{real} \to \texttt{real}$ representing a continuous real-valued function $f : [0,1] \to [0,1]$ and computes $\int_0^1 f\,dx$ to within $\varepsilon = 2^{-n}$. Essentially, the presented algorithm finds a partition of $[0,1]$ such that on every subinterval $[x_i, x_{i+1}]$ of the partition the variation of f does not exceed ε, and returns a Riemann sum $\sum_{i=0}^{N}(f\,x_i) \cdot (x_{i+1} - x_i)$. For that, it is sufficient to know a value of $f\,x$ to within ε in every point $x \in [0,1]$. For $F : \texttt{real} \to \texttt{real}$ representing f and $\varepsilon = 2^{-k}$, we define $F|_\varepsilon$ that represents f within ε by

$$F|_{\varepsilon=2^{-k}}\,x = \lambda i : \texttt{nat}.\ \textit{if}\ i < k\ \textit{then}\ F\,x\,i\ \textit{else}\ \texttt{0}.$$

That is, $F|_\varepsilon$ yields first k bits of output of F precisely and replaces the rest bits with zeros. Obviously, if F is strongly continuous then so is $F|_\varepsilon$.

```
exception OneIsReached

let integrate′ (F : real → real) n (x : real) (acc : real) =
    let (y, m) = Mod_ℕ F|_{ε=2^{-n}} x in
    let acc′ = acc + y · 2^{-m} in
    try
        let x′ = x + 2^{-m} in
        integrate′ F n x′ acc′
    with
        OneIsReached → acc′

let integrate (F : real → real) n =
    integrate′ F n 0^∞ 0^∞
```

Figure 1.4.: Exact integration algorithm in ML

The algorithm for exact integration by Longley is given in Figure 1.4. The algorithm proceeds as follows. Put $x_0 = 0$ which is represented by $\texttt{0}^\infty$, and let $(y_0, m_0) = \texttt{Mod}_\mathbb{N}\,F|_\varepsilon\,\texttt{0}^\infty$. Then only m_0 input bits are required for $F|_\varepsilon\,\texttt{0}^\infty$ to produce y_0 and hence, $F|_\varepsilon$ returns the same result when applied to any stream with $\texttt{0}$'s at the first m_0 positions, including $\texttt{0}^{m_0}\texttt{1}^\infty$. This means that $|f\,x - y_0| \leq \varepsilon$, for all $x \in [x_0, x_0 + \delta_0]$ with $\delta_0 = 2^{-m_0}$. Hence, $\int_{x_0}^{x_1} f\,dx$ is approximated by $y_0 \cdot \delta_0$ to within $\varepsilon \cdot \delta_0$. We put $x_1 = x_0 + \delta_0$ and repeat the process. In this way, a partition $\{x_i\}_{i\in[0,N]}$ is constructed such that $|f\,x_i - y_i| \leq \varepsilon$ for all $x \in [x_i, x_{i+1}]$. One can show that the algorithm indeed terminates at some step N and $x_N = 1$.

Theorem 1.5.10 ([Lon99]). *The exact integration algorithm terminates.* □

Since for each $i \in [0, N-1]$, $\int_{x_i}^{x_{i+1}} f\,dx$ is approximated by $y_i \cdot \delta_i$ to within $\varepsilon \cdot \delta_i$ and $\sum_i \delta_i = 1$, the algorithm produces a correct result.

On the figure above, $\texttt{integrate}'$ takes as parameters a function $F : \texttt{real} \to \texttt{real}$, a natural n such that $\varepsilon = 2^{-n}$ is the required precision of integral computation, a current point x in which the value of F is computed, and an accumulator for a result *acc*. The addition operation raises the exception `OneIsReached` when the sum reaches the value 1. The value 2^{-m} is represented by $\texttt{0}^m\texttt{10}^\infty$. Multiplication of y by 2^{-m} can be efficiently implemented as a "right shift" of y by m positions.

We don't give a formal proof for the correctness of the algorithm, but the key point of such a proof would be that purity of $F^\sharp$ (strong continuity of F) is a sufficient condition for the correctness of $\texttt{integrate}'$, i.e., $F^\sharp$ should neither change a state nor raise any exceptions by itself.

1.5.2. Formal Reasoning About Programs

Our results are applicable for verification of algorithms that take pure second-order functionals as input. Using the representation result, one can always assume that the input is given directly by means of a strategy tree which enables naturally the reasoning by induction. For a concretely given input, e.g., in a form of a program defined in some restricted language, one might establish its purity in the sense of relational parametricity or use the result of Parametricity Theorem which guarantees purity of definable functionals.

In the previous subsection, we considered a design of certified algorithms for exact integration which are based on the extraction of intentional information from pure functionals representing computable real functions of type `real` $\to$ `real`. The next chapter presents another case study — formalization and verification of fixpoint solver **RLD** in Coq.

As shown by Keuchel and Schrijvers in [KS12], the results of Section 1.2 can be extended and applied for "modular reasoning in the purely functional setting of polymorphic monadic mixin components". Similarly as pure monadically parametric second-order functionals are characterized by first-order objects — strategy trees, the authors show that polymorphic monadic mixin components correspond to monomorphically-typed first-order tree-like representations. The latter allows to eliminate a higher-order parameter of the mixin component.

1.6. Conclusion

The main result of this chapter is in providing two equivalent characterizations of monadically parametric second-order functionals in both the total and the partial settings. The first characterization is extensional and is based on preservation of relations similar to the parametricity by Reynolds [Rey83]. In order to adapt the technique to monadic case, we introduced a notion of the *acceptable monadic relation*. We should note that the latter also appears in [Voi09] (named as *monad action*) where it was used to derive free theorems for monadic programs in the sense of Wadler [Wad89]. The second, intentional, characterization of monadically parametric second-order functionals is based on representation of those as strategy trees that define question-answer dialogues. In [Voi09] however, this intentional characterization was not considered.

Resumptions, a notion similar to strategy trees, appear in O'Hearn and Reynolds' paper [OR00] for intentional characterization of Algol's procedures. The work presented in the thesis differs from loc.cit. by a more general monadic formulation and by generalization of the extensional characterization to monads other than the state monad. In [KS12], it was mentioned that strategy trees also can be interpreted as *coroutines* also known in functional programming as the coroutine monad, or the resumption monad. *Decision trees* as strategies for sequentially realizable functionals appear in [Lon02], but not in the monadic context.

The Parametricity Theorem states basically that every functional implemented in $\lambda_{\to}$ with monadic semantics is monadically parametric. We argue that second-order monadically parametric functionals are essentially strongly continuous functionals.

As a special case, we have considered a notion of parametricity for second-order functionals polymorphic in states. The results of [HKS10b] were generalized for arbitrary second-order types. Also, a more complete formalization of Theorem 1.3.35 is provided in the thesis.

Finally, we have formulated several applications of the notion of purity, including the verification of algorithms that take pure second-order functionals as input. Among such algorithms are generic local fixpoint algorithms and algorithms for exact real arithmetic.

Open questions

Extraction of strategies using other types of effects? What kind of monads allow for extraction of a strategy tree or other kind of strategy? The studied cases include *State*, *Cont*, T_{Tree}. Are there more examples? For instance, it is seemingly impossible to use exceptions for this purpose. The exception monad is unable to distinguish the trees $\mathsf{Que}\ 0\,(\lambda b.\mathsf{Que}\ 0\,(\lambda b'.\mathsf{Ans}\ b'))$ and $\mathsf{Que}\ (0, \lambda b.\mathsf{Ans}\ b)$. In both cases, if k raises no exceptions, we arrive to the same answer b. If it does, only one question is asked and the exception is propagated.

Generalization to higher types. Connection with game semantics. Apparently, it is possible to give a definition of monadic parametricity similar to Definition 1.2.17 for all types of $\lambda_{\to}$. However, the question is left open what are corresponding strategies for types of order higher than two. It could be the case that strategies in the sense of game semantics, like in [HO00, AM96, AMJ94], are the appropriate generalization. However, then it is not clear yet how one could characterise their existence by parametricity. Generally, connection with the game semantics should be better understood. Another possible approach is in utilizing of Kripke relations of varying arity as in [JT93]. This might be an interesting question for further investigation.

It is hard to think of a practical example of a parametric higher-order functional. One example however that may come into our minds is a kind of algorithm for solving differential or integral equations from analysis that takes as arguments a third-order differentiation/integration functional F and some real-value parameter function f. Characterization of higher-order monadically parametric functionals, though, seems to be mostly of academic interest.

2. Verified Generic Fixpoint Algorithms

2.1. Introduction

Many problems in programs analysis and other application areas may be expressed in terms of solutions to constraint systems of the form $\mathbf{x} \sqsupseteq F_{\mathbf{x}}$, $\mathbf{x} \in V$, where V is a set of variables, or unknowns, and $F_{\mathbf{x}}$ is a function returning a value in a semi-lattice of abstract properties $\mathbb{D}$. Given a constraint system $\mathcal{S}$ of the above form representing a particular problem, the goal is then to compute efficiently a solution to $\mathcal{S}$ that is, an assignment σ from variables V to abstract values from $\mathbb{D}$ such that all the constraints are satisfied, i.e., $\sigma\,\mathbf{x} \sqsupseteq F_{\mathbf{x}}\,\sigma$, for $\mathbf{x} \in V$.

The main tool for solution of such constraint systems is a fixpoint computation algorithm. Starting with least "bottom" values for unknowns, the algorithm iteratively picks a next unknown $\mathbf{x}$ according to some evaluation strategy and tries to satisfy the constraint for $\mathbf{x}$. It continues to increase values of unknowns iteratively until a required solution is found.

Fixpoint algorithms are often reimplemented and reinvented for many particular application domains and even kinds of analysis. For example, in the verified compiler CompCert [Ler09], one can find two versions of the same solver instantiated for the forward and backward analyses. By contrast, *generic* solvers do *not* make any assumptions on the application domain. Generic reusable implementations are very useful since they can be freely instantiated for a wide range of applications: they are parametric in a lattice of abstract values $\mathbb{D}$ and do not depend on the kind of right-hand sides. Additionally, such implementations allow to separate the iteration logic from the logic of application itself which may ease the reasoning about the correctness of the tool.

Another important quality of a "good" solver is its *well-behavedness* in a certain sense. The *exact* solver returns a minimal solution for complete lattices $\mathbb{D}$ and monotonic constraint systems. In order to apply the solver with widening and narrowing acceleration techniques, it must additionally implement a *chaotic iteration* strategy in the sense of Cousot and Cousot [CC77b]. Intuitively, the latter means that evaluations of right-hand sides are performed atomically, i.e., two consecutive accesses to a variable $\mathbf{x}$ during the evaluation of the right-hand side return same values.

In practise, a full solution to the constraint system is not always needed. One might only be interested in values for a small subset of unknowns — a *local* solution. Starting with a given list of *interesting* variables, the *local* solver explores the system and evaluates only those unknowns whose values are necessary to satisfy the interesting constraints. The variables needed to compute a local solution are not known in advance. Moreover, the dependencies between unknowns may even change during the evaluation. It is the task of the fixpoint solver to avoid unnecessary computations and use an optimal strategy for picking the constraints.

Fixpoint algorithms are at the heart of many analysis tools and compilers. If these tools are to be trusted, the fixpoint algorithm must be verified. Since effective fixpoint solvers often exhibit a very intricate behaviour, they are hard to prove correct and their implementation is error-prone. At the same time, testing gives no guarantees about correctness of solvers. In practice, it may happen that even those solvers that were repeatedly used on real-life samples, are erroneous. That was exactly the case with the initial version of the solver **RLD** for that a counterexample was found while we were trying to prove its correctness, despite the fact that it worked correctly in all the practical cases. After the error was discovered, the author inspected another fixpoint solver used in the static analyser Goblint [VV09] and found a bug in its implementation!

The simplest workaround is to implement a certified validator that a posteriori checks that a result returned by the untrusted solver is consistent and indeed satisfies the given constraint system. Proving correctness of the verifier is significantly easier that proving correctness of the solver. However, the author thinks this is still a poor solution for safety-critical tools which must comply highest safety standards. A certificate for the checker does not guarantee that the solver will produce anything meaningful. In contrast, a certificate for the solver supplied with termination contracts would guarantee that the solver always return correct results. What one should do when the validator once gives a negative answer?

The main contribution of this chapter is the development of the generic local exact certified fixpoint solver **RLDE**.

Related Work Since verification of fixpoint solvers is a rather non-trivial and time consuming task, there were not so many attempts previously to develop an efficient certified local fixpoint solver. For example, in the recent paper [BLMP13], only a fixpoint checker is formally verified while the non-verified solver based on general iteration techniques by Bourdoncle [Bou93] is implemented in OCaml.

The CompCert verified compiler project [Ler09] and other known publications on certified analysis [BCDdS02, KN03, BGL04, CGD04] formalize variants of Kildall's worklist algorithm [Kil73]. This algorithm is a standard tool for the data flow analysis [Muc97]. It is generic in the sense that it does not depend on the application domain and thus, can be used with any abstract join-semilattice $\mathbb{D}$. Kildall's algorithm operates on the control flow graph of the given program. That is, each vertex of the graph represents an instruction in the program. There is an edge between vertices p and q if q can be executed immediately after p. In order to apply the algorithm, the control graph must be given explicitly, for example, through a function *succs* which maps each p to a set of successor instructions. The function *succs* must be precomputed statically. Thus, the algorithm cannot be considered as a generic one in our sense.

Other attempts to construct a local generic solver are made by Le Charlier and Van Hentenryck in [CH92] which presented the top-down solver **TD** and by Fecht and Seidl in [FS99] presenting the worklist based recursive solver with time-stamps. For these solvers, pen-and-paper arguments for the partial correctness are provided. In both publications, it is assumed that right-hand sides are given as computable function implemented in some programming language. However, the purity of right-hand sides as a necessary condition for correctness of the solvers is not mentioned. To the author's knowledge, neither of these solvers has ever been verified formally, by means of a theorem prover or a proof assistant.

The Coq Proof Assistant

The interactive proof assistant Coq [Coq12] is based on the formal language *Calculus of Inductive Constructions* which is an extension of original Calculus of Construction of Coquand and Huet [CH88] by inductive definitions [CP88, PPM89]. The calculus is strongly normalizing which intuitively means that every computation in Coq terminates.

Terms and types The specification language of Coq is strongly typed, that is, every term definable in the language has a type. Every type, in its turn, is also a term of another type which is called *sort*. Coq has the following built-in sorts

– `Set`, the sort of data types and program specifications

– `Prop`, the sort of logical proposition

– `Type`, the sort of `Set` and `Prop`

For example, the type of terms `42` and `2 + 3` is `nat`, the inductive type of natural numbers. The term `nat` in turn has type `Set`. The term `5 :: nil` has type `list nat`, and `list` is of type `Type` $\to$ `Type`, that is for any type A : `Type`, `list` A is a type.

The keywords `Definition` and `Fixpoint` are used in Coq to define non-recursive terms and recursive functions, respectively. The Coq's underlying logic supports only structural recursion over well-founded structures (see the paragraph on Inductive definitions below) that generalizes a primitive recursion. In the case of recursive definitions, Coq uses a syntactic termination check in order to preserve the strong normalization. It requires that some function parameter (also called a *principal* parameter) structurally decreases within every recursive call. For example, the function that computes a sum of two natural numbers is defined with the first argument being a principal argument as follows.

```
Fixpoint plus (n m : nat) : nat :=
  match n with
    | 0 => m
    | S p => S (plus p m)
  end.
```

The function performs a recursive call on `p` which obtained from `n` by pattern matching and thus, is structurally smaller than `n`.

Propositions The Curry-Howard correspondence [SU06] is at the heart of Coq. A proposition P is a term of the sort `Prop`. Moreover, at the same time, P itself is a type. There are two distinguished propositions `True` and `False` of type `Prop` (which are inductively defined) in Coq. One can define propositions using implications (which are usual arrows of Coq's type system), universal quantification (which are dependent types) and inductive constructions for conjunction, disjunction and existential quantifications.

To prove a proposition P means to show that P is *inhabited*, that is, to construct a proof term p of type P. It is the user who is in charge to construct p which will be only type-checked by the kernel of Coq. There are two ways to build p. First, the user may define p explicitly which is however not always convenient since a term may be very large even in the case of a simple proposition. The second way, is to use a suite of *tactics*, i.e., commands for interactive construction of proofs. The formulated proposition is posed as goal in a context of initial assumptions (possibly empty). The user then applies a sequence of tactics in order to rewrite, simplify, transform the goal or assumptions, introduce new assumptions, decompose the goal into simpler goals, or solve the goal. The proof process ends when all the subgoals generated during the process are solved. Actually, tactics may be used for construction of terms of any type and not only propositions.

A type of predicates (relations) on a set A is a type $A \to$ `Prop` of functions from A to `Prop`. In this chapter, we shall think of logical relations in the above sense, and we shall write a usual arrow $\to$ for logical implications instead of double arrows $\implies$.

Inductive definitions Coq allows to introduce inductive types that consist of well-founded tree-shaped structures. Such a type may represent an infinite set, but each element of it is constructed in a "well-founded" manner. New inductive definitions in Coq are introduced by means of the keyword `Inductive`. For example, the type of natural numbers `nat` is defined in Coq's standard library inductively as

```
Inductive nat : Set :=
  | O : nat
  | S : nat -> nat.
```

providing two *constructors*, a zero and a successor function, correspondingly. Thus, every structure of type `nat` is either `O` or can be produced from `O` by finitely many applications of `S`.

The inductive reasoning in Coq is very convenient since the system automatically generates induction principles from inductive definitions. For example, for `nat` the usual induction scheme is generated

```
nat_ind
   : forall P : nat -> Prop,
     P 0 -> (forall n : nat, P n -> P (S n)) -> forall n : nat, P n
```

Coq imposes certain restrictions on what it accepts as a legal inductive definition. Every well-formed inductive definition must satisfy the *positivity condition* (for a precise definition refer to the Coq manual [Coq12], Section 4.5.3). Violation of the condition would lead to inconsistency of the calculus since it would be possible to break the strong normalization property and construct a proof of `False`.

Extraction Since every constructive proof corresponds to a certain program by Curry-Howard isomorphism, it is possible to effectively *extract* this program. Coq implements a mechanism of extraction of ML programs (in OCaml, Haskell, or Scheme) from proofs.

For a further reading on Coq, refer to the Coq tutorial [GP07] and the Coq'Art textbook [BC04].

2.2. RLD Solver

This section is organized as follows. In Subsection 2.2.1 we present the solver **RLD**. In Subsection 2.2.2 we prove that the solver is partially correct. In Subsection 2.2.3 we define a subclass of right-hand sides for which **RLD** is exact. In Subsection 2.2.4 we present an exact modification **RLDE**. In Subsection 2.2.5 we provide sufficient conditions for termination of **RLD** (**RLDE**).

2.2.1. Description of RLD

In what follows, we give an informal description of the algorithm **RLD** and provide it's stateful implementation in ML-like language. We do not define yet formally the notions of *constraint system* and *solver*, but instead appeal to intuitive understanding. The precise definitions and discussion on sufficient conditions for correctness of the solver follow in Subsection 2.2.2.

The algorithm **RLD** performs on a constraint system denoted by

$$\mathbf{x} \sqsupseteq F_{\mathbf{x}}, \quad \mathbf{x} \in V,$$

with a fixed set of *variables* (or *unknowns*) V over a *bounded join-semilattice*

$$\mathbb{D} = (D, \sqcup, \sqsubseteq, \bot)$$

that consists of a carrier D equipped with a partial ordering $\sqsubseteq$ and a least upper bound operation $\sqcup$, and has a distinguished least element $\bot$. We assume that the binary operation $\sqcup$ is total, that is, for every $x, y \in D$ there exists a least upper bound $x \sqcup y$ of x and y, i.e., $x, y \sqsubseteq x \sqcup y$ and for all z such that $x, y \sqsubseteq z$, $x \sqcup y \sqsubseteq z$ holds. Generally, we require neither completeness nor the ascending chain condition for $\mathbb{D}$. We assume that for every $\mathbf{x} \in V$, right-hand side $F_{\mathbf{x}}$ is a function of type $(V \to D) \to D$ implemented in some programming language.

One basic idea of the algorithm **RLD** is in pre-evaluation of variables instead of using their current values. Namely, as soon as a value of some variable $\mathbf{y}$ is requested during evaluation of the right-hand side $F_{\mathbf{x}}$, the algorithm does not naively use the current value for $\mathbf{y}$ but instead, it first tries to compute a best possible approximation for $\mathbf{y}$ relative to a current variable assignment. This allows to reduce the overall number of performed iterations. The second idea is to minimize the amount of touched variables. The algorithm runs for a finite list of *interesting* variables provided by the user. **RLD** tries to solve only those interesting variables and the variables they transitively depend on. For that, the algorithm relies on self-observation in order to discover variable dependencies (with respect to the current variable assignment) as they are encountered during evaluation of right-hand sides. The solver records the information about dependencies in a dedicated data-structure by means of *side-effects*.

The algorithm **RLD** maintains the following mutable data structures (we use the array notation to denote access to components of these structures).

1. Finite structure σ mapping variables from V to abstract values from D. Since the overall size of V can be large or even infinite, the algorithm tracks only finite number of observed variables. To extend σ to all unknowns from V, we define the auxiliary function
$$\sigma_\perp\,\mathbf{x} = \begin{cases} \sigma\,[\mathbf{x}] & \text{if } \mathbf{x} \in \mathrm{dom}(\sigma) \\ \perp & \text{otherwise} \end{cases}$$
that returns a current value of $\sigma\,[\mathbf{x}]$ if it is defined and $\perp$ otherwise. The function $\sigma_\perp$ releases us of having to track manually which variables were already introduced to σ and which not and thus, to avoid problems by evaluations of right-hand sides.

2. Finite map *infl* that stores dependencies between variables. More exactly, for a variable $\mathbf{x}$, *infl*$[\mathbf{x}]$ returns an over-approximation of a set of variables $\mathbf{y}$, for which evaluation of $F_\mathbf{y}$ on the current $\sigma_\perp$ depends on $\mathbf{x}$. Again, we track only finite number of observed variables and define the auxiliary function
$$\mathit{infl}_\emptyset\,\mathbf{x} = \begin{cases} \mathit{infl}\,[\mathbf{x}] & \text{if } \mathbf{x} \in \mathrm{dom}(\mathit{infl}) \\ \emptyset & \text{otherwise.} \end{cases}$$
In practical implementations, *infl* can be define as a finite mapping either to lists of V or to finite sets of V. However, different implementation of *infl* may lead to different orders of evaluations of variables and different outputs of the solver.

3. Finite set *stable* $\subseteq V$. Intuitively, if variable $\mathbf{x}$ is marked as “stable” then either $\mathbf{x}$ is already “solved” in the sense that a computation for $\mathbf{x}$ has completed and σ gives a solution for $\mathbf{x}$ (i.e., $\sigma_\perp\,\mathbf{x} \sqsupseteq F_\mathbf{x}\,\sigma_\perp$ holds) and all those variables $\mathbf{x}$ transitively depends on, or $\mathbf{x}$ is “called” and it is in the call stack of `solve` function and its value is being processed.

The structures receive initial values $\sigma_{init} = \emptyset$, $\mathit{stable}_{init} = \emptyset$, $\mathit{infl}_{init} = \emptyset$.

The algorithm **RLD** proceeds as follows (see Fig. 2.1).

– The function `solve_all` is invoked by `main` for a list $X \subseteq V$ of interesting variables from the initial state. The function `solve_all` calls the recursive worker function `solve` for every $\mathbf{x} \in X$ in turn.

```
let $\sigma_\perp$ x =
  if x $\in$ $\mathrm{dom}(\sigma)$ then $\sigma$[x] else $\perp$

let $\mathit{infl}_\emptyset$ x =
  if x $\in$ $\mathrm{dom}(\mathit{infl})$ then $\mathit{infl}$[x] x else $\emptyset$

let extract_work x =
  let $\mathit{work}$ = $\mathit{infl}_\emptyset$ x in
  $\mathit{infl}$[x] := $\emptyset$;
  $\mathit{stable}$ := $\mathit{stable} \setminus \mathit{work}$;
  $\mathit{work}$

let rec evalget x y =
  solve y;
  $\mathit{infl}$[y] := $\mathit{infl}$[y] $\cup$ {x};
  $\sigma_\perp$ y

and solve x =
  if (x $\notin$ $\mathit{stable}$) then begin
    $\mathit{stable}$ := $\mathit{stable}$ $\cup$ {x};
    let $d$ = $F$ x (evalget x) in
    let $\mathit{cur}$ = $\sigma_\perp$ x in
    let $\mathit{new}$ = $\mathit{cur} \sqcup d$ in
      if ($\mathit{new} \not\sqsubseteq \mathit{cur}$) then begin
        $\sigma$[x] := $\mathit{new}$;
        let $\mathit{work}$ = extract_work x in
          solve_all $\mathit{work}$
      end
  end

and solve_all X =
  foreach x $\in$ X do solve x

let main X =
  $\sigma$ := $\emptyset$; $\mathit{infl}$ := $\emptyset$; $\mathit{stable}$ := $\emptyset$;
  solve_all X;
  $(\sigma_\perp, \mathit{stable})$
```

Figure 2.1.: The recursive solver tracking local dependencies (**RLD**)

- The function `solve` when called for some variable $\mathbf{x}$ first checks whether $\mathbf{x}$ is already marked as being *stable*. If so, the function returns; otherwise, the algorithm adds $\mathbf{x}$ to the set *stable* and tries to satisfy the constraint $\sigma\,\mathbf{x} \sqsupseteq F_{\mathbf{x}}\,\sigma$. For that, it evaluates a value d of the right-hand side $F_{\mathbf{x}}$ by invoking $F_{\mathbf{x}}$ (`evalget` $\mathbf{x}$).
- After the latter returns, `solve` computes $\mathit{new} = \mathit{cur} \sqcup d$ with *cur* being a current value of $\sigma_\bot\mathbf{x}$, and compares *new* and *cur*. If *new* is subsumed by *cur*, the constraint for $\mathbf{x}$ is satisfied, and `solve` returns. Otherwise, the value of σ for $\mathbf{x}$ gets updated with *new*. Since the current value of $\sigma\,[\mathbf{x}]$ has changed, all constraints of variables $\mathbf{y}$ dependent on $\mathbf{x}$ may not be satisfied any more and must be re-evaluated. For that, the function `solve` removes those variables from *stable* (*destabilizes* them) by invoking `extract_work` function and schedules them for reevaluation. The function `extract_work` returns the set of those variables as *work* and additionally resets *infl* $[\mathbf{x}]$, since some of dependencies may be outdated. This allows to keep precision of the information stored by *infl*. Then `solve_all` *work* is recursively called.
- As we mentioned, the right-hand side $F_{\mathbf{x}}$ is not evaluated directly on $\sigma_\bot$, but on the partially applied stateful function `evalget` $\mathbf{x}$. For every variable $\mathbf{y}$ accessed by $F_{\mathbf{x}}$, `evalget` $\mathbf{x}\,\mathbf{y}$ first computes a better approximation for $\mathbf{y}$ by recursive call to `solve` $\mathbf{y}$. Additionally, `evalget` keeps a track of variables $\mathbf{y}$ visited during evaluation of $F_{\mathbf{x}}$. We say that $\mathbf{y}$ *influences* $\mathbf{x}$, or $\mathbf{x}$ *depends on* $\mathbf{y}$. The discovered dependencies are stored in *infl*. Only after *infl* structure is updated appropriately, the current value of $\sigma\,\mathbf{y}$ is returned by `evalget` $\mathbf{x}\,\mathbf{y}$.
- Note that the structure *stable* prevents the solver from infinite recursive descent into solving $\mathbf{x}$ if $\mathbf{x}$ depends on itself like, say, for $F_{\mathbf{x}}\,\sigma = \sigma\,\mathbf{x}$. If `solve` was once called for $\mathbf{x}$ then $\mathbf{x}$ is marked as *stable*, and further recursive calls to `solve` $\mathbf{x}$ immediately return.

The comparison $\mathit{new} \not\sqsubseteq \mathit{cur}$ can be replaced by the equality check $\mathit{new} \neq \mathit{cur}$. This may be beneficial in the case if every element of the (semi-)lattice $\mathbb{D}$ has a unique representation and the equality operation can be implemented more efficiently than the ordering relation.

The following two distinguishing features of **RLD** must be emphasized. The first feature is in how the solver behaves when a variable $\mathbf{x}$ changes its value. The solver performs destabilizations only *locally*, i.e., only for variables that are influenced by $\mathbf{x}$ immediately, and triggers a reevaluation of these variables at once. This aspect distinguishes **RLD**, for example, from another local solver, the top-down solver **TD** [CH92] by Le Charlier and Van Hentenryck which destabilizes recursively all the variables that also indirectly (transitively) depend on $\mathbf{x}$. Thus, this feature of **RLD** leads to more algorithmic shortcuts allowing to avoid unnecessary recomputations. The second feature is in how **RLD** evaluate right-hand sides. During evaluation of $F_{\mathbf{x}}$(`eval` $\mathbf{x}$) two consecutive accesses to some $\mathbf{y}$ may yield different values. The reason is that $\mathbf{y}$ may get recomputed in-between due to some variable dependencies. That makes evaluations of right-hand sides essentially *non-atomic* and thus, **RLD** does *not* belong to the family of *chaotic iteration* [CC77b] schemes. However, this feature is rather a deficiency of the solver than its advantage, as we will see later.

In the following subsections, we show that **RLD** is a local solver in a certain sense and returns parts of a minimal solutions for a subclass of monotonic constraint systems.

2.2.2. Correctness

Notice that the algorithm **RLD** is formulated in Figure 2.1 as being applicable to any right-hand side function F implemented in ML. However, it only makes sense to apply the algorithm to constraint functions that do not produce any side-effects by themselves.

In this subsection, we formulate precisely sufficient conditions for partial correctness of **RLD** and prove that it indeed belongs to the family of local generic solvers.

Which right-hand side functions F are allowed? As we mentioned, the algorithm uses side-effects for extracting intentional information about variable dependencies of its functional argument F while the latter is implemented in some specification language, for example, in ML. However, if we pose no restrictions on F, we might experience a problem if the specification language provides non-functional features. Consider, for instance, the following two ML-snippets

```
let a = ref 0
let f : int → int =
  fun x → incr a; x + 1
```

and

```
exception Exn
let g : int → int =
  fun x → raise Exn
```

that define legal ML-functions f and g of type `int → int`. However, both functions produce effects: f mutates a store and g raises an exception, but these effectful behaviours are not reflected in ML-types of functions. If F is allowed to produce any kind of effects, correctness of the algorithm cannot be guaranteed. For example, if F modifies any of the structures maintained by **RLD** or influences its flow of control by any other means, the result returned by the solver when applied to such F might not make any sense.

In order to reason about the solver **RLD** formally, we will provide a purely functional implementation of the algorithm by means of passing of the world as a state parameter. We prove the correctness of **RLD** relative to a subclass of right-hand sides represented by monadically parametric functionals of type

$$V \to \textstyle\prod_T.(V \to TD) \to TD$$

in the sense of Definition 1.2.17. This means that effects of the evaluation of $(F_{\mathbf{x}})_T\,\sigma$ are attributed only to the effects produced by the effectful function σ, for any monad T. We shall omit the index of a projection if it is clear from context. We remind that Theorems 1.2.25 and 1.2.26 imply that there exists a one-to-one correspondence between monadically parametric functions of the above type and *strategy trees* of type $V \to \mathit{Tree}_{V,D,D}$. Let us remind the main definitions related to strategy trees.

Definition 2.2.1. Given sets A, B and C, the set of strategy trees $\mathit{Tree}_{A,B,C}$ is a minimal set generated by constructors:

– $\mathsf{Ans} : C \to \mathit{Tree}_{A,B,C}$

– $\mathsf{Que} : A \to (B \to \mathit{Tree}_{A,B,C}) \to \mathit{Tree}_{A,B,C}$

Definition 2.2.2. Given a monad T, we define the function

$$[\![\cdot]\!]_T : \mathit{Tree}_{A,B,C} \to (A \to T\,B) \to T\,C$$

recursively by

– $[\![\mathsf{Ans}\ c]\!]_T = \lambda k.\,\mathsf{val}_T\,c$

– $[\![\mathsf{Que}\ a\,f]\!]_T = \lambda k.\,\mathsf{bind}_T\,(k\,a)\,(\lambda b.[\![f\,b]\!]_T\,k)$.

Define $[\![t]\!] = \Lambda T.[\![t]\!]_T$. We will also refer to $[\![\cdot]\!]$ as the *monadic interpreter* of strategy trees. We write $[\![\cdot]\!]^*$ for $[\![\cdot]\!]_{Id}$.

Example 2.2.3. For state monad State_S, we have

– $[\![\mathsf{Ans}\ c]\!]_S = \lambda s.(c, s)$

– $[\![\mathsf{Que}\ a\,f]\!]_S = \lambda k.\lambda s.\mathit{let}\ (b_1, s_1) = k\,a\,s\ \mathit{in}\ [\![f\,b_1]\!]_S\,k\,s_1$.

Example 2.2.4. Given $h = \mathsf{val}_{\mathit{State}_S} \circ \sigma = \lambda a.\lambda s.(\sigma\,a, s)$ with $\sigma : A \to B$, that is, h has no effects on state, then $[\![t]\!]_S\,h\,s = (c, s_1)$ implies $s = s_1$, for all t, s, s_1, and c.

Based on results of the previous chapter, we give

Definition 2.2.5. The right-hand side functional $F : V \to \prod_T.(V \to TD) \to TD$ is *monadically parametric* (*pure*) if there exists a (unique) strategy function $\overline{F} \in V \to \mathit{Tree}_{V,D,D}$ such that $F_{\mathbf{x}} = [\![\overline{F}_{\mathbf{x}}]\!]$ extensionally, for all $\mathbf{x} \in V$.

Example 2.2.6. Given a pure F and $\sigma : V \to D$, $(F_{\mathbf{x}})_{\mathit{State}_S}(\mathsf{val}_{\mathit{State}_S} \circ \sigma)\,s = (d, s_1)$ implies $s = s_1$, for all s, s_1, and d.

We introduce a monadic version of the instrumentation function `instr` that allows for extraction of modulus of continuity from pure functionals.

$$\mathtt{instr}_T : \textstyle\prod_{A,B}.(A \to TB) \to (A \to \mathit{StateT}_{\mathtt{list}\,(A\times B)}TB)$$

by

$$\mathtt{instr}_T\,A\,B\,f = \lambda a.\lambda l.\,\mathsf{bind}_T\,(f\,a)\,(\lambda b.\,\mathsf{val}_T(b, l \mathbin{+\!+} \mathtt{[}(a, b)\mathtt{]}))$$

where StateT is a state monad transformer and ++ is the list append function. Intuitively, $\texttt{instr}_T\, f$ records the arguments accessed by f along with respective values returned by f. Instantiated for State_S, we have

$$\texttt{instr}_{\mathit{State}_S}\, A\, B\, f = \lambda a.\lambda l.\lambda s.\mathit{let}\ (b, s_1) = f\, a\, s\ \ \mathit{in}\ \ ((b, l \mathbin{++} [(a,b)]), s_1).$$

Intuitively, a variable $\mathbf{x}$ depends on a variable $\mathbf{y}$ relative to the variable assignment σ if a value $\sigma\,\mathbf{y}$ is accessed during the evaluation of $F_{\mathbf{x}}\,\sigma$. By means of instr, we define a notion of variable dependencies formally. We remind a definition of the *modulus of continuity* functional (see Definition 1.5.1). For $t \in \mathit{Tree}_{A,B,C}$ and $\sigma : A \to B$, define

$$\texttt{deps}\, t\, \sigma = \mathit{snd}([\![t]\!]_{\mathit{State}_{\texttt{list}\,(A\times B)}}\, (\texttt{instr}_{\mathit{Id}}\, \sigma)\, [\,])$$

which yields a *query trace* of computation of $[\![t]\!]^*\sigma$, i.e., a list of queried unknowns $a : A$ along with received answers $\sigma\, a : B$. Notice that $(a,b) \in \texttt{deps}\, t\, \sigma$ implies $b = \sigma\, a$. For a monadically parametric $F : \prod_T.(A \to TB) \to TC$ and its strategy tree $t \in \mathit{Tree}_{A,B,C}$, by $\texttt{deps}\, F\, \sigma$ we denote $\texttt{deps}\, t\, \sigma$.

Definition 2.2.7. For a pure right-hand side function $F : V \to \prod_T.(V \to D) \to D$, we say that a variable $\mathbf{x}$ *depends on* a variable $\mathbf{y}$ (or $\mathbf{y}$ *influences* $\mathbf{x}$) *relative to the variable assignment* $\sigma : V \to D$ if $(\mathbf{y}, \sigma\,\mathbf{y}) \in \texttt{deps}\, F_{\mathbf{x}}\, \sigma$.

Solutions of constraint systems Below we define precisely the notions of a solution and a local solution to the constraint system $\mathcal{S} = (V, \mathbb{D}, F)$ over a semilattice $\mathbb{D}$ with a set of unknowns V and a monadically parametric functional

$$F : V \to \textstyle\prod_T.(V \to TD) \to TD.$$

Definition 2.2.8. We say that the variable assignment $\sigma : V \to D$ is a *solution to the constraint system* $\mathcal{S}$ if $\sigma\,\mathbf{x} \sqsupseteq (F_{\mathbf{x}})_{\mathit{Id}}\,\sigma$ holds for all $\mathbf{x} \in V$.

In contrast, a *local* solution σ satisfies constraints only for a subset of variables and their dependencies as formulated below.

Definition 2.2.9. Let $X \subseteq V$. We say that the pair (σ, X') is a *local solution to* $\mathcal{S}$ *relative to* X if

1. $X \subseteq X'$;

2. for all $\mathbf{x} \in X'$ and $\mathbf{y}$, $(\mathbf{y}, \sigma\,\mathbf{y}) \in \texttt{deps}\, F_{\mathbf{x}}\, \sigma$ implies $\mathbf{y} \in X'$;

3. $\sigma\,\mathbf{x} \sqsupseteq (F_{\mathbf{x}})_{\mathit{Id}}\,\sigma$ holds for all $\mathbf{x} \in X'$.

In particular, this means that the restriction $\sigma\restriction_{X'}$ is a solution to the constraint system $(X', F\restriction_{X'})$.

Note that if $\mathbb{D}$ has a greatest element $\top_{\mathbb{D}}$, we can trivially extend any local solution (σ, X') to a global one by putting $\top_{\mathbb{D}}$ for all $\mathbf{x} \in V \setminus X'$ as

$$\sigma_{X'}\,\mathbf{x} = \begin{cases} \sigma\,\mathbf{x} & \mathbf{x} \in X' \\ \top_{\mathbb{D}} & \text{otherwise.} \end{cases}$$

Lemma 2.2.10. *Let (σ, X') be a local solution to the constraint system $\mathcal{S} = (V, \mathbb{D}, F)$ relative to $X \subseteq V$. Then $\sigma_{X'}$ is a solution to $\mathcal{S}$.*

Proof. We take $\mathbf{x} \in V$ and show $\sigma_{X'}\,\mathbf{x} \sqsupseteq (F_{\mathbf{x}})_{Id}\,\sigma_{X'}$. If $\mathbf{x} \notin X'$, the statement is obvious. In the case $\mathbf{x} \in X'$, by Lemma 1.5.6, it is sufficient to show $\sigma\,\mathbf{x} \sqsupseteq [\![\overline{F}_{\mathbf{x}}]\!]^{*}\sigma$ and $\sigma\,\mathbf{z} = \sigma_{X'}\,\mathbf{z}$, for all $\mathbf{z}$ such that $(\mathbf{z}, \_) \in \mathtt{deps}\,\overline{F}_{\mathbf{x}}\,\sigma$, which both follow from the locality of σ. $\square$

Definition 2.2.11. We say that a partial function

$$\mathcal{A}_{V,\mathbb{D}} : (V \to \textstyle\prod_{T}.(V \to TD) \to TD) \times \mathcal{P}_{fin}(V) \rightharpoonup (V \to D) \times \mathcal{P}_{fin}(V)$$

parametrically polymorphic in V and $\mathbb{D}$ is (a denotational semantics of) a *local solver* if given a constraint system $\mathcal{S} = (V, \mathbb{D}, F)$ over a bounded join-semilattice $\mathbb{D}$ for a set of unknowns V with a pure F, $\mathcal{A}$ when applied to a pair (F, X) for a finite set $X \subseteq V$ of interesting variables yields a local solution (σ, X') of $\mathcal{S}$ relative to X whenever it terminates.

Definition 2.2.12. We say that the solver $\mathcal{A}_{V,\mathbb{D}}$ is a *chaotic iteration solver* if it maintains a mapping $\sigma : V \to D$ and, when computing for a given $\mathcal{S}$, it performs a sequence of updates $\sigma_0, \sigma_1, \ldots$ starting from an initial σ_0 such that the following is true. On every step i, a variable $\mathbf{x} \in V$ is selected which is updated with respect to the current σ_i. That is,

$$\sigma_{i+1}\,\mathbf{y} = \begin{cases} \sigma_i\,\mathbf{x} \sqcup (F_{\mathbf{x}})_{Id}\,\sigma_i & \text{if } \mathbf{x} = \mathbf{y} \\ \sigma_i\,\mathbf{y} & \text{otherwise.} \end{cases}$$

The class of chaotic iteration solvers is important since they allow for using with widening and narrowing operators [CC77a, Cou81] or a combined operator [ASV13].

Erroneous optimization Our experience shows that design of fixpoint algorithms is error-prone.

When starting to reason about the algorithm **RLD** from Figure 2.1, one might feel tempted to avoid to call `solve` for variables whose reevaluation has already been triggered, but has not yet been completed. We note that the meaning of the set *stable* is twofold. First, it contains all the variables $\mathbf{x}$ for which a call `solve` $\mathbf{x}$ has already terminated. Those variables are solved, i.e., their corresponding constraints are satisfied. Second, it contains variables being currently processed, for which reevaluation of right-hand sides is triggered but not yet finished, and corresponding constraints may not be satisfied. Therefore, it seems reasonable to distinguish this kind of “called” variables and prevent them from redundant destabilization since their recomputation is pending.

```
let $\sigma_\bot$ x = if x $\in$ $\mathrm{dom}(\sigma)$ then $\sigma$[x] else $\bot$

let $infl_\emptyset$ x = if x $\in$ $\mathrm{dom}(infl)$ then $infl$[x] else []

let extract_work x =
  let work = $infl_\emptyset$ x in
  infl[x] := [];
  (* do _not_ reevaluate called variables: *)
  let work' = filter ($\lambda$x.x $\notin$ called) work in
  iter ($\lambda$x.stable := stable $\setminus$ {x}) work';
  work'

let rec evalget x y =
  solve y;
  infl[y] := x :: infl[y];
  $\sigma_\bot$ y

and solve x =
  if (x $\notin$ stable) then begin
    stable := stable $\cup$ {x};
    (* mark x as called: *)
    called := called $\cup$ {x};
    let $d$ = $F$ x (evalget x) in
    (* remove x from called: *)
    called := called $\setminus$ {x};
    let cur = $\sigma_\bot$ x in
    let new = cur $\sqcup$ $d$ in
      if (new $\not\sqsubseteq$ cur) then begin
        $\sigma$[x] := new;
        let work = extract_work x in
          solve_all work
      end
  end

and solve_all X = iter solve X

let main X =
  $\sigma$ := $\emptyset$; infl := $\emptyset$; stable := $\emptyset$; called := $\emptyset$;
  solve_all X;
  ($\sigma_\bot$, stable)
```

Figure 2.2.: The erroneous optimization of **RLD**

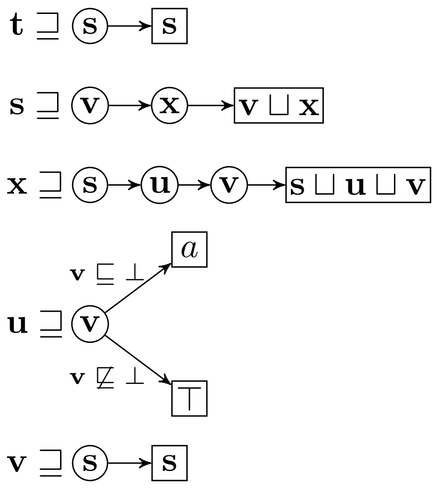

Figure 2.3.: Counterexample for the erroneous optimization

By *destabilization* of a variable we mean removing of the variable from the set of stable variables with subsequent reevaluation of the respective right-hand side. The above observation would lead to the following optimization.

We introduce an extra data structure, the set of variables *called* with an initial value $called_{init} = \emptyset$ (see Fig. 2.2, occurrences of *called* are highlighted). It distinguishes a subset of "stable" variables which are currently being processed. Variable $\mathbf{x}$ is added to *called* just before a reevaluation of $F_{\mathbf{x}}$ starts and gets removed from *called* right after the evaluation returns. The function `extract_work` does *not* destabilize variables from $infl[\mathbf{x}]$ which currently belong to *called*. Thus, recomputation for those variables is not triggered. One can show that $called \subseteq stable$ is invariant for every function of the algorithm, i.e., if $called \subseteq stable$ holds before a function call then the property holds after the call, whenever it terminates.

This optimization appears to be wrong as shown by the counterexample that appears in Fig.2.3. It should be noted however that the counterexample works only if *infl* structure maps variables to *lists* of variables and recording of new dependencies is implemented as

$$infl[\mathbf{y}] := x :: infl[\mathbf{y}]$$

Still, a similar counterexample can be constructed for any other implementation of *infl* (like sets or other kinds of collections). The counterexample defines a constraint system over the three-element lattice $\mathbb{D} = (\{\perp, a, \top\}, \sqsubseteq, \sqcup)$ with $\perp \sqsubset a \sqsubset \top$. Figure 2.3 represents right-hand sides of the constraint system by means of strategy trees. Here, round nodes denote queries to variables while rectangle boxes denote answers expressed in terms of constants and received values of queried variables. Conceptually, query nodes have one outgoing edge for every value of $\mathbb{D}$ corresponding to every possible received value for the variable. In the figure, however, we merge edges with equal subtrees for the sake of simplicity. In the example, the right-hand sides could be represented in a ML-like language by

$$\begin{aligned}
F_{\mathbf{t}}\, k &= \texttt{let}\ s_1 = k\,\mathbf{s}\ \texttt{in}\ s_1\\
F_{\mathbf{s}}\, k &= \texttt{let}\ v_1 = k\,\mathbf{v}\ \texttt{in let}\ x_1 = k\,\mathbf{x}\ \texttt{in}\ v_1 \sqcup x_1\\
F_{\mathbf{x}}\, k &= \texttt{let}\ s_1 = k\,\mathbf{s}\ \texttt{in let}\ u_1 = k\,\mathbf{u}\ \texttt{in let}\ v_1 = k\,\mathbf{v}\ \texttt{in}\ s_1 \sqcup u_1 \sqcup v_1\\
F_{\mathbf{u}}\, k &= \texttt{let}\ v_1 = k\,\mathbf{v}\ \texttt{in if}\ (v_1 \sqsubseteq \bot)\ \texttt{then}\ a\ \texttt{else}\ \top\\
F_{\mathbf{v}}\, k &= \texttt{let}\ s_1 = k\,\mathbf{s}\ \texttt{in}\ s_1
\end{aligned}$$

Let us trace the computations done by the solver when `solve_all [t]` is called from the initial state (the full trace can be found in Appendix B.1).

The algorithm calls `solve t` which in turn recursively calls `solve s`. During the run of `solve s`, the algorithm recursively computes new values of variables **v**, **x**, and **u** in that order. For the sake of brevity, we skip a description of those steps (cf. lines 1–54 in Appendix B.1), but we note that they lead to a change in $\sigma[\mathbf{s}]$. Before the new value of $\sigma[\mathbf{s}] = a$ is returned by `solve s`, the algorithm recomputes all the variables dependent on **s**. These are variables from $\mathit{infl}[\mathbf{s}] = \texttt{[x; v]}$. Thus, the algorithm resets $\mathit{infl}[\mathbf{s}]$ to `[]` and removes both **x** and **v** from *stable* and *called* (lines 60–64). The state prior to the call `solve_all [x; v]` is

$$\begin{aligned}
\sigma\ &= \{\mathbf{s} \mapsto a, \mathbf{u} \mapsto a, \mathbf{x} \mapsto a\}\\
\mathit{infl}\ &= \{\mathbf{u} \mapsto \texttt{[x]}, \mathbf{v} \mapsto \texttt{[x; u; s]}, \mathbf{x} \mapsto \texttt{[s]}\}\\
\mathit{stable}\ &= \{\mathbf{s}, \mathbf{t}, \mathbf{u}\}\\
\mathit{called}\ &= \{\mathbf{t}\}
\end{aligned}$$

1. `solve x` is invoked, and the variable **x** is put back into the sets *stable* and *called.* The state prior to reevaluation of the right-hand side $F_{\mathbf{x}}$ is

$$\begin{aligned}
\sigma\ &= \{\mathbf{s} \mapsto a, \mathbf{u} \mapsto a, \mathbf{x} \mapsto a\}\\
\mathit{infl}\ &= \{\mathbf{u} \mapsto \texttt{[x]}, \mathbf{v} \mapsto \texttt{[x; u; s]}, \mathbf{x} \mapsto \texttt{[s]}\}\\
\mathit{stable}\ &= \{\mathbf{s}, \mathbf{t}, \mathbf{u}, \mathbf{x}\}\\
\mathit{called}\ &= \{\mathbf{t}, \mathbf{x}\}
\end{aligned}$$

During the evaluation of $F_{\mathbf{x}}$ (`evalget x`) the algorithm traverses the tree $\overline{F}_{\mathbf{x}}$ and tries to solve variables **s**, **u** and **v** in turn.

- Since $\mathbf{s}, \mathbf{u} \in \mathit{stable}$, the algorithm does not descend into solving them. The structures σ, *stable* and *called* are not changed, but the solver records that **x** depends on **s** and **u**, i.e., **x** is added to $\mathit{infl}[\mathbf{s}]$ and $\mathit{infl}[\mathbf{u}]$ (lines 71–75).
- The algorithm recomputes **v** (a call to `solve v`), that gets a large value a since $\sigma[\mathbf{s}] = a$ (lines 76–88). Since $\sigma[\mathbf{v}]$ has increased, variables influenced by **v** must be recomputed. These are variables from $\mathit{infl}[\mathbf{v}] = \texttt{[x; u; s]}$. However, the algorithm does *not* destabilize **x** since $\mathbf{x} \in \mathit{called}$ at the moment (lines 89–93). Thus, the variable **u** is destabilized and recomputed (since $\mathbf{u} \in \mathit{stable} \setminus \mathit{called}$) and gets a

greater value $\top$ (lines 95–107). At this point, although *infl*$[\mathbf{u}] = [\mathbf{x};\mathbf{x}]$, the variable $\mathbf{x}$ is again *not* recomputed since $\mathbf{x} \in$ *called* (line 108). Thus, the value of $\mathbf{x}$ remains as before, $\sigma[\mathbf{x}] = a$. Then $\mathbf{s}$ gets recomputed, but this does not lead to a change in the state (lines 115–120). Finally, `solve` $\mathbf{s}$ returns (line 121).

2. `solve` $\mathbf{v}$ is called, but $\mathbf{v} \in$ *stable* $\setminus$ *called* at this moment (lines 125–126).

Finally, the algorithm returns the variable assignment

$$\sigma_1 = \{\mathbf{s} \mapsto a, \mathbf{t} \mapsto a, \mathbf{u} \mapsto \top, \mathbf{v} \mapsto a, \mathbf{x} \mapsto a\}$$

which is not a solution since the constraint for $\mathbf{x}$ is not satisfied, as

$$\sigma_1\,\mathbf{x} = a \sqsubset \top = (F_{\mathbf{x}})_{\mathit{Id}}\,\sigma_1.$$

The example demonstrates how small, seemingly correct, modifications of the algorithm may lead to subtle errors. The erroneously optimized version was used for some time in practice, and was featured in the draft of the book [SWH10]. The counterexample was one of our guiding motivations for a rigorous verification of the fixpoint algorithm **RLD**.

Functional implementation In the functional implementation of algorithm **RLD**, the global state is made explicit, and passed into function calls by means of a separate parameter. Accordingly, the modified state together with the computed value (if there is any) are jointly returned. The type of a state is

$$\texttt{type state} = (V \rightharpoonup D) \times (V \rightharpoonup \texttt{list}\,V) \times \mathcal{P}_{\mathit{fin}}(V)$$

where `list` V is a type of lists of elements of V and $\mathcal{P}_{\mathit{fin}}(V)$ is a type of finite sets of elements of V. The three components correspond to the finite (partial) map σ, the finite (partial) map *infl*, and the set *stable* of the imperative implementation, respectively. We implement and verify the version of **RLD** with *infl* mapping variables to lists of variables.

To facilitate the state handling, we introduce the following auxiliary functions:

– `getval` : `state` $\to V \to D$ implements the function $\sigma_\perp$;

– `setval` : $V \to D \to$ `state` $\to$ `state` when applied to $\mathbf{x}$ and d updates the current value of $\sigma\,[\mathbf{x}]$ with d;

– `get_stable` : `state` $\to \mathcal{P}_{\mathit{fin}}(V)$ extracts the component *stable* of a given state;

– `is_stable` : $V \to$ `state` $\to$ `bool` checks if a given variable $\mathbf{x}$ is in *stable*;

– `add_stable` : $V \to$ `state` $\to$ `state` adds a given variable to *stable*;

– `rem_stable` : $V \to$ `state` $\to$ `state` removes a given variable from *stable*;

– `get_infl` : $V \to$ `state` $\to$ `list` V implements the function $\mathit{infl}_\emptyset$;

```
let extract_work x = fun $s$ →
  let $w$ = get_infl x $s$ in
  let $s_0$ = rem_infl x $s$ in
  let $s_1$ = foldl (fun $s$ y →rem_stable y $s$) $s_0$ $w$ in
    ($w$, $s_1$)

let rec evalget x y : $State_{\mathsf{state}}\, D$ = fun $s$ →
  let $s_0$ = solve y $s$ in
  let $s_1$ = add_infl y x $s_0$ in
    (getval $s_1$ y, $s_1$)

and solve x = fun $s$ →
  if is_stable x $s$ then $s$ else
    let $s_0$ = add_stable x $s$ in
    let ($d$, $s_1$) = $F$ x (evalget x) $s_0$ in
    let cur = getval $s_1$ x in
    let new = cur $\sqcup$ $d$ in
      if (new $\sqsubseteq$ cur) then $s_1$ else
        let $s_2$ = setval x new $s_1$ in
        let ($w$, $s_3$) = extract_work x $s_2$ in
          solve_all $w$ $s_3$

and solve_all $w$ = fun $s$ →
  match $w$ with
  | [] → $s$
  | x :: xs → solve_all (solve x $s$) xs

let main X =
  let $s_{init}$ = $(\emptyset, \emptyset, \emptyset)$ in
  let $s$ = solve_all X $s_{init}$ in
  (getval $s$, get_stable $s$)
```

Figure 2.4.: Functional implementation of **RLD** with explicit state passing

– `add_infl` : $V \to V \to$ `state` $\to$ `state` when applied to variables $\mathbf{y}$ and $\mathbf{x}$ updates the *infl*-component of a state as $\mathit{infl}[\mathbf{y}] := \mathbf{x} :: \mathit{infl}[\mathbf{y}]$;

– `rem_infl` : $V \to$ `state` $\to$ `state` when applied to a variable $\mathbf{x}$ resets the component $\mathit{infl}[\mathbf{x}]$ in a given state to `[]`.

The auxiliary function `extract_work` : $V \to$ `state` $\to$ `list` $V \times$ `state` applied to a variable $\mathbf{x}$ puts $w = \mathit{infl}[\mathbf{x}]$ that stores (an over-approximation of) a set of variables immediately influenced by $\mathbf{x}$ relative to a current variable assignment, resets $\mathit{infl}[\mathbf{x}]$ to `[]`, and subtracts w from the component *stable* of a given state. The mutually recursive functions `evalget`, `solve` and `solve_all` of the algorithm are then given in Figure 2.4. Provided a list of interesting variables $X \subseteq V$, the algorithm calls the function `solve_all` from the initial state $s_{init} = (\emptyset, \emptyset, \emptyset)$.

From now on, **RLD** refers to this functional implementation. Our goal is to prove

Theorem 2.2.13. *The algorithm* **RLD** *is a local solver.*

The proof argument consists of the four main steps.

1. Implementation of the functional program in Coq.

2. Instrumentation of the functional program by means of auxiliary data structures — ghost variables.

3. Providing (strong enough) invariants for the instrumented program.

4. Deduction of the correctness statement from invariants.

Step 1. Formalization in Coq The implementation of the algorithm in Coq is not straightforward since Coq does *not* support a general recursion. The underlying logic of Coq (Calculus of Inductive Constructions) only supports recursion over inductively defined types that generalizes the primitive recursion. The recursive definition is accepted by Coq only if it is *provably* terminating.

As generality is a desirable property of **RLD**, we neither make any assumptions concerning the semilattice $\mathbb{D}$ (e.g., with respect to the ascending chain property), nor do we assume finiteness of the set of variables V. Therefore, termination of the algorithm cannot be guaranteed in general. However, even having those assumed, it is not clear how one could implement the algorithm in the general setting. For example, it is definitely not possible to give a straightforward implementation using the standard `Fixpoint` tool since the definition by `Fixpoint` must provide a *structurally* decreasing argument. Another opportunity could be in use of the `Function` command which generalizes `Fixpoint` [Coq12]. It still requires a decreasing argument, but not necessarily a structurally decreasing one. Instead, a *measure* or a *well-founded* relation must be provided that guarantees termination of all recursive calls. In other words, `Function` provides Noetherian induction over well-founded ordered types. As we will see in Subsection 2.2.5, it appears to be possible to define such a well-founded relation along in the context of necessary assumption, but some more problems arise at this point. Currently, `Function` does not support mutually defined functions. Even if we unfold bodies of all the functions except of `solve` and try to pass explicitly the function

```
fun s →
  let s0 = solve y s in
  let s1 = add_infl y x s0 in
    (getval s1 y, s1)
```

as an argument to a right-hand side $F_{\mathtt{x}}$, Coq complains again since `Function` does *not* allow for λ-expressions with recursive calls inside. The presence of mutually inductive definitions makes the algorithm very hard to implement in the general setting. Despite being possible theoretically, such implementation would also significantly complicate the formal reasoning.

In view of the mentioned difficulties, our formalization of the algorithm in Coq relies on a representation of partial functions through their graphs. For that, we define an interpreter $[\![\cdot]\!]^{\#}_{\mathtt{state}}$ of strategy trees operating on graphs of stateful functions. Recall that we consider relations as function to `Prop`.

Definition 2.2.14. Given a strategy tree t and a graph k of a partial function of type $A \to \mathit{State}_{\mathtt{state}} B$, i.e., k is of type $A \to S \to B \times S \to \mathtt{Prop}$, $[\![t]\!]^{\#}_{\mathtt{state}}\, k$ defines a graph of a function of type $\mathit{State}_{\mathtt{state}} C$ inductively by

– $[\![\mathsf{Ans}\, c]\!]^{\#}_{\mathtt{state}}\, k\, s\, (c, s)$ for all $s : \mathtt{state}$, $c : C$;

– if $k\, a\, s\, (b, s_1)$ and $[\![f\, b]\!]^{\#}_{\mathtt{state}}\, k\, s_1\, (c, s_2)$ then $[\![\mathsf{Que}\, a\, f]\!]^{\#}_{\mathtt{state}}\, k\, s\, (c, s_2)$, for all $s, s_1, s_2 : \mathtt{state}$, $a : A$, $b : B$, $c : C$, $f : B \to \mathit{Tree}_{A,B,C}$.

The following mutual recursive definitions of relations mimic the functional implementation of the algorithm. Since well-formed inductive definition in Coq must satisfy the positivity condition, we implement a separate relation for the partially applied function `evalget x`. Thus, for `evalget` we have

```
Inductive EvalGet :
  Var.t -> Var.t -> state -> D.t * state -> Prop :=
  | EvalGet0 :
      forall x y s s0 s1 d,
        Solve y s s0 ->
        s1 = add_infl y x s0 ->
        d = getval s0 y ->
        EvalGet x y s (d, s1)

with EvalGet_x :
  Var.t -> (Var.t -> state -> D.t * state -> Prop) -> Prop :=
  | EvalGet_x0 :
      forall x (f : Var.t -> state -> D.t * state -> Prop),
            (forall y s0 ds1,
               f y s0 ds1 -> EvalGet x y s0 ds1) ->
            EvalGet_x x f
```

`EvalGet` defines a graph of `evalget`. `EvalGet_x` x f holds if f is a subrelation of `EvalGet` x.

```
    with Wrap_Eval_x :
      Var.t -> (Var.t -> state -> D.t * state -> Prop) ->
      @Tree Var.t D.t D.t ->
      state -> D.t * state -> Prop :=
      | Wrap_Eval_x0 :
        forall x f t s0 ds1,
          EvalGet_x x f ->
          [[t]]# f s0 ds1 ->
          Wrap_Eval_x x f t s0 ds1

    with Eval_rhs :
      Var.t ->
      state -> D.t * state -> Prop :=
      | Eval_rhs0 :
        forall x f s0 ds1,
          EvalGet_x x f ->
          Wrap_Eval_x x f (rhs x) s0 ds1 ->
          Eval_rhs x s0 ds1
```

To simulate the evaluation of a right-hand side, we implement two relations `Wrap_Eval_x` and `Eval_rhs`. The former implements reevaluation of $[\![t]\!]^{\#}_{\texttt{state}}$ for arbitrary $t : \mathit{Tree}_{V,D,D}$ and f, which is a subrelation of the graph of stateful `evalget` x. The latter obtained by substitution of t by `rhs` x, a strategy tree for F_x, which is pure by assumption. The separation into `Wrap_Eval_x` and `Eval_rhs` is needed for proof development purposes. We construct stronger invariants for `Wrap_Eval_x` that can be proven by induction on t and deduce the necessary invariant for `Eval_rhs`. Note that in the definition of `Eval_rhs` we universally quantify over all subrelations f of `EvalGet` x and thus, `EvalGet` x itself is included, clearly. The direct application of $[\![t]\!]^{\#}$ to (`EvalGet` x) were not possible here since it would violate the positivity condition for `EvalGet`, as we mentioned before.

The relation for functions `solve` and `solve_all` are defined by

```
    with Solve :
      Var.t -> state -> state -> Prop :=
      | Solve0 :
          forall x s, is_stable x s -> Solve x s s
      | Solve1 :
          forall x d s s2,
          ~ is_stable x s ->
          let s1 := prepare x s in
          Eval_rhs x s1 (d, s2) ->
          let cur := getval s2 x in
          let new := D.join cur d in
          D.Leq new cur ->
          Solve x s s2
      | Solve2 :
          forall x d s s2 s5 s6 work,
          ~ is_stable x s ->
          let s1 := prepare x s in
          Eval_rhs x s1 (d, s2) ->
          let cur := getval s2 x in
          let new := D.join cur d in
          ~ D.Leq new cur ->
          let s4 := setval x new s2 in
          (work, s5) = extract_work x s4 ->
          SolveAll work s5 s6 ->
          Solve x s s6

    with SolveAll :
      list Var.t -> state -> state -> Prop :=
      | SolveAll0 :
          forall s, SolveAll [] s s
      | SolveAll2 :
          forall x xs s s1 s2,
            Solve x s s1 ->
            SolveAll xs s1 s2 ->
            SolveAll (x :: xs) s s2.
```

`Solve` and `SolveAll` have separate constructors for each `if`-branch and `match`-case in `solve` and `solve_all`. The helper function `prepare` adds variable **x** to *stable*

```
    Definition prepare x s := add_stable x s.
```

The function `extract_work` (with the helper function `handle_work`) extracts a list of variables that are subject for reevaluation

```
    Definition handle_work (w : list Var.t) (s : state) :=
        let f s x := rem_stable x s in
        List.fold_left f w s.

    Definition extract_work x (s : state) : (list Var.t * state)
      := let w := get_infl s x in
         let s := rem_infl x s in
         let s := handle_work w s in
           (w, s).
```

The defined predicates relate a state before the call of a respective function with a value (if there is any) and a state after the function returns. Therefore, they can be used to reason about properties of the algorithm even if its termination is not guaranteed, such as partial correctness. We show that each of the defined relations is indeed a graph of a partial function. By structural induction on definition of predicates we prove

Lemma 2.2.15. *The relations* `EvalGet`, `Wrap_Eval_x`, `Eval_rhs`, `Solve`, `SolveAll` *are graphs of partial functions, i.e.,*

$$\begin{aligned}
&(\forall x, y, s, p, p'.\, \texttt{EvalGet}\ x\ y\ s\ p \wedge \texttt{EvalGet}\ x\ y\ s\ p' \to p = p') \wedge \\
&(\forall x, y, f, f', s, p, p'. \\
&\qquad \texttt{EvalGet\_x}\ x\ f \wedge \texttt{EvalGet\_x}\ x\ f' \wedge f\, y\, s\, p \wedge f'\, y\, s\, p' \to p = p') \wedge \\
&(\forall x, f, f', t, s, p, p'.\, \texttt{Wrap\_Eval\_x}\ x\ f\ t\ s\ p \wedge \texttt{Wrap\_Eval\_x}\ x\ f'\ t\ s\ p' \wedge \\
&\quad (\forall y, s_1, p_1, p_1', f\ y\ s_1\ p_1 \wedge f'\ y\ s_1\ p_1' \to p_1 = p_1') \to p = p') \wedge \\
&(\forall x, s, p, p'.\, \texttt{Eval\_rhs}\ x\ s\ p \wedge \texttt{Eval\_rhs}\ x\ s\ p' \to p = p') \wedge \\
&(\forall x, s, s_1, s_1'.\, \texttt{Solve}\ x\ s\ s_1 \wedge \texttt{Solve}\ x\ s\ s_1' \to s_1 = s_1') \wedge \\
&(\forall w, s, s_1, s_1'.\, \texttt{SolveAll}\ w\ s\ s_1 \wedge \texttt{SolveAll}\ w\ s\ s_1' \to s_1 = s_1').
\end{aligned}$$

Proof. By induction on structure of relations. □

Step 2. Instrumentation When trying to capture the behaviour of the solver by means of invariants, we run into difficulties to formulate invariants strong enough for proving the partial correctness. The internal logic of the algorithm is hard to express in terms of the values of structures presented in the implementation alone. Our solution is to introduce additional components into the `state` that do not affect the operational behaviour of the algorithm but store an auxiliary information. The auxiliary data structures appear in the program as *ghost* structures, i.e., they are not allowed to appear in case distinctions or pattern matching constructions and may not be written into ordinary structures. Thus, they influence neither the flow of control of the program nor the values of usual structures but allow to express extra-functional properties of behaviour of the program. The technique of introducing auxiliary ghost structures originally appeared for verification of parallel programs [OG76, Han73]. Currently they are used also in other application areas, e.g., in the specification language JML [PBB$^+$04].

We introduce two auxiliary data structures: the sets *called* and *queued* of variables. Intuitively, *called* stores a subset of variables from *stable* currently being processed, for which reevaluation by `solve` has been started and not yet returned. The structure *queued* stores a set of variables that have been destabilized (queued for reevaluation), i.e., removed from the set *stable* by the algorithm, and have not yet been reevaluated by `solve`. Thus, sets *stable* and *queued* have no common elements. The union of the sets gives a set of all the variables ever touched by the algorithm.

Accordingly, the type `state′` for the instrumented implementation is

$$\texttt{type state}' = (V \rightharpoonup D) \times (V \rightharpoonup \texttt{list}\, V) \times \mathcal{P}_{\mathit{fin}}(V) \times \mathcal{P}_{\mathit{fin}}(V) \times \mathcal{P}_{\mathit{fin}}(V)\,.$$

The five components correspond to the finite (partial) map σ, the finite (partial) map *infl*, and the sets *stable*, *called*, and *queued*, respectively. We shall use a record notation to denote components of states, e.g., $s.\mathit{stable}$, $s.\mathit{called}$ etc. We introduce the following auxiliary functions and relations:

- `add_called` : $V \to$ `state'` $\to$ `state'` adds a given variable to *called*;
- `rem_called` : $V \to$ `state'` $\to$ `state'` removes a given variable from *called*;
- `add_queued` : $V \to$ `state'` $\to$ `state'` adds a given variable to *queued*;
- `rem_queued` : $V \to$ `state'` $\to$ `state'` removes a given variable from *queued*;
- `is_called` : $V \to$ `state'` $\to$ `Prop` checks if a given variable is in *called*;
- `is_solved` : $V \to$ `state'` $\to$ `Prop` checks if a given variable is in $(\mathit{stable} \setminus \mathit{called})$.

The last two relations will be used in definitions of invariants.

To capture the information about evaluations of right-hand sides, we provide `instrR` that is a relational version of `instr` defined for graphs of stateful functions of type $A \to \mathit{State}_S B$.

Definition 2.2.16. For sets A, B, S and a relation $f : A \to S \to B \times S \to$ `Prop`, we define `instrR` $f : A \to (S \times \texttt{list}\,(A \times B)) \to B \times (S \times \texttt{list}\,(A \times B)) \to$ `Prop` by

$$(\texttt{instrR}\, f)\, a\, (s, l)\, (b, (s_1, l \mathbin{++} [(a, b)])) \quad \text{iff} \quad f\, a\, s\, (b, s_1)$$

for all a, b, s, s_1, l of respective types.

In the instrumented implementation, we use the relational monadic tree interpreter instantiated with $\mathit{State}_{\texttt{state'} \times \texttt{list}\,(V \times D)}$. If $[\![t]\!]^{\#}_{\texttt{state'} \times \texttt{list}\,(V \times D)}\, (\texttt{instrR}\, f)\, (s, l)\, (d, (s_1, l_1))$ holds then l_1 equals l suffixed with all the variables accessed by f along with the respective values returned by f, in the order they were met.

The instrumented relations `Wrap_Eval_x` and `Eval_rhs` are given below. The changes relative to non-instrumented versions are indicated by the comments.

```
(* ... *)
with Wrap_Eval_x :
  Var.t -> (Var.t -> state' -> D.t * state' -> Prop) ->
  @Tree Var.t D.t D.t ->
  state' * list (Var.t * D.t) ->
  D.t * (state' * list (Var.t * D.t)) -> Prop :=
  | Wrap_Eval_x0 :
    forall x f t sl dsl0,
      EvalGet_x x f ->
      (* use instrumented f *)
      let f' := instrR f in
```

```
          (* it additionally returns a list of visited variables... *)
          (* along with their values: *)
          [[t]]# f' sl dsl0 ->
        Wrap_Eval_x x f t sl dsl0

  with Eval_rhs :
    Var.t ->
    state' -> D.t * (state' * list (Var.t * D.t)) -> Prop :=
    | Eval_rhs0 :
      forall x f s dsl0,
        EvalGet_x x f ->
        (* the list of accessed variables is initially [] *)
        Wrap_Eval_x x f (rhs x) (s,[]) dsl0 ->
        Eval_rhs x s dsl0
  (* ... *)
```

The instrumented function `extract_work` for a given $\mathbf{x} : V$ additionally removes all the variables influenced by $\mathbf{x}$ from the set *called* and adds them to the set *queued* for the current state, as defined below.

```
  Definition handle_work (w : list Var.t) (s : state') :=
      let f s x :=
        (* remove x from called: *)
        let s := rem_called x s in
        let s := rem_stable x s in
        (* add x to queued: *)
        let s := add_queued x s in
        s in
      List.fold_left f w s.

  Definition extract_work (x : Var.t) (s : state')
    : (list Var.t * state')
    := let w := get_infl s x in
       let s := rem_infl x s in
       let s := handle_work w s in
         (w, s).
```

For the instrumented `Solve`, a given $\mathbf{x} : V$ is removed from *queued* and added to *called* right before the reevaluation of the right-hand side for $\mathbf{x}$, and gets removed from *called* right after the reevaluation terminates.

```
  (* ... *)
  with Solve :
    Var.t -> state' -> state' -> Prop :=
    | Solve0 :
          forall x s, is_stable x s -> Solve x s s
      | Solve1 :
          forall x d s s2 ps,
          ~ is_stable x s ->
          (* remove from x queued and add it to called: *)
          let s1 := prepare x s in
          Eval_rhs x s1 (d, (s2, ps)) ->
          (* remove x from called: *)
          let s3 := rem_called x s2 in
          let cur := getval s3 x in
          let new := D.join cur d in
          D.Leq new cur ->
          Solve x s s3
      | Solve2 :
          forall x d s s2 s5 s6 ps work,
          ~ is_stable x s ->
          (* remove x from queued and add it to called: *)
          let s1 := prepare x s in
          Eval_rhs x s1 (d, (s2, ps)) ->
          (* remove x from called: *)
          let s3 := rem_called x s2 in
          let cur := getval s3 x in
          let new := D.join cur d in
          ~ D.Leq new cur ->
          let s4 := setval x new s3 in
          (* add destabilized variables to queued and *)
          (* remove them from called and stable: *)
          (work, s5) = extract_work x s4 ->
          SolveAll work s5 s6 ->
          Solve x s s6
    (* ... *)
```

where

```
    Definition prepare x s :=
      let s1 := rem_queued x s in
      let s2 := add_stable x s1 in
      let s3 := add_called x s2 in
        s3.
```

All the rest definitions remain unchanged. As in the non-instrumented case, we prove by structural induction the following technical lemma.

Lemma 2.2.17. *The instrumented relations* `EvalGet`, `Wrap_Eval_x`, `Eval_rhs`, `Solve`, `SolveAll` *are graphs of partial functions.* □

It is intuitively clear that the instrumentation does not alter the relevant behaviour of the algorithm and thus, the subsequent verification of the instrumented version also establishes the correctness of the original one. Below, we provide a rigorous proof of that statement.

For the rest of this paragraph let us use a primed notation, e.g., $\mathtt{state}'$, $\mathtt{Solve}'$ etc. for names of the instrumented versions while using unprimed ones for the original versions of structures and relations. First, we define the simulation relation

$$\sim\subseteq \mathtt{state} \times \mathtt{state}'$$

as the graph of the projection from $\mathtt{state}'$ to $\mathtt{state}$. We will use infix notation for $\sim$ below. Formally, $s \sim s'$ iff $s.\sigma = s'.\sigma$, $s.\mathit{infl} = s'.\mathit{infl}$ and $s.\mathit{stable} = s'.\mathit{stable}$ hold. For graphs of stateful functions $f : X \to \mathtt{state} \to Y \times \mathtt{state} \to \mathtt{Prop}$ and $f' : X \to \mathtt{state}' \to Y \times \mathtt{state}' \to \mathtt{Prop}$ we define a lifted simulation by

$$\begin{aligned} f \sim^{\#} f' \quad \text{iff} \quad & \forall x, s, s', y, s_1.\, s \sim s' \wedge f\, x\, s\, (y, s_1) \to \\ & \exists y', s_1'.\, f'\, x\, s'\, (y', s_1') \wedge y = y' \wedge s_1 \sim s_1'. \end{aligned}$$

Then, we show by induction that each component of the algorithm simulates its instrumented version. Thus, they yield equal results when started in related states after discarding the instrumentation. The following lemma holds.

Lemma 2.2.18 (simulation)**.**

$$\begin{aligned}
&(\forall x, y, s, s', s_1, d.\, \mathit{EvalGet}\; x\; y\; s\; (d, s_1) \wedge s \sim s' \to \\
&\qquad \exists d', s_1'.\, \mathit{EvalGet}'\; x\; y\; s'\; (d', s_1') \wedge d = d' \wedge s_1 \sim s_1') \wedge \\
&(\forall x, f.\, \mathit{EvalGet\_x}\; x\; f \to \exists f'.\, \mathit{EvalGet\_x}'\; x\; f' \wedge f \sim^{\#} f') \wedge \\
&(\forall x, t, s, s', d, s_1, l'.\, \mathit{Wrap\_Eval\_x}\; x\; t\; s\; (d, s_1) \wedge s \sim s' \to \\
&\qquad \exists d', s_1', l_1'.\, \mathit{Wrap\_Eval\_x}'\; x\; t\; (s', l')\; (d', (s_1', l_1')) \wedge d = d' \wedge s_1 \sim s_1') \wedge \\
&(\forall x, s, s', d, s_1.\, \mathit{Eval\_rhs}\; x\; s\; (d, s_1) \wedge s \sim s' \to \\
&\qquad \exists d', s_1', l'.\, \mathit{Eval\_rhs}'\; x\; s'\; (d', (s_1', l')) \wedge d = d' \wedge s_1 \sim s_1') \wedge \\
&(\forall x, s, s', s_1.\, \mathit{Solve}\; x\; s\; s_1 \wedge s \sim s' \to \exists s_1'.\, \mathit{Solve}'\; x\; s'\; s_1' \wedge s_1 \sim s_1') \wedge \\
&(\forall w, s, s', s_1.\, \mathit{SolveAll}\; w\; s\; s_1 \wedge s \sim s' \to \exists s_1'.\, \mathit{SolveAll}'\; w\; s'\; s_1' \wedge s_1 \sim s_1')
\end{aligned}$$

Proof. By induction on structure of relations. □

Step 3. Invariants Before formulating invariants, we first provide several technical lemmas that relate finite sequences of pairs of $A \times B$ with functions $f : A \to B$ and traces generated by f in a tree $t \in \mathit{Tree}_{A,B,C}$, for sets A, B, C. Remind a definition of the function $\mathtt{subtree} : \mathit{Tree}_{A,B,C} \to \mathtt{list}\,(A \times B) \to \mathit{Tree}_{A,B,C}$. When applied to t and ps, $\mathtt{subtree}\; t\, \mathit{ps}$ yields a subtree of t which is obtained if we use B-components of elements of ps as choice values for Que-branchings.

```
Fixpoint subtree (t : Tree A B C) (ps : list (A * B))
  : Tree A B C :=
  match t, ps with
    | _, nil => t
    | Ans _, _ => t
    | Que x k, (_, b) :: r => subtree (k b) r
  end.
```

Further, we refer to finite sequences of type `list` $(A \times B)$ as *paths*. We say that a path ps : `list` $(A \times B)$ is *valid* for $f : A \to B$ if it agrees with f on all its elements.

```
Definition valid (f : A -> B) (ps : list (A * B)) : Prop :=
  forall p, In p ps ->
    let (a, b) := p in b = f a.
```

We say that a path ps : `list` $(A \times B)$ is *legal* for $t \in \mathit{Tree}_{A,B,C}$ if t and ps satisfy the following predicate.

```
Fixpoint legal (t : Tree A B C) (ps : list (A * B)) : Prop :=
  match t, ps with
    | _, nil => True
    | Ans _, _ :: _ => False
    | Que x k, (a, b) :: r => a = x /\ legal (k b) r
  end.
```

Intuitively, `legal` $t\,ps$ expresses that one can walk in t along ps (always staying within t), for every $(a, b) \in ps$ using the value b as an argument of the Que-branching function. For example, for the tree $t \in \mathit{Tree}_{\{\mathbf{x},\mathbf{y}\},\mathbb{N},\mathbb{N}}$ defined by

$$t = \mathsf{Que}\,\mathbf{x}\,(\lambda x.\texttt{if}\ x = 0\ \texttt{then}\ (\mathsf{Ans}\ 1)\ \texttt{else}\ (\mathsf{Que}\,\mathbf{y}\,(\lambda y.\mathsf{Ans}\ y)))$$

paths `[]`, `[`$(\mathbf{x}, 0)$`]`, `[`$(\mathbf{x}, 1), (\mathbf{y}, 0)$`]` are legal, while `[`$(\mathbf{x}, 0), (\mathbf{y}, 0)$`]` is not.

We show that traces generated by `deps` are valid and legal paths leading to Ans-leaves.

Lemma 2.2.19. *For all $t : \mathit{Tree}_{A,B,C}$ and $f : A \to B$, the following holds*

- *valid f (deps $t\,f$);*
- *legal t (deps $t\,f$);*
- *subtree t (deps $t\,f$) $=$ Ans $([\![t]\!]^* f)$.*

Proof. By induction on t. □

The reverse is also true as shown by the following lemma.

Lemma 2.2.20. *Given $t \in \mathit{Tree}_{A,B,C}$, ps : list $(A \times B)$, $f : A \to B$, and $c : C$, if valid $f\,ps$, legal $t\,ps$ and subtree $t\,ps$ $=$ Ans c hold then $ps =$ deps $t\,f$ and $c = [\![t]\!]^* f$.*

Proof. By induction on *ps*. □

In what follows, we construct invariants for every component of the instrumented implementation. We start first with some simple specifications.

```
Definition Inv_0 (s : state') : Prop :=
  let '(sigma, infl, stable, called, queued) := s in
    VS.Subset called stable /\
    VS.Empty (VS.inter stable queued).
```

`Inv_0` asserts that in the state s, the set *called* is a subset of *stable* and sets of *stable* and *queued* variables do not intersect. By structural induction, one can show that `Inv_0` is indeed invariant for all the components of the algorithm. For instance, we have that if `Inv_0` s holds for s : `state`$'$ then `Solve` $x\,s\,s'$ implies `Inv_0` s', for all $x : V$, $s' : state'$, and similarly for other components of the solver.

The following specification relates the three state components, *stable*, *called* and *queued*, before and after a terminating function call.

```
Definition Inv_1 (s s' : state') : Prop :=
  let '(_, _, stable, called, queued) := s in
  let '(_, _, stable', called',  queued') := s' in
    VS.Subset stable  stable' /\
    VS.Subset called' called  /\
    VS.Subset queued' queued.
```

Merely, `Inv_1` asserts that the set *stable* always grows, and *called* and *queued* components may only shrink. We call a state s (a transition from s to s') *consistent* if `Inv_0` s (respectively, `Inv_1` $s\,s'$) holds. Again, one can show by induction that all main functions of the algorithm modify states consistently, i.e., `Inv_1` is a valid transition relation for every main function of the algorithm. Since $s_{init}.called = s_{init}.queued = \emptyset$, we have $s.called = s.queued = \emptyset$ for any $x : V$ and s : `state` such that `Solve` $x\,s_{init}\,s$. However, `Inv_0` and `Inv_1` alone are not sufficient to prove correctness of the solver since they do not speak about σ and *infl* components. As we will see, to relate all the components is a non-trivial task indeed.

In order to provide strong invariants for the algorithm, we define several more auxiliary relations. The relation

```
Definition Inv_corr (s : state') : Prop :=
  let sigma := getval s in
    (forall x : Var.t,
       is_solved x s ->
       let val := [[rhs x]]* sigma in
       D.Leq val (sigma x)) /\
    (forall (x v : Var.t) d,
       is_solved x s ->
```

```
           In (v,d) (deps (rhs x) sigma) ->
           In x (get_infl s v) /\
           (is_stable v s \/ is_queued v s)).
```

specifies which states can be considered as correct. Remind that `is_solved` $x\,s$ abbreviates $x \in s.stable \setminus s.called$. The first conjunct of `Inv_corr` asserts that for every variable x that is "solved" in s the constraint for x is satisfied, i.e., `getval` $s\,x \sqsupseteq [\![$`rhs` $x]\!]^*($`getval` $s)$ holds (hence the name *solved*). The second conjunct states how dependencies of x are treated by the solver. Namely, it assert that $\mathit{infl}_\emptyset x$ over-approximates the set of actual dependencies of x returned by `deps` (`rhs` x) (`getval` s). Moreover, every v influencing x relative to `getval` s must be either marked as stable or as queued. Thus, reasoning about dependencies is made possible with respect to a set of destabilized variables distinguished by means of the auxiliary data structure *queued*.

The relation

```
Definition Inv_corr_var (s : state') x :=
  let '(_, _, stable, _, queued) := s in
  let sigma := getval s in
    forall v d,
      In (v,d) (deps (rhs x) sigma) ->
      In x (get_infl s v) /\
      (is_stable v s \/ is_queued v s).
```

specifies that dependencies of x are treated correctly by the solver. It partially repeats the second conjunct of `Inv_corr`, but the point is that `is_solved` $x\,s$ for a given x may not hold. For example, x is marked as *called* at the start of `solve`, and `Inv_corr` allows to reason about dependencies of x during the run of `solve` x.

```
Definition Inv_sigma (s s' : state') : Prop :=
  forall z : Var.t, D.Leq (getval s z) (getval s' z).
```

The transition relation `Inv_sigma` asserts that current values of variables in s' are larger than those in s.

The relation

```
Definition Inv_sigma_infl (s0 s1 : state') : Prop :=
  forall z,
    let d0 := getval s0 z in
    let d1 := getval s1 z in
    (D.Leq d1 d0 -> incl (get_infl s0 z) (get_infl s1 z)) /\
    (~ D.Leq d1 d0 ->
     forall u,
       In u (get_infl s0 z) -> is_solved u s1).
```

relates changes in the structures σ and *infl*. Here, `incl` implements a set inclusion on lists. `Inv_sigma_infl` asserts that, for every variable $z : V$, if the value of z did *not* increase then $s'.\mathit{infl}$ stores more influences of z; otherwise, if the value of z is altered in `state'`, then all the variables influenced by z in `state` are solved in `state'`. Now we construct invariants for each component of the algorithm.

– Invariant for `EvalGet`.

  The following specification relates arguments $x, y : V$, $s :$ `state'` with the result value $(d, s') : D \times$ `state'` of the call `evalget` $x\,y\,s$ whenever it terminates.

```
Definition Inv_EvalGet
    (x y : Var.t) (s : state') (p : D.t * state') : Prop :=
  let (d,s') := p in
  Inv_0 s ->
  Inv_corr s ->
```

  When starting from a consistent and correct state s,

```
  Inv_0 s' /\
  Inv_1 s s' /\
  Inv_corr s' /\
```

  a new consistent and correct state s' is returned (the transition from s to s' is consistent)

```
  d = getval s' y /\
```

  along with a new value of y;

```
  is_stable y s' /\
```

  moreover, y is stable in s' (since `solve` y is triggered recursively from `evalget` $x\,y\,s$);

```
  Inv_sigma s s' /\
```

  current variable assignment can not decrease (since σ always gets updated accumulatively);

```
  Inv_sigma_infl s s' /\
```

  variable dependencies are processed correctly;

```
  In x (get_infl s' y).
```

  additionally, the solver records that y influences x (since y was met in the right-hand side for x). For the rest of invariants, only distinctive parts of specifications are explained.

– Invariant for `EvalGet_x`.

  The definition is straightforward since `EvalGet_x` models a partial application of `evalget` to x.

```
Definition Inv_EvalGet_x
  (x : Var.t)
  (f : Var.t -> state' -> D.t * state' -> Prop) : Prop :=
    forall y s ds1,
      f y s ds1 -> Inv_EvalGet x y s ds1.
```

– Invariant for `Wrap_Eval_x`.

The following specification relates an input $(s, ps) : \mathtt{state}' \times \mathtt{list}\,(V \times D)$ with the result value $(d, (s', ps')) : D \times (\mathtt{state}' \times \mathtt{list}\,(V \times D))$ of the call

$$[\![t]\!]_{\mathtt{state}' \times \mathtt{list}\,(V \times D)}(\mathtt{instr}_{\mathtt{state}'}\,(\mathtt{evalget}\,x))\,(s, ps)$$

whenever it terminates. Remind that the function proceeds recursively on t which is a subtree of $\mathtt{rhs}\,x$ taking as a parameter a list ps of already visited variables along with their received values. The invariant states that

```
  Definition Inv_Wrap_Eval_x
    (x : Var.t)
    (f : Var.t -> state' -> D.t * state' -> Prop)
    (t : Tree Var.t D.t D.t)
    (sl : state' * list (Var.t * D.t))
    (dsl' : D.t * (state' * list (Var.t * D.t))) : Prop :=
    let (s, ps) := sl in
    let '(d, (s', ps')) := dsl' in
    Inv_0 s ->
    Inv_corr s ->
    (forall p,
       In p ps ->
       is_stable (fst p) s /\ D.Leq (snd p) (getval s (fst p))) ->
```

if ps is a partial path of stable variables such that the recorded values do not exceed current values of the recorded variables

```
    legal (rhs x) ps ->
    subtree (rhs x) ps = t ->
```

and ps is a legal path in $\mathtt{rhs}\,x$ leading to a given subtree t

```
    Inv_0 s' /\
    Inv_1 s s' /\
    (forall p,
       In p ps' -> D.Leq (snd p) (getval s' (fst p))) /\
    Inv_corr s' /\
    Inv_sigma s s' /\
    Inv_sigma_infl s s' /\
```

then a longer path ps' (that actually extends ps) of stable visited variables is returned together with a consistent and correct state s', and new recorded values of variables from ps' are smaller than their values in s' (we recall that all updates are accumulative);

```
legal (rhs x) ps' /\
subtree (rhs x) ps' = Ans d /\
```

the result path ps' is a legal path in `rhs` x and leads to an answer d;

```
(is_called x s' ->
 valid (getval s) ps ->
 (forall p,
    In p ps -> In x (get_infl s (fst p))) ->
```

moreover, if $x \in s'.called$ (and therefore, $x \in s.called$ by `Inv_1` $s\,s'$) and ps is valid relative to `getval` s and if all the variables mentioned in the path ps are recorded as dependencies of x

```
 valid (getval s') ps' /\
 (forall p,
    In p ps' -> In x (get_infl s' (fst p))) /\
 Inv_corr_var s' x).
```

then ps' is valid also for `getval` s' and all the variables mentioned in ps' are recorded as dependencies of x. We notice that from the fact $x \in s'.called$ one can deduce that none of variables influencing x and mentioned in ps changed its value. Otherwise, x would be recursively recomputed and removed from the *called* component which would contradict the assertion $x \in s'.called$. Since `is_called` $x\,s'$ holds and `Inv_corr` s' speaks only about variables not from $s'.called$, one additionally needs the proposition `Inv_corr_var` $s'\,x$ in order to reason about dependencies of x. When x gets removed from *called* by `solve` the second conjunct of `Inv_corr` for x is automatically fulfilled thanks to `Inv_corr_var` $s'\,x$ part.

– Invariant for `Eval_rhs`.

Since `Eval_rhs` is defined through `Wrap_Eval_x` by putting `[]` as an initial partial path in t, we obtain the following specification.

```
Definition Inv_Eval_rhs
  (x : Var.t) (s : state')
  (dsl' : D.t * (state' * list (Var.t * D.t))) : Prop :=
  let '(d, (s', ps)) := dsl' in
  Inv_0 s ->
  Inv_corr s ->
  Inv_0 s' /\
  Inv_1 s s' /\
  Inv_corr s' /\
  Inv_sigma s s' /\
  Inv_sigma_infl s s' /\
  (forall p,
     In p ps -> D.Leq (snd p) (getval s' (fst p))) /\
  legal (rhs x) ps /\
  subtree (rhs x) ps = Ans d /\
(is_called x s' ->
 d = [[rhs x]]* (getval s') /\
 deps (rhs x) (getval s') = ps /\
 (forall p,
    In p ps -> In x (get_infl s' (fst p))) /\
 Inv_corr_var s' x).
```

Note that in the case `is_called` $x\,s'$ we can deduce that ps is a trace of `getval` s' in `rhs` x and thus, `valid` (`getval` s') ps holds. By Lemma 2.2.20, we then conclude $d = [\![\texttt{rhs}\,x]\!]^*(\texttt{getval}\,s')$.

– Invariant for `Solve`.

  The specification

```
Definition Inv_Solve (x : Var.t) (s s' : state') : Prop :=
  Inv_0 s ->
  Inv_corr s ->
  Inv_0 s' /\
  Inv_1 s s' /\
  Inv_sigma s s' /\
  Inv_sigma_infl s s' /\
  Inv_corr s' /\
  (~ is_stable x s -> is_solved x s').
```

  relates arguments x and s with the result state s' of the call of `solve` $x\,s$ whenever it terminates. If the state s is consistent and correct then so is s'. In the case $x \notin \mathit{stable}$, x is eventually solved in s'.

– Invariant for `SolveAll`. The specification

```
Definition Inv_SolveAll
  (w : list Var.t) (s s' : state') : Prop :=
  Inv_0 s ->
  Inv_corr s ->
  (forall x, In x w -> ~ is_called x s) ->
  Inv_0 s' /\
  Inv_1 s s' /\
  Inv_sigma s s' /\
  Inv_sigma_infl s s' /\
  Inv_corr s' /\
  (* all variables from worklist are solved: *)
  (VS.Subset
     (VS.union (of_list w) (get_solved s))
     (get_solved s')).
```

  relates the arguments w : `list` V and s : `state'` with the result s' : `state'` of the call `solve_all` $w\,s$ whenever it terminates. It states that all the variables solved in

s together with variables from w are solved in s'. In view of `Inv_0` s', we conclude that none of the variables from w are in $s'.\mathit{queued}$. Note that $w = \mathit{infl}[x]$ (for some x) may contain spurious (outdated) dependencies, i.e., variables not dependent on x on the current variable assignment, which still get recomputed, though. However, it is safe to reevaluate them as confirmed by `Inv_corr` s' that asserts correctness of the final state s', i.e., all the needed dependencies are appropriately processed.

We show that the above defined relations are invariants of the respective functions.

Lemma 2.2.21 (invariants)**.** *For the instrumented implementation of* **RLD***, the following is true.*

$$(\forall x, y, s, p.\, \mathtt{EvalGet}\ x\ y\ s\ p \to \mathtt{Inv\_EvalGet}\ x\ y\ s\ p) \wedge$$
$$(\forall x, f.\, \mathtt{EvalGet\_x}\ x\ f \to \mathtt{Inv\_EvalGet\_x}\ x\ f) \wedge$$
$$(\forall x, t, s, p.\, \mathtt{Wrap\_Eval\_x}\ x\ t\ s\ p \to \mathtt{Inv\_Wrap\_Eval\_x}\ x\ t\ s\ p \wedge$$
$$(\forall x, s, p.\, \mathtt{Eval\_rhs}\ x\ s\ p \to \mathtt{Inv\_Eval\_rhs}\ x\ s\ p \wedge$$
$$(\forall x, s, s'.\, \mathtt{Solve}\ x\ s\ s' \to \mathtt{Inv\_Solve}\ x\ s\ s') \wedge$$
$$(\forall w, s, s'.\, \mathtt{SolveAll}\ w\ s\ s' \to \mathtt{Inv\_SolveAll}\ w\ s\ s')$$

Proof. By induction on structure of relations. `Inv_Wrap_Eval_x` and `Inv_SolveAll` are proved by extra induction on t and w, respectively. □

The proof is direct but not that short taking around 800 lines of Coq code together with necessary technical lemmas.

Proof of Theorem 2.2.13 Having verified the invariants, we now prove that Theorem 2.2.13 holds, i.e., **RLD** is a local solver. Let s_{init} : `state` be an initial state with $s_{\mathit{init}}.\mathit{stable} = \emptyset$ and $s_{\mathit{init}}.\sigma = s_{\mathit{init}}.\mathit{infl} = \emptyset$. Assume that **RLD** applied to (F, X) with pure F terminates, and let s : `state` be such that `SolveAll` $X\, s_{\mathit{init}}\, s$. According to Definition 2.2.11, we have to show that

1. $X \subseteq$ `get_stable` s;
2. `is_stable` $s\,x$ and $(y, \_) \in$ `deps` F_x (`getval` s) imply `is_stable` $s\,y$, for all $x, y : V$;
3. `getval` $s\,x \sqsupseteq (F_x)_{\mathit{Id}}$ (`getval` s) holds for every $x \in$ `get_stable` s.

We exploit the invariants for the instrumented solver and the simulation lemma. Until the end of this paragraph, we use primed notation for instrumented functions and states. Let s'_{init} be an instrumented initial state with $s'_{\mathit{init}}.\mathit{stable} = s'_{\mathit{init}}.\mathit{called} = s'_{\mathit{init}}.\mathit{queued} = \emptyset$ and $s'_{\mathit{init}}.\mathit{sigma} = s'_{\mathit{init}}.\mathit{infl} = \emptyset$. Obviously, $s_{\mathit{init}} \sim s'_{\mathit{init}}$ holds. By Lemma 2.2.18, there exists s' : `state'` such that

$$\mathtt{SolveAll'}\ X\ s'_{\mathit{init}}\, s' \quad \text{and} \quad s \sim s'.$$

By Lemma 2.2.21,

$$\mathtt{Inv\_SolveAll}\ x\, s'_{\mathit{init}}\, s'$$

holds. Since the premises of `Inv_SolveAll` $x\, s'_{init}\, s'$ (namely, these are `Inv_0` s'_{init}, `Inv_corr` s'_{init} and $\forall x. x \in X \to \neg$`is_called'` $x\, s'_{init}$) are trivially fulfilled, we have

$$\texttt{Inv\_1}\; s'_{init}\, s' \tag{a}$$

$$\texttt{Inv\_corr}\; s' \tag{b}$$

$$X \cup (s'_{init}.stable \setminus s'_{init}.called) \subseteq (s'.stable \setminus s'.called) \tag{c}$$

Since (a) asserts $s'_{init}.called \supseteq s'.called$, we deduce $s'.called = \emptyset$. Hence, from (c) we conclude

$$X \subseteq s'.stable\,. \tag{d}$$

From (b), we obtain

$$\forall x \in s'.stable.\, \texttt{getval}\, s'\, x \sqsupseteq [\![\texttt{rhs}\, x]\!]^* \,(\texttt{getval}\, s') \tag{e}$$

and

$$\forall x \in s'.stable. \forall y.\, (y, \_) \in \texttt{deps}\,(\texttt{rhs}\, x)\,(\texttt{getval}\, s') \to y \in s'.stable\,. \tag{f}$$

Since $s \sim s'$, (d), (e) and (f) prove the theorem. □

Notice that instead of the two step proof using simulation relation, one could alternatively eliminate ghost structures directly from proofs for the instrumented version and construct a direct proof for the original version of the solver. The invariants for the original version can be obtained by existentially quantifying the instrumentation components in respective invariants for the instrumented version. To show that such existentially quantified invariants are indeed preserved, one eliminates the existential quantifiers in premises yielding fixed but arbitrary instrumentation values of the starting state. Then these instrumentation values are updated by means of the auxiliary functions `add_queued`, `rem_queued`, etc., imitating the algorithm, and used as existential witnesses in the conclusion statement. The remaining proof obligations then follow step by step the corresponding proofs for the instrumented versions. For a formal account of this proof-transforming procedure in the context of Hoare logic, refer to [HP07]. We will follow this approach in future version of development since it will allow to reduce the size of Coq code, although with some overhead for the formulation of invariants and the maintenance of proofs.

2.2.3. Exactness

In many applications, right-hand sides F arise as monotonic functions. In this case, one would expect that a "good" solver produces more precise results. However, as shown by the example below, **RLD** is *not* an exact solver generally, i.e., it may fail to return a precise solution even if the right-hand sides are monotonic functions.

Definition 2.2.22. We say that the monadically parametric function $F : \prod_T .(A \to TB) \to TC$ is *monotonic* if it is monotonic for *Id*, i.e., for all $f_1, f_2 : A \to B$ such that $\forall x : A. f_1\, x \sqsubseteq_B f_2\, x$, we have $F_{Id}\, f_1 \sqsubseteq_C F_{Id}\, f_2$.

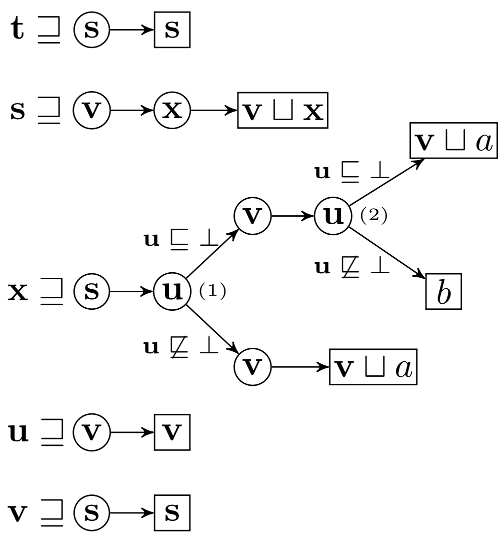

Figure 2.5.: Counterexample to the monotonic case

Definition 2.2.23. We say that the constraint system $(V, \mathbb{D}, F)$ with monadically parametric $F : V \to \prod_T .(V \to TD) \to TD$ is *monotonic* if $F_{\mathbf{x}}$ is monotonic for every $\mathbf{x} \in V$.

Theorem 2.2.24 (Knaster-Tarski). *Given a complete lattice D and a monotonic function $f : D \to D$, i.e., such that*

$$\forall d_1, d_2 : D.\, d_1 \sqsubseteq d_2 \textit{ implies } f\, d_1 \sqsubseteq f\, d_2$$

the set of fixed points of f in D forms a complete lattice. □

Corollary 2.2.25. *For every monotonic constraint system $\mathcal{S} = (V, \mathbb{D}, F)$ over a complete lattice $\mathbb{D}$, there exists a least solution $\mu : V \to D$ of $\mathcal{S}$ such that $\mu\, \mathbf{x} = (F_{\mathbf{x}})_{Id}\, \mu$ holds for all $\mathbf{x} \in V$.* □

Definition 2.2.26. We say that a local solver $\mathcal{A}_{V,D}$ is *exact* if, for any monotonic constraint system $(V, \mathbb{D}, F)$ over a complete lattice $\mathbb{D}$, $\mathcal{A}$ when applied to a pair (F, X) for a finite set $X \subseteq V$ of interesting variables, if it terminates, returns a local solution (σ, X') of $\mathcal{S}$ relative to X such that for the least solution μ of $\mathcal{S}$, $\mu \restriction_{X'} = \sigma \restriction_{X'}$ holds.

Consider a constraint system with pure right-hand sides defined by strategy trees as in Figure 2.5 over the lattice $\mathbb{D} = (\{\bot, a, b, \top\}, \sqsubseteq, \sqcup)$ where $\bot \sqsubset a, b \sqsubset \top$, and a and b are incomparable. These right-hand sides could be represented in a ML-like language by

$$F_{\mathbf{t}}\, k = \texttt{let } s_1 = k\, \mathbf{s} \texttt{ in } s_1$$
$$F_{\mathbf{s}}\, k = \texttt{let } v_1 = k\, \mathbf{v} \texttt{ in let } x_1 = k\, \mathbf{x} \texttt{ in } v_1 \sqcup x_1$$

```
$F_{\mathbf{x}}\, k = $ let $s_1 = k\,\mathbf{s}$ in let $u_1 = k\,\mathbf{u}$ in
            if $(u_1 \sqsubseteq \bot)$ then
                let $v_1 = k\,\mathbf{v}$ in
                let $u_2 = k\,\mathbf{u}$ in
                if $(u_2 \sqsubseteq \bot)$ then $v_1 \sqcup a$ else $b$
            else
                let $v_1 = k\,\mathbf{v}$ in $v_1 \sqcup a$
$F_{\mathbf{u}}\, k = $ let $v_1 = k\,\mathbf{v}$ in $v_1$
$F_{\mathbf{v}}\, k = $ let $s_1 = k\,\mathbf{s}$ in $s_1$
```

Although a result of the query $k\,\mathbf{s}$ is not used to produce a return value of $F_{\mathbf{x}}\,k$, the dependency on $\mathbf{s}$ triggers a recomputation of $\mathbf{x}$ once a value of $\mathbf{s}$ changes. Note that there is a path in a tree for $F_{\mathbf{x}}$ for that variable $\mathbf{u}$ is queried twice. These two Que-nodes are marked by (1) and (2) in the above figure.

It is not difficult to check that all the right-hand sides are monotonic functions and the least solution μ of the system is

$$\mu\, x = a, \quad x \in \{\mathbf{t}, \mathbf{s}, \mathbf{x}, \mathbf{u}, \mathbf{v}\}.$$

Suppose, we want to compute a local solution relative to $X = \{\mathbf{t}\}$. Will **RLD** return an *exact* local solution for $\{\mathbf{t}\}$? The answer is *negative*. To figure out the problem, let us introduce a ghost structure *called* to **RLD** and trace computations done by the solver. Consider the following implementation with the type of states

$$\texttt{type state}'' = (V \rightharpoonup D) \times (V \rightharpoonup \texttt{list}\, V) \times \mathcal{P}_{fin}(V) \times \mathcal{P}_{fin}(V)$$

where as before, components correspond to the finite (partial) map σ, the finite (partial) map *infl*, and the sets *stable*, *called*, respectively.

```
let extract_work x = fun s →
  let w = get_infl x s in
  let s0 = rem_infl x s in
  let s1 =
    foldl (fun s y →rem_stable y (rem_called y s)) s0 w in
    (w,s1)

(* ... *)
and solve x = fun s →
  if is_stable x s then s else
    let s0 = add_called x (add_stable x s) in
    let (d,s1) = Fx (evalget x) s0 in
    let s2 = rem_called x s1 in
    let cur = getval s2 x in
    let new = cur ⊔ d in
    if (new ⊑ cur) then $s_2$ else
      let $s_3$ = setval x new $s_2$ in
      let $(w, s_4)$ = extract_work x $s_3$ in
        solve_all $w$ $s_4$
(* ... *)
```

The rest of functions are as in Figure 2.4. We trace the computations performed when `solve_all [t]` is called from the initial state s_{init} (the full trace can be found in Appendix B.2).

From the initial state, `solve t` is called, which in turn recursively invokes `solve s`. During the run of `solve s`, the algorithm performs the following recursive evaluations. First, it tries to recompute $\mathbf{v}$, which is however only depends on $\mathbf{s}$, and the latter is stable, hence $\mathbf{v}$ does not change its value. Then the algorithm proceeds with variables $\mathbf{x}$, $\mathbf{u}$ and $\mathbf{v}$, in that order. For brevity, we skip a description of these computation steps (cf. lines 1–56 in Appendix B.2), but we note that $\sigma[\mathbf{s}]$ gets finally updated. Before this new value of $\sigma[\mathbf{s}] = a$ is returned by `solve s`, the algorithm must recompute all the variables dependent on $\mathbf{s}$. These are variables from $\mathit{infl}[\mathbf{s}] = [\mathbf{x}; \mathbf{v}]$ (line 57). Thus, $\mathit{infl}[\mathbf{s}]$ is reset to `[]`, and $\mathbf{x}$ and $\mathbf{v}$ are removed from *stable* and *called* (lines 58–61). The state prior to the call `solve_all [x; v]` is

$$\begin{array}{ll} \sigma & = \{\mathbf{s} \mapsto a, \mathbf{x} \mapsto a\} \\ \mathit{infl} & = \{\mathbf{u} \mapsto [\mathbf{x}; \mathbf{x}], \mathbf{v} \mapsto [\mathbf{x}; \mathbf{u}; \mathbf{s}], \mathbf{x} \mapsto [\mathbf{s}]\} \\ \mathit{stable} & = \{\mathbf{s}, \mathbf{t}, \mathbf{u}\} \\ \mathit{called} & = \{\mathbf{t}\} \end{array}$$

1. For recomputation of $\mathbf{x}$, `solve x` is called, and variable $\mathbf{x}$ is put back into *stable* and *called*. Thus, the state prior to reevaluation of the right-hand side $\overline{F}_{\mathbf{x}}$ is

$$\begin{array}{ll} \sigma & = \{\mathbf{s} \mapsto a, \mathbf{x} \mapsto a\} \\ \mathit{infl} & = \{\mathbf{u} \mapsto [\mathbf{x}, \mathbf{x}], \mathbf{v} \mapsto [\mathbf{x}; \mathbf{u}; \mathbf{s}], \mathbf{x} \mapsto [\mathbf{s}]\} \\ \mathit{stable} & = \{\mathbf{s}, \mathbf{t}, \mathbf{u}, \mathbf{x}\} \\ \mathit{called} & = \{\mathbf{t}, \mathbf{x}\} \end{array}$$

During the evaluation of $[\![\overline{F}_{\mathbf{x}}]\!]$ (`evalget x`) the algorithm traverses the tree $\overline{F}_{\mathbf{x}}$ and recomputes encountered variables as described below.

– Since $\mathbf{s}, \mathbf{u} \in \mathit{stable}$, the algorithm does not descend into solving them. The structures σ, *stable* and *called* do not change, but the solver records that $\mathbf{x}$ depends on $\mathbf{s}$ and $\mathbf{u}$, i.e., $\mathbf{x}$ is added to $\mathit{infl}[\mathbf{s}]$ and $\mathit{infl}[\mathbf{u}]$ (lines 69–72).

– Since $\sigma[\mathbf{u}] = \bot$, the algorithm continues with the upper branch of (1) in $\overline{F}_{\mathbf{x}}$. Thus, it recomputes $\mathbf{v}$, which gets a larger value a (as $\sigma[\mathbf{s}] = a$) (lines 73–81). Since the value of $\sigma[\mathbf{v}]$ has changed, variables influenced by $\mathbf{v}$ must be recomputed before `solve v` returns. These are variables from $\mathit{infl}[\mathbf{v}] = [\mathbf{x}; \mathbf{u}; \mathbf{s}]$. They get removed from the sets *stable* and *called*, and get recomputed by calling consequently `solve` $\mathbf{x}$, `solve` $\mathbf{u}$, and `solve` $\mathbf{s}$ (lines 91–155). At the end end of their run, the state components are (the *infl* component is omitted)

$$\begin{aligned}\sigma &= \{\mathbf{s} \mapsto a, \mathbf{u} \mapsto a, \mathbf{v} \mapsto a, \mathbf{x} \mapsto a\}\\ \mathit{stable} &= \{\mathbf{s}, \mathbf{t}, \mathbf{u}, \mathbf{v}, \mathbf{x}\}\\ \mathit{called} &= \{\mathbf{t}\}\end{aligned}$$

Thus, the value $\sigma[\mathbf{u}]$ is altered and equals a. The algorithm returns from `solve` $\mathbf{v}$ and continues traversing $\overline{F}_\mathbf{x}$. Notice that change in $\sigma[\mathbf{v}]$ caused another round of a recursive recomputation of $\mathbf{x}$, while the parent evaluation of $F_\mathbf{x}$ has not yet terminated. After the call to `solve` $\mathbf{v}$ returns, we observe that $\mathbf{x}$ is solved, i.e., $\mathbf{x} \in \mathit{stable} \setminus \mathit{called}$. This is a crucial observation for a modification of the solver considered in the next subsection.

– The solver hits the branching (2) in $\overline{F}_\mathbf{x}$, and $\mathbf{u} \in \mathit{stable}$ (lines 157–158). This time, the solver must follow the lower branch of (2), with $\mathbf{u} \not\sqsubseteq \bot$, as $\sigma[\mathbf{u}] = a$, and it hits an Ans-leaf with a "bad" value b.

Thus, evaluation of $F_\mathbf{x}$ (`evalget` $\mathbf{x}$) finishes with a current state having

$$\sigma = \{\mathbf{s} \mapsto a, \mathbf{u} \mapsto a, \mathbf{v} \mapsto a, \mathbf{x} \mapsto a\}$$

and $\sigma[\mathbf{x}]$ gets updated with $a \sqcup b = \top$. Since $\sigma[\mathbf{x}]$ has strictly increased, more recomputations are triggered which we omit here (lines 159–304).

2. For recomputation of $\mathbf{v}$, `solve` $\mathbf{v}$ is called, but $\mathbf{v}$ is solved at this moment, i.e., $\mathbf{v} \in \mathit{stable} \setminus \mathit{called}$, and the call returns immediately (lines 305–306).

Finally, the algorithm terminates and returns a solution $\sigma_1 : V \to D$ such that $\sigma_1\, \mathbf{x} = \top$ and thus, σ_1 is not minimal. The reason why the "bad" value b is reached in $\overline{F}_\mathbf{x}$ during the run of the solver is that $\mathbf{u}$ changes its value between branchings (1) and (2). While passing through $\overline{F}_\mathbf{x}$, the algorithm picks different branches at (1) and at (2) — the upper one, $\mathbf{u} \sqsubseteq \bot$, in the former case and the lower one, This demonstrates that computations of right-hand sides are *not* atomic in general. $\mathbf{u} \not\sqsubseteq \bot$, in the latter case. Note that the Ans-leaf b is *not* reachable by $[\![\overline{F}_\mathbf{x}]\!]^*\sigma$, for any effect-free variable assignment $\sigma : V \to D$.

Notice that the counterexample also reveals that **RLD** is not a chaotic iteration solver that makes him hardly usable with widening/narrowing operators in general. Indeed, for the variable assignment σ_0 such that $\sigma_0\, \mathbf{s} = \sigma_0\, \mathbf{u} = \sigma_0\, \mathbf{v} = a$, we have $[\![\overline{F}_\mathbf{x}]\!]^*\sigma_0 = a$ which is strictly smaller than $\top$, while the top value was assigned to $\sigma[\mathbf{x}]$ once the Ans-leaf b was reached.

Although **RLD** does not return a minimal solution generally, it is however exact for a subclass of monotonic strategy functions.

Definition 2.2.27. We say that a strategy tree t has an *unique-lookup* property if for any legal path in t any variable $\mathbf{v} : V$ is queried at most once on this path. Formally,

```
Definition uniq_lookup t :=
  forall ps, legal t ps -> NoDup (fst (split ps)).
```

where `split` is a standard Coq function splitting a list of pairs into a pair of lists, and `NoDup` asserts that a list has no duplicated elements in it.

For example, the property does not hold for the tree $\overline{F}_{\mathbf{x}}$ as in Figure 2.5, since there exists a path through $\overline{F}_{\mathbf{x}}$ such that the variable $\mathbf{u}$ is met twice, at the Que-nodes (1) and (2).

Theorem 2.2.28. *Given a monotonic constraint system $\mathcal{S} = (V, \mathbb{D}, F)$ over the complete lattice $\mathbb{D}$ with pure F such that $\overline{F}_x$ enjoys the unique-lookup property for every $x \in V$,* **RLD** *when applied to (F, X) — if it terminates — returns a local solution (σ, X') such that $\sigma \restriction_{X'} = \mu \restriction_{X'}$ for the least solution μ of $\mathcal{S}$.*

Proof. To show this, we construct invariants for every function of the algorithm that relate a current variable assignment with an arbitrary solution of the constraint system. Namely, we establish that for every solution $\mu : V \to D$ the condition `getval` $s \sqsubseteq \mu$ is preserved by all components of the algorithm. This implies the assertion of the theorem. We define the following invariants. Each of them has an extra premise *isMonotone*$(F) \wedge$ *hasUniqueLookup*$(\texttt{rhs})$ which is omitted for brevity.

$$\begin{aligned}
&\texttt{Inv\_EvalGet}_{mon}\ x\ y\ s\ (d, s') \equiv \\
&\quad \forall \mu : V \to D.\, (\forall z. \mu\, z \sqsupseteq [\![\texttt{rhs}\, z]\!]^* \mu) \wedge \texttt{getval}\, s \sqsubseteq \mu \to \texttt{getval}\, s' \sqsubseteq \mu \\
&\texttt{Inv\_Wrap\_Eval\_x}_{mon}\ x\ f\ t\ (s, l)\ (d, (s', l')) \equiv \\
&\quad \forall \mu : V \to D.\, (\forall z. \mu\, z \sqsupseteq [\![\texttt{rhs}\, z]\!]^* \mu) \wedge \texttt{getval}\, s \sqsubseteq \mu \to \texttt{getval}\, s' \sqsubseteq \mu \\
&\texttt{Inv\_eval\_rhs}_{mon}\ x\ s\ (d, s') \equiv \\
&\quad \forall \mu : V \to D.\, (\forall z. \mu\, z \sqsupseteq [\![\texttt{rhs}\, z]\!]^* \mu) \wedge \texttt{getval}\, s \sqsubseteq \mu \to \\
&\quad \texttt{getval}\, s' \sqsubseteq \mu \wedge d \sqsubseteq \texttt{getval}\, s'\, x \\
&\texttt{Inv\_EvalGet}_{mon}\ x\ s\ s' \equiv \\
&\quad \forall \mu : V \to D.\, (\forall z. \mu\, z \sqsupseteq [\![\texttt{rhs}\, z]\!]^* \mu) \wedge \texttt{getval}\, s \sqsubseteq \mu \to \texttt{getval}\, s' \sqsubseteq \mu \\
&\texttt{Inv\_solve\_all}_{mon}\ w\ s\ s' \equiv \\
&\quad \forall \mu : V \to D.\, (\forall z. \mu\, z \sqsupseteq [\![\texttt{rhs}\, z]\!]^* \mu) \wedge \texttt{getval}\, s \sqsubseteq \mu \to \texttt{getval}\, s' \sqsubseteq \mu
\end{aligned}$$

By induction on structure of relations we prove

Lemma 2.2.29. *For the instrumented implementation of* **RLD** *the following is true.*

$$\begin{aligned}
&(\forall x, y, s, r'.\, \textit{EvalGet}'\ x\ y\ s\ r' \to \textit{Inv\_EvalGet}_{mon}\ x\ y\ s\ r') \wedge \\
&(\forall x, f.\, \textit{EvalGet\_x}'\ x\ f \to \textit{Inv\_EvalGet\_x}_{mon}\ x\ f) \wedge \\
&(\forall x, t, r, r'.\, \textit{Wrap\_Eval\_x}\ x\ t\ r\ r' \to \textit{Inv\_Wrap\_Eval\_x}_{mon}\ x\ t\ r\ r') \wedge \\
&(\forall x, s, r'.\, \textit{Eval\_rhs}\ x\ s\ r' \to \textit{Inv\_Eval\_rhs}_{mon}\ x\ s\ r') \wedge \\
&(\forall x, s, s'.\, \textit{Solve}'\ x\ s\ s' \to \textit{Inv\_Solve}_{mon}\ x\ s\ s') \wedge \\
&(\forall w, s, s'.\, \textit{SolveAll}'\ w\ s\ s' \to \textit{Inv\_SolveAll}_{mon}\ w\ s\ s')
\end{aligned}$$

□

Let s_{init} : $\texttt{state}'$ be an initial state with $\sigma = \mathit{infl} = \emptyset$ and $\mathit{stable} = \mathit{called} = \mathit{queued} = \emptyset$, and let μ be a least solution to $\mathcal{S}$ which exists by Corollary 2.2.25. Assume that **RLD** when applied to (F, X) terminates and let s be a state returned by the call $\texttt{solve\_all}\, X\, s_{init}$ that is, $\texttt{SolveAll}'\, X\, s_{init}\, s$ holds. Since $\texttt{getval}\, s_{init}\, x = \bot_D \sqsubseteq \mu\, x$, for all $x : V$, by Lemma 2.2.29, we have

$$\texttt{getval}\, s \sqsubseteq \mu. \tag{2.1}$$

We need to show that $\texttt{getval}\, s\, x = \mu\, x$ holds for all $x \in X'$.

Let $\sigma_{X'}$ be an extension of $\texttt{getval}\, s$ defined by

$$\sigma_{X'}\, x = \begin{cases} \texttt{getval}\ s\ x & x \in X' \\ \top_D & \text{otherwise.} \end{cases}$$

By Lemma 2.2.10, $\sigma_{X'}$ is a solution to $\mathcal{S}$. Since X' is $\texttt{deps}$-closed,

$$[\![\overline{F}_x]\!]^*\,(\texttt{getval}\ s) = [\![\overline{F}_x]\!]^*\,\sigma_{X'} \tag{2.2}$$

holds by Lemma 1.5.6, for all $x \in X'$. For $x \in X'$, we get

$$\begin{aligned} \mu\, x &\sqsupseteq \texttt{getval}\ s\ x && \text{by (2.1)} \\ &\sqsupseteq [\![\overline{F}_x]\!]^*\,(\texttt{getval}\, s) && \text{by Theorem 2.2.13} \\ &= [\![\overline{F}_x]\!]^*\,\sigma_{X'} && \text{by (2.2)} \\ &\sqsupseteq [\![\overline{F}_x]\!]^*\,\mu && \text{by monotonicity of } F \text{ and minimality of } \mu \\ &= \mu\, x && \text{since } \mu \text{ is a solution to } \mathcal{S}. \end{aligned}$$

Therefore, $\mu\, x = \texttt{getval}\ s\ x = [\![\overline{F}_x]\!]^*\,(\texttt{getval}\, s)$ holds, for all $x \in X'$. This proves Theorem 2.2.28. □

Although theoretically, one can always transform a definable pure function F into G such that $F_{Id}\,\sigma = G_{Id}\,\sigma$ and $\overline{G}$ has a unique-lookup property, practically that requires availability of the source code of F.

Apparently, **RLD** implements a chaotic iteration strategy when restricted to the class of unique-lookup right-hand sides, but we have no formal proof of this conjecture yet.

2.2.4. RLDE

In this subsection we present a modification of the solver **RLD** that is exact (called **RLDE**).

The idea of improvement comes from a careful inspection of behaviour of the instrumented **RLD** and its invariants. Considering the example displayed in Figure 2.5, we observed that for the unknown $\mathbf{x}$, during a computation of the right-hand side $F_{\mathbf{x}}\,(\texttt{evalget}\,\mathbf{x})$, another variable $\mathbf{v}$ gets a strictly larger value which in turn causes another round of recursive recomputation for $\mathbf{x}$. We have also mentioned that after

```
exception IsSolved

let $\sigma_\bot$ x = if x $\in \mathrm{dom}(\sigma)$ then $\sigma$[x] else $\bot$

let $infl_\emptyset$ x = if x $\in \mathrm{dom}(infl)$ then $infl$[x] x else $\emptyset$

let extract_work x =
  let $work$ = $infl_\emptyset$ x in
  $infl$[x] := $\emptyset$;
  $stable$ := $stable \setminus work$;
  $called$ := $called \setminus work$;
  $work$

let rec evalget x y =
  solve y;
  $infl$[y] := $infl$[y] $\cup$ {x};
  if not (x $\in$ $called$) then
    raise IsSolved
  else
    $\sigma_\bot$ y

and solve x =
  if (x $\notin$ $stable$) then begin
    $stable$ := $stable$ $\cup$ {x};
    $called$ := $called$ $\cup$ {x};
    try
      let $d$ = $F$ x (evalget x) in
      $called$ := $called$ $\setminus$ {x};
      let $cur$ = $\sigma_\bot$ x in
      let $new$ = $cur \sqcup d$ in
        if ($new \not\sqsubseteq cur$) then begin
          $\sigma$[x] := $new$;
          let $work$ = extract_work x in
            solve_all $work$
        end
    with IsSolved $\to$ ()
  end

and solve_all X = foreach x $\in$ X do solve x

let main X =
  $\sigma$ := $\emptyset$; $infl$ := $\emptyset$; $stable$ := $\emptyset$; $called$ := $\emptyset$;
  solve_all X;
  $(\sigma_\bot, stable)$
```

Figure 2.6.: The recursive solver tracking local dependencies, exact (**RLDE**)

`solve` **v** returns, **x** does *not* belong to *called* any more. Notice also that the invariant `Inv_Wrap_Eval_x` (namely, its `Inv_corr` part) guarantees that in this case (after the internal recursive call to `solve` **v** returns) **x** is solved, i.e., $\mathbf{x} \in \mathit{stable} \setminus \mathit{called}$ holds and the constraint for **x** is satisfied. This is a crucial observation that allows us to conclude that it is safe to interrupt computation of $F_\mathbf{x}$ (`evalget` **x**) as soon as it is discovered that **x** is solved. This allows to avoid bad cases like in the previous example when the reached Ans-leaf b spoils the result solution, although it is not reachable in $\overline{F}_\mathbf{x}$ when using any effect-free variable assignment $\sigma : V \to D$, as well as to avoid unnecessary computations of nodes that are not reachable in this sense.

Thus, the idea of improvement is to check whether **x** is still in *called* while computing the right-hand side for **x** and interrupt as soon as the condition fails. To implement this idea, we adjust the original **RLD** in the following way (Figure 2.6). Similar to the instrumented version of **RLD**, we introduce the data structure *called*, which stores a set of suspended variables for that `solve` was called but the latest call has not yet terminated. When `solve` **x** is called, **x** is added to both *stable* and *called* sets. In `evalget` **x y**, after `solve` **y** returns and *infl* is updated accordingly, we check if $\mathbf{x} \in \mathit{called}$. If yes, we proceed normally, otherwise the exception `IsSolved` is raised and thus, the current evaluation of $F_\mathbf{x}$ (`evalget` **x**) gets cancelled. If the exception is caught in `solve`, the latter returns immediately. Otherwise, we remove **x** from *called*, update the current value of **x** and recompute the variables dependent on **x** if needed. The purely functional implementation can be found in Appendix B.3.

We prove the modified algorithm **RLDE** correct. Moreover, whenever it terminates it returns an exact solution if the input constraint system is monotonic defined over a complete lattice. We have

Theorem 2.2.30. *The algorithm* **RLDE** *is an* exact *local solver.* □

The proof is similar to the proofs of Theorems 2.2.13 and 2.2.28. The purely functional implementation using the exception transformer monad *ErrorT* can be found in Appendix. The invariants are similar as for **RLD**, with small modifications. Below is one of invariants for the instrumented version implemented in Coq.

```
Definition Inv_Eval_rhs
  (x : Var.t) (s : state')
  (dsl' : option D.t * (state' * list (Var.t * D.t))) :=
  let '(od, (s', ps)) := dsl' in
  is_called x s ->
  Inv_0 s ->
  Inv_corr s ->
  Inv_0 s' /\
  Inv_1 s s' /\
  Inv_corr s' /\
  Inv_sigma s s' /\
  Inv_sigma_infl s s' /\
    (forall p,
       In p ps ->
       D.Leq (snd p) (getval s' (fst p))) /\
    legal (rhs x) ps /\
    (od = error -> ~ is_called x s') /\
    (forall d, od = value d -> subtree (rhs x) ps = Ans d) /\
    (forall d,
       od = value d ->
       is_called x s' /\
       d = [[rhs x]]* (getval s') /\
       deps (rhs x) (getval s') = ps /\
       (forall p,
          In p ps -> In x (get_infl s' (fst p))) /\
       Inv_corr_var s' x) /\
    (* monotonic case *)
    (forall mu,
      is_monotone rhs ->
      is_solution rhs mu ->
      leqF (getval s) mu ->
      leqF (getval s') mu /\
      (forall d, od = value d -> D.Leq d (mu x))).
```

Proposition 2.2.31. **RLDE** *is a chaotic iteration solver.*

Proof. Variable **x** gets updated only in the case if no exception was raised during evaluation of the right-hand side. The invariant `Inv_Eval_rhs` states that in this case $d = [\![\texttt{rhs x}]\!]^* (\texttt{getval } s')$, i.e., d is a value of the right-hand side on a *current* variable assignment. ☐

The last proposition is not implemented in Coq.

Since **RLDE** is a chaotic iteration solver, it is possible to use it with widening and narrowing operators or a combined operator. For that, the condition $new \not\sqsubseteq cur$ must be replaced by $new \neq cur$.

2.2.5. Termination and Complexity

In what follows, we provide sufficient conditions for termination of **RLD** and **RLDE** and analyse the complexity of the algorithms.

To prove termination of a functional program it is sufficient to provide a *termination argument*, i.e., a *well-founded* (also called *noetherian*) relation $(A, \prec_A)$ such that every recursive call is carried on with an argument value smaller with respect to $\prec_A$ [Wal94, GWB98]. Intuitively, a binary relation $(A, \prec_A)$ is well-founded if it forbids infinite descending chains of the form $\cdots \prec a_i \prec_A \cdots \prec_A a_1 \prec_A a_0$. For example, suppose the interpreter has to evaluate a term, say $f\, a_0$, for $a_0 \in A$, that leads necessarily to a recursive evaluation of another term $f\,a_1$, $a_1 \in A$, from which in turn a recursive computation of another term $f\,a_2$, $a_2 \in A$ arises, etc. The sequence of evaluated arguments $a_0, a_1, a_2, \cdots$ forms a *trace* of computation of the term $f\,a_0$. Intuitively, the program terminates if every trace is finite, i.e., there always exists k such that the term $f\,a_k$ can be evaluated without further recursive calls to f. Thus, providing a well-founded relation $(A, \prec_A)$ such that $a_{i+1} \prec a_i$ holds for every pair of adjacent values from each trace guarantees termination of the algorithm.

We begin with the definition of well-foundedness.

Definition 2.2.32. We say that $(A, \prec_A)$ is a *noetherian* relation iff $\prec_A$ is a binary relation on A and there exists no countable infinite descending chains. That is

$$noe\,(A, \prec_A) \iff \forall c : \mathbb{N} \to A. \neg(\forall i. c_{i+1} \prec_A c_i).$$

Further, we will omit the index A if it is clear from context.

Example 2.2.33. $(\mathbb{N}, <)$ is noetherian. □

Example 2.2.34. If B is a finite set, then $(2^B, \supset)$ is noetherian. □

In the constructive world of Coq, the notion of well-foundedness is defined by means of the inductive predicate of *accessibility*.

```
Inductive Acc (A : Type) (R : A -> A -> Prop) (x : A) : Prop :=
  Acc_intro : (forall y : A, R y x -> Acc R y) -> Acc R x
```

The predicates essentially states that an element x is accessible iff all the smaller elements are accessible. For example, in $(\mathbb{N}, <)$, 0 is accessible since there are no numbers smaller than it, 1 is accessible since 0 is accessible, 2 is accessible since both 0 and 1 are accessible, and so on.

Definition 2.2.35. We say that $(A, \prec_A)$ is *well-founded* iff all its elements are accessible. Formally,

$$wf\,(A, \prec_A) \iff \forall x : A.\, \texttt{Acc}(x)$$

For well-founded relations, the well-founded induction scheme holds.

Theorem 2.2.36 (well-founded induction scheme)**.** *For a well-founded relation* $(A, \prec_A)$*,*

$$(\forall x : A.\, (\forall y : A.\, y \prec_A x \to P(y)) \to P(x)) \to \forall a : A. P(a)$$

holds for all properties P*.* □

We can show that in the classical setting both notions coincide.

Proposition 2.2.37. *For a well-founded relation* $(A, \prec_A)$*,*

$$noe\,(A, \prec_A) \iff wf\,(A, \prec_A)$$

is provable on assumption of the excluded middle axiom.

Proof. $\Rightarrow$) Suppose $(A, \prec_A)$ is noetherian, but not well-founded. Then there exists an element a_0 such that $\neg\mathtt{Acc}(a_0)$. Then $\neg(\forall y. y \prec_A a_0 \to \mathtt{Acc}(y))$ holds which is classically equivalent to $\exists y. y \prec_A a_0 \wedge \neg\mathtt{Acc}(y)$. Take a_1 such that $a_1 \prec a_0$ and $\neg\mathtt{Acc}(a_1)$. Repeating the argument, we construct $a_2 \prec a_1$ such that $\neg\mathtt{Acc}(a_2)$, and so on. Thus, there exists an infinite decreasing chain $\cdots \prec a_2 \prec a_1 \prec a_0$. Contradiction.

$\Leftarrow$) Suppose $(A, \prec_A)$ is well-founded. We show by well-founded induction a stronger statement $\forall x : A. P(x)$ with

$$P(x) \equiv \forall c : \mathbb{N} \to A. \neg(c_0 = x \wedge \forall i. c_{i+1} \prec c_i).$$

Take x such that the induction hypothesis

$$\forall y. y \prec x \to P(y) \tag{IH}$$

holds. To show $P(x)$, take a chain c, and suppose $\forall i. c_{i+1} \prec c_i \wedge c_0 = x$. Construct $c' : \mathbb{N} \to A$ as $c'_i = c_{i+1}$, for all i, and take $y = c'_0 = c_1$. Using (IH) with y and c', we get $\neg(c'_0 = y \wedge \forall i. c'_{i+1} \prec c'_i)$ which contradicts the assumption on c. □

Below, we provide some standard constructions of well-founded relations.

Definition 2.2.38. For binary relations $(A, \leq_A)$ and $(B, \leq_B)$, we define a *symmetric product* relation $(A \times B, \leq_{\text{sym}})$ by

$$(a_1, b_1) \leq_{\text{sym}} (a_2, b_2) \iff (a_1 \leq_A a_2 \wedge b_1 = b_2) \vee (a_1 = a_2 \wedge b_1 \leq_B b_2).$$

Lemma 2.2.39. *The symmetric product $(A \times B, \prec_{sym})$ of well-founded relations $(A, \prec_A)$ and $(B, \prec_B)$ is well-founded.*

Proof. By well-founded induction on both arguments. □

In the proof of termination, we use the lexicographical product construction defined as follows.

Definition 2.2.40. For binary relations $(A, \leq_A)$ and $(B, \leq_B)$, a *lexicographical product* $(A \times B, \leq_{\text{lex}})$ is defined by

$$(a_1, b_1) \leq_{\text{lex}} (a_2, b_2) \iff a_1 \leq_A a_2 \vee a_1 = a_2 \wedge b_1 \leq_B b_2.$$

Lemma 2.2.41. *The lexicographical product $(A \times B, \prec_{lex})$ of well-founded relations $(A, \prec_A)$ and $(B, \prec_B)$ is well-founded.*

Proof. By well-founded induction on both arguments. □

Lemma 2.2.42. *(inverse image) Given a well-founded relation $(B, \prec_B)$ and $f : A \to B$, define $a_1 \prec_A a_2$ by $f\,a_1 \prec_B f\,a_2$. Then $(A, \prec_A)$ is well-founded.* □

Definition 2.2.43. We say that a binary relation $(A, \leq)$

1. satisfies the *ascending chain condition* if $(A, >)$ is well-founded, i.e., there is no infinite strictly ascending chains $a : \mathbb{N} \to D$ with $a_0 < a_1 < \cdots < a_k < \cdots$;
2. has a *finite height* h if for all ascending chains $a : \mathbb{N} \to D$, $a_0 \leq a_1 \leq \cdots$, there exists $k \leq h$ such that $a_k = a_{k+1} = \ldots$, and h is minimal with such property

where $x < y \iff x \leq y \wedge x \neq y$.

Let $V \to \mathbb{D}$ be a set of functions from V to a partially ordered set $\mathbb{D} = (D, \sqsubseteq)$. Consider a pointwise ordering on $V \to \mathbb{D}$, i.e., for $\sigma_1, \sigma_2 : V \to \mathbb{D}$, we define $\sqsubseteq_{V \to \mathbb{D}}$ by

$$\sigma_1 \sqsubseteq_{V\to\mathbb{D}} \sigma_2 \iff \sigma_1\, x \sqsubseteq_{\mathbb{D}} \sigma_2\, x, \text{ for all } x \in V.$$

The following is true.

Lemma 2.2.44. *Given a join-semilattice $\mathbb{D} = (D, \sqsubseteq, \sqcup)$ and finite V,*

1. $(V \to \mathbb{D}, \sqsubseteq_{V\to\mathbb{D}})$ is a join-semilattice;

2. $(V \to \mathbb{D}, \sqsubseteq_{V\to\mathbb{D}})$ satisfies the ascending chain condition whenever $\mathbb{D}$ does;

3. $(V \to \mathbb{D}, \sqsubseteq_{V\to\mathbb{D}})$ has a finite height $h \cdot |V|$ whenever $\mathbb{D}$ has a height h. □

Theorem 2.2.45. *For any finite constraint system $S = (V, \mathbb{D}, F)$ with monadically parametric F over a join-semilattice $\mathbb{D} = (D, \sqcup, \sqsubseteq)$ that satisfies the ascending chain condition,*

1. the algorithm **RLD** *(***RLDE***) when called with (F, X), for a finite $X \subseteq V$, terminates;*

2. furthermore, if $\mathbb{D}$ has a finite height h and m is a limit of number of different variables occurring in $\overline{F}_{\mathbf{x}}$, for all $\mathbf{x} \in V$, then the algorithm terminates in $\mathcal{O}(h \cdot m \cdot |V|)$.

Proof. 1) Perhaps to some surprise, the construction of a termination argument is fairly simple. Recall that the invariants for the algorithms imply that values of variables cannot decrease (the `Inv_sigma` conjunct) and the state component *s.stable* of stable variables tends to grow after every function call (the `Inv_1` conjunct). The only point in the algorithm where *stable* temporarily shrinks is when extracting the worklist by `extract_work` function while the only possible reason for destabilization is a strictly increased value of some variable x. This gives us a clue on how to construct the termination argument for **RLD**.

We introduce a well-founded relation $(\mathtt{state}, \prec_{\mathtt{state}})$ on states as a lexicographical product on *sigma* and *stable* components ignoring the value of *infl* component. Thus, $p \prec_{\mathtt{state}} q$ iff either for some variable $\mathbf{x} : V$ the value of $\mathbf{x}$ is strictly larger in p than in q and all the rest variables did not decrease, or (in the case all the respective values are same) there are more stable variables in p than in q.

Lemma 2.2.46. $(\mathit{state}, \prec_{\mathit{state}})$ *is well-founded.*

Proof. $\prec_{\mathsf{state}}$ is a well-founded relation as a lexicographical product of well-founded relations — the reverse ordering on functions $\sqsupset_{V\to\mathbb{D}}$ and the reverse set inclusion $\supset_{2^V}$ on subsets of V.

The lemma is fully formalized in Coq. The formal proof is straightforward but quite cumbersome despite the apparent simplicity of informal arguments. In broad strokes, the formal proof evolves as follows. First, given a binary relation $(A, \leq_A)$, one defines a type of finite products A^n (vectors) and a pointwise product of relations $(A^n, \leq^n)$ on vectors using Coq's dependent types. The inductive definitions are

```
Inductive vector {A} : nat -> Type :=
  | vec_nil : vector 0
  | vec_cons : forall n, A -> vector n -> vector (S n).
```

and

```
Inductive lp_vector {A} {leA : relation A}
  : forall n, relation (vector A n) :=
  | lp_nil : lp_vector (n:=0) (vec_nil A) (vec_nil A)
  | lp_cons :
      forall a a’ n v v’,
        leA a a’ ->
        lp_vector (n:=n) v v’ ->
        lp_vector (n:=S n) (vec_cons a v) (vec_cons a’ v’).
```

respectively. Second, one shows that if $(A, \leq_A)$ satisfies the ascending chain condition then $(A^n, \leq^n)$ also does. Third, one defines a natural embedding $\phi : (V \to \mathbb{D}) \to D^n$ for $n = |V|$ preserving the pointwise ordering. The latter allows to establish well-foundedness of $(V \to \mathbb{D}, \sqsupset_{V\to\mathbb{D}})$ using Lemmas 2.2.39 and 2.2.42 and the ascending chain condition. Fourth, one shows that $(2^V, \supset_{2^V})$. Finally, one infers well-foundedness of $\prec_{\mathsf{state}}$. The whole implementation takes more than 400 lines of Coq code. $\square$

We define the equivalence of states $\equiv_{\mathsf{state}}$ by

```
Definition eq_state s1 s2 :=
  getval s1 = getval s2 /\ get_stable s1 = get_stable s2.
```

and define $\preceq_{\mathsf{state}} = \prec_{\mathsf{state}} \cup \equiv_{\mathsf{state}}$ by

```
Definition precEq_state s1 s2
  := prec_state s1 s2 \/ eq_state s1 s2.
```

The following transitivity properties hold.

Lemma 2.2.47. *1.* $\forall q, r, s : \mathsf{state}. q \prec_{\mathsf{state}} r \to r \prec_{\mathsf{state}} s \to q \prec_{\mathsf{state}} s$

2. $\forall q, r, s : \mathsf{state}. q \preceq_{\mathsf{state}} r \to r \prec_{\mathsf{state}} s \to q \prec_{\mathsf{state}} s$

3. $\forall q, r, s : \mathsf{state}. q \prec_{\mathsf{state}} r \to r \preceq_{\mathsf{state}} s \to q \prec_{\mathsf{state}} s$

4. $\forall q, r, s : \mathit{state}. q \preceq_{\mathit{state}} r \to r \preceq_{\mathit{state}} s \to q \preceq_{\mathit{state}} s$ $\square$

For each function of the algorithm, we show that whenever it terminates on the input state s_1 and returns a state s_2 then $s_2 \preceq_{\texttt{state}} s_1$ that is, in s_2 all the variable have larger values and the set of stable variables grows. By structural induction on definition of the graph of **RLD** (**RLDE**) we prove the following lemma.

```
Lemma precEq_invariant :
  (forall x y s1 ds2,
     EvalGet x y s1 ds2 ->
     let (d,s2) := ds2 in precEq_state s2 s1) /\
  (forall x f,
     EvalGet_x x f ->
     forall y s1 ds2,
       f y s1 = ds2 -> let (d,s2) := ds2 in precEq_state s2 s1) /\
  (forall x f t s1 ds2,
     Wrap_Eval_x x f t s1 ds2 ->
     let (d,s2) := ds2 in precEq_state s2 s1) /\
  (forall x s1 ds2,
     Eval_rhs x s1 ds2 ->
     let (d,s2) := ds2 in precEq_state s2 s1) /\
  (forall x s1 s2,
     Solve x s1 s2 -> precEq_state s2 s1) /\
  (forall w s1 s2,
     SolveAll w s1 s2 -> precEq_state s2 s1).
```

Consider, for example, a proof for the invariant for `EvalGet`, the simplest one. Given $x, y : V$, s, s_0, s_1 : `state` and $d : \mathbb{D}$ such that `Solve` $y\ s\ s_0$ and the induction hypothesis $s_0 \preceq_{\texttt{state}} s$ holds, and let $s_1 =$ `add_infl` $y\,x\,s_0$, $d =$ `getval` $s_0\,y$, one needs to show that $s_1 \preceq_{\texttt{state}} s$. This immediately follows from Lemma 2.2.47, since $s_1 \equiv_{\texttt{state}} s_0$. To prove the `Wrap_Eval_x` part one requires extra induction on t. Finally, we formulate

```
Theorem termination (Hwell : well_founded prec_state) :
  forall x s1, exists s2, Solve x s1 s2.
```

which claims that every call to `solve` $x\,s$ terminates, for all $x : V$, s : `state`. The sketch of the proof appears below. To show $\forall s, x. \exists s'.$ `Solve` $x\ s\ s'$, we apply the well-founded induction scheme for $\prec_{\texttt{state}}$ (Theorem 2.2.36). Take s : `state`, $x : V$ and assume the induction hypothesis

$$\forall q : \texttt{state}.\, q \prec_{\texttt{state}} s \to \forall x : V.\, \exists r : \texttt{state}.\, \texttt{Solve}\ x\ q\ r \qquad \text{(IH)}$$

The goal is to show that there exists s' such that `Solve` $x\ s\ s'$.

1. If `is_stable` $x\,s$ take $s' = s$.

2. If $\neg$`is_stable` $x\,s$, define $s_0 =$ `add_stable` $x\,s$. Then $s_0 \prec_{\texttt{state}} s$ holds. Using (IH) we can show that

$$\forall y, q.\, q \prec_{\texttt{state}} s \to \exists p_2 : \mathbb{D} \times V.\, \texttt{EvalGet}\ x\,y\,s_1\,p_2 \tag{2.3}$$

Indeed, given y and q such that $q \prec_{\texttt{state}} s$, by (IH) there exists r : `state` such that `Solve` $y\,q\,r$ holds. Define $r_1 =$ `add_infl` $y\,x\,r$ and $d =$ `getval` $r\,y$. Then `EvalGet` $x\,y\,q\,(d, r_1)$ holds. Define a relation

$$f = \lambda y.\lambda q : \texttt{state}.\lambda p : \mathbb{D} \times \texttt{state}.\, \texttt{EvalGet}\ x\,y\,q\,p.$$

One can easily check that f satisfies `EvalGet_x` $x\,f$. Then using (2.3), we prove by induction on strategy tree

$$\forall t : \mathit{Tree}.\forall q : \texttt{state}.\, q \prec s \to \exists p : \mathbb{D} \times \texttt{state}.\, \texttt{Wrap\_Eval\_x}\ x\,f\,t\,q\,p \tag{2.4}$$

Using (2.4) with the strategy tree $\overline{F}_x$, we conclude that there exist d, s_1 such that `Eval_rhs` $x\,s_0\,(d, s_1)$. Define values $\mathit{cur} =$ `getval` $s_1\,x$ and $\mathit{new} = d \sqcup \mathit{cur}$. Two cases are possible.

a) $\mathit{new} \sqsubseteq \mathit{cur}$. Put $s' = s_1$.

b) $\mathit{new} \not\sqsubseteq \mathit{cur}$. Let $s_2 =$ `setval` $x\,\mathit{new}\,s_1$ and $(w, s_3) =$ `extract_work` $x\,s_2$. One can show that $s_3 \prec s$ holds. By induction on lists, we the show

$$\forall l : \texttt{list}\, V, q : \texttt{state}, q \prec_{\texttt{state}} s \to \exists s_4, \texttt{SolveAll}\ l\,q\,s_4 \tag{2.5}$$

Applying (2.5) with w and s_3, we get s_4. Finally, put $s' = s_4$.

Termination is proven.

2) First, we note that each variable **x** can be destabilized (i.e., removed from the set *stable*) at most $h \cdot |V|$ times. The variable **x** can be destabilized only if some other variable **y** strictly increases its value (as the values of variables are updated accumulatively). For **y**, this may happen at most h times since h is the height of $\mathbb{D}$. Therefore, every variable **x** depending on **y** can be destabilized at most h times. By the assumption that m is a limit of number of variables **x** can depend on, we obtain that each **x** can be destabilized at most $h \cdot m$ times. Thus, during computation only $h \cdot m \cdot |V|$ reevaluations of right-hand sides are possible, since every evaluation of the right-hand side $F_{\mathbf{x}}$ can be triggered only if **x** is *not* in *stable*, i.e., it was destabilized. The assertion of the theorem follows. The complexity argument is not formalized in Coq.

Theorem 2.2.45 proven. □

2.2.6. The Totalized Version of RLD and Extraction

Although **RLD** (and its exact modification) is proven correct, it is not executable in the relational form implemented in Coq and cannot be extracted as an OCaml program. In order to utilize Coq's extraction facility, we apply a standard trick of *totalization* of the algorithm by bounding a priori the maximal depth of recursion. For that, we pass an additional natural parameter to each main function of the algorithm. This parameter keeps a bound for a number of possible recursive calls. Each consecutive recursive call is performed with an old value of n (received from the caller) decreased by one. Once a maximum depth of recursion is reached that is, n reaches 0, the algorithm aborts with an exception. The algorithm is then used with a threshold value n_0. In practice, n_0 can always be chosen suitably large such that this depth of recursion is never reached.

This trick allows to define the algorithm by primitive recursion on a principal natural parameter n and thus, the Coq's extraction mechanism is applicable. The totalized version of **RLD** is given in Figure 2.7. *Error* denotes the exception monad, and *StateT* is the state monad transformer. For efficiency reasons, one can slightly tweak the algorithms and avoid an extra recursive call to `solve_all` by distinguishing a case in `solve_all` when a worklist w contains exactly one element.

Coq's extraction tool is applicable to this implementation. The problem of correctness of the extracted implementation of the algorithm in ML is thus reduced to correctness of the Coq's extraction facility.

2.3. Conclusion

We have presented a certified solver **RLD** and proved that it is exact for a special subclass of constraint systems, but not exact in general. We have presented a novel solver **RLDE** which is exact and moreover, belongs to the class of chaotic iteration solvers. The latter fact makes it possible to adapt **RLDE** for using with widening and narrowing operators or a combined operator introduced in [ASV13]. This version of **RLDE** differs from the version published in the technical report [HKS13] and is more optimal. The current version of the solver became possible by virtue of characterization of purity in the general monadic framework rather than for states only.

We have formulated sufficient conditions for termination of both presented solvers. These are the finiteness of a constraint system and the ascending chain condition of a (semi-)lattice.

We have performed some experimental evaluations of **RLD** results of which are not included in the thesis. During experiments, an interesting phenomenon was revealed. On all available benchmarks, **RLD** and **RLDE** perform better if the list of variables w is reversed prior to calling `solve_all` w. It is interesting to investigate if there is any fundamental reason for such a behaviour or is it just a coincidence, or benchmark specific phenomenon. Tweaked in that way versions of the solvers perform 10–20% better than **TD** by Le Charlier and Van Hentenryck on our benchmarks while the original versions perform comparably equal to **TD**.

Further possible directions of work include versions of **RLD** (**RLDE**) that support constraint systems with side-effects and systems with multiple constraints for the same variable.

```
Fixpoint solve (n : nat) (x : Var.t) (s : state) : Error state :=
  match n with
    | 0 => error
    | S k =>
        if is_stable_b x s then
          value s
        else
          let s0 := add_stable x s in
          do p <- F x (evalget k x) s0;
          let (d, s1) := p in
          let cur := getval s1 x in
          let new := D.join cur d in
            if D.leq new cur then
              value s1
            else
              let s2 := setval x new s1 in
              let (w, s3) := extract_work x s2 in
                solve_all k w s3
  end

with solve_all (n : nat) (w : list Var.t) (s : state) : Error state :=
  match n with
    | 0 => error
    | S k =>
        match w with
          | [] => value s
          (*| [x] => solve k x s*)
          | x :: l => (solve k x s) >>= solve_all k l
        end
  end

with evalget (n : nat) (x y : Var.t) : (StateT state) Error D.t :=
  match n with
    | 0 => fun s => error
    | S k =>
        fun s =>
          do s1 <- solve k y s;
          let s2 := add_infl y x s1 in
          let d := getval s1 y in
          value (d, s2)
  end.
```

Figure 2.7.: The totalized version of **RLD** implemented in Coq

References

[AJ94] Samson Abramsky and Achim Jung. Domain theory. In *Handbook of Logic in Computer Science*, pages 1–168. Clarendon Press, 1994.

[AM96] 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.

[AMJ94] 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.

[ASV13] Kalmer Apinis, Helmut Seidl, and Vesal Vojdani. How to combine widening and narrowing for non-monotonic systems of equations. In Hans-Juergen Boehm and Cormac Flanagan, editors, *PLDI*, pages 377–386. ACM, 2013.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[BCDdS02] Gilles Barthe, Pierre Courtieu, Guillaume Dufay, and Simão Melo de Sousa. Tool-assisted specification and verification of the javacard platform. In Hélène Kirchner and Christophe Ringeissen, editors, *AMAST*, volume 2422 of *Lecture Notes in Computer Science*, pages 41–59. Springer, 2002.

[BGL04] Yves Bertot, Benjamin Grégoire, and Xavier Leroy. A structured approach to proving compiler optimizations based on dataflow analysis. In Filliâtre et al. [FPMW06], pages 66–81.

[BHK13] Andrej Bauer, Martin Hofmann, and Aleksandr Karbyshev. On monadic parametricity of second-order functionals. In Frank Pfenning, editor, *FoSSaCS*, volume 7794 of *Lecture Notes in Computer Science*, pages 225–240. Springer, 2013.

[BKV10] Nick Benton, Andrew Kennedy, and Carsten Varming. Formalizing domains, ultrametric spaces and semantics of programming languages, 2010. Submitted to Math. Struct. in Comp. Science.

[BLMP13] Sandrine Blazy, Vincent Laporte, André Maroneze, and David Pichardie. Formal verification of a C value analysis based on abstract interpretation. In *SAS*, 2013. To appear.

[Bou93] François Bourdoncle. Efficient chaotic iteration strategies with widenings. In *In Proceedings of the International Conference on Formal Methods in Programming and their Applications*, pages 128–141. Springer-Verlag, 1993.

[CC77a] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors, *POPL*, pages 238–252. ACM, 1977.

[CC77b] Patrick Cousot and Radhia Cousot. Automatic synthesis of optimal invariant assertions: Mathematical foundations. In *Proceedings of the 1977 symposium on Artificial intelligence and programming languages*, pages 1–12, New York, NY, USA, 1977. ACM.

[CGD04] Solange Coupet-Grimal and William Delobel. A uniform and certified approach for two static analyses. In Filliâtre et al. [FPMW06], pages 115–137.

[CH88] Thierry Coquand and Gérard P. Huet. The calculus of constructions. *Inf. Comput.*, 76(2/3):95–120, 1988.

[CH92] Baudouin Le Charlier and Pascal Van Hentenryck. A universal top-down fixpoint algorithm. Technical Report CS-92-25, Brown University, Providence, RI 02912, 1992.

[Cou81] Patrick Cousot. Semantic foundations of program analysis. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.

[CP88] Thierry Coquand and Christine Paulin. Inductively defined types. In Per Martin-Löf and Grigori Mints, editors, *Conference on Computer Logic*, volume 417 of *Lecture Notes in Computer Science*, pages 50–66. Springer, 1988.

[Esc12] Martín Escardó. Continuity of Gödel's system T functionals. Online, `http://www.cs.bham.ac.uk/~mhe/dialogue/dialogue.html`, July 2012.

[FPMW06] Jean-Christophe Filliâtre, Christine Paulin-Mohring, and Benjamin Werner, editors. *Types for Proofs and Programs, International Workshop, TYPES 2004, Jouy-en-Josas, France, December 15-18, 2004, Revised Selected Papers*, volume 3839 of *Lecture Notes in Computer Science*. Springer, 2006.

[Fre90] Peter J. Freyd. Recursive types reduced to inductive types. In *LICS*, pages 498–507. IEEE Computer Society, 1990.

[Fre91] Peter J. Freyd. Remarks on algebraically compact categories. In *Applications of Categories in Computer Science. Proceedings of the LMS Symposium, Durham*, volume 177, pages 95–106, 1991.

[FS99] Christian Fecht and Helmut Seidl. A faster solver for general systems of equations. *Sci. Comput. Program.*, 35(2):137–161, 1999.

[GMT08] Georges Gonthier, Assia Mahboubi, and Enrico Tassi. A Small Scale Reflection Extension for the Coq system. Rapport de recherche RR-6455, INRIA, 2008.

[GP07] Eduardo Giménez and Castéran Pierre. A tutorial on [co-]inductive types in coq, August 2007. `http://coq.inria.fr/documentation`.

[GWB98] Jürgen Giesl, Christoph Walther, and Jürgen Brauburger. Termination analysis for functional programs. In *Automated Deduction — A Basis for Applications, Vol. III, Applied Logic Series 10*, pages 135–164. Kluwer, 1998.

[Han73] Per Brinch Hansen. Concurrent programming concepts. *ACM Comput. Surv.*, 5(4):223–245, 1973.

[HKS10a] 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.

[HKS10b] 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.

[HKS13] Martin Hofmann, Aleksandr Karbyshev, and Helmut Seidl. On the verification of local generic solvers. Technical report, Technische Universität München, 2013.

[HO00] 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.

[HP07] Martin Hofmann and Mariela Pavlova. Elimination of ghost variables in program logics. In Gilles Barthe and Cédric Fournet, editors, *Proc. Trustworthy Global Computing, LNCS 4912*, volume 4912 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2007.

[Jon83] Cliff B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, 1983.

[JT93] 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.

[KA13] Aleksandr Karbyshev and Kalmer Apinis. Solvers: verified fixpoint algorithms. `https://github.com/karbyshev/solvers/`, 2013.

[Kar13] Aleksandr Karbyshev. Purity: the accompanying Coq implementation. `https://github.com/karbyshev/purity/`, 2013.

[Kat05] 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.

[Kil73] Gary A. Kildall. A unified approach to global program optimization. In Patrick C. Fischer and Jeffrey D. Ullman, editors, *POPL*, pages 194–206. ACM Press, 1973.

[KN03] Gerwin Klein and Tobias Nipkow. Verified bytecode verifiers. *Theor. Comput. Sci.*, 3(298):583–626, 2003.

[KS12] Steven Keuchel and Tom Schrijvers. Modular monadic reasoning, a (co-)routine. In Ralf Hinze, editor, *Implementation and Application of Functional Languages, 24th Symposium, Pre-Proceedings*, 2012.

[Ler09] Xavier Leroy. A formally verified compiler back-end. *J. Autom. Reasoning*, 43(4):363–446, 2009.

[Lon99] John Longley. When is a functional program not a functional program? In *ICFP*, pages 1–7, 1999.

[Lon02] John Longley. The sequentially realizable functionals. *Ann. Pure Appl. Logic*, 117(1-3):1–93, 2002.

[Mog91] Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

[OG76] Susan S. Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM*, 19(5):279–285, 1976.

[OR00] Peter W. O'Hearn and John C. Reynolds. From algol to polymorphic linear lambda-calculus. *J. ACM*, 47(1):167–223, 2000.

[PBB$^+$04] Mariela Pavlova, Gilles Barthe, Lilian Burdy, Marieke Huisman, and Jean-Louis Lanet. Enforcing high-level security properties for applets. In Jean-Jacques Quisquater, Pierre Paradinas, Yves Deswarte, and Anas Abou El Kalam, editors, *CARDIS*, pages 1–16. Kluwer, 2004.

[Pit96] Andrew M. Pitts. Relational properties of domains. *Inf. Comput.*, 127(2):66–90, 1996.

[PPM89] Frank Pfenning and Christine Paulin-Mohring. Inductively defined types in the calculus of constructions. In Michael G. Main, Austin Melton, Michael W. Mislove, and David A. Schmidt, editors, *Mathematical Foundations of Programming Semantics*, volume 442 of *Lecture Notes in Computer Science*, pages 209–228. Springer, 1989.

[Rey83] John C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, pages 513–523, 1983.

[Sim98] 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.

[SU06] Morten Heine Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard Isomorphism, Volume 149 (Studies in Logic and the Foundations of Mathematics).* Elsevier Science Inc., New York, NY, USA, 2006.

[SWH10] Helmut Seidl, Reinhard Wilhelm, and Sebastian Hack. *Übersetzerbau: Analyse und Transformation.* Springer Verlag, 2010.

[Coq12] The Coq Development Team. *The Coq proof assistant reference manual.* TypiCal Project (formerly LogiCal), 2012. Version 8.4.

[Voi09] Janis Voigtländer. Free theorems involving type constructor classes: functional pearl. In Graham Hutton and Andrew P. Tolmach, editors, *ICFP*, pages 173–184. ACM, 2009.

[VV09] Vesal Vojdani and Varmo Vene. Goblint: Path-sensitive data race analysis. In *Annales Univ. Sci. Budapest., Sect. Comp*, volume 30, pages 141–155, 2009.

[Wad89] Philip Wadler. Theorems for free! In *FPCA*, pages 347–359, 1989.

[Wad95] Philip Wadler. Monads for functional programming. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*, pages 24–52. Springer, 1995.

[Wal94] Christoph Walther. On proving the termination of algorithms by machine. *Artif. Intell.*, 71(1):101–157, 1994.

Appendices

A. Appendix to Chapter 1

A.1. Proof of Theorem 1.2.11

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

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

Thus, assume

$$\forall c.\,[\![c]\!]_T\,[\![\tau^c]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}}\,[\![c]\!]_{T'} \tag{A.1}$$

and

$$\forall x.\,(\eta\,x)\,[\![\tau^c]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}}\,(\eta'\,x). \tag{A.2}$$

(Const) $(\mathsf{val}_T[\![c]\!]_T)\,T^{\mathsf{rel}}([\![\tau]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}})\,(\mathsf{val}_{T'}[\![c]\!]_{T'})$ follows from acceptability of T^{rel} and the assumption (A.1).

(Var) $(\mathsf{val}_T(\eta\,x))\,T^{\mathsf{rel}}([\![\tau]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}})\,(\mathsf{val}_{T'}(\eta\,x))$ follows from acceptability of T^{rel} and the assumption (A.2).

(Abs) Assume that $\Gamma, x : \tau_1 \vdash e : \tau_2$ and

$$[\![e]\!]_T(\eta_1)\,T^{\mathsf{rel}}([\![\tau_2]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}})\,[\![e]\!]_{T'}(\eta_1') \tag{A.3}$$

for all η_1 for $(\Gamma, x : \tau_1)$ and T and η_1' for $(\Gamma, x : \tau_1)$ and T'. To show is

$$(\mathsf{val}_T(\lambda v.[\![e]\!]_T(\eta[x{\mapsto}v])))\,T^{\mathsf{rel}}([\![\tau_1 \to \tau_2]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}})\,(\mathsf{val}_{T'}(\lambda v.[\![e]\!]_{T'}(\eta'[x{\mapsto}v]))).$$

For that, we apply acceptability of T^{rel} (for val) and take $v\,[\![\tau_1]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}}\,v'$. It is left to show

$$([\![e]\!]_T(\eta[x{\mapsto}v]))\,T^{\mathsf{rel}}([\![\tau_2]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}})\,([\![e]\!]_{T'}(\eta[x{\mapsto}v']))$$

and it follows from the induction hypothesis (A.3).

(App) Assume that $\Gamma \vdash e_1 : \tau_1 \to \tau_2$, $\Gamma \vdash e_2 : \tau_1$ such that

$$[\![e_1]\!]_T(\eta_1)\,T^{\mathsf{rel}}([\![\tau_1 \to \tau_2]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}})\,[\![e_1]\!]_{T'}(\eta_1') \tag{A.4}$$

for all η_1 for Γ and T and η_1' for Γ and T', and

$$[\![e_2]\!]_T(\eta_2)\,T^{\mathsf{rel}}([\![\tau_1]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}})\,[\![e_2]\!]_{T'}(\eta_2') \tag{A.5}$$

for all η_2 for Γ and T and η_2' for Γ and T'. To show is

$$(\mathsf{bind}_T([\![e_1]\!]_T(\eta))\,(\mathsf{bind}_T([\![e_2]\!]_T(\eta))))\,T^{\mathsf{rel}}([\![\tau_2]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}})$$
$$(\mathsf{bind}_{T'}([\![e_1]\!]_{T'}(\eta'))\,(\mathsf{bind}_{T'}([\![e_2]\!]_{T'}(\eta')))).$$

Applying acceptability of T^{rel} (for bind), we obtain two goals:

$$[\![e_1]\!]_T(\eta)\, T^{\mathsf{rel}}([\![\tau_1]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}} \dot{\to} T^{\mathsf{rel}}([\![\tau_2]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}})\, [\![e_1]\!]_{T'}(\eta')$$

which holds by (A.4), and

$$\mathsf{bind}_T([\![e_2]\!]_T(\eta))\, (([\![\tau_1]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}} \dot{\to} T^{\mathsf{rel}}([\![\tau_2]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}})) \dot{\to} T^{\mathsf{rel}}([\![\tau_2]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}}))\, \mathsf{bind}_{T'}([\![e_2]\!]_{T'}(\eta')).$$

For the latter, assume $f : [\![\tau_1]\!]_T \to T[\![\tau_2]\!]_T$ and $f' : [\![\tau_1]\!]_{T'} \to T'[\![\tau_2]\!]_{T'}$ such that

$$f\, ([\![\tau_1]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}} \dot{\to} T^{\mathsf{rel}}([\![\tau_2]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}})\, f'. \tag{A.6}$$

It is sufficient to show

$$(\mathsf{bind}_T([\![e_2]\!]_T(\eta))\, f)\, T^{\mathsf{rel}}([\![\tau_2]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}})\, (\mathsf{bind}_{T'}([\![e_2]\!]_{T'}(\eta'))\, f')$$

which follows from acceptability of T^{rel} (for bind) and assumptions (A.5), (A.6).
(Prod) Assume that $\Gamma \vdash e_1 : \tau_1$, $\Gamma \vdash e_2 : \tau_2$ such that

$$[\![e_1]\!]_T(\eta_1)\, T^{\mathsf{rel}}([\![\tau_1]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}})\, [\![e_1]\!]_{T'}(\eta'_1) \tag{A.7}$$

for all η_1 for Γ and T and η'_1 for Γ and T', and

$$[\![e_2]\!]_T(\eta_2)\, T^{\mathsf{rel}}([\![\tau_2]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}})\, [\![e_2]\!]_{T'}(\eta'_2) \tag{A.8}$$

for all η_2 for Γ and T and η'_2 for Γ and T'. To show is

$$\begin{aligned}&\mathsf{bind}_T([\![e_1]\!]_T(\eta))\, (\mathsf{bind}_T([\![e_2]\!]_T(\eta)) \circ (\mathit{curry}\ \mathsf{val}_T^{[\![\tau_1]\!]_T \times [\![\tau_2]\!]_T}))\\ &\qquad T^{\mathsf{rel}}([\![\tau_1]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}} \dot{\times} [\![\tau_2]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}})\\ &\qquad\qquad \mathsf{bind}_{T'}([\![e_1]\!]_{T'}(\eta))\, (\mathsf{bind}_{T'}([\![e_2]\!]_{T'}(\eta)) \circ (\mathit{curry}\ \mathsf{val}_{T'}^{[\![\tau_1]\!]_{T'} \times [\![\tau_2]\!]_{T'}})).\end{aligned}$$

Using acceptability of T^{rel} (for bind) we obtain the following two goals:

$$[\![e_1]\!]_T(\eta)\, T^{\mathsf{rel}}([\![\tau_1]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}})\, [\![e_1]\!]_{T'}(\eta')$$

which holds by (A.7) and

$$\begin{aligned}&(\mathsf{bind}_T([\![e_2]\!]_T(\eta)) \circ (\mathit{curry}\ \mathsf{val}_T))\, \Big([\![\tau_1]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}} \dot{\to} T^{\mathsf{rel}}([\![\tau_1]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}} \dot{\times} [\![\tau_2]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}})\Big)\\ &\qquad (\mathsf{bind}_{T'}([\![e_2]\!]_{T'}(\eta')) \circ (\mathit{curry}\ \mathsf{val}_{T'})).\end{aligned}$$

For the latter, take x, x' such that $x\, [\![\tau_1]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}}\, x'$ and show, after simplification,

$$\begin{aligned}&(\mathsf{bind}_T([\![e_2]\!]_T(\eta))\, (\mathit{curry}\ \mathsf{val}_T\ x))\, T^{\mathsf{rel}}([\![\tau_1]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}} \dot{\times} [\![\tau_2]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}})\\ &\qquad (\mathsf{bind}_{T'}([\![e_2]\!]_{T'}(\eta))\, (\mathit{curry}\ \mathsf{val}_{T'}\ x')).\end{aligned}$$

Again, we apply acceptability of T^{rel} (for bind) and get two goals. The first one

$$[\![e_2]\!]_T(\eta)\, T^{\mathsf{rel}}([\![\tau_2]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}})\, [\![e_2]\!]_{T'}(\eta')$$

follows from (A.8). To prove the second one we take y, y' such that $y\, [\![\tau_2]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}}\, y'$ and show

$$\mathsf{val}_T(x,y)\, T^{\mathsf{rel}}([\![\tau_1]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}} \dot{\times} [\![\tau_2]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}})\ \mathsf{val}_T(x',y').$$

Indeed, it follows from acceptability of T^{rel} (for val) and the assumptions on x, x' and y, y'.

(FST) Assume that $\Gamma \vdash e : \tau_1 \times \tau_2$,

$$[\![e]\!]_T(\eta_1)\, T^{\mathsf{rel}}([\![\tau_1 \times \tau_2]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}})\, [\![e]\!]_{T'}(\eta_1') \tag{A.9}$$

for all η_1 for Γ and T and η_2' for Γ and T'. To show is

$$(\mathsf{bind}_T([\![e]\!]_T(\eta))\,(\mathsf{val}_T \circ \pi_1))\, T^{\mathsf{rel}}([\![\tau_1]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}})\, (\mathsf{bind}_{T'}([\![e]\!]_{T'}(\eta))\,(\mathsf{val}_{T'} \circ \pi_1)).$$

Applying acceptability of T^{rel} (for bind), we obtain two goals:

$$[\![e]\!]_T(\eta)\, T^{\mathsf{rel}}([\![\tau_1 \times \tau_2]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}})\, [\![e]\!]_{T'}(\eta')$$

which holds by (A.9) and

$$(\mathsf{val}_T \circ \pi_1)\, ([\![\tau_1 \times \tau_2]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}} \dot{\to} T^{\mathsf{rel}}([\![\tau_1]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}}))\, (\mathsf{val}_{T'} \circ \pi_1).$$

For the latter, take p, p' such that $p\, [\![\tau_1 \times \tau_2]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}}\, p'$ and show

$$(\mathsf{val}_T(\mathit{fst}\ p))\, T^{\mathsf{rel}}([\![\tau_1]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}})\, (\mathsf{val}_{T'}(\mathit{fst}\ p')).$$

It holds by acceptability of T^{rel} (for val) and the assumption on x, x'.

(SND) Similarly to (FST).

(LET) Assume that $\Gamma \vdash e_1 : \tau_1$, $\Gamma, x : \tau_1 \vdash e_2 : \tau_2$ and

$$[\![e_1]\!]_T(\eta_1)\, T^{\mathsf{rel}}([\![\tau_1]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}})\, [\![e_1]\!]_{T'}(\eta_1') \tag{A.10}$$

for all η_1 for Γ and T and η_1' for Γ and T', and

$$[\![e_2]\!]_T(\eta_2)\, T^{\mathsf{rel}}([\![\tau_2]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}})\, [\![e]\!]_{T'}(\eta_2') \tag{A.11}$$

for all η_2 for $(\Gamma, x : \tau_1)$ and T and η_2' for $(\Gamma, x : \tau_1)$ and T'. We show

$$\begin{aligned}(\mathsf{bind}_T([\![e_1]\!]_T(\eta))\,(\lambda v.[\![e_2]\!]_T(\eta[x{\mapsto}v])))\, T^{\mathsf{rel}}([\![\tau_2]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}})&\\ (\mathsf{bind}_{T'}([\![e_1]\!]_{T'}(\eta))\,(\lambda v.[\![e_2]\!]_{T'}(\eta'[x{\mapsto}v]))).&\end{aligned}$$

After application of acceptability of T^{rel} (for bind) the two new goals are

$$[\![e_1]\!]_T(\eta)\, T^{\mathsf{rel}}([\![\tau_1]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}})\, [\![e_1]\!]_{T'}(\eta')$$

which holds by (A.10) and

$$\lambda v.[\![e_2]\!]_T(\eta[x{\mapsto}v])\,([\![\tau_1]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}} \dot{\to} T^{\mathsf{rel}}([\![\tau_2]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}}))\,\lambda v.[\![e_2]\!]_{T'}(\eta'[x{\mapsto}v]).$$

For the second one, take v, v' such that $v\,([\![\tau_1]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}})\,v'$ and show

$$([\![e_2]\!]_T(\eta[x{\mapsto}v]))\,T^{\mathsf{rel}}([\![\tau_2]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}})\,([\![e_2]\!]_{T'}(\eta[x{\mapsto}v'])).$$

It follows from the induction hypothesis (A.11).

(Rec) (for the partial case only) Assume that $\Gamma, x : \tau_1, f : \tau_1 \to \tau_2 \vdash e : \tau_2$ and

$$[\![e]\!]_T(\eta_1)\,T^{\mathsf{rel}}([\![\tau_2]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}})\,[\![e]\!]_{T'}(\eta_1') \tag{A.12}$$

for all η_1 for $(\Gamma, x : \tau_1, f : \tau_1 \to \tau_2)$ and T and η_1' for $(\Gamma, x : \tau_1, f : \tau_1 \to \tau_2)$ and T'. We have to establish

$$\begin{aligned}(\mathsf{val}_T(\mathit{fix}\,(\lambda h.\lambda v.[\![e]\!]_T(\eta[f{\mapsto}h][x{\mapsto}v]))))\,T^{\mathsf{rel}}([\![\tau_1 \to \tau_2]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}})&\\ (\mathsf{val}_{T'}(\mathit{fix}\,(\lambda h.\lambda v.[\![e]\!]_{T'}(\eta'[f{\mapsto}h][x{\mapsto}v])))).&\end{aligned}$$

Apply acceptability of T^{rel} (for val) and take v, v' such that $v\,([\![\tau_1]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}})\,v'$. We show

$$\mathit{fix}\,(\lambda h.\lambda v.[\![e]\!]_T(\eta[f{\mapsto}h][x{\mapsto}v]))\,[\![\tau_1 \to \tau_2]\!]^{\mathsf{rel}}\,\mathit{fix}\,(\lambda h.\lambda v.[\![e]\!]_{T'}(\eta'[f{\mapsto}h][x{\mapsto}v])).$$

Since $[\![\tau_1 \to \tau_2]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}} = [\![\tau_1]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}} \dot{\to} T^{\mathsf{rel}}([\![\tau_2]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}})$ is admissible by Lemma 1.2.10, we apply the fixpoint induction principle (Lemma 1.1.9). First,

$$\bot\,([\![\tau_1]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}} \dot{\to} T^{\mathsf{rel}}([\![\tau_2]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}}))\,\bot$$

holds by strictness of T^{rel}. Second, we show that for all $h : [\![\tau_1]\!]^{\mathsf{rel}}_T \to T[\![\tau_2]\!]^{\mathsf{rel}}_T$ and $h' : [\![\tau_1]\!]^{\mathsf{rel}}_{T'} \to T'[\![\tau_2]\!]^{\mathsf{rel}}_{T'}$ such that $h\,([\![\tau_1]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}} \dot{\to} T^{\mathsf{rel}}([\![\tau_2]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}}))\,h'$

$$\lambda v.[\![e]\!]_T(\eta[f{\mapsto}h][x{\mapsto}v])\,([\![\tau_1]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}} \dot{\to} T^{\mathsf{rel}}([\![\tau_2]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}}))\,\lambda v.[\![e]\!]_{T'}(\eta[f{\mapsto}h'][x{\mapsto}v])$$

holds. For that, we take v, v' such that $v\,([\![\tau_1]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}})\,v'$ and show

$$[\![e]\!]_T(\eta[f{\mapsto}h][x{\mapsto}v])\,T^{\mathsf{rel}}([\![\tau_2]\!]^{\mathsf{rel}}_{T^{\mathsf{rel}}})\,[\![e]\!]_{T'}(\eta[f{\mapsto}h'][x{\mapsto}v']).$$

Indeed, it holds by induction hypothesis (A.12). □

B. Appendix to Chapter 2

B.1. Trace of cex to the erroneous modification of RLD

Below is the full trace of counterexample from Fig. 2.3 to the erroneous optimization of **RLD**. The counterexample only works for the implementation as in Fig. 2.2.

```
* solve t
   | sigma  = [||]
   | infl   = [||]
   | stable = {t}
   | called = {t}
   evaluate [[t]]
      * solve s
         | sigma  = [||]
         | infl   = [||]
         | stable = {s, t}
         | called = {s, t}
         evaluate [[s]]
            * solve v
               | sigma  = [||]
               | infl   = [||]
               | stable = {s, t, v}
               | called = {s, t, v}
               evaluate [[v]]
                  * solve s
                     s is stable, return
               constraint for v is satisfied: Bot >= Bot
               return
            * solve x
               | sigma  = [||]
               | infl   = [|s -> [v], v -> [s]|]
               | stable = {s, t, v, x}
               | called = {s, t, x}
               evaluate [[x]]
                  * solve s
                     s is stable, return
                  * solve u
                     | sigma  = [||]
                     | infl   = [|s -> [x, v], v -> [s]|]
                     | stable = {s, t, u, v, x}
                     | called = {s, t, u, x}
                     evaluate [[u]]
                        * solve v
                           v is stable, return
                     value of u has increased: Bot < ()
                     | sigma  = [|u -> ()|]
                     | infl   = [|s -> [x, v], v -> [u, s]|]
                     | stable = {s, t, u, v, x}
                     | called = {s, t, x}
                     extracted worklist = []
                     return
                  * solve v
            v is stable, return
      value of x has increased: Bot < ()
      | sigma  = [|u -> (), x -> ()|]
      | infl   = [|s -> [x, v], u -> [x], v -> [x, u, s]|]
      | stable = {s, t, u, v, x}
      | called = {s, t}
      extracted worklist = []
      return
value of s has increased: Bot < ()
| sigma  = [|s -> (), u -> (), x -> ()|]
| infl   = [|s -> [x, v], u -> [x], v -> [x, u, s], x -> [s]|]
| stable = {s, t, u, v, x}
| called = {t}
extracted worklist = [x, v]
| sigma  = [|s -> (), u -> (), x -> ()|]
| infl   = [|u -> [x], v -> [x, u, s], x -> [s]|]
| stable = {s, t, u}
| called = {t}
recompute variables from [x, v]:
* solve x
   | sigma  = [|s -> (), u -> (), x -> ()|]
   | infl   = [|u -> [x], v -> [x, u, s], x -> [s]|]
   | stable = {s, t, u, x}
   | called = {t, x}
   evaluate [[x]]
      * solve s
         s is stable, return
      * solve u
         u is stable, return
      * solve v
         | sigma  = [|s -> (), u -> (), x -> ()|]
         | infl   = [|s -> [x], u -> [x, x], v -> [x, u, s], x -> [s]|]
         | stable = {s, t, u, v, x}
         | called = {t, v, x}
         evaluate [[v]]
            * solve s
               s is stable, return
         value of v has increased: Bot < ()
         | sigma  = [|s -> (), u -> (), v -> (), x -> ()|]
         | infl   = [|s -> [v, x], u -> [x, x], v -> [x, u, s], x -> [s]|]
         | stable = {s, t, u, v, x}
         | called = {t, x}
         extracted worklist = [u, s]
         | sigma  = [|s -> (), u -> (), v -> (), x -> ()|]
         | infl   = [|s -> [v, x], u -> [x, x], x -> [s]|]
         | stable = {t, v, x}
         | called = {t, x}
         recompute variables from [u, s]:
         * solve u
            | sigma  = [|s -> (), u -> (), v -> (), x -> ()|]
            | infl   = [|s -> [v, x], u -> [x, x], x -> [s]|]
            | stable = {t, u, v, x}
            | called = {t, u, x}
            evaluate [[u]]
               * solve v
                  v is stable, return
            value of u has increased: () < Top
            | sigma  = [|s -> (), u -> Top, v -> (), x -> ()|]
            | infl   = [|s -> [v, x], u -> [x, x], v -> [u], x -> [s]|]
            | stable = {t, u, v, x}
            | called = {t, x}
            extracted worklist = []
```

```
                  return
               * solve s
                  | sigma  = [|s -> (), u -> Top, v -> (), x -> ()|]
                  | infl   = [|s -> [v, x], v -> [u], x -> [s]|]
                  | stable = {s, t, u, v, x}
                  | called = {s, t, x}
                  evaluate [[s]]
                     * solve v
                        v is stable, return
                     * solve x
                        x is stable, return
                  constraint for s is satisfied: () >= ()
                  return
               return
         constraint for x is satisfied: () >= ()
         return
      * solve v
         v is stable, return
      return
   value of t has increased: Bot < ()
   | sigma  = [|s -> (), t -> (), u -> Top, v -> (), x -> ()|]
   | infl   = [|s -> [t, v, x], v -> [x, s, u], x -> [s, s]|]
   | stable = {s, t, u, v, x}
   | called = {}
   extracted worklist = []
   return
```

B.2. Trace of cex to the monotonic case for RLD

Below is the full trace of counterexample from Fig. 2.5 to monotonic case for **RLD**. The counterexample demonstrate that **RLD** is *not* an exact solver. The counterexample only works for the implementation as described in Subsection 2.2.3.

```
* solve t
   | sigma  = [||]
   | infl   = [||]
   | stable = {t}
   | called = {t}
   evaluate [[t]]
      * solve s
         | sigma  = [||]
         | infl   = [||]
         | stable = {s, t}
         | called = {s, t}
         evaluate [[s]]
            * solve v
               | sigma  = [||]
               | infl   = [||]
               | stable = {s, t, v}
               | called = {s, t, v}
               evaluate [[v]]
                  * solve s
                     s is stable, return
               constraint for v is satisfied: bot >= bot
               return
            * solve x
               | sigma  = [||]
               | infl   = [|s -> [v], v -> [s]|]
               | stable = {s, t, v, x}
            | called = {s, t, x}
            evaluate [[x]]
               * solve s
                  s is stable, return
               * solve u
                  | sigma  = [||]
                  | infl   = [|s -> [x, v], v -> [s]|]
                  | stable = {s, t, u, v, x}
                  | called = {s, t, u, x}
                  evaluate [[u]]
                     * solve v
                        v is stable, return
                  constraint for u is satisfied: bot >= bot
                  return
               * solve v
                  v is stable, return
               * solve u
                  u is stable, return
            value of x has increased: bot < a
            | sigma  = [|x -> a|]
            | infl   = [|s -> [x, v], u -> [x, x], v -> [x, u, s]|]
            | stable = {s, t, u, v, x}
            | called = {s, t}
            extracted worklist = []
            return
      value of s has increased: bot < a
      | sigma  = [|s -> a, x -> a|]
      | infl   = [|s -> [x, v], u -> [x, x], v -> [x, u, s], x -> [s]|]
      | stable = {s, t, u, v, x}
      | called = {t}
      extracted worklist = [x, v]
      | sigma  = [|s -> a, x -> a|]
      | infl   = [|u -> [x, x], v -> [x, u, s], x -> [s]|]
      | stable = {s, t, u}
      | called = {t}
      recompute variables from [x, v]:
      * solve x
         | sigma  = [|s -> a, x -> a|]
         | infl   = [|u -> [x, x], v -> [x, u, s], x -> [s]|]
         | stable = {s, t, u, x}
         | called = {t, x}
         evaluate [[x]]
            * solve s
               s is stable, return
            * solve u
               u is stable, return
            * solve v
               | sigma  = [|s -> a, x -> a|]
               | infl   = [|s -> [x], u -> [x, x, x], v -> [x, u, s], x -> [s]|]
               | stable = {s, t, u, v, x}
               | called = {t, v, x}
               evaluate [[v]]
                  * solve s
                     s is stable, return
               value of v has increased: bot < a
               | sigma  = [|s -> a, v -> a, x -> a|]
               | infl   = [|s -> [v, x], u -> [x, x, x], v -> [x, u, s], x -> [s]|]
               | stable = {s, t, u, v, x}
               | called = {t, x}
               extracted worklist = [x, u, s]
               | sigma  = [|s -> a, v -> a, x -> a|]
               | infl   = [|s -> [v, x], u -> [x, x, x], x -> [s]|]
               | stable = {t, v}
               | called = {t}
               recompute variables from [x, u, s]:
               * solve x
                  | sigma  = [|s -> a, v -> a, x -> a|]
                  | infl   = [|s -> [v, x], u -> [x, x, x], x -> [s]|]
                  | stable = {t, v, x}
                  | called = {t, x}
                  evaluate [[x]]
                     * solve s
                        | sigma  = [|s -> a, v -> a, x -> a|]
                        | infl   = [|s -> [v, x], u -> [x, x, x], x -> [s]|]
                        | stable = {s, t, v, x}
                        | called = {s, t, x}
                        evaluate [[s]]
                           * solve v
                              v is stable, return
                           * solve x
                              x is stable, return
                        constraint for s is satisfied: a >= a
                        return
                     * solve u
                        | sigma  = [|s -> a, v -> a, x -> a|]
                        | infl   = [|s -> [x, v, x], u -> [x, x, x], v -> [s], x -> [s, s]|]
                        | stable = {s, t, u, v, x}
                        | called = {t, u, x}
                        evaluate [[u]]
                           * solve v
                              v is stable, return
                        value of u has increased: bot < a
                        | sigma  = [|s -> a, u -> a, v -> a, x -> a|]
                        | infl   = [|s -> [x, v, x], u -> [x, x, x], v -> [u, s], x -> [s, s]|]
                        | stable = {s, t, u, v, x}
                        | called = {t, x}
                        extracted worklist = [x, x, x]
                        | sigma  = [|s -> a, u -> a, v -> a, x -> a|]
                        | infl   = [|s -> [x, v, x], v -> [u, s], x -> [s, s]|]
                        | stable = {s, t, u, v}
                        | called = {t}
                        recompute variables from [x, x, x]:
                        * solve x
                           | sigma  = [|s -> a, u -> a, v -> a, x -> a|]
                           | infl   = [|s -> [x, v, x], v -> [u, s], x -> [s, s]|]
                           | stable = {s, t, u, v, x}
                           | called = {t, x}
                           evaluate [[x]]
                              * solve s
                                 s is stable, return
                              * solve u
                                 u is stable, return
                              * solve v
                                 v is stable, return
                           constraint for x is satisfied: a >= a
                           return
                        * solve x
                           x is stable, return
                        * solve x
                           x is stable, return
                        return
                     * solve v
                        v is stable, return
                  constraint for x is satisfied: a >= a
```

```
         return
      * solve u
         u is stable, return
      * solve s
         s is stable, return
      return
   * solve u
      u is stable, return
value of x has increased: a < top
| sigma  = [|s -> a, u -> a, v -> a, x -> top|]
| infl   = [|s -> [x, x, v, x], u -> [x, x, x], v -> [x, x, x, u, s], x -> [s, s]|]
| stable = {s, t, u, v, x}
| called = {t}
extracted worklist = [s, s]
| sigma  = [|s -> a, u -> a, v -> a, x -> top|]
| infl   = [|s -> [x, x, v, x], u -> [x, x, x], v -> [x, x, x, u, s]|]
| stable = {t, u, v, x}
| called = {t}
recompute variables from [s, s]:
* solve s
   | sigma  = [|s -> a, u -> a, v -> a, x -> top|]
   | infl   = [|s -> [x, x, v, x], u -> [x, x, x], v -> [x, x, x, u, s]|]
   | stable = {s, t, u, v, x}
   | called = {s, t}
   evaluate [[s]]
      * solve v
         v is stable, return
      * solve x
         x is stable, return
   value of s has increased: a < top
   | sigma  = [|s -> top, u -> a, v -> a, x -> top|]
   | infl   = [|s -> [x, x, v, x], u -> [x, x, x], v -> [s, x, x, x, u, s], x -> [s]|]
   | stable = {s, t, u, v, x}
   | called = {t}
   extracted worklist = [x, x, v, x]
   | sigma  = [|s -> top, u -> a, v -> a, x -> top|]
   | infl   = [|u -> [x, x, x], v -> [s, x, x, x, u, s], x -> [s]|]
   | stable = {s, t, u}
   | called = {t}
   recompute variables from [x, x, v, x]:
   * solve x
      | sigma  = [|s -> top, u -> a, v -> a, x -> top|]
      | infl   = [|u -> [x, x, x], v -> [s, x, x, x, u, s], x -> [s]|]
      | stable = {s, t, u, x}
      | called = {t, x}
      evaluate [[x]]
         * solve s
            s is stable, return
         * solve u
            u is stable, return
         * solve v
            | sigma  = [|s -> top, u -> a, v -> a, x -> top|]
            | infl   = [|s -> [x], u -> [x, x, x, x], v -> [s, x, x, x, u, s], x -> [s]|]
            | stable = {s, t, u, v, x}
            | called = {t, v, x}
            evaluate [[v]]
               * solve s
                  s is stable, return
            value of v has increased: a < top
            | sigma  = [|s -> top, u -> a, v -> top, x -> top|]
            | infl   = [|s -> [v, x], u -> [x, x, x, x], v -> [s, x, x, x, u, s], x -> [s]|]
            | stable = {s, t, u, v, x}
| called = {t, x}
extracted worklist = [s, x, x, x, u, s]
| sigma  = [|s -> top, u -> a, v -> top, x -> top|]
| infl   = [|s -> [v, x], u -> [x, x, x, x], x -> [s]|]
| stable = {t, v}
| called = {t}
recompute variables from [s, x, x, x, u, s]:
* solve s
   | sigma  = [|s -> top, u -> a, v -> top, x -> top|]
   | infl   = [|s -> [v, x], u -> [x, x, x, x], x -> [s]|]
   | stable = {s, t, v}
   | called = {s, t}
   evaluate [[s]]
      * solve v
         v is stable, return
      * solve x
         | sigma  = [|s -> top, u -> a, v -> top, x -> top|]
         | infl   = [|s -> [v, x], u -> [x, x, x, x], v -> [s], x -> [s]|]
         | stable = {s, t, v, x}
         | called = {s, t, x}
         evaluate [[x]]
            * solve s
               s is stable, return
            * solve u
               | sigma  = [|s -> top, u -> a, v -> top, x -> top|]
               | infl   = [|s -> [x, v, x], u -> [x, x, x, x], v -> [s], x -> [s]|]
               | stable = {s, t, u, v, x}
               | called = {s, t, u, x}
               evaluate [[u]]
                  * solve v
                     v is stable, return
               value of u has increased: a < top
               | sigma  = [|s -> top, u -> top, v -> top, x -> top|]
               | infl   = [|s -> [x, v, x], u -> [x, x, x, x], v -> [u, s], x -> [s]|]
               | stable = {s, t, u, v, x}
               | called = {s, t, x}
               extracted worklist = [x, x, x, x]
               | sigma  = [|s -> top, u -> top, v -> top, x -> top|]
               | infl   = [|s -> [x, v, x], v -> [u, s], x -> [s]|]
               | stable = {s, t, u, v}
               | called = {s, t}
               recompute variables from [x, x, x, x]:
               * solve x
                  | sigma  = [|s -> top, u -> top, v -> top, x -> top|]
                  | infl   = [|s -> [x, v, x], v -> [u, s], x -> [s]|]
                  | stable = {s, t, u, v, x}
                  | called = {s, t, x}
                  evaluate [[x]]
                     * solve s
                        s is stable, return
                     * solve u
                        u is stable, return
                     * solve v
                        v is stable, return
                  constraint for x is satisfied: top >= top
                  return
               * solve x
                  x is stable, return
               * solve x
                  x is stable, return
               * solve x
                  x is stable, return
```

```
                                       return
                                    * solve v
                                       v is stable, return
                                 constraint for x is satisfied: top >= top
                                 return
                           constraint for s is satisfied: top >= top
                           return
                        * solve x
                           x is stable, return
                        * solve x
                           x is stable, return
                        * solve x
                           x is stable, return
                        * solve u
                           u is stable, return
                        * solve s
                           s is stable, return
                        return
                  constraint for x is satisfied: top >= top
                  return
               * solve x
                  x is stable, return
               * solve v
                  v is stable, return
               * solve x
                  x is stable, return
               return
            * solve s
               s is stable, return
            return
         * solve v
            v is stable, return
         return
   value of t has increased: bot < top
   | sigma  = [|s -> top, t -> top, u -> top, v -> top, x -> top|]
   | infl   = [|s -> [t, x, x, v, x], u -> [x, x], v -> [x, x, x, u, s], x -> [s, s]|]
   | stable = {s, t, u, v, x}
   | called = {}
   extracted worklist = []
   return
The result solution:
[|s -> top, t -> top, u -> top, v -> top, x -> top|]
```

B.3. Functional implementation of RLDE

```
let extract_work x = fun $s$ $\rightarrow$
  let $w$ = get_infl x $s$ in
  let $s_0$ = rem_infl x $s$ in
  let $s_1$ = foldl (fun $s$ y $\rightarrow$rem_stable y (rem_called y $s$)) $s_0$ $w$ in
    ($w$, $s_1$)

let rec evalget x y : $\mathit{ErrorT}(\mathit{State}_{\mathtt{state}})\,D$ = fun $s$ $\rightarrow$
  let $s_0$ = solve y $s$ in
  let $s_1$ = add_infl y x $s_0$ in
    if not (is_called x $s_1$) then
      (error, $s_1$)
    else
      (value (getval $s_1$ y), $s_1$)

and solve x = fun $s$ $\rightarrow$
  if is_stable x $s$ then $s$ else
    let $s_0$ = add_stable x $s$ in
    let $s_1$ = add_called x $s_0$ in
    let ($o$, $s_2$) = $F$ x (evalget x) $s_1$ in
      match $o$ with
      | error $\rightarrow$ $s_2$
      | value $d$ $\rightarrow$
          let $s_3$ = rem_called x $s_2$ in
          let $\mathit{cur}$ = getval $s_3$ x in
          let $\mathit{new}$ = $\mathit{cur} \sqcup d$ in
            if ($\mathit{new} \sqsubseteq \mathit{cur}$) then $s_3$ else
            let $s_4$ = setval x $\mathit{new}$ $s_3$ in
            let ($w$, $s_5$) = extract_work x $s_4$ in
              solve_all $w$ $s_5$

and solve_all $w$ = fun $s$ $\rightarrow$
  match $w$ with
  | [] $\rightarrow$ $s$
  | x :: xs $\rightarrow$ solve_all (solve x $s$) xs

let main X =
  let $s_{init}$ = $(\emptyset,\emptyset,\emptyset)$ in
  let $s$ = solve_all X $s_{init}$ in
    (getval $s$, get_stable $s$)
```

Figure B.1.: Functional implementation of **RLDE** with explicit state passing