Imperative Abstractions for Functional Actions

Walter Guttmann

Institut für Programmiermethodik und Compilerbau,
Universität Ulm, 89069 Ulm, Germany
walter.guttmann@uni-ulm.de

Abstract

We elaborate our relational model of non-strict, imperative computations. The theory is extended to support infinite data structures. To facilitate their use in programs, we extend the programming language by concepts such as procedures, parameters, partial application, algebraic data types, pattern matching and list comprehensions. For each concept, we provide a relational semantics. Abstraction is further improved by programming patterns such as fold, unfold and divide-and-conquer. To support program reasoning, we prove laws such as fold-map fusion, otherwise known from functional programming languages. We give examples to show the use of our concepts in programs.

Keywords: fold, higher-order procedures, imperative programming, infinite data structures, lazy evaluation, non-strictness, program semantics, relations, unfold

---

1. Introduction

One of the motivations for lazy evaluation in functional programming is that it helps to improve the modularity of programs [21]. To obtain the benefits also in an imperative context, our previous works [17, 18] develop a relational model of non-strict computations. We have recently described how to extend this model by infinite data structures [19]. The basic language introduced in these works is sufficient to implement programs that construct and use infinite data structures. However, the resulting implementations are difficult to work with and hard to read, since they are entirely defined in terms of rather low-level constructs. For example, consider our implementation of the 'unfaithful' prime number sieve [25] (definitions of the basic constructs are provided in Section 3):

$$\begin{aligned}
\mathit{primes} &= \mathit{from2} \mathbin{;} \mathit{sieve} \\
\mathit{from2} &= \boldsymbol{var}\ c{\leftarrow}2 \mathbin{;} (\nu R.\ \boldsymbol{var}\ t{\leftarrow}c \mathbin{;} c{\leftarrow}c{+}1 \mathbin{;} R^{+t} \mathbin{;} \mathit{xs}{\leftarrow}t{:}\mathit{xs} \mathbin{;} \boldsymbol{end}\ t) \mathbin{;} \boldsymbol{end}\ c \\
\mathit{sieve} &= \nu R.\ \boldsymbol{var}\ p{\leftarrow}\mathit{head}(\mathit{xs}) \mathbin{;} \mathit{xs}{\leftarrow}\mathit{tail}(\mathit{xs}) \mathbin{;} \mathit{remove} \mathbin{;} R^{+p} \mathbin{;} \mathit{xs}{\leftarrow}p{:}\mathit{xs} \mathbin{;} \boldsymbol{end}\ p \\
\mathit{remove} &= \nu R.\ \boldsymbol{var}\ q, t{\leftarrow}p, \mathit{head}(\mathit{xs}) \mathbin{;} \mathit{xs}{\leftarrow}\mathit{tail}(\mathit{xs}) \mathbin{;} R^{+q,t} \mathbin{;} p{\leftarrow}q \mathbin{;} \mathit{div} \mathbin{;} \boldsymbol{end}\ q, t \\
\mathit{div} &= (\mathbb{1} \blacktriangleleft p|t \blacktriangleright \mathit{xs}{\leftarrow}t{:}\mathit{xs})
\end{aligned}$$

The relation *from2* generates and assigns to xs the infinite sequence $2, 3, 4, \ldots$ and *sieve* successively and recursively removes all multiples of the first element of xs from the rest. Due to the non-strict semantics, this program can be executed in such a way that only so many prime numbers are computed as actually required. However, it does not achieve the conciseness of its Haskell equivalent:

$$\begin{aligned}
\mathit{primes} &= \mathit{sieve}\ [2..] \\
\mathit{sieve}\ (p : \mathit{xs}) &= p : \mathit{sieve}\ [\ x \mid x \leftarrow \mathit{xs},\ p \nmid x\ ]
\end{aligned}$$

This is due to parameters, pattern matching and succinct notations such as list comprehensions available in Haskell. In a nutshell, we need to support these and further constructs to obtain a practical language. To achieve that using relations is the present paper's goal.

Section 2 gives the relational basics. A compendium of relations modelling a selection of programming constructs is presented in Section 3, where we also establish algebraic properties such as left and right unit laws, isotony, determinacy, totality and continuity. Our relational theory describes non-strict computations, which are able to yield defined results in spite of undefined inputs. The framework can also be applied to programs with infinite data structures, as demonstrated by examples constructing and modifying infinite lists. Parts of Sections 2 and 3 are derived from previous works [17, 18, 19] that also contain a detailed motivation of the general approach and particular decisions which we do not repeat in the present paper. Other parts, in particular most of Sections 3.3 and 3.4, are new and reflect the changes to the theory necessary to include function types and recursive data types.

The language extensions start with Section 4, where we introduce procedure declarations and calls with the two parameter passing mechanisms *call by value* and *call by reference*. As shown in Section 5, our procedures are amenable to partial application. Algebraic data types and pattern matching are treated in Section 6. In the remainder of the paper, we apply the concepts of Sections 4–6 to develop several patterns of higher-order programming, another key to improve modularity [21]. In particular, Sections 7 and 9 show how to express in our framework the class of fold- and unfold-computations on (finite and infinite) lists and trees. They are well-known in functional programming languages and include such operations as *map* and *concat*, the building blocks of list comprehensions as discussed in Section 8. Throughout the paper, we illustrate the concepts by examples.

Appendices A and B state and prove basic facts about parallel composition, as well as directed sets and fixpoints in partial orders. They support the theoretical development in Section 3.

In short, the contributions of this paper are the extension of our relational model of imperative, non-strict computations [17, 18, 19] by abstractions for higher-order programming, parameters and pattern matching, also in the presence of infinite data structures, and the full elaboration of the underlying theory.

2. Preliminaries

In this section we recall from [18, 19] the foundations of our relational model of imperative, non-deterministic, non-strict programs in the presence of infinite data structures. We also introduce terminology, notation and conventions used in this paper.

2.1. Variables

Characteristic features of imperative programming are variables, states and statements. We assume an infinite supply $x_1, x_2, \ldots$ of variables. Associated with each variable x_i is its type or range D_i, a set comprising all values the variable can take. Each D_i shall contain two special elements $\perp$ and ∞ with the following intuitive meaning. If the variable x_i has the value $\perp$ *and* this value is needed, the execution of the program aborts. If the variable x_i has the value ∞ *and* this value is needed, the execution of the program does not terminate. Hence $\perp$ and ∞ represent the results of undefined and non-terminating computations, respectively, in a non-strict setting. Further structure is imposed on D_i in Section 2.3.

A state is given by the values of a finite but unbounded number of variables $x_1, \ldots, x_m$ which we abbreviate as $\vec{x}$. Let $1..m$ denote the first m positive integers. Let $\vec{x}_I$ denote the subsequence of $\vec{x}$ comprising those x_i with $i \in I$ for a subset $I \subseteq 1..m$. By writing $\vec{x}{=}a$ where $a \in \{\infty, \perp\}$ we express that $x_i{=}a$ for each $i \in 1..m$. Let $D_I =_{\mathrm{def}} \prod_{i \in I} D_i$ denote the Cartesian product of the ranges of the variables x_i with $i \in I$. A state is an element $\vec{x} \in D_{1..m}$.

The effect of statements is to transform states into new states. We therefore distinguish the values of a variable x_i before and after the execution of a statement. The input value is denoted just as the variable by x_i and the output value is denoted by x_i'. In particular, both $x_i \in D_i$ and $x_i' \in D_i$. The output state $(x_1', \ldots, x_n')$ is abbreviated as $\vec{x}'$. Statements may introduce new variables into the state and remove variables from the state; then $m \neq n$.

2.2. Relations

A computation is modelled as a relation $R = R[\vec{x}, \vec{x}'] \subseteq D_{1..m} \times D_{1..n}$. An element $(\vec{x}, \vec{x}') \in R$ intuitively means that the execution of R with input values $\vec{x}$ *may* yield the output values $\vec{x}'$. The image of a state $\vec{x}$ is given by $R(\vec{x}) =_{\text{def}} \{\vec{x}' \mid (\vec{x}, \vec{x}') \in R\}$. Non-determinism is modelled by having $|R(\vec{x})| > 1$.

Another way to state the type of the relation is $R : D_{1..m} \leftrightarrow D_{1..n}$. The framework employed is that of heterogeneous relation algebra [30, 31]. We omit any notational distinction of the types of relations and their operations and assume type-correctness in their use. We also write $R[\vec{x}_{1..m}, \vec{x}'_{1..n}] : D_{1..m} \leftrightarrow D_{1..n}$ to state the names $\vec{x}_{1..m}$ and $\vec{x}_{1..n}$ of the input and output variables, respectively.

We denote the zero, identity and universal relations by $\mathbb{⊥}$, $\mathbb{I}$ and $\mathbb{T}$, respectively. Lattice join, meet and order of relations are denoted by $\cup$, $\cap$ and $\subseteq$, respectively. The Boolean complement of R is $\overline{R}$, and the converse (transposition) of R is $R^\smile$. Relational (sequential) composition of P and Q is denoted by $P \mathbin{;} Q$ and PQ. Converse has highest precedence, followed by sequential composition, followed by meet and join with lowest precedence.

A relation R is a vector iff $R\mathbb{T} = R$, total iff $R\mathbb{T} = \mathbb{T}$, univalent iff $R^\smile R \subseteq \mathbb{I}$, surjective iff $R^\smile$ is total and injective iff $R^\smile$ is univalent. A relation is a mapping iff it is both total and univalent. Frequently used relational facts are

- the Dedekind law $PQ \cap R \subseteq (P \cap RQ^\smile)(P^\smile R \cap Q)$,
- the Schröder equivalences $PQ \subseteq R \Leftrightarrow P^\smile \overline{R} \subseteq \overline{Q} \Leftrightarrow \overline{R}Q^\smile \subseteq \overline{P}$,
- $(R \cap P)Q = R \cap PQ$ if R is a vector,
- $R(P \cap Q) = RP \cap RQ$ if R is univalent, and
- $\overline{RP} = R\overline{P}$ and $PR \subseteq Q \Leftrightarrow P \subseteq QR^\smile$ if R is a mapping.

We call a set S of relations *co-directed* iff it is directed with respect to $\supseteq$, that is, if $S \neq \emptyset$ and any two relations $P, Q \in S$ have a lower bound $R \in S$ with $R \subseteq P$ and $R \subseteq Q$.

Relational constants representing computations may be specified by set comprehension as, for example, in

$$R = \{(\vec{x}, \vec{x}') \mid x'_1{=}x_2 \wedge x'_2{=}1\} = \{(\vec{x}, \vec{x}') \mid x'_1{=}x_2\} \cap \{(\vec{x}, \vec{x}') \mid x'_2{=}1\} .$$

We abbreviate such a comprehension by its constituent predicate, that is, we write $R = (x'_1{=}x_2) \cap (x'_2{=}1)$. In doing so, we use the identifier x in a generic way, possibly decorated with an index, a prime or an arrow. It follows, for example, that $\vec{x}{=}\vec{c}$ is a vector for every constant $\vec{c}$.

To form heterogeneous relations and, more generally, to change their dimensions, we use the following projection operation. Let I, J, K and L be index sets such that $I \cap K = \emptyset = J \cap L$. The dimensions of $R : D_{I\cup K} \leftrightarrow D_{J\cup L}$ are restricted by

$$(\exists \vec{x}_K, \vec{x}'_L : R) =_{\text{def}} \{(\vec{x}_I, \vec{x}'_J) \mid \exists \vec{x}_K, \vec{x}'_L : (\vec{x}_{I\cup K}, \vec{x}'_{J\cup L}) \in R\} : D_I \leftrightarrow D_J .$$

We abbreviate the case $L = \emptyset$ as $(\exists \vec{x}_K : R)$ and the case $K = \emptyset$ as $(\exists \vec{x}'_L : R)$. Observe that $(\exists \vec{x}_K : \mathbb{I}) \mathbin{;} R = (\exists \vec{x}_K : R)$ and $R \mathbin{;} (\exists \vec{x}'_L : \mathbb{I}) = (\exists \vec{x}'_L : R)$.

Defined in terms of the projection, we furthermore use the following relational parallel composition operator, similar to that of [4, 5, 27]. The parallel composition of the relations $P : D_I \leftrightarrow D_J$ and $Q : D_K \leftrightarrow D_L$ is

$$P \| Q =_{\text{def}} (\exists \vec{x}'_K : \mathbb{I}) \mathbin{;} P \mathbin{;} (\exists \vec{x}_L : \mathbb{I}) \cap (\exists \vec{x}'_I : \mathbb{I}) \mathbin{;} Q \mathbin{;} (\exists \vec{x}_J : \mathbb{I}) : D_{I\cup K} \leftrightarrow D_{J\cup L} .$$

If necessary, we write $P\,{}_I\|_K\,Q$ to clarify the partition of $I \cup K$ (a more detailed notation would also clarify the partition of $J \cup L$). Parallel composition shall have lower precedence than meet and join. Appendix A discusses several properties of parallel composition.

A *chain* is a possibly empty, totally ordered subset of a partially ordered set. Appendix B discusses properties of directed sets and fixpoints in partial orders.

2.3. Types

The state of an imperative program is given by the values of its variables, taken from the ranges D_i introduced above. To properly deal with infinite data structures, we assume that the ranges are algebraic semilattices [11], which are complete semilattices having a basis of finite elements. These structures are closed under the constructions described below and adequate for our results.

In particular, each D_i is a partial order with a least element in which suprema of directed sets exist. We denote by $\preccurlyeq : D_i \leftrightarrow D_i$ the order on D_i, let ∞ be its least element, and write $\sup S$ for the supremum of the directed set S with respect to $\preccurlyeq$. The corresponding strict order is $\prec =_{\mathrm{def}} \preccurlyeq \cap \bar{\mathbb{I}}$. The dual order of $\preccurlyeq$ is denoted by $\succcurlyeq =_{\mathrm{def}} \preccurlyeq^{\smile}$. An order similar to $\preccurlyeq$, in which $\perp$ is the least element, is discussed in [18].

Our data types are constructed as follows. Elementary types, such as the Boolean values $\mathit{Bool} =_{\mathrm{def}} \{\infty, \perp, \mathit{true}, \mathit{false}\}$ and the integer numbers $\mathit{Int} =_{\mathrm{def}} \mathbb{Z} \cup \{\infty, \perp\}$, are defined as flat partial orders, that is, $x \preccurlyeq y \Leftrightarrow_{\mathrm{def}} x = \infty \vee x = y$. Thus $\perp$ is treated like any other value except ∞, with regard to $\preccurlyeq$. The union of a finite number of types D_i is given by their separated sum $\{\infty, \perp\} \cup \{(i, x) \mid x \in D_i\}$ ordered by $x \preccurlyeq y \Leftrightarrow_{\mathrm{def}} x{=}\infty \vee x{=}\perp{=}y \vee (x{=}(i, x_i) \wedge y{=}(i, y_i) \wedge x_i \preccurlyeq_{D_i} y_i)$. The product of a finite number of types D_i is $D_I = \prod_{i \in I} D_i$ ordered by the pointwise extension of $\preccurlyeq$, that is, $\vec{x}_I \preccurlyeq \vec{y}_I \Leftrightarrow_{\mathrm{def}} \forall i \in I : x_i \preccurlyeq_{D_i} y_i$. Values of function types are ordered pointwise and $\preccurlyeq$-continuous, that is, they distribute over suprema of directed sets. Recursive data types are built by the inverse limit construction, see [28].

In [18] the ranges D_i are restricted to flat orders, which is not sufficient for infinite data structures. The extension to algebraic semilattices is introduced in [19].

The product construction plays a double role. It is not only used to build compound data types but also to represent the state of a computation with several variables. Hence the elements of the state $\vec{x} \in D_{1..m}$ are ordered by $\preccurlyeq$ and we may write $\vec{x} \preccurlyeq \vec{x}'$ to express that $x_i \preccurlyeq x_i'$ for every variable x_i.

3. Programming Constructs

In this section we elaborate our model of non-strict computations. We first recall from [18, 19] the definitions of a number of basic programming constructs. While offering brief explanations, we refer to those papers for further intuition about their choice. In Sections 3.2–3.4 we prove several algebraic properties about the programs: isotony, unit laws, determinacy, totality and continuity. First applications with infinite lists are considered in Section 3.5.

3.1. Basic Constructs

Of major importance is the order $\preccurlyeq$ on states, which we take as the new relation modelling skip, denoted also by $\mathbb{1} =_{\mathrm{def}} \preccurlyeq$. The intention underlying the definition of $\mathbb{1}$ is to enforce an upper closure of the image of each state with respect to $\preccurlyeq$, as in [16]. Our selection of constructs is inspired by [20] and rich enough to yield a basic programming and specification language.

Definition 1. We use the following relations and operations:

skip	$\mathbb{1} =_{\mathrm{def}} \preccurlyeq$
assignment	$(\vec{x} \leftarrow \vec{e}) =_{\mathrm{def}} \mathbb{1} ; (\vec{x}' {=} \vec{e}) ; \mathbb{1}$
variable declaration	$\boldsymbol{var}\ \vec{x}_K =_{\mathrm{def}} (\exists \vec{x}_K : \mathbb{1})$
variable undeclaration	$\boldsymbol{end}\ \vec{x}_K =_{\mathrm{def}} (\exists \vec{x}'_K : \mathbb{1})$
parallel composition	$P \| Q$
sequential composition	$P ; Q$
conditional	$(P \blacktriangleleft b \blacktriangleright Q) =_{\mathrm{def}} b{=}\infty \cup (b{=}\perp \cap \vec{x}'{=}\perp) \cup (b{=}\mathit{true} \cap P) \cup (b{=}\mathit{false} \cap Q)$
non-deterministic choice	$P \cup Q$
conjunction of co-directed set S	$\bigcap_{P \in S} P$
greatest fixpoint	$\nu f =_{\mathrm{def}} \bigcup \{R \mid f(R) = R\}$

Sequential composition, non-deterministic choice, conjunction and fixpoint are just the familiar operations of relation algebra. The recursive specification $R = f(R)$ is resolved as the greatest fixpoint $\nu(\lambda R.f(R))$ which we abbreviate as $\nu R.f(R)$. In particular, the iteration ***while*** b ***do*** P is just $\nu R.(P \mathbin{;} R \blacktriangleleft b \blacktriangleright \mathbb{1})$. By using the greatest fixpoint we obtain demonic non-determinism according to [5, 33]. For example, the endless loop is $(\nu R.R) = \mathbb{T}$, which absorbs any relation in a non-deterministic choice.

The assignment uses the mapping $\vec{x}'{=}\vec{e}$, where each expression $e \in \vec{e}$ may depend on the input values $\vec{x}$ of the variables, and yields exactly one value $e(\vec{x})$ from the expression's type. Thus $\vec{e}$, viewed as a function from the input to the output values, is the mapping $\vec{x}'{=}\vec{e}$. We write $(\vec{x}{\leftarrow}e)$ to assign the same expression e to all variables. Conditions are expressions with values in $Bool$ that may depend on the input $\vec{x}$. If b is a condition, the relation $b{=}c$ is a vector for each $c \in Bool$. The effect of an undefined condition in a conditional statement is to set all variables of the current state undefined. The assignment shall have higher precedence than sequential composition. The conditional shall associate to the right with lower precedence than sequential composition.

Expressions occurring on the right hand side of assignments and as conditions are assumed to be $\preccurlyeq$-continuous, hence also $\preccurlyeq$-isotone. We assume that the language of expressions contains basic operators for arithmetic, comparison, composition as well as injection and projection required in connection with data structures. Some intuition is provided by the following examples, demonstrating that computations in our setting are indeed non-strict.

Example 2. Assignments, their composition and conditionals elaborate as follows.

1. We have $(\vec{x}{\leftarrow}\vec{e}) = \{(\vec{x},\vec{x}') \mid \vec{e}(\vec{x}){\preccurlyeq}\vec{x}'\}$, thus the successor states of $\vec{x}$ under this assignment comprise the usual successor $\vec{e}(\vec{x})$ and its upper closure with respect to $\preccurlyeq$. In particular, $(\vec{x}\leftarrow\infty) = \mathbb{T}$ and $(\vec{x}{\leftarrow}\vec{c}) = (\vec{x}'{=}\vec{c})$ for each $\preccurlyeq$-maximal $\vec{c} \in D_{1..n}$. We can therefore replace the term $b{=}\infty \cup (b{=}\bot \cap \vec{x}'{=}\bot)$ in the conditional's definition by $(b{=}\infty \cap \vec{x}{\leftarrow}\infty) \cup (b{=}\bot \cap \vec{x}{\leftarrow}\bot)$.
2. The composition of two assignments amounts to $(\vec{x}\leftarrow\vec{e}) \mathbin{;} (\vec{x}\leftarrow f(\vec{x})) = (\vec{x}\leftarrow f(\vec{e}))$. In particular, $(x_1,x_2\leftarrow\bot,2) \mathbin{;} (x_1\leftarrow x_2) = (x_1,x_2\leftarrow 2,2)$ and $\mathbb{T} \mathbin{;} (x_1,x_2\leftarrow 2,2) = (x_1,x_2,\vec{x}_{3..n}\leftarrow 2,2,\infty)$. If all expressions $\vec{e}$ are constant we have $\mathbb{T} \mathbin{;} (\vec{x}{\leftarrow}\vec{e}) = (\vec{x}{\leftarrow}\vec{e})$.
3. Recalling how relational constants are specified, and using $\vec{x}_{1..m}$ as input variables, we obtain for the condition b that $(b{=}c) = \{(\vec{x},\vec{x}') \mid b(\vec{x}){=}c\} : D_{1..m} \leftrightarrow D_{1..n}$ for arbitrary $D_{1..n}$ depending on the context. The law $(P \blacktriangleleft b \blacktriangleright P) = P$ holds if b is defined, but not in general since an implementation cannot check if both branches of a conditional are equal.

Variables $\vec{x}_K$ are added to and removed from the current state by ***var*** $\vec{x}_K$ and ***end*** $\vec{x}_K$, respectively, which are projection operators adapted to satisfy the algebraic properties below. They are the only constructs to obtain inhomogeneous relations. For convenience, we introduce the let-construct for local variables. Moreover, as a special instance of relational parallel composition, we distinguish the *alphabet extension* [20].

Definition 3. Let $P : D_I \leftrightarrow D_J$ be a (possibly heterogeneous) relation and K such that $I \cap K = J \cap K = \emptyset$. The alphabet extension of P by the variables $\vec{x}_K$ is $P^{+\vec{x}_K} =_{\mathrm{def}} P\,{}_I\|_K\,\mathbb{1}$. Local variables are provided by

$$\begin{aligned} &\textbf{\textit{let}}\ \vec{x}_K\ \textbf{\textit{in}}\ P &&=_{\mathrm{def}} \textbf{\textit{var}}\ \vec{x}_K \mathbin{;} P \mathbin{;} \textbf{\textit{end}}\ \vec{x}_K \\ &\textbf{\textit{let}}\ \vec{x}_K{\leftarrow}\vec{e}_K\ \textbf{\textit{in}}\ P &&=_{\mathrm{def}} \textbf{\textit{var}}\ \vec{x}_K{\leftarrow}\vec{e}_K \mathbin{;} P \mathbin{;} \textbf{\textit{end}}\ \vec{x}_K \end{aligned}$$

The latter uses the initialised variable declaration $(\textbf{\textit{var}}\ \vec{x}_K{\leftarrow}\vec{e}_K) =_{\mathrm{def}} \textbf{\textit{var}}\ \vec{x}_K \mathbin{;} (\vec{x}_K{\leftarrow}\vec{e}_K)$.

For example, the alphabet extension is used to hide local variables from recursive calls. The values of $\vec{x}_K$ are preserved, while $\vec{x}_I$ is transformed to $\vec{x}_J$ by P. The scope of the let-construct shall extend as far to the right as possible.

3.2. Isotony and Neutrality

Observe the use of $\mathbb{1}$ in the definitions of assignment and variable (un)declaration. This is to establish skip as a left and a right unit of sequential composition.

Definition 4. $\mathscr{H}_L(P) \Leftrightarrow_{\text{def}} \mathbb{1} \mathbin{;} P = P$ and $\mathscr{H}_R(P) \Leftrightarrow_{\text{def}} P \mathbin{;} \mathbb{1} = P$ and $\mathscr{H}_E(P) \Leftrightarrow_{\text{def}} \mathscr{H}_L(P) \wedge \mathscr{H}_R(P)$.

An equivalent formulation of the latter is $\mathscr{H}_E(P) \Leftrightarrow \mathbb{1} \mathbin{;} P \mathbin{;} \mathbb{1} = P$. We first record several facts about our programming constructs and neutrality for later use.

Lemma 5.

1. *$\mathscr{H}_E(\mathbb{1})$ and $\mathbb{I} \subseteq \mathbb{1}$.*
2. *$\mathscr{H}_L(\vec{x}'{=}\vec{e} \mathbin{;} \mathbb{1})$ and hence $(\vec{x}{\leftarrow}\vec{e}) = (\vec{x}'{=}\vec{e}) \mathbin{;} \mathbb{1}$.*
3. *$\boldsymbol{var}\ \vec{x}_K = (\exists \vec{x}_K : \mathbb{I}) \mathbin{;} \mathbb{1} = \mathbb{1} \mathbin{;} (\exists \vec{x}_K : \mathbb{I})$ and hence $\mathscr{H}_E(\boldsymbol{var}\ \vec{x}_K)$.*
4. *$\boldsymbol{end}\ \vec{x}_K = \mathbb{1} \mathbin{;} (\exists \vec{x}'_K : \mathbb{I}) = (\exists \vec{x}'_K : \mathbb{I}) \mathbin{;} \mathbb{1}$ and hence $\mathscr{H}_E(\boldsymbol{end}\ \vec{x}_K)$.*
5. *Let $P : D_I \leftrightarrow D_J$ and $Q : D_K \leftrightarrow D_L$ satisfy $\mathscr{H}_E$. Then $P \| Q = \boldsymbol{end}\ \vec{x}_K \mathbin{;} P \mathbin{;} \boldsymbol{var}\ \vec{x}_L \cap \boldsymbol{end}\ \vec{x}_I \mathbin{;} Q \mathbin{;} \boldsymbol{var}\ \vec{x}_J$.*

Proof.

1. The claims amount to transitivity and reflexivity of $\preccurlyeq$.
2. We have $\mathbb{1} \mathbin{;} (\vec{x}'{=}\vec{e}) \mathbin{;} \mathbb{1} \subseteq (\vec{x}'{=}\vec{e}) \mathbin{;} \mathbb{1} \mathbin{;} \mathbb{1} = (\vec{x}'{=}\vec{e}) \mathbin{;} \mathbb{1} \subseteq \mathbb{1} \mathbin{;} (\vec{x}'{=}\vec{e}) \mathbin{;} \mathbb{1}$ by $\preccurlyeq$-isotony of $\vec{e}$ and part 1.
3. $\boldsymbol{var}\ \vec{x}_K = (\exists \vec{x}_K : \mathbb{1}) = (\exists \vec{x}_K : \mathbb{I}) \mathbin{;} \mathbb{1}$ and this equals $\mathbb{1} \mathbin{;} (\exists \vec{x}_K : \mathbb{I})$ since, letting $J = I \cup K$,

$$\begin{aligned}(\vec{x}_I, \vec{z}_J) \in (\exists \vec{x}_K : \mathbb{I}) \mathbin{;} \mathbb{1} &\Leftrightarrow (\exists \vec{y}_J : (\exists \vec{x}_K : \vec{x}_J{=}\vec{y}_J) \wedge \vec{y}_J{\preccurlyeq}\vec{z}_J) \Leftrightarrow (\exists \vec{y}_J : \vec{x}_I{=}\vec{y}_I \wedge \vec{y}_J{\preccurlyeq}\vec{z}_J) \Leftrightarrow \vec{x}_I{\preccurlyeq}\vec{z}_I \ ,\\ (\vec{x}_I, \vec{z}_J) \in \mathbb{1} \mathbin{;} (\exists \vec{x}_K : \mathbb{I}) &\Leftrightarrow (\exists \vec{y}_I : \vec{x}_I{\preccurlyeq}\vec{y}_I \wedge \exists \vec{y}_K : \vec{y}_J{=}\vec{z}_J) \Leftrightarrow (\exists \vec{y}_I : \vec{x}_I{\preccurlyeq}\vec{y}_I \wedge \vec{y}_I{=}\vec{z}_I) \Leftrightarrow \vec{x}_I{\preccurlyeq}\vec{z}_I \ .\end{aligned}$$

4. $\boldsymbol{end}\ \vec{x}_K = (\exists \vec{x}'_K : \mathbb{1}) = \mathbb{1} \mathbin{;} (\exists \vec{x}'_K : \mathbb{I})$ and this equals $(\exists \vec{x}'_K : \mathbb{I}) \mathbin{;} \mathbb{1}$ since, letting $I = J \cup K$,

$$\begin{aligned}(\vec{x}_I, \vec{z}_J) \in (\exists \vec{x}'_K : \mathbb{I}) \mathbin{;} \mathbb{1} &\Leftrightarrow (\exists \vec{y}_J : (\exists \vec{y}_K : \vec{x}_I{=}\vec{y}_I) \wedge \vec{y}_J{\preccurlyeq}\vec{z}_J) \Leftrightarrow (\exists \vec{y}_J : \vec{x}_J{=}\vec{y}_J \wedge \vec{y}_J{\preccurlyeq}\vec{z}_J) \Leftrightarrow \vec{x}_J{\preccurlyeq}\vec{z}_J \ ,\\ (\vec{x}_I, \vec{z}_J) \in \mathbb{1} \mathbin{;} (\exists \vec{x}'_K : \mathbb{I}) &\Leftrightarrow (\exists \vec{y}_I : \vec{x}_I{\preccurlyeq}\vec{y}_I \wedge \exists \vec{z}_K : \vec{y}_I{=}\vec{z}_I) \Leftrightarrow (\exists \vec{y}_I : \vec{x}_I{\preccurlyeq}\vec{y}_I \wedge \vec{y}_J{=}\vec{z}_J) \Leftrightarrow \vec{x}_J{\preccurlyeq}\vec{z}_J \ .\end{aligned}$$

5. By parts 3 and 4 we obtain

$$\begin{aligned}&\boldsymbol{end}\ \vec{x}_K \mathbin{;} P \mathbin{;} \boldsymbol{var}\ \vec{x}_L \cap \boldsymbol{end}\ \vec{x}_I \mathbin{;} Q \mathbin{;} \boldsymbol{var}\ \vec{x}_J\\ =\ &(\exists \vec{x}'_K : \mathbb{1}) \mathbin{;} P \mathbin{;} (\exists \vec{x}_L : \mathbb{1}) \cap (\exists \vec{x}'_I : \mathbb{1}) \mathbin{;} P \mathbin{;} (\exists \vec{x}_J : \mathbb{1})\\ =\ &(\exists \vec{x}'_K : \mathbb{I}) \mathbin{;} \mathbb{1} \mathbin{;} P \mathbin{;} \mathbb{1} \mathbin{;} (\exists \vec{x}_L : \mathbb{I}) \cap (\exists \vec{x}'_I : \mathbb{I}) \mathbin{;} \mathbb{1} \mathbin{;} Q \mathbin{;} \mathbb{1} \mathbin{;} (\exists \vec{x}_J : \mathbb{I})\\ =\ &\mathbb{1}P\mathbb{1} \| \mathbb{1}Q\mathbb{1}\\ =\ &P \| Q \ .\end{aligned}$$ □

The main result of this section shows isotony and the unit laws for our programs. These properties are necessary to obtain determinacy, totality and continuity in the following sections. In particular, isotony is important for the existence of fixpoints.

Theorem 6. *Let $X \in \{E, L, R\}$ and consider the constructs of Definition 1.*

1. *Functions composed of constants and those constructs are $\subseteq$-isotone.*
2. *The relations satisfying $\mathscr{H}_X$ form a complete lattice.*
3. *Relations composed of constants satisfying $\mathscr{H}_X$ and those constructs satisfy $\mathscr{H}_X$.*

Proof.

1. The operations ; and $\cup$ are isotone. The operation $\bigcap$ is pointwise isotone, that is, $\bigcap_{i \in I} P_i \subseteq \bigcap_{i \in I} Q_i$ if $P_i \subseteq Q_i$ for each $i \in I$. The operations $\cdot \blacktriangleleft \cdot \blacktriangleright \cdot$ and $\|$ are composed of these and hence isotone. The fixpoint operator ν is isotone [11, Rule 8.28 and duality]. Functions composed using isotone operations are isotone.
2. The function $\lambda P.(\mathbb{1} \mathbin{;} P)$ is a closure operator (isotone, increasing and idempotent) by isotony of ; and Lemma 5.1. Its image, comprising the relations that satisfy $\mathscr{H}_L$, thus forms a complete lattice [11, Proposition 7.2]. The same argument applies to $\lambda P.(P \mathbin{;} \mathbb{1})$ and $\mathscr{H}_R$, as well as $\lambda P.(\mathbb{1} \mathbin{;} P \mathbin{;} \mathbb{1})$ and $\mathscr{H}_E$.

3. Nested recursions are treated by assuming that the free variables are constants satisfying $\mathscr{H}_X$ and showing that the characteristic functions preserve $\mathscr{H}_X$. The proof is by structural induction with the following cases:
   * constant satisfying $\mathscr{H}_X$: trivial.
   * skip, assignment and variable (un)declaration: by Lemma 5.
   * sequential composition: by associativity.
   * non-deterministic choice: by distributivity of ; over $\cup$.
   * (arbitrary) conjunction: apply [11, Proposition 7.2] to the closure operators of part 2.
   * conditional: We first show $(P \blacktriangleleft b \blacktriangleright Q) = b{\preccurlyeq}\infty \cup (b{\preccurlyeq}\bot \cap \vec{x}{\leftarrow}\bot) \cup (b{\preccurlyeq}\mathit{true} \cap P) \cup (b{\preccurlyeq}\mathit{false} \cap Q)$. The inequality $\subseteq$ is clear since $b{=}c \subseteq b{\preccurlyeq}c$ for each $c \in \mathit{Bool}$. The reverse inequality follows since $b{\preccurlyeq}c \cap S = (b{=}\infty \cup b{=}c) \cap S = (b{=}\infty \cap S) \cup (b{=}c \cap S) \subseteq b{=}\infty \cup (b{=}c \cap S)$ for any relation S. Now assume $\mathscr{H}_X(P)$ and $\mathscr{H}_X(Q)$, then $\mathscr{H}_X(P \blacktriangleleft b \blacktriangleright Q)$ follows by the cases choice, conjunction and assignment above, if we can show $\mathscr{H}_E(b{\preccurlyeq}c)$ for each $c \in \mathit{Bool}$. But this holds by $\preccurlyeq$-isotony of b since $(\vec{x}, \vec{x}') \in \mathbb{1} \mathbin{;} (b{\preccurlyeq}c) \mathbin{;} \mathbb{1} \Leftrightarrow (\exists \vec{y}, \vec{z} : \vec{x}{\preccurlyeq}\vec{y} \wedge b(\vec{y}){\preccurlyeq}c \wedge \vec{z}{\preccurlyeq}\vec{x}') \Rightarrow b(\vec{x}){\preccurlyeq}c \Leftrightarrow (\vec{x}, \vec{x}') \in (b{\preccurlyeq}c)$.
   * parallel composition: Assume $\mathscr{H}_L(P)$ and $\mathscr{H}_L(Q)$, then $\mathbb{1}(P \| Q) = (\mathbb{1} \| \mathbb{1})(P \| Q) = \mathbb{1}P \| \mathbb{1}Q = P \| Q$ by Lemmas 38.5 and 38.4 in Appendix A. Assume $\mathscr{H}_R(P)$ and $\mathscr{H}_R(Q)$, then similarly $(P \| Q)\mathbb{1} = (P \| Q)(\mathbb{1} \| \mathbb{1}) = P\mathbb{1} \| Q\mathbb{1} = P \| Q$.
   * greatest fixpoint: To show $\mathscr{H}_X(\nu f)$, observe that f is isotone by part 1 and preserves $\mathscr{H}_X$ by the induction hypothesis. By the case conjunction above, the relations satisfying $\mathscr{H}_X$ are closed under infima of chains. Therefore $\mathscr{H}_X(\nu f)$ by Corollary 42 in Appendix B. $\square$

3.3. Determinacy and Totality

Our next goal is to establish continuity, see Section 3.4. As a preparatory step, we describe deterministic computations. This is because unbounded non-determinism breaks continuity as shown, for example, in [13, Chapter 9] and [8, Section 5.7]. Although Definition 1 admits only finite choice, we obtain unbounded non-determinism if it is used within (recursively constructed) infinite data structures, see Example 17.

This can be remedied in either of two ways: by restriction to orders with finite height or to deterministic programs. The former approach [18] suffices for basic data structures, but excludes functions as values and infinite data structures. In this paper, we follow [19] and obtain continuity by not using the non-deterministic choice. While the restriction to deterministic programs may seem harsh, it is characteristic of many programming languages and does not preclude the use of non-deterministic choice for specification purposes. We characterise deterministic computations in our context by the following condition $\mathscr{H}_D$ whose use was suggested by a referee.

Definition 7. The pointwise least elements of the relation P with respect to $\preccurlyeq$ are given by $\operatorname{lea} P =_{\mathrm{def}} P \cap \overline{\overline{P}\succcurlyeq}$, similarly to constructions in [31, Chapter 3.3]. Let $\mathscr{H}_D(P)$ hold iff $\operatorname{lea} P$ is total.

Hence $\mathscr{H}_D$ holds iff the image set $P(\vec{x})$ of every input $\vec{x}$ has a least element. The pointwise least elements with respect to $\preccurlyeq$ account for the upper closure. We first record several facts about determinacy for later use.

Lemma 8.

1. *The relation* $\operatorname{lea} P$ *is univalent, and hence* $\mathscr{H}_D(P)$ *holds iff* $\operatorname{lea} P$ *is a mapping.*
2. *Let* $\mathscr{H}_D(P)$*, then* P *is total.*
3. *Let* $\mathscr{H}_D(P)$*, then* $P \subseteq (\operatorname{lea} P){\preccurlyeq}$*. Let* $\mathscr{H}_R(P)$*, then* $P \supseteq (\operatorname{lea} P){\preccurlyeq}$*.*
4. *Let* P *be a mapping and* $\mathscr{H}_D(Q)$*, then* $\mathscr{H}_D(PQ)$*.*

PROOF.

1. $(\operatorname{lea} P)^{\smile}(\operatorname{lea} P) = (P \cap \overline{P\overline{\succcurlyeq}})^{\smile}(P \cap \overline{P\overline{\succcurlyeq}}) \subseteq \overline{P\overline{\succcurlyeq}}^{\smile} P \cap P^{\smile}\overline{P\overline{\succcurlyeq}} \subseteq \succcurlyeq^{\smile} \cap \succcurlyeq \; = \mathbb{I}$ using the Schröder law in the third step, and antisymmetry of $\preccurlyeq$ in the final step.
2. $P\mathbb{T} \supseteq (\operatorname{lea} P)\mathbb{T} = \mathbb{T}$.
3. $(\operatorname{lea} P)\preccurlyeq \; \subseteq P\preccurlyeq \; = P$ by $\mathscr{H}_R(P)$. By $\mathscr{H}_D(P)$ we have $P = P \cap \mathbb{T} = P \cap (\operatorname{lea} P)\mathbb{T} = P \cap (\operatorname{lea} P)(\preccurlyeq \cup \overline{\preccurlyeq}) \subseteq (\operatorname{lea} P)\preccurlyeq \cup \, (P \cap \overline{P\overline{\succcurlyeq}}\,\overline{\preccurlyeq}) \subseteq (\operatorname{lea} P)\preccurlyeq$ since $P \cap \overline{P\overline{\succcurlyeq}}\,\overline{\preccurlyeq} \subseteq (\overline{P\overline{\succcurlyeq}} \cap P\overline{\succcurlyeq})\overline{\preccurlyeq} \subseteq \mathbb{L}$ by the Dedekind law.
4. $\operatorname{lea}(PQ) = PQ \cap \overline{PQ\overline{\succcurlyeq}} = PQ \cap P\overline{Q\overline{\succcurlyeq}} = P(Q \cap \overline{Q\overline{\succcurlyeq}}) = P \operatorname{lea} Q$ since P is a mapping. Hence $\operatorname{lea}(PQ)\mathbb{T} = P(\operatorname{lea} Q)\mathbb{T} = P\mathbb{T} = \mathbb{T}$. □

As expected, we have to exclude the non-deterministic choice from the following result which shows determinacy for our programming constructs.

Theorem 9. *Relations composed of constants satisfying $\mathscr{H}_D$ and $\mathscr{H}_E$ and the constructs of Definition 1 without the choice operator satisfy $\mathscr{H}_D$.*

Proof. Nested recursions are treated by assuming that the free variables are constants satisfying $\mathscr{H}_D$ and $\mathscr{H}_E$ and showing that the characteristic functions preserve $\mathscr{H}_D$ and $\mathscr{H}_E$. The proof is by structural induction with the following cases:

* constant satisfying $\mathscr{H}_D$ and $\mathscr{H}_E$: trivial.

* skip: $\operatorname{lea} \mathbb{1} = \; \preccurlyeq \cap \overline{\preccurlyeq\overline{\succcurlyeq}} \subseteq \; \preccurlyeq \cap \overline{\mathbb{I}\overline{\succcurlyeq}} = \; \preccurlyeq \cap \succcurlyeq \; = \mathbb{I} = \; \preccurlyeq \cap \, \mathbb{I} \subseteq \; \preccurlyeq \cap \overline{\preccurlyeq\overline{\succcurlyeq}}$ since $\preccurlyeq\overline{\succcurlyeq} \subseteq \overline{\mathbb{I}}$ by the Schröder law, hence $\operatorname{lea} \mathbb{1} = \mathbb{I}$ is total.

* assignment: $(\vec{x} \leftarrow \vec{e}) = (\vec{x}' = \vec{e}) \; ; \mathbb{1}$ by Lemma 5.2, hence $\mathscr{H}_D(\vec{x} \leftarrow \vec{e})$ by Lemma 8.4 since $\vec{x}' = \vec{e}$ is a mapping and $\mathscr{H}_D(\mathbb{1})$ by the case skip above. In particular, $\mathscr{H}_D(\mathbb{T})$ by choosing $\vec{e} = \infty$.

* variable declaration: Let $\boldsymbol{var}\ \vec{x}_K : D_I \leftrightarrow D_{I \cup K}$, then $\boldsymbol{var}\ \vec{x}_K = \{(\vec{x}_I, \vec{x}'_{I \cup K}) \mid \vec{x}_I \preccurlyeq \vec{x}'_I\} = P \; ; \mathbb{1}$ using the mapping $P = \{(\vec{x}_I, \vec{x}'_{I \cup K}) \mid \vec{x}'_I = \vec{x}_I \wedge \vec{x}'_K = \infty\}$, hence $\mathscr{H}_D(\boldsymbol{var}\ \vec{x}_K)$ by Lemma 8.4.

* variable undeclaration: $\boldsymbol{end}\ \vec{x}_K = (\exists \vec{x}'_K : \mathbb{I}) \; ; \mathbb{1}$ by Lemma 5.4, hence $\mathscr{H}_D(\boldsymbol{end}\ \vec{x}_K)$ by Lemma 8.4 since $\exists \vec{x}'_K : \mathbb{I}$ is a mapping.

* sequential composition: To show $\mathscr{H}_D(PQ)$, observe that $\mathscr{H}_D(P)$ and $\mathscr{H}_D(Q)$ by the induction hypothesis, and $\mathscr{H}_L(Q)$ by Theorem 6.3. Then $PQ \subseteq (\operatorname{lea} P)\preccurlyeq Q = (\operatorname{lea} P)Q \subseteq PQ$ by Lemma 8.3, hence $PQ = (\operatorname{lea} P)Q$, thus $\mathscr{H}_D(PQ)$ by Lemma 8.4 since $\operatorname{lea} P$ is a mapping by Lemma 8.1.

* conjunction of co-directed set S: For each $P \in S$ we have $\mathscr{H}_D(P)$ by the induction hypothesis and $\mathscr{H}_R(P)$ by Theorem 6.3. To show $\mathscr{H}_D(\bigcap S)$ we construct for every $\vec{x}$ an $\vec{x}'$ such that $(\vec{x}, \vec{x}') \in \operatorname{lea} \bigcap S$; let $\vec{x}$ be given. For each $P \in S$ there is an $\vec{x}'_P$ such that $(\vec{x}, \vec{x}'_P) \in \operatorname{lea} P$ by $\mathscr{H}_D(P)$, hence $(\vec{x}, \vec{x}'_P) \in P$ and $(\vec{x}, \vec{y}) \in P \Rightarrow \vec{x}'_P \preccurlyeq \vec{y}$ for every $\vec{y}$. The set $M =_{\mathrm{def}} \{\vec{x}'_P \mid P \in S\}$ is directed since S is co-directed and $P(\vec{x}) \subseteq Q(\vec{x}) \Rightarrow \vec{x}'_Q \preccurlyeq \vec{x}'_P$ for any $P, Q \in S$. Hence $\vec{x}' =_{\mathrm{def}} \sup M$ exists. For each $P \in S$ we have $(\vec{x}, \vec{x}'_P) \in P$ and $\vec{x}'_P \preccurlyeq \vec{x}'$, hence $(\vec{x}, \vec{x}') \in P\preccurlyeq \; = P$ by $\mathscr{H}_R(P)$, thus $(\vec{x}, \vec{x}') \in \bigcap S$. Let $\vec{y}$ be given such that $(\vec{x}, \vec{y}) \in \bigcap S$, hence $(\vec{x}, \vec{y}) \in P$ and $\vec{x}'_P \preccurlyeq \vec{y}$ for each $P \in S$, thus $\vec{x}' \preccurlyeq \vec{y}$. Therefore $(\vec{x}, \vec{x}') \in \operatorname{lea} \bigcap S$.

* conditional: Observe that $(P \blacktriangleleft b \blacktriangleright Q) = \bigcup_{i=1}^{4} b{=}c_i \cap R_i$ using $c_{1..4} = \infty, \bot, \mathit{true}, \mathit{false}$ and $R_{1..4} = \vec{x} \leftarrow \infty, \vec{x} \leftarrow \bot, P, Q$. For each $i, j \in 1..4$ we have that $b{=}c_i$ is a vector and $b{=}c_i \subseteq \overline{b{=}c_j}$ if $i \neq j$. Hence

$$\begin{aligned} \operatorname{lea}(P \blacktriangleleft b \blacktriangleright Q) &= (\textstyle\bigcup_{i=1}^{4} b{=}c_i \cap R_i) \cap \overline{(\bigcup_{j=1}^{4} b{=}c_j \cap R_j)\overline{\succcurlyeq}} = \bigcup_{i=1}^{4} b{=}c_i \cap R_i \cap \overline{\bigcup_{j=1}^{4} (b{=}c_j \cap R_j)\overline{\succcurlyeq}} \\ &= \textstyle\bigcup_{i=1}^{4} b{=}c_i \cap R_i \cap \bigcap_{j=1}^{4} \overline{b{=}c_j \cap R_j\overline{\succcurlyeq}} = \bigcup_{i=1}^{4} b{=}c_i \cap R_i \cap \bigcap_{j=1}^{4} \overline{b{=}c_j} \cup \overline{R_j\overline{\succcurlyeq}} \\ &= \textstyle\bigcup_{i=1}^{4} b{=}c_i \cap R_i \cap (\overline{b{=}c_i} \cup \overline{R_i\overline{\succcurlyeq}}) = \bigcup_{i=1}^{4} b{=}c_i \cap R_i \cap \overline{R_i\overline{\succcurlyeq}} = \bigcup_{i=1}^{4} b{=}c_i \cap \operatorname{lea} R_i \end{aligned}$$

  and therefore $(\operatorname{lea}(P \blacktriangleleft b \blacktriangleright Q))\mathbb{T} = \bigcup_{i=1}^{4} (b{=}c_i \cap \operatorname{lea} R_i)\mathbb{T} = \bigcup_{i=1}^{4} b{=}c_i \cap (\operatorname{lea} R_i)\mathbb{T} = \bigcup_{i=1}^{4} (b{=}c_i) = \mathbb{T}$ since $\mathscr{H}_D(R_i)$ by the case assignment above and the induction hypothesis.

* parallel composition: Assume $\mathscr{H}_D(P)$ and $\mathscr{H}_D(Q)$ by the induction hypothesis, then P and Q are total by Lemma 8.2, and by Lemma 38 in Appendix A we obtain

$$\begin{aligned}
\mathrm{lea}(P\|Q) &= (P\|Q) \cap \overline{(P\|Q)\overline{\succcurlyeq}} = (P\|Q) \cap \overline{(P\|Q)\overline{\succcurlyeq\|\succcurlyeq}} = (P\|Q) \cap \overline{(P\|Q)((\overline{\succcurlyeq}\|\mathbb{T}) \cup (\mathbb{T}\|\overline{\succcurlyeq}))} \\
&= (P\|Q) \cap \overline{(P\|Q)(\overline{\succcurlyeq}\|\mathbb{T}) \cup (P\|Q)(\mathbb{T}\|\overline{\succcurlyeq})} = (P\|Q) \cap \overline{(P\overline{\succcurlyeq}\|Q\mathbb{T}) \cup (P\mathbb{T}\|Q\overline{\succcurlyeq})} \\
&= (P\|Q) \cap \overline{P\overline{\succcurlyeq}\|\mathbb{T}} \cap \overline{\mathbb{T}\|Q\overline{\succcurlyeq}} = (P\|Q) \cap (\overline{P\overline{\succcurlyeq}}\|\mathbb{T}) \cap (\mathbb{T}\|\overline{Q\overline{\succcurlyeq}}) = P \cap \overline{P\overline{\succcurlyeq}}\|Q \cap \overline{Q\overline{\succcurlyeq}}
\end{aligned}$$

and therefore $(\mathrm{lea}(P\|Q))\mathbb{T} = ((\mathrm{lea}\,P)\|(\mathrm{lea}\,Q))(\mathbb{T}\|\mathbb{T}) = (\mathrm{lea}\,P)\mathbb{T}\|(\mathrm{lea}\,Q)\mathbb{T} = \mathbb{T}\|\mathbb{T} = \mathbb{T}$.

* greatest fixpoint: To show $\mathscr{H}_D(\nu f)$, apply Corollary 42 in Appendix B to the set S of relations satisfying $\mathscr{H}_D$ and $\mathscr{H}_E$. This set is closed under infima of chains as shown in the cases assignment and conjunction above and in Theorem 6.3. Moreover, S is closed under f by the induction hypothesis and Theorem 6.3. Finally, f is isotone by Theorem 6.1. $\square$

An important consequence is that our programs are total, hence for every input state there exists an output state. Totality holds also for computations using the non-deterministic choice.

Theorem 10. *Relations composed of constants satisfying $\mathscr{H}_D$ and $\mathscr{H}_E$ and the constructs of Definition 1 are total.*

Proof. Let R be such a relation and S the relation obtained from R by replacing every non-deterministic choice $P \cup Q$ with P. Then $\mathscr{H}_D(S)$ by Theorem 9, hence S is total by Lemma 8.2. But $S \subseteq R$ by Theorem 6.1, hence R is total, too. $\square$

3.4. Continuity

A function f on relations is called *co-continuous* iff it distributes over infima of co-directed sets of relations, formally $f(\bigcap S) = \bigcap_{P \in S} f(P)$ for every co-directed set S. Instead of the chains used in [18] we now switch to co-directed sets to match the algebraic semilattice structure on value ranges, see [1] for the correspondence. The importance of co-continuity comes from the permission to represent the greatest fixpoint νf by the constructive $\bigcap_{n \in \mathbb{N}} f^n(\mathbb{T})$ according to Kleene's theorem. This enables the approximation of νf by repeatedly unfolding f, which simulates recursive calls of the modelled computation. We use this, for instance, in Example 15 and Theorem 32 below. The following condition $\mathscr{H}_C$ generalises $\preccurlyeq$-continuity to relations.

Definition 11. Let $\mathscr{H}_C(P)$ hold iff $(\forall \vec{x} \in S : (\vec{x}, \vec{x}') \in P) \Rightarrow (\sup S, \vec{x}') \in P$ for every directed set S ordered by $\preccurlyeq$.

Several facts about $\mathscr{H}_C$ are recorded in the following lemma. It shows a close correspondence of our programs to continuous mappings, to be used for higher-order procedures in Section 7, and the conditions under which sequential composition distributes over infima of co-directed sets.

Lemma 12.

1. *Let P be a mapping. Then P is $\preccurlyeq$-continuous iff $\mathscr{H}_C(P\preccurlyeq)$ and P is $\preccurlyeq$-isotone.*
2. *Let P satisfy $\mathscr{H}_D(P)$ and $\mathscr{H}_R(P)$. Then $\mathrm{lea}\,P$ is $\preccurlyeq$-continuous iff $\mathscr{H}_C(P)$ and $\mathscr{H}_L(P)$.*
3. *Let Q satisfy $\mathscr{H}_L(Q)$. Then $\mathscr{H}_C(Q)$ iff for every co-directed set S of relations satisfying $\mathscr{H}_D$ and $\mathscr{H}_R$ we have $(\bigcap S)Q = \bigcap_{P \in S} PQ$.*
4. *Let S be a co-directed set such that $\mathscr{H}_L(Q)$ for each $Q \in S$, and let P be such that $\mathscr{H}_D(P)$. Then $P(\bigcap S) = \bigcap_{Q \in S} PQ$.*
5. *Let $\mathscr{H}_C(P)$ and $\mathscr{H}_C(Q)$. Then $\mathscr{H}_C(P \cup Q)$.*

Proof.

1. For the forward implication, let S be a directed set ordered by $\preccurlyeq$ and $\vec{x}'$ such that $(\vec{x}, \vec{x}') \in P\preccurlyeq$ for each $\vec{x} \in S$, hence $P(\vec{x})\preccurlyeq\vec{x}'$. By continuity, $P(\sup S) = \sup\{P(\vec{x}) \mid \vec{x} \in S\} \preccurlyeq \vec{x}'$ or $(\sup S, \vec{x}') \in P\preccurlyeq$. Hence $\mathscr{H}_C(P\preccurlyeq)$ holds, while isotony immediately follows from continuity.
For the backward implication, let S be a directed set ordered by $\preccurlyeq$. Then $T =_{\text{def}} \{P(\vec{x}) \mid \vec{x} \in S\}$ is directed since P is isotone, hence $\sup T$ exists and satisfies $(\vec{x}, \sup T) \in P\preccurlyeq$ for each $\vec{x} \in S$ since $P(\vec{x})\preccurlyeq \sup T$. Thus $(\sup S, \sup T) \in P\preccurlyeq$ by $\mathscr{H}_C(P\preccurlyeq)$, that is, $P(\sup S)\preccurlyeq \sup T$. The reverse inequality holds since $P(\vec{x})\preccurlyeq P(\sup S)$ for each $\vec{x} \in S$ by isotony of P. Thus $P(\sup S) = \sup T$, showing continuity.
2. Assume $\mathscr{H}_D(P)$ and $\mathscr{H}_R(P)$, then $\operatorname{lea} P$ is a mapping by Lemma 8.1 and $P = (\operatorname{lea} P)\preccurlyeq$ by Lemma 8.3. By part 1 it suffices to show $\mathscr{H}_C((\operatorname{lea} P)\preccurlyeq)$ iff $\mathscr{H}_C(P)$, which is immediate, and that $\operatorname{lea} P$ is $\preccurlyeq$-isotone iff $\mathscr{H}_L(P)$. But $\mathscr{H}_L(P)$ implies $\preccurlyeq(\operatorname{lea} P) \subseteq \preccurlyeq P = P = (\operatorname{lea} P)\preccurlyeq$ which states isotony [24, 12, 29], and that in turn implies $\mathscr{H}_L(P)$ by $\preccurlyeq P = \preccurlyeq(\operatorname{lea} P)\preccurlyeq \subseteq (\operatorname{lea} P)\preccurlyeq\preccurlyeq = (\operatorname{lea} P)\preccurlyeq = P$.
3. For the forward implication, assume $\mathscr{H}_L(Q)$ and $\mathscr{H}_C(Q)$. Let S be a co-directed set of relations such that $\mathscr{H}_D(P)$ and $\mathscr{H}_R(P)$ for each $P \in S$. By the meet property it suffices to show $\bigcap_{P\in S} PQ \subseteq (\bigcap S)Q$. Let $(\vec{x}, \vec{x}') \in \bigcap_{P\in S} PQ$, hence for each $P \in S$ there is a $\vec{y}_P$ such that $(\vec{x}, \vec{y}_P) \in \operatorname{lea} P$ and $(\vec{y}_P, \vec{x}') \in Q$ since $PQ = (\operatorname{lea} P)\preccurlyeq Q = (\operatorname{lea} P)Q$ by Lemma 8.3. The set $M =_{\text{def}} \{\vec{y}_P \mid P \in S\}$ is directed since S is co-directed and $U(\vec{x}) \subseteq V(\vec{x}) \Rightarrow \vec{y}_V\preccurlyeq\vec{y}_U$ for any $U, V \in S$. Hence $\vec{y} =_{\text{def}} \sup M$ exists and satisfies $(\vec{y}, \vec{x}') \in Q$ by $\mathscr{H}_C(Q)$. Moreover $\vec{y}_P\preccurlyeq\vec{y}$ for each $P \in S$, hence $(\vec{x}, \vec{y}) \in (\operatorname{lea} P)\preccurlyeq = P$ by Lemma 8.3. Thus $(\vec{x}, \vec{y}) \in \bigcap S$ and $(\vec{x}, \vec{x}') \in (\bigcap S)Q$.
For the backward implication, assume $\mathscr{H}_L(Q)$ and $\bigcap_{P\in S} PQ = (\bigcap S)Q$ for every co-directed set S such that $\mathscr{H}_R(P)$ and $\mathscr{H}_D(P)$ for each $P \in S$. To show $\mathscr{H}_C(Q)$, let T be a directed set ordered by $\preccurlyeq$ and $\vec{x}'$ such that $(\vec{x}, \vec{x}') \in Q$ for each $\vec{x} \in T$. Define $P_{\vec{x}} =_{\text{def}} \{(\vec{v}, \vec{y}) \mid \vec{x}\preccurlyeq\vec{y}\}$, then $\mathscr{H}_D(P_{\vec{x}})$ since $(\operatorname{lea} P_{\vec{x}})(\vec{v}) = \{\vec{x}\}$ for every $\vec{v}$, and $\mathscr{H}_R(P_{\vec{x}})$ since $P_{\vec{x}}\preccurlyeq = \{(\vec{v}, \vec{z}) \mid \exists\vec{y} : \vec{x}\preccurlyeq\vec{y}\wedge\vec{y}\preccurlyeq\vec{z}\} = \{(\vec{v}, \vec{z}) \mid \vec{x}\preccurlyeq\vec{z}\} = P_{\vec{x}}$. The set of relations $S =_{\text{def}} \{P_{\vec{x}} \mid \vec{x} \in T\}$ is co-directed since T is directed and $\vec{y}\preccurlyeq\vec{z} \Rightarrow P_{\vec{z}} \subseteq P_{\vec{y}}$ for any $\vec{y}, \vec{z} \in T$. For an arbitrary $\vec{v}$ we have $(\vec{v}, \vec{x}) \in P_{\vec{x}}$ and thus $(\vec{v}, \vec{x}') \in P_{\vec{x}}Q$ for each $\vec{x} \in T$, whence $(\vec{v}, \vec{x}') \in \bigcap_{\vec{x}\in T} P_{\vec{x}}Q = (\bigcap S)Q$ by the assumption. Therefore $\vec{y}$ exists such that $(\vec{y}, \vec{x}') \in Q$ and $(\vec{v}, \vec{y}) \in P_{\vec{x}}$ for each $\vec{x} \in T$, hence also $\vec{x}\preccurlyeq\vec{y}$. Thus $\sup T\preccurlyeq\vec{y}$ and $(\sup T, \vec{x}') \in \preccurlyeq Q = Q$ by $\mathscr{H}_L(Q)$.
4. $P(\bigcap S) \subseteq \bigcap_{Q\in S} PQ \subseteq \bigcap_{Q\in S}(\operatorname{lea} P)\preccurlyeq Q = \bigcap_{Q\in S}(\operatorname{lea} P)Q = (\operatorname{lea} P)(\bigcap S) \subseteq P(\bigcap S)$ by Lemmas 8.3 and 8.1.
5. Let S be a directed set ordered by $\preccurlyeq$ and $\vec{x}'$ such that $(\vec{x}, \vec{x}') \in P \cup Q$ for each $\vec{x} \in S$. Define $S_P =_{\text{def}} \{\vec{x} \mid \vec{x} \in S \wedge (\vec{x}, \vec{x}') \in P\}$ and $S_Q =_{\text{def}} \{\vec{x} \mid \vec{x} \in S \wedge (\vec{x}, \vec{x}') \in Q\}$. By Theorem 39 in Appendix B one of the following three cases holds:
   * S_P is directed but $S\setminus S_P$ is not, and $\sup S = \sup S_P$. Then $(\sup S, \vec{x}') = (\sup S_P, \vec{x}') \in P \subseteq P\cup Q$ by $\mathscr{H}_C(P)$.
   * $S \setminus S_P$ is directed but S_P is not, and $\sup S = \sup(S \setminus S_P)$. Then $(\sup S, \vec{x}') = (\sup(S \setminus S_P), \vec{x}') \in Q \subseteq P \cup Q$ by $\mathscr{H}_C(Q)$ since $S \setminus S_P \subseteq S_Q$.
   * Both S_P and $S \setminus S_P$ are directed, and $\sup S = \sup S_P$ or $\sup S = \sup(S \setminus S_P)$. Then continue as in the first or the second case, respectively. ☐

The previous lemma also shows that non-deterministic choice preserves $\mathscr{H}_C$, but since it does not preserve $\mathscr{H}_D$, this operator is not included in the following closure and continuity results.

Theorem 13. *Relations composed of constants satisfying $\mathscr{H}_C$, $\mathscr{H}_D$ and $\mathscr{H}_E$ and the constructs of Definition 1 without the choice operator satisfy $\mathscr{H}_C$.*

Proof. Nested recursions are treated by assuming that the free variables are constants satisfying $\mathscr{H}_C$, $\mathscr{H}_D$ and $\mathscr{H}_E$ and showing that the characteristic functions preserve $\mathscr{H}_C$, $\mathscr{H}_D$ and $\mathscr{H}_E$. The proof is by structural induction with the following cases:

* constant satisfying $\mathscr{H}_C$, $\mathscr{H}_D$ and $\mathscr{H}_E$: trivial.
* skip: $\mathscr{H}_C(\mathbb{1})$ follows by Lemma 12.1 since $\mathbb{I}$ is a $\preccurlyeq$-continuous mapping and $\mathbb{I}\preccurlyeq = \mathbb{1}$.

* assignment: $(\vec{x}{\leftarrow}\vec{e}) = (\vec{x}'{=}\vec{e})\mathbb{1}$ by Lemma 5.2, and $\mathscr{H}_C((\vec{x}'{=}\vec{e})\mathbb{1})$ by Lemma 12.1 since $\vec{x}'{=}\vec{e}$ is the $\preccurlyeq$-continuous function $\vec{e}$.

* variable declaration: Let S be a directed set and $\vec{x}'_{I\cup K}$ such that $(\vec{x}_I, \vec{x}'_{I\cup K}) \in \exists\vec{x}_K : \mathbb{1}$ for each $\vec{x}_I \in S$, hence $\vec{x}_I \preccurlyeq \vec{x}'_I$. Then $\sup S \preccurlyeq \vec{x}'_I$, whence $(\sup S, \vec{x}'_{I\cup K}) \in \exists\vec{x}_K : \mathbb{1}$.

* variable undeclaration: ***end*** $\vec{x}_K = (\exists\vec{x}'_K : \mathbb{I})\mathbb{1}$ by Lemma 5.4, and the mapping $\exists\vec{x}'_K : \mathbb{I}$ is $\preccurlyeq$-continuous since suprema are taken pointwise, whence $\mathscr{H}_C((\exists\vec{x}'_K : \mathbb{I})\mathbb{1})$ by Lemma 12.1.

* sequential composition: To show $\mathscr{H}_C(PQ)$, observe that $\mathscr{H}_C(P)$ and $\mathscr{H}_C(Q)$ by the induction hypothesis, $\mathscr{H}_D(P)$ by Theorem 9, and $\mathscr{H}_E(P)$, $\mathscr{H}_L(Q)$ and $\mathscr{H}_L(PQ)$ by Theorem 6.3. By Lemma 12.3 it thus suffices to show $(\bigcap S)PQ = \bigcap_{R\in S} RPQ$ for every co-directed set S of relations satisfying $\mathscr{H}_D$ and $\mathscr{H}_R$. Let such an S be given, then $(\bigcap S)P = \bigcap_{R\in S} RP = \bigcap T$ for $T =_{\mathrm{def}} \{RP \mid R \in S\}$ by Lemma 12.3 using $\mathscr{H}_C(P)$ and $\mathscr{H}_L(P)$. The set T is co-directed since S is, and its elements satisfy $\mathscr{H}_R(RP)$ by Theorem 6.3 and $\mathscr{H}_D(RP)$ by the case sequential composition in Theorem 9. Therefore $(\bigcap S)PQ = (\bigcap T)Q = \bigcap_{R\in S} RPQ$ again by Lemma 12.3 using $\mathscr{H}_C(Q)$ and $\mathscr{H}_L(Q)$.

* (arbitrary) conjunction: To show $\mathscr{H}_C(\bigcap S)$ for a set S of relations satisfying $\mathscr{H}_C$, let T be directed and $\vec{x}'$ such that $(\vec{x}, \vec{x}') \in \bigcap S$ for each $\vec{x} \in T$, hence $(\vec{x}, \vec{x}') \in P$ for each $P \in S$ and $\vec{x} \in T$. By $\mathscr{H}_C(P)$ we obtain $(\sup T, \vec{x}') \in P$ for each $P \in S$, thus $(\sup T, \vec{x}') \in \bigcap S$.

* conditional: Assuming $\mathscr{H}_C(P)$ and $\mathscr{H}_C(Q)$ by the induction hypothesis, $\mathscr{H}_C(P \blacktriangleleft b \blacktriangleright Q)$ follows by Lemma 12.5 and the cases conjunction and assignment above, if we can show $\mathscr{H}_C(b{=}c)$ for each $c \in \mathit{Bool}$. To this end, let S be directed and $\vec{x}'$ such that $(\vec{x}, \vec{x}') \in (b{=}c)$ for each $\vec{x} \in S$, hence $b(\vec{x}) = c$. By $\preccurlyeq$-continuity of b we have $b(\sup S) = \sup_{\vec{x}\in S} b(\vec{x}) = \sup_{\vec{x}\in S} c = c$, thus $(\sup S, \vec{x}') \in (b{=}c)$.

* parallel composition: by Lemma 5.5 and the cases conjunction, sequential composition and variable (un)declaration above.

* greatest fixpoint: To show $\mathscr{H}_C(\nu f)$, apply Corollary 42 in Appendix B to the set S of relations satisfying $\mathscr{H}_C$, $\mathscr{H}_D$ and $\mathscr{H}_E$. This set is closed under infima of chains as shown in the case conjunction above and in Theorems 9 and 6.3. Moreover, S is closed under f by the induction hypothesis and Theorems 9 and 6.3. Finally, f is isotone by Theorem 6.1. $\square$

The main result of this section shows continuity for our programs, allowing us to compute fixpoints by Kleene's theorem.

Theorem 14. *Functions composed of constants satisfying $\mathscr{H}_C$, $\mathscr{H}_D$ and $\mathscr{H}_E$ and the constructs of Definition 1 without the choice operator are co-continuous, that is, they distribute over infima of co-directed sets of relations satisfying $\mathscr{H}_C$, $\mathscr{H}_D$ and $\mathscr{H}_E$.*

PROOF. By Theorems 13, 9 and 6.3, we can assume that the variables introduced by the ν operator range over relations satisfying $\mathscr{H}_C$, $\mathscr{H}_D$ and $\mathscr{H}_E$. These variables are free in the subterms of the ν operator, whence we show that every function composed of the allowed constructs and free variables is co-continuous in each of its free variables. The proof is by structural induction with the following cases:

* free variable: the identity function is co-continuous.

* (arbitrary) constant, including skip, assignment, variable (un)declaration: trivial.

* sequential composition: Let X be a free variable of $P \,;\, Q = (P \,;\, Q)(X) = P(X) \,;\, Q(X)$, and S a co-directed set of relations satisfying $\mathscr{H}_C$, $\mathscr{H}_D$ and $\mathscr{H}_E$. Define $P(S) =_{\mathrm{def}} \{P(A) \mid A \in S\}$ and similarly $Q(S)$ and $(P \,;\, Q)(S)$. By the induction hypothesis, P and Q are co-continuous in X, hence it remains to show the third step of

$$(P \,;\, Q)(\textstyle\bigcap S) = P(\bigcap S) \,;\, Q(\bigcap S) = (\bigcap P(S)) \,;\, (\bigcap Q(S)) = \bigcap_{C\in S} P(C) \,;\, Q(C) = \bigcap (P \,;\, Q)(S) \,.$$

The sets $P(S)$ and $Q(S)$ are co-directed by Theorem 6.1. Moreover $\mathscr{H}_C(\bigcap Q(S))$ and $\mathscr{H}_L(\bigcap Q(S))$ by Theorems 13 and 6.3, and $\mathscr{H}_D(P(A))$, $\mathscr{H}_R(P(A))$ and $\mathscr{H}_L(Q(A))$ for each $A \in S$ by Theorems 9 and 6.3. Thus by Lemmas 12.3 and 12.4

$$(\bigcap P(S)) \mathbin{;} (\bigcap Q(S)) = \bigcap_{A \in S} P(A) \mathbin{;} \bigcap Q(S) = \bigcap_{A \in S} \bigcap_{B \in S} P(A) \mathbin{;} Q(B) = \bigcap_{C \in S} P(C) \mathbin{;} Q(C) .$$

For the last step observe that $\bigcap_{C \in S} P(C) \mathbin{;} Q(C) \subseteq P(D) \mathbin{;} Q(D) \subseteq P(A) \mathbin{;} Q(B)$ using any lower bound $D \in S$ of A and B.

* (arbitrary) conjunction: Let X be a free variable of $\bigcap S = (\bigcap S)(X) = \bigcap_{P \in S} P(X)$, and T a co-directed set of relations satisfying $\mathscr{H}_C$, $\mathscr{H}_D$ and $\mathscr{H}_E$. By the induction hypothesis,

$$(\bigcap S)(\bigcap T) = \bigcap_{P \in S} P(\bigcap T) = \bigcap_{P \in S} \bigcap_{A \in T} P(A) = \bigcap_{A \in T} \bigcap_{P \in S} P(A) = \bigcap_{A \in T} (\bigcap S)(A) .$$

* conditional: The claim follows by the induction hypothesis and the cases conjunction and constant above, if we can show that also $\cup$ preserves co-continuity. To this end, let X be a free variable of $P \cup Q = (P \cup Q)(X) = P(X) \cup Q(X)$, and S a co-directed set of relations satisfying $\mathscr{H}_C$, $\mathscr{H}_D$ and $\mathscr{H}_E$. Then

$$\begin{aligned}(P \cup Q)(\bigcap S) &= P(\bigcap S) \cup Q(\bigcap S) = (\bigcap_{A \in S} P(A)) \cup (\bigcap_{B \in S} Q(B)) = \bigcap_{A \in S} \bigcap_{B \in S} P(A) \cup Q(B) \\ &= \bigcap_{C \in S} P(C) \cup Q(C) = \bigcap_{C \in S} (P \cup Q)(C) .\end{aligned}$$

* parallel composition: by Lemma 5.5 and the cases conjunction, sequential composition and constant above.

* greatest fixpoint: Let Y be a free variable of $\nu f = (\nu f)(Y) = \nu X.f(X,Y)$, and S a co-directed set of relations satisfying $\mathscr{H}_C$, $\mathscr{H}_D$ and $\mathscr{H}_E$. For $A \in S$ define $g_A(X) =_{\mathrm{def}} f(X,A)$ and $h(X) =_{\mathrm{def}} f(X, \bigcap S)$. Then g_A and h are co-continuous by the induction hypothesis since $\bigcap S$ satisfies $\mathscr{H}_C$, $\mathscr{H}_D$ and $\mathscr{H}_E$ by Theorems 13, 9 and 6.3. Therefore, if we can show $h^n(\mathbb{T}) = \bigcap_{A \in S} g_A^n(\mathbb{T})$, the claim follows by using Kleene's theorem twice in

$$\begin{aligned}(\nu f)(\bigcap S) &= \nu X.f(X, \bigcap S) = \nu h = \bigcap_{n \in \mathbb{N}} h^n(\mathbb{T}) = \bigcap_{n \in \mathbb{N}} \bigcap_{A \in S} g_A^n(\mathbb{T}) \\ &= \bigcap_{A \in S} \bigcap_{n \in \mathbb{N}} g_A^n(\mathbb{T}) = \bigcap_{A \in S} \nu g_A = \bigcap_{A \in S} \nu X.f(X,A) = \bigcap_{A \in S} (\nu f)(A) .\end{aligned}$$

We prove $h^n(\mathbb{T}) = \bigcap_{A \in S} g_A^n(\mathbb{T})$ by induction. The basis follows by $h^0(\mathbb{T}) = \mathbb{T} = \bigcap_{A \in S} \mathbb{T} = \bigcap_{A \in S} g_A^0(\mathbb{T})$, and the step by

$$\begin{aligned}h^{n+1}(\mathbb{T}) &= h(h^n(\mathbb{T})) = h(\bigcap_{A \in S} g_A^n(\mathbb{T})) = \bigcap_{A \in S} h(g_A^n(\mathbb{T})) = \bigcap_{A \in S} f(g_A^n(\mathbb{T}), \bigcap S) \\ &= \bigcap_{A \in S} \bigcap_{B \in S} f(g_A^n(\mathbb{T}), B) = \bigcap_{C \in S} f(g_C^n(\mathbb{T}), C) = \bigcap_{C \in S} g_C(g_C^n(\mathbb{T})) \\ &= \bigcap_{C \in S} g_C^{n+1}(\mathbb{T}) ,\end{aligned}$$

since f is isotone by Theorem 6.1, hence the set $\{g_A^n(\mathbb{T}) \mid A \in S\}$ is co-directed, its elements satisfy $\mathscr{H}_C$, $\mathscr{H}_D$ and $\mathscr{H}_E$ since $\mathbb{T}$ satisfies and g_A preserves these properties by Theorems 13, 9 and 6.3, and h and f are co-continuous. □

We thus obtain a theory of non-strict computations over infinite data structures by restricting ourselves to deterministic programs. Future work shall investigate whether another trade-off is possible to reconcile non-determinism and infinite data structures. Theorems 6, 10 and 14 are the main results to ensure that the application of our theory is meaningful. These theorems also apply to all programming constructs we introduce in the remainder of this paper, since they are composed of the basic constructs of Definition 1 without the choice operator.

3.5. Application

Having completed the foundations, let us see the theory at work. To this end, we recall the construction of the infinite list of natural numbers $[0..] = 0 : 1 : 2 : 3 : \ldots$ from [19]. We assume that the type of lists of integers has been defined as $\mathit{IntList} = \mathit{Nil} + (\mathit{Int} : \mathit{IntList})$ with non-strict constructors $:$ and Nil. Such types are further discussed in Section 6.

Example 15. Our program to compute $[0..]$ should have two variables xs and c to hold the result and to count, respectively. The solution is to increment the value of c before the recursive call and to construct the sequence afterwards. The value of c is saved across the recursive call in the local variable t by the alphabet extension:

$$P = f(P) =_{\mathrm{def}} \boldsymbol{let}\ t{\leftarrow}c\ \boldsymbol{in}\ c{\leftarrow}c{+}1 \ ;\ P^{+t} \ ;\ xs{\leftarrow}t{:}xs\ .$$

This recursion is used as a part of the program *from2* in Section 1. We confirm that it computes the infinite list $[c..] = c : c{+}1 : c{+}2 : c{+}3 : \ldots$ by calculating the greatest fixpoint of f. Using Theorem 14 we obtain $\nu f = \bigcap_{n\in\mathbb{N}} f^n(\top)$ where

$$\begin{aligned}
f^0(\top) &= \top \\
f^1(\top) &= \boldsymbol{let}\ t{\leftarrow}c\ \boldsymbol{in}\ c{\leftarrow}c{+}1 \ ;\ \top^{+t} \ ;\ xs{\leftarrow}t{:}xs \\
&= \boldsymbol{let}\ t{\leftarrow}c\ \boldsymbol{in}\ c{\leftarrow}c{+}1 \ ;\ (\top_{xs,c}\|_t \mathbb{1}) \ ;\ xs{\leftarrow}t{:}xs \\
&= \boldsymbol{let}\ t{\leftarrow}c\ \boldsymbol{in}\ c{\leftarrow}c{+}1 \ ;\ (xs, c, t{\leftarrow}\infty, \infty, t) \ ;\ xs{\leftarrow}t{:}xs \\
&= \boldsymbol{let}\ t{\leftarrow}c\ \boldsymbol{in}\ c{\leftarrow}\infty \ ;\ xs{\leftarrow}t{:}\infty \\
&= xs, c{\leftarrow}c{:}\infty, \infty \\
f^2(\top) &= \boldsymbol{let}\ t{\leftarrow}c\ \boldsymbol{in}\ c{\leftarrow}c{+}1 \ ;\ (xs, c{\leftarrow}c{:}\infty, \infty)^{+t} \ ;\ xs{\leftarrow}t{:}xs \\
&= \boldsymbol{let}\ t{\leftarrow}c\ \boldsymbol{in}\, (xs, c, t{\leftarrow}c{+}1{:}\infty, \infty, t) \ ;\ xs{\leftarrow}t{:}xs \\
&= xs, c{\leftarrow}c{:}c{+}1{:}\infty, \infty \\
f^3(\top) &= xs, c{\leftarrow}c{:}c{+}1{:}c{+}2{:}\infty, \infty
\end{aligned}$$

Identities described in Example 2 are applied to calculate $f^1(\top)$. We thus obtain for the n-th approximation of the fixpoint $f^n(\top) = (xs, c{\leftarrow}c : c{+}1 : c{+}2 : \ldots : c{+}n{-}1 : \infty, \infty)$ and therefore $c{\leftarrow}0 \ ;\ \nu f = (xs, c{\leftarrow}[0..], \infty)$.

Example 16. Consider the infinite list $[c..]$ constructed in the previous example and stored in the variable xs. We now show how to remove all even numbers from it. The solution is to construct a new list, again saving the value of each element across the recursive call:

$$Q = g(Q) =_{\mathrm{def}} \boldsymbol{let}\ h{\leftarrow}\mathit{head}(xs)\ \boldsymbol{in}\ xs{\leftarrow}\mathit{tail}(xs) \ ;\ Q^{+h} \ ;\ (\mathbb{1} \blacktriangleleft 2|h \blacktriangleright xs{\leftarrow}h{:}xs)\ ,$$

similarly to the program *remove* in Section 1. Using Theorem 14 and Lemma 12.4 we obtain $xs{\leftarrow}[c..] \ ;\ \nu g = xs{\leftarrow}[c..] \ ;\ \bigcap_{n\in\mathbb{N}} g^n(\top) = \bigcap_{n\in\mathbb{N}} xs{\leftarrow}[c..] \ ;\ g^n(\top)$. Observe that

$$\begin{aligned}
xs{\leftarrow}[c..] \ ;\ g^{n+1}(\top) &= xs{\leftarrow}[c..] \ ;\ \boldsymbol{let}\ h{\leftarrow}\mathit{head}(xs)\ \boldsymbol{in}\ xs{\leftarrow}\mathit{tail}(xs) \ ;\ (g^n(\top))^{+h} \ ;\ (\mathbb{1} \blacktriangleleft 2|h \blacktriangleright xs{\leftarrow}h{:}xs) \\
&= xs{\leftarrow}[c..] \ ;\ \boldsymbol{let}\ h{\leftarrow}c\ \boldsymbol{in}\ xs{\leftarrow}\mathit{tail}(xs) \ ;\ (g^n(\top))^{+h} \ ;\ (\mathbb{1} \blacktriangleleft 2|h \blacktriangleright xs{\leftarrow}h{:}xs) \\
&= xs{\leftarrow}[c..] \ ;\ xs{\leftarrow}\mathit{tail}(xs) \ ;\ g^n(\top) \ ;\ (\mathbb{1} \blacktriangleleft 2|c \blacktriangleright xs{\leftarrow}c{:}xs) \\
&= xs{\leftarrow}[c{+}1..] \ ;\ g^n(\top) \ ;\ (\mathbb{1} \blacktriangleleft 2|c \blacktriangleright xs{\leftarrow}c{:}xs)\ ,
\end{aligned}$$

and hence

$$\begin{aligned}
xs{\leftarrow}[c..] \ ;\ g^{n+2}(\top) &= xs{\leftarrow}[c{+}1..] \ ;\ g^{n+1}(\top) \ ;\ (\mathbb{1} \blacktriangleleft 2|c \blacktriangleright xs{\leftarrow}c{:}xs) \\
&= xs{\leftarrow}[c{+}2..] \ ;\ g^n(\top) \ ;\ (\mathbb{1} \blacktriangleleft 2|c{+}1 \blacktriangleright xs{\leftarrow}c{+}1{:}xs) \ ;\ (\mathbb{1} \blacktriangleleft 2|c \blacktriangleright xs{\leftarrow}c{:}xs) \\
&= xs{\leftarrow}[c{+}2..] \ ;\ g^n(\top) \ ;\ (xs{\leftarrow}c{+}1{:}xs \blacktriangleleft 2|c \blacktriangleright xs{\leftarrow}c{:}xs) \\
&= xs{\leftarrow}[c{+}2..] \ ;\ g^n(\top) \ ;\ xs{\leftarrow}2\lfloor \tfrac{c}{2} \rfloor{+}1{:}xs\ .
\end{aligned}$$

We thus obtain for the n-th approximation $xs{\leftarrow}[c..] \ ;\ g^n(\top) = (xs{\leftarrow}2\lfloor\frac{c}{2}\rfloor{+}1 : 2\lfloor\frac{c}{2}\rfloor{+}3 : \ldots : 2\lfloor\frac{c+n}{2}\rfloor{-}1 : \infty)$ and therefore $xs{\leftarrow}[c..] \ ;\ \nu g = xs{\leftarrow}[2\lfloor\frac{c}{2}\rfloor{+}1, 2\lfloor\frac{c}{2}\rfloor{+}3\,..]$ which retains in xs the odd numbers of $[c..]$.

Example 17. Consider the recursively specified program $R = h(R) = R \mathbin{;} (xs \leftarrow 1{:}xs \cup xs \leftarrow 2{:}xs)$. Since it uses the choice operator, we cannot apply Theorem 14 to obtain co-continuity of h. But also without Kleene's theorem we can see that R assigns to xs any of the infinite lists containing only the elements 1 and 2. There are $2^{|\mathbb{N}|}$ such lists, which shows that even finite choice can lead to unbounded non-determinism. However, the output values of xs are a finitely generable set [32].

4. Procedures and Parameters

Most imperative programming languages support the abstraction of statements into procedures. They usually carry parameters to clarify the interface between caller and callee. Any non-local variables must be accessed via the parameters. On the other hand, the caller cannot access local variables of the called procedure. Two prominent parameter passing mechanisms are *by value* and *by reference*. We implement both, but make two restrictions on the latter: references are to variables of the state only, and aliasing is not allowed.

Definition 18. The declaration $P(\textbf{\textit{val}}\, \vec{x}_I{:}D_I, \textbf{\textit{ref}}\, \vec{x}_J{:}D_J) = R$ abbreviates the equation

$$P = \textbf{\textit{var}}\ \vec{x}_{I\cup J} \leftarrow \overrightarrow{in}_{I\cup J} \mathbin{;} \textbf{\textit{end}}\ \overrightarrow{in}_{I\cup J} \mathbin{;} R \mathbin{;} \textbf{\textit{var}}\ \overrightarrow{out}_J \leftarrow \vec{x}_J \mathbin{;} \textbf{\textit{end}}\ \vec{x}_{I\cup J}\ .$$

It introduces the procedure P with value parameters $\vec{x}_I$ of type D_I, reference parameters $\vec{x}_J$ of type D_J, and body R. Recursive calls of P are permitted in the body R and resolved as the greatest fixpoint. The special variables $\overrightarrow{in}_{I\cup J}$ and $\overrightarrow{out}_J$ are used for parameter passing. The types of P and R are $P[\overrightarrow{in}_{I\cup J}, \overrightarrow{out}'_J] : D_{I\cup J} \leftrightarrow D_J$ and $R[\vec{x}_{I\cup J}, \vec{x}'_{I\cup J}] : D_{I\cup J} \leftrightarrow D_{I\cup J}$, respectively.

If they are clear from the context, we omit types and write $P(\textbf{\textit{val}}\, \vec{x}_I, \textbf{\textit{ref}}\, \vec{x}_J) = R$. For convenience, we allow that $I \cap J \neq \emptyset$. There are thus three kinds of parameters:

* x_i where $i \in I$ and $i \notin J$ is passed by value. It corresponds to a local variable whose initial value is determined by the caller and whose final value is discarded.
* x_i where $i \notin I$ and $i \in J$ is passed by value and result [15]. It corresponds to a local variable initialised with the value of a variable of the caller, which in turn takes the final value of x_i. By this mechanism, we can pass back results to the caller, but the called procedure works on a local copy. Since aliasing is not allowed, this amounts to passing by reference.
* x_i where $i \in I$ and $i \in J$ is similar, except that a separate value is determined by the caller to initialise x_i. In this case, we call the mechanism passing *by value and reference.* This is just to enable a more convenient notation, see the procedure *slr* in Section 9.

The caller must supply values for $\vec{x}_I$ and distinct (names of) variables for $\vec{x}_J$.

Definition 19. The procedure call $P(\vec{e}_I, \vec{x}_J)$ corresponding to the declaration of Definition 18 abbreviates the relation

$$\textbf{\textit{var}}\ \overrightarrow{in}_{I\cup J} \leftarrow \vec{e}_I \vec{x}_{J\setminus I} \mathbin{;} P^{+\vec{x}_K} \mathbin{;} \vec{x}_J \leftarrow \overrightarrow{out}_J \mathbin{;} \textbf{\textit{end}}\ \overrightarrow{out}_J\ .$$

It passes the values $\vec{e}_I$ to initialise the local variables $\vec{x}_I$ of the called procedure and the variables $\vec{x}_J$ as references. The expressions $\vec{e}_I$ may depend on the state $\vec{x}_K$ of the caller, and $J \subseteq K$. The alphabet extension of P by $\vec{x}_K$ saves the caller's variables' values across the call. The type of the call is $P(\vec{e}_I, \vec{x}_J)[\vec{x}_K, \vec{x}'_K] : D_K \leftrightarrow D_K$.

For an index $i \in I \cap J$, the caller supplies both a value e_i and a variable x_i. Instead of the value of x_i, the value e_i is used to initialise the corresponding formal parameter, and the result is stored in x_i after the call.

As an example, let us reconsider the generation of the natural numbers.

Example 20. Let $\mathit{enumFrom}(\textbf{\textit{val}}\ c, \textbf{\textit{ref}}\ \mathit{xs}) = \mathit{enumFrom}(c{+}1, \mathit{xs})\ ;\ \mathit{xs}{\leftarrow}c{:}\mathit{xs}$. Using this declaration, we show that $\mathit{enumFrom}(2, \mathit{xs}) = \mathit{xs}{\leftarrow}[2..]$ where the infinite list $2 : 3 : 4 : \ldots$ is denoted by $[2..]$. First,

$$\begin{aligned}
&\mathit{enumFrom}\\
= {}& \textbf{\textit{var}}\ c, \mathit{xs}{\leftarrow}\mathit{in}_c, \mathit{in}_{\mathit{xs}}\ ;\ \textbf{\textit{end}}\ \mathit{in}_c, \mathit{in}_{\mathit{xs}}\ ;\ \mathit{enumFrom}(c{+}1, \mathit{xs})\ ;\ \mathit{xs}{\leftarrow}c{:}\mathit{xs}\ ;\ \textbf{\textit{var}}\ \mathit{out}_{\mathit{xs}}{\leftarrow}\mathit{xs}\ ;\ \textbf{\textit{end}}\ c, \mathit{xs}\\
= {}& \textbf{\textit{var}}\ c, \mathit{xs}{\leftarrow}\mathit{in}_c, \mathit{in}_{\mathit{xs}}\ ;\ \textbf{\textit{end}}\ \mathit{in}_c, \mathit{in}_{\mathit{xs}}\ ;\ \textbf{\textit{var}}\ \mathit{in}_c, \mathit{in}_{\mathit{xs}}{\leftarrow}c{+}1, \mathit{xs}\ ;\ \mathit{enumFrom}^{+c,\mathit{xs}}\ ;\ \mathit{xs}{\leftarrow}\mathit{out}_{\mathit{xs}}\ ;\\
& \textbf{\textit{end}}\ \mathit{out}_{\mathit{xs}}\ ;\ \mathit{xs}{\leftarrow}c{:}\mathit{xs}\ ;\ \textbf{\textit{var}}\ \mathit{out}_{\mathit{xs}}{\leftarrow}\mathit{xs}\ ;\ \textbf{\textit{end}}\ c, \mathit{xs}\\
= {}& \textbf{\textit{var}}\ c, \mathit{xs}{\leftarrow}\mathit{in}_c, \mathit{in}_{\mathit{xs}}\ ;\ \mathit{in}_c{\leftarrow}c{+}1\ ;\ \mathit{enumFrom}^{+c,\mathit{xs}}\ ;\ \mathit{xs}{\leftarrow}\mathit{out}_{\mathit{xs}}\ ;\ \mathit{xs}{\leftarrow}c{:}\mathit{xs}\ ;\ \mathit{out}_{\mathit{xs}}{\leftarrow}\mathit{xs}\ ;\ \textbf{\textit{end}}\ c, \mathit{xs}\\
= {}& \textbf{\textit{var}}\ c{\leftarrow}\mathit{in}_c\ ;\ \mathit{in}_c{\leftarrow}\mathit{in}_c{+}1\ ;\ \mathit{enumFrom}^{+c}\ ;\ \mathit{out}_{\mathit{xs}}{\leftarrow}c{:}\mathit{out}_{\mathit{xs}}\ ;\ \textbf{\textit{end}}\ c\ .
\end{aligned}$$

The argument proceeds analogously to Example 15, except that we have heterogeneous relations now, starting with $\mathbb{T} = \textbf{\textit{end}}\ \mathit{in}_c, \mathit{in}_{\mathit{xs}}\ ;\ \textbf{\textit{var}}\ \mathit{out}_{\mathit{xs}}$. We obtain that $\mathit{enumFrom} = \textbf{\textit{var}}\ \mathit{out}_{\mathit{xs}}{\leftarrow}[\mathit{in}_c..]\ ;\ \textbf{\textit{end}}\ \mathit{in}_c, \mathit{in}_{\mathit{xs}}$. Assuming the state of the caller has variables $\vec{x}_K$ in addition to xs, this implies

$$\begin{aligned}
&\mathit{enumFrom}(2, \mathit{xs})\\
= {}& \textbf{\textit{var}}\ \mathit{in}_c, \mathit{in}_{\mathit{xs}}{\leftarrow}2, \mathit{xs}\ ;\ \mathit{enumFrom}^{+\mathit{xs},\vec{x}_K}\ ;\ \mathit{xs}{\leftarrow}\mathit{out}_{\mathit{xs}}\ ;\ \textbf{\textit{end}}\ \mathit{out}_{\mathit{xs}}\\
= {}& \textbf{\textit{var}}\ \mathit{in}_c, \mathit{in}_{\mathit{xs}}{\leftarrow}2, \mathit{xs}\ ;\ (\textbf{\textit{var}}\ \mathit{out}_{\mathit{xs}}{\leftarrow}[\mathit{in}_c..]\ ;\ \textbf{\textit{end}}\ \mathit{in}_c, \mathit{in}_{\mathit{xs}})^{+\mathit{xs},\vec{x}_K}\ ;\ \mathit{xs}{\leftarrow}\mathit{out}_{\mathit{xs}}\ ;\ \textbf{\textit{end}}\ \mathit{out}_{\mathit{xs}}\\
= {}& \textbf{\textit{var}}\ \mathit{in}_c, \mathit{in}_{\mathit{xs}}{\leftarrow}2, \mathit{xs}\ ;\ \textbf{\textit{var}}\ \mathit{out}_{\mathit{xs}}{\leftarrow}[\mathit{in}_c..]\ ;\ \textbf{\textit{end}}\ \mathit{in}_c, \mathit{in}_{\mathit{xs}}\ ;\ \mathit{xs}{\leftarrow}\mathit{out}_{\mathit{xs}}\ ;\ \textbf{\textit{end}}\ \mathit{out}_{\mathit{xs}}\\
= {}& \textbf{\textit{var}}\ \mathit{out}_{\mathit{xs}}{\leftarrow}[2..]\ ;\ \mathit{xs}{\leftarrow}\mathit{out}_{\mathit{xs}}\ ;\ \textbf{\textit{end}}\ \mathit{out}_{\mathit{xs}}\\
= {}& \mathit{xs}{\leftarrow}[2..]\ .
\end{aligned}$$

To elaborate the interaction of procedure declaration and call, let us introduce a convenient abstraction to express that the values of certain variables do not change.

Definition 21. Let I, J and K be index sets such that $I \cap J = I \cap K = \emptyset$ and let $P : \vec{x}_{I\cup J} \leftrightarrow \vec{x}_{I\cup K}$. Then

$$(\textbf{\textit{const}}\ \vec{x}_I : P) =_{\mathrm{def}} \textbf{\textit{var}}\ \vec{t}_I{\leftarrow}\vec{x}_I\ ;\ P^{+\vec{t}_I}\ ;\ \vec{x}_I{\leftarrow}\vec{t}_I\ ;\ \textbf{\textit{end}}\ \vec{t}_I\ .$$

The values of $\vec{x}_I$ are stored in the temporary variables $\vec{t}_I$ and restored after P.

The scope of ***const*** shall extend as far to the right as possible. The following lemma describes how ***const*** commutes with several programming constructs.

Lemma 22. *Let I and J be index sets such that $I \cap J = \emptyset$.*

1. $\textbf{\textit{const}}\ \vec{x}_I : \vec{x}_I{\leftarrow}\vec{e}_I\ ;\ P = (\textbf{\textit{var}}\ \vec{x}_I{\leftarrow}\vec{e}_I\ ;\ P\ ;\ \textbf{\textit{end}}\ \vec{x}_I)^{+\vec{x}_I}$*, provided $\vec{e}_I$ does not use the variables $\vec{x}_I$.*
2. $\textbf{\textit{const}}\ \vec{x}_I : \textbf{\textit{var}}\ \vec{x}_J{\leftarrow}\vec{e}_J\ ;\ P = \textbf{\textit{var}}\ \vec{x}_J{\leftarrow}\vec{e}_J\ ;\ \textbf{\textit{const}}\ \vec{x}_I : P$*.*
3. $\textbf{\textit{const}}\ \vec{x}_I : P\ ;\ \vec{x}_J{\leftarrow}\vec{e}_J = (\textbf{\textit{const}}\ \vec{x}_{I\cup J} : P)\ ;\ \vec{x}_J{\leftarrow}\vec{e}_J$*, provided $\vec{e}_J$ does not use the variables $\vec{x}_{I\cup J}$.*
4. $\textbf{\textit{const}}\ \vec{x}_I : P\ ;\ \textbf{\textit{end}}\ \vec{x}_J = (\textbf{\textit{const}}\ \vec{x}_I : P)\ ;\ \textbf{\textit{end}}\ \vec{x}_J$*.*
5. $\textbf{\textit{const}}\ \vec{x}_I : P^{+\vec{x}_J} = (\textbf{\textit{const}}\ \vec{x}_I : P)^{+\vec{x}_J}$*.*

Proof.

1. $$\begin{aligned}
&\textbf{\textit{const}}\ \vec{x}_I : \vec{x}_I{\leftarrow}\vec{e}_I\ ;\ P\\
= {}& \textbf{\textit{var}}\ \vec{t}_I{\leftarrow}\vec{x}_I\ ;\ (\vec{x}_I{\leftarrow}\vec{e}_I\ ;\ P)^{+\vec{t}_I}\ ;\ \vec{x}_I{\leftarrow}\vec{t}_I\ ;\ \textbf{\textit{end}}\ \vec{t}_I\\
= {}& \textbf{\textit{var}}\ \vec{t}_I{\leftarrow}\vec{x}_I\ ;\ (\textbf{\textit{end}}\ \vec{x}_I\ ;\ \textbf{\textit{var}}\ \vec{x}_I{\leftarrow}\vec{e}_I\ ;\ P)^{+\vec{t}_I}\ ;\ \textbf{\textit{end}}\ \vec{x}_I\ ;\ \textbf{\textit{var}}\ \vec{x}_I{\leftarrow}\vec{t}_I\ ;\ \textbf{\textit{end}}\ \vec{t}_I\\
= {}& \textbf{\textit{var}}\ \vec{t}_I{\leftarrow}\vec{x}_I\ ;\ \textbf{\textit{end}}\ \vec{x}_I\ ;\ (\textbf{\textit{var}}\ \vec{x}_I{\leftarrow}\vec{e}_I\ ;\ P\ ;\ \textbf{\textit{end}}\ \vec{x}_I)^{+\vec{t}_I}\ ;\ \textbf{\textit{var}}\ \vec{x}_I{\leftarrow}\vec{t}_I\ ;\ \textbf{\textit{end}}\ \vec{t}_I\\
= {}& \textbf{\textit{var}}\ \vec{t}_I{\leftarrow}\vec{x}_I\ ;\ (\textbf{\textit{var}}\ \vec{x}_I{\leftarrow}\vec{e}_I\ ;\ P\ ;\ \textbf{\textit{end}}\ \vec{x}_I)^{+\vec{t}_I,\vec{x}_I}\ ;\ \textbf{\textit{end}}\ \vec{x}_I\ ;\ \textbf{\textit{var}}\ \vec{x}_I{\leftarrow}\vec{t}_I\ ;\ \textbf{\textit{end}}\ \vec{t}_I\\
= {}& \textbf{\textit{var}}\ \vec{t}_I\ ;\ (\textbf{\textit{var}}\ \vec{x}_I{\leftarrow}\vec{e}_I\ ;\ P\ ;\ \textbf{\textit{end}}\ \vec{x}_I)^{+\vec{t}_I,\vec{x}_I}\ ;\ \vec{t}_I{\leftarrow}\vec{x}_I\ ;\ \vec{x}_I{\leftarrow}\vec{t}_I\ ;\ \textbf{\textit{end}}\ \vec{t}_I\\
= {}& (\textbf{\textit{var}}\ \vec{x}_I{\leftarrow}\vec{e}_I\ ;\ P\ ;\ \textbf{\textit{end}}\ \vec{x}_I)^{+\vec{x}_I}\ ;\ \textbf{\textit{var}}\ \vec{t}_I\ ;\ \vec{t}_I{\leftarrow}\vec{x}_I\ ;\ \textbf{\textit{end}}\ \vec{t}_I\\
= {}& (\textbf{\textit{var}}\ \vec{x}_I{\leftarrow}\vec{e}_I\ ;\ P\ ;\ \textbf{\textit{end}}\ \vec{x}_I)^{+\vec{x}_I}\ .
\end{aligned}$$
2. $$\begin{aligned}
&\textbf{\textit{const}}\ \vec{x}_I : \textbf{\textit{var}}\ \vec{x}_J{\leftarrow}\vec{e}_J\ ;\ P\\
= {}& \textbf{\textit{var}}\ \vec{t}_I{\leftarrow}\vec{x}_I\ ;\ (\textbf{\textit{var}}\ \vec{x}_J{\leftarrow}\vec{e}_J\ ;\ P)^{+\vec{t}_I}\ ;\ \vec{x}_I{\leftarrow}\vec{t}_I\ ;\ \textbf{\textit{end}}\ \vec{t}_I\\
= {}& \textbf{\textit{var}}\ \vec{t}_I{\leftarrow}\vec{x}_I\ ;\ \textbf{\textit{var}}\ \vec{x}_J{\leftarrow}\vec{e}_J\ ;\ P^{+\vec{t}_I}\ ;\ \vec{x}_I{\leftarrow}\vec{t}_I\ ;\ \textbf{\textit{end}}\ \vec{t}_I\\
= {}& \textbf{\textit{var}}\ \vec{x}_J{\leftarrow}\vec{e}_J\ ;\ \textbf{\textit{var}}\ \vec{t}_I{\leftarrow}\vec{x}_I\ ;\ P^{+\vec{t}_I}\ ;\ \vec{x}_I{\leftarrow}\vec{t}_I\ ;\ \textbf{\textit{end}}\ \vec{t}_I\\
= {}& \textbf{\textit{var}}\ \vec{x}_J{\leftarrow}\vec{e}_J\ ;\ \textbf{\textit{const}}\ \vec{x}_I : P\ .
\end{aligned}$$

3. $\quad (\textbf{\textit{const}}\ \vec{x}_{I\cup J} : P)\ ;\ \vec{x}_J \leftarrow \vec{e}_J$
$= \textbf{\textit{var}}\ \vec{t}_{I\cup J} \leftarrow \vec{x}_{I\cup J}\ ;\ P^{+\vec{t}_{I\cup J}}\ ;\ \vec{x}_{I\cup J} \leftarrow \vec{t}_{I\cup J}\ ;\ \textbf{\textit{end}}\ \vec{t}_{I\cup J}\ ;\ \vec{x}_J \leftarrow \vec{e}_J$
$= \textbf{\textit{var}}\ \vec{t}_{I\cup J} \leftarrow \vec{x}_{I\cup J}\ ;\ P^{+\vec{t}_{I\cup J}}\ ;\ \vec{x}_{I\cup J} \leftarrow \vec{t}_{I\cup J}\ ;\ \vec{x}_J \leftarrow \vec{e}_J\ ;\ \textbf{\textit{end}}\ \vec{t}_{I\cup J}$
$= \textbf{\textit{var}}\ \vec{t}_{I\cup J} \leftarrow \vec{x}_{I\cup J}\ ;\ P^{+\vec{t}_{I\cup J}}\ ;\ \vec{x}_I \leftarrow \vec{t}_I\ ;\ \vec{x}_J \leftarrow \vec{e}_J\ ;\ \textbf{\textit{end}}\ \vec{t}_{I\cup J}$
$= \textbf{\textit{var}}\ \vec{t}_I \leftarrow \vec{x}_I\ ;\ P^{+\vec{t}_I}\ ;\ \vec{x}_J \leftarrow \vec{e}_J\ ;\ \vec{x}_I \leftarrow \vec{t}_I\ ;\ \textbf{\textit{end}}\ \vec{t}_I$
$= \textbf{\textit{var}}\ \vec{t}_I \leftarrow \vec{x}_I\ ;\ (P\ ;\ \vec{x}_J \leftarrow \vec{e}_J)^{+\vec{t}_I}\ ;\ \vec{x}_I \leftarrow \vec{t}_I\ ;\ \textbf{\textit{end}}\ \vec{t}_I$
$= \textbf{\textit{const}}\ \vec{x}_I : P\ ;\ \vec{x}_J \leftarrow \vec{e}_J\ .$

4. $\quad \textbf{\textit{const}}\ \vec{x}_I : P\ ;\ \textbf{\textit{end}}\ \vec{x}_J$
$= \textbf{\textit{var}}\ \vec{t}_I \leftarrow \vec{x}_I\ ;\ (P\ ;\ \textbf{\textit{end}}\ \vec{x}_J)^{+\vec{t}_I}\ ;\ \vec{x}_I \leftarrow \vec{t}_I\ ;\ \textbf{\textit{end}}\ \vec{t}_I$
$= \textbf{\textit{var}}\ \vec{t}_I \leftarrow \vec{x}_I\ ;\ P^{+\vec{t}_I}\ ;\ \vec{x}_I \leftarrow \vec{t}_I\ ;\ \textbf{\textit{end}}\ \vec{t}_I\ ;\ \textbf{\textit{end}}\ \vec{x}_J$
$= (\textbf{\textit{const}}\ \vec{x}_I : P)\ ;\ \textbf{\textit{end}}\ \vec{x}_J\ .$

5. $\quad (\textbf{\textit{const}}\ \vec{x}_I : P)^{+\vec{x}_J}$
$= (\textbf{\textit{var}}\ \vec{t}_I \leftarrow \vec{x}_I\ ;\ P^{+\vec{t}_I}\ ;\ \vec{x}_I \leftarrow \vec{t}_I\ ;\ \textbf{\textit{end}}\ \vec{t}_I)^{+\vec{x}_J}$
$= \textbf{\textit{var}}\ \vec{t}_I \leftarrow \vec{x}_I\ ;\ P^{+\vec{t}_I,\vec{x}_J}\ ;\ \vec{x}_I \leftarrow \vec{t}_I\ ;\ \textbf{\textit{end}}\ \vec{t}_I$
$= \textbf{\textit{const}}\ \vec{x}_I : P^{+\vec{x}_J}\ .$ □

Theorem 23. *Consider the declaration* $P(\textbf{\textit{val}}\ \vec{x}_I, \textbf{\textit{ref}}\ \vec{x}_J) = R$ *together with the call* $P(\vec{x}_I, \vec{x}_J) : \vec{x}_K \leftrightarrow \vec{x}_K$*. Then*

$$P(\vec{x}_I, \vec{x}_J) = (\textbf{\textit{const}}\ \vec{x}_{I\setminus J} : R)^{+\vec{x}_{K\setminus(I\cup J)}} = \textbf{\textit{const}}\ \vec{x}_{I\setminus J} : R^{+\vec{x}_{K\setminus(I\cup J)}}\ .$$

This shows that the call preserves the variables $\vec{x}_{I\setminus J}$ passed by value and the variables $\vec{x}_{K\setminus(I\cup J)}$ not passed at all. Only the values of the variables $\vec{x}_J$ may be modified by the call $P(\vec{x}_I, \vec{x}_J)$.

Proof. Let $Q =_{\text{def}} R\ ;\ \textbf{\textit{var}}\ \overrightarrow{out}_J \leftarrow \vec{x}_J$. Using Lemma 22,

$P(\vec{x}_I, \vec{x}_J)$
$= \textbf{\textit{var}}\ \overrightarrow{in}_{I\cup J} \leftarrow \vec{x}_I\vec{x}_{J\setminus I}\ ;\ P^{+\vec{x}_K}\ ;\ \vec{x}_J \leftarrow \overrightarrow{out}_J\ ;\ \textbf{\textit{end}}\ \overrightarrow{out}_J$
$= \textbf{\textit{var}}\ \overrightarrow{in}_{I\cup J} \leftarrow \vec{x}_{I\cup J}\ ;\ (\textbf{\textit{var}}\ \vec{x}_{I\cup J} \leftarrow \overrightarrow{in}_{I\cup J}\ ;\ \textbf{\textit{end}}\ \overrightarrow{in}_{I\cup J}\ ;\ Q\ ;\ \textbf{\textit{end}}\ \vec{x}_{I\cup J})^{+\vec{x}_K}\ ;\ \vec{x}_J \leftarrow \overrightarrow{out}_J\ ;\ \textbf{\textit{end}}\ \overrightarrow{out}_J$
$= \textbf{\textit{var}}\ \overrightarrow{in}_{I\cup J} \leftarrow \vec{x}_{I\cup J}\ ;\ (\textbf{\textit{const}}\ \vec{x}_{I\cup J} : \vec{x}_{I\cup J} \leftarrow \overrightarrow{in}_{I\cup J}\ ;\ \textbf{\textit{end}}\ \overrightarrow{in}_{I\cup J}\ ;\ Q)^{+\vec{x}_{K\setminus(I\cup J)}}\ ;\ \vec{x}_J \leftarrow \overrightarrow{out}_J\ ;\ \textbf{\textit{end}}\ \overrightarrow{out}_J$
$= (\textbf{\textit{var}}\ \overrightarrow{in}_{I\cup J} \leftarrow \vec{x}_{I\cup J}\ ;\ (\textbf{\textit{const}}\ \vec{x}_{I\cup J} : \vec{x}_{I\cup J} \leftarrow \overrightarrow{in}_{I\cup J}\ ;\ \textbf{\textit{end}}\ \overrightarrow{in}_{I\cup J}\ ;\ Q)\ ;\ \vec{x}_J \leftarrow \overrightarrow{out}_J\ ;\ \textbf{\textit{end}}\ \overrightarrow{out}_J)^{+\vec{x}_{K\setminus(I\cup J)}}$
$= ((\textbf{\textit{const}}\ \vec{x}_{I\cup J} : \textbf{\textit{var}}\ \overrightarrow{in}_{I\cup J} \leftarrow \vec{x}_{I\cup J}\ ;\ \vec{x}_{I\cup J} \leftarrow \overrightarrow{in}_{I\cup J}\ ;\ \textbf{\textit{end}}\ \overrightarrow{in}_{I\cup J}\ ;\ Q)\ ;\ \vec{x}_J \leftarrow \overrightarrow{out}_J\ ;\ \textbf{\textit{end}}\ \overrightarrow{out}_J)^{+\vec{x}_{K\setminus(I\cup J)}}$
$= ((\textbf{\textit{const}}\ \vec{x}_{I\cup J} : R\ ;\ \textbf{\textit{var}}\ \overrightarrow{out}_J \leftarrow \vec{x}_J)\ ;\ \vec{x}_J \leftarrow \overrightarrow{out}_J\ ;\ \textbf{\textit{end}}\ \overrightarrow{out}_J)^{+\vec{x}_{K\setminus(I\cup J)}}$
$= (\textbf{\textit{const}}\ \vec{x}_{I\setminus J} : R\ ;\ \textbf{\textit{var}}\ \overrightarrow{out}_J \leftarrow \vec{x}_J\ ;\ \vec{x}_J \leftarrow \overrightarrow{out}_J\ ;\ \textbf{\textit{end}}\ \overrightarrow{out}_J)^{+\vec{x}_{K\setminus(I\cup J)}}$
$= (\textbf{\textit{const}}\ \vec{x}_{I\setminus J} : R)^{+\vec{x}_{K\setminus(I\cup J)}}$
$= \textbf{\textit{const}}\ \vec{x}_{I\setminus J} : R^{+\vec{x}_{K\setminus(I\cup J)}}\ .$ □

Of particular interest is the case that R itself modifies $\vec{x}_J$ only, since then $P(\vec{x}_I, \vec{x}_J) = R^{+\vec{x}_{K\setminus(I\cup J)}}$ holds. If moreover R is composed of variable (un)declarations, assignments, sequential composition and conditionals with defined conditions only, the alphabet extension distributes and we can replace the call $P(\vec{x}_I, \vec{x}_J)$ simply by the body R. Our calculations in Sections 7 and 8 use Theorem 23 in this way.

Parameter passing in a relational context is treated in [20, Section 9.2] by λ-expressions. Our approach avoids this irregularity by using relations only. Further approaches to parameter passing use predicate transformers, see [2, 3, 10] and references therein.

5. Partial Application

Given a procedure $P(\textbf{\textit{val}}\ \vec{x}_I, \textbf{\textit{ref}}\ \vec{x}_J) = R$ and a subset $K \subseteq I$ of its value parameters, we are interested in fixing the values of $\vec{x}_K$. These values shall be determined by expressions $\vec{e}_K$ in the state where the partial application $P_{\vec{x}_K \leftarrow \vec{e}_K}$ is constructed. This is useful, for example, because the partially supplied procedure can itself be passed as a parameter to a higher-order procedure, as discussed in Section 7. Fixing parameters passed by value and reference is also supported, since it requires only a slight modification.

Definition 24. Consider the declaration $P(\textbf{\textit{val}}\ \vec{x}_I, \textbf{\textit{ref}}\ \vec{x}_J) = R$, an index set $K \subseteq I$, the variables $\vec{x}_K$ and the constants $\vec{c}_K \in D_K$. The partial application of P fixing the values of $\vec{x}_K$ to $\vec{c}_K$ is

$$P_{\vec{x}_K \leftarrow \vec{c}_K} =_{\mathrm{def}} \textbf{\textit{end}}\ \overrightarrow{in}_{K\cap J} \ ; \textbf{\textit{var}}\ \overrightarrow{in}_K \leftarrow \vec{c}_K \ ; P \ .$$

This uses the relation P introduced by the declaration according to Definition 18. The type of the partially supplied procedure is $P_{\vec{x}_K \leftarrow \vec{c}_K}[\overrightarrow{in}_{(I\setminus K)\cup J}, \overrightarrow{out}'_J] : D_{(I\setminus K)\cup J} \leftrightarrow D_J$.

Observe that the same type is obtained by a declaration with signature $P(\textbf{\textit{val}}\ \vec{x}_{I\setminus K}, \textbf{\textit{ref}}\ \vec{x}_J)$. Moreover, the construction is such that the procedure call $P_{\vec{x}_K \leftarrow \vec{c}_K}(\vec{e}_{I\setminus K}, \vec{x}_J)$ works as if such a declaration was actually available. To see this, assume that the state comprises variables $\vec{x}_L$, then

$$\begin{aligned}
& P_{\vec{x}_K \leftarrow \vec{c}_K}(\vec{e}_{I\setminus K}, \vec{x}_J) \\
= \ & \textbf{\textit{var}}\ \overrightarrow{in}_{(I\setminus K)\cup J} \leftarrow \vec{e}_{I\setminus K}\vec{x}_{J\setminus(I\setminus K)} \ ; (P_{\vec{x}_K \leftarrow \vec{c}_K})^{+\vec{x}_L} \ ; \vec{x}_J \leftarrow \overrightarrow{out}_J \ ; \textbf{\textit{end}}\ \overrightarrow{out}_J \\
= \ & \textbf{\textit{var}}\ \overrightarrow{in}_{(I\setminus K)\cup J} \leftarrow \vec{e}_{I\setminus K}\vec{x}_{J\setminus(I\setminus K)} \ ; (\textbf{\textit{end}}\ \overrightarrow{in}_{K\cap J} \ ; \textbf{\textit{var}}\ \overrightarrow{in}_K \leftarrow \vec{c}_K \ ; P)^{+\vec{x}_L} \ ; \vec{x}_J \leftarrow \overrightarrow{out}_J \ ; \textbf{\textit{end}}\ \overrightarrow{out}_J \\
= \ & \textbf{\textit{var}}\ \overrightarrow{in}_{(I\setminus K)\cup J} \leftarrow \vec{e}_{I\setminus K}\vec{x}_{J\setminus(I\setminus K)} \ ; \textbf{\textit{end}}\ \overrightarrow{in}_{K\cap J} \ ; \textbf{\textit{var}}\ \overrightarrow{in}_K \leftarrow \vec{c}_K \ ; P^{+\vec{x}_L} \ ; \vec{x}_J \leftarrow \overrightarrow{out}_J \ ; \textbf{\textit{end}}\ \overrightarrow{out}_J \\
= \ & \textbf{\textit{var}}\ \overrightarrow{in}_{(I\cup J)\setminus K} \leftarrow \vec{e}_{I\setminus K}\vec{x}_{J\setminus I} \ ; \textbf{\textit{var}}\ \overrightarrow{in}_K \leftarrow \vec{c}_K \ ; P^{+\vec{x}_L} \ ; \vec{x}_J \leftarrow \overrightarrow{out}_J \ ; \textbf{\textit{end}}\ \overrightarrow{out}_J \\
= \ & \textbf{\textit{var}}\ \overrightarrow{in}_{I\cup J} \leftarrow \vec{e}_{I\setminus K}\vec{c}_K\vec{x}_{J\setminus I} \ ; P^{+\vec{x}_L} \ ; \vec{x}_J \leftarrow \overrightarrow{out}_J \ ; \textbf{\textit{end}}\ \overrightarrow{out}_J \\
= \ & P(\vec{e}_{I\setminus K}\vec{c}_K, \vec{x}_J) \ .
\end{aligned}$$

Hence the partial application correctly supplies the values $\vec{c}_K$ for $\vec{x}_K$. This calculation also shows why the additional $\textbf{\textit{end}}\ \overrightarrow{in}_{K\cap J}$ is needed for parameters passed by value and reference.

More generally, we would like to fix the values of $\vec{x}_K$ by arbitrary expressions $\vec{e}_K$ of the current state, rather than just constants $\vec{c}_K$. However, expressions $\vec{e}_K$ referring to the state $\vec{x}_L$ cannot be passed around in a referentially transparent manner. To see this, consider replacing $\vec{c}_K$ by $\vec{e}_K$ in the third line of the above calculation: this would not even be meaningful, because $\vec{x}_L$ is hidden by the alphabet extension. A similar problem occurs in functional programming languages, where an expression may be transported to and evaluated in an environment different from that of its construction. An implementation would typically use a closure to store the necessary values. We do not formalise closures, since this is not necessary for our calculations and reasoning. Thus, we allow the construction of $P_{\vec{x}_K \leftarrow \vec{e}_K}$ with the understanding that $\vec{e}_K$ is evaluated in the state where the construction takes place.

Example 25. Consider the procedure $div(\textbf{\textit{val}}\ p{:}Int, t, \textbf{\textit{ref}}\ xs) = (\mathbb{1} \blacktriangleleft p|t \blacktriangleright xs \leftarrow t{:}xs)$. Assuming the state $\vec{x}$ contains variables $q{:}Int$, t and xs, we obtain for each $C \in Int$:

$$\begin{aligned}
& (q{=}C) \cap div_{p\leftarrow q}(t, xs) \\
= \ & (q{=}C) \cap div_{p\leftarrow C}(t, xs) \\
= \ & (q{=}C) \cap div(C, t, xs) \\
= \ & (q{=}C) \cap \textbf{\textit{var}}\ in_p, in_t, in_{xs} \leftarrow C, t, xs \ ; div^{+\vec{x}} \ ; xs \leftarrow out_{xs} \ ; \textbf{\textit{end}}\ out_{xs} \\
= \ & (q{=}C) \cap \textbf{\textit{var}}\ in_p, in_t, in_{xs} \leftarrow q, t, xs \ ; div^{+\vec{x}} \ ; xs \leftarrow out_{xs} \ ; \textbf{\textit{end}}\ out_{xs} \\
= \ & (q{=}C) \cap div(q, t, xs) \ .
\end{aligned}$$

Hence $div_{p\leftarrow q}(t, xs) = div(q, t, xs)$.

6. Algebraic Data Types and Pattern Matching

In Section 2.3 we have described how to construct sum, product, function and recursive types from elementary types. A convenient notation for sum, product and recursive types is Haskell's `data` declaration [26]:

$$\begin{aligned}
\textbf{\textit{data}}\ D = \ & C_1\ D_{1,1}\ D_{1,2}\ \dots\ D_{1,k_1} \\
| \ & C_2\ D_{2,1}\ D_{2,2}\ \dots\ D_{2,k_2} \\
& \dots \\
| \ & C_n\ D_{n,1}\ D_{n,2}\ \dots\ D_{n,k_n}
\end{aligned}$$

where $n \in \mathbb{N}^+$ and $k_i \in \mathbb{N}$ for each $1 \le i \le n$. By this declaration we obtain

1. the new (possibly recursive) data type $D = \{\infty, \bot\} \cup \sum_{1 \le i \le n} \prod_{1 \le j \le k_i} D_{i,j}$,
2. non-strict constructor functions $C_i : \prod_{1 \le j \le k_i} D_{i,j} \to D \setminus \{\infty, \bot\}$,
3. observer functions $isC_i : D \to Bool$, and
4. selector functions $selC_i : D \to \prod_{1 \le j \le k_i} D_{i,j}$ and $selC_{i,j} : D \to D_{i,j}$.

Constructors, observers and selectors are $\preccurlyeq$-continuous and satisfy

$$(isC_i(e), selC_i(e), selC_{i,j}(e)) = \begin{cases} (\infty, \infty, \infty) & \text{if } e = \infty, \\ (\bot, \bot, \bot) & \text{if } e = \bot, \\ (true, \vec{e}, e_j) & \text{if } e = C_i(\vec{e}), \\ (false, \bot, \bot) & \text{if } e = C_k(\vec{e}) \text{ and } i \ne k. \end{cases}$$

Example 26. The declaration ***data*** *IntList* = *Cons Int IntList* | *Nil* yields the recursive type of integer lists, together with the functions

$$\begin{aligned} Cons &: Int \times IntList \to IntList \\ Nil &: IntList \\ isCons &: IntList \to Bool \\ isNil &: IntList \to Bool \\ head &: IntList \to Int \\ tail &: IntList \to IntList \end{aligned}$$

Binary trees with integer nodes are obtained similarly by ***data*** *IntTree* = *Node IntTree Int IntTree* | *Leaf*.

Pattern matching is used to access field values without directly using observers and selectors. We support the following four kinds of patterns:

$$\begin{array}{rll} pat = & \_ & \text{wild card} \\ | & v & \text{variable} \\ | & \overrightarrow{pat} & \text{tuple} \\ | & C\ pat & \text{constructor} \end{array}$$

Constants are covered by nullary constructors. Matching is performed by the case statement.

Definition 27. The case statement is:

$$\begin{array}{lcl} \textbf{\textit{case}}\ e\ \textbf{\textit{of}} & & \\ \quad pat_1 \to P_1 & & vars(pat_1, e)\ ;\ P_1\ ;\ endvars(pat_1) \blacktriangleleft match(pat_1, e) \blacktriangleright \\ \quad pat_2 \to P_2 & =_{\text{def}} & vars(pat_2, e)\ ;\ P_2\ ;\ endvars(pat_2) \blacktriangleleft match(pat_2, e) \blacktriangleright \\ \quad \ldots & & \ldots \\ \quad pat_k \to P_k & & vars(pat_k, e)\ ;\ P_k\ ;\ endvars(pat_k) \blacktriangleleft match(pat_k, e) \blacktriangleright \vec{x} \leftarrow \bot \end{array}$$

It uses the auxiliary functions *match*, *vars* and *endvars* for matching, variable binding and removal, respectively. The following condition checks whether the pattern *pat* matches the value of e:

$$match(pat, e) = \begin{cases} true & \text{if } pat \text{ is a wild card or a variable,} \\ \bigwedge_{i \in I} match(pat_i, e_i) & \text{if } pat = \overrightarrow{pat}_I \text{ and } e = \vec{e}_I, \\ isC(e) \vartriangle match(pat', selC(e)) & \text{if } pat = C\ pat', \\ false & \text{otherwise.} \end{cases}$$

The sequential conjunction $b \vartriangle c$ yields c if $b = true$, and b otherwise. The last case of *match* indicates that a tuple pattern fails to match the value of e, which is not a tuple or one of different size. A static type checker may prevent this. The variables of a pattern are declared and bound by the relation

$$vars(pat, e) = \begin{cases} \mathbb{1} & \text{if } pat \text{ is a wild card,} \\ \textbf{\textit{var}}\ v \leftarrow e & \text{if } pat \text{ is the variable } v, \\ \succ_{i \in I} vars(pat_i, e_i) & \text{if } pat = \overrightarrow{pat}_I \text{ and } e = \vec{e}_I, \\ vars(pat', selC(e)) & \text{if } pat = C\ pat'. \end{cases}$$

The iterated sequential composition $\gg_{i \in I} R_i$ performs the computations R_i in some sequence (it does not matter which). All variables of a pattern are undeclared by the relation

$$endvars(pat) = \begin{cases} \mathbb{1} & \text{if } pat \text{ is a wild card,} \\ \textbf{\textit{end}}\ v & \text{if } pat \text{ is the variable } v, \\ \gg_{i \in I} endvars(pat_i) & \text{if } pat = \overrightarrow{pat}_I, \\ endvars(pat') & \text{if } pat = C\ pat'. \end{cases}$$

Variables in a pattern must be distinct from each other and from variables of the state. Each relation P_i includes the variables of its pattern pat_i in its type.

With some more effort, the case statement may be extended to support guarded patterns as well. Patterns can be applied profitably to match against parameters in procedure declarations.

Example 28. Consider the data type *IntList* of lists of integers. Then,

$$\begin{aligned} & \textbf{\textit{case}}\ xs\ \textbf{\textit{of}}\ Cons(h,t) \to P_1 \\ & \qquad\qquad Nil \to P_2 \\ = {} & vars(Cons(h,t),xs) \mathbin{;} P_1 \mathbin{;} endvars(Cons(h,t)) \blacktriangleleft match(Cons(h,t),xs) \blacktriangleright \\ & (vars(Nil,xs) \mathbin{;} P_2 \mathbin{;} endvars(Nil) \blacktriangleleft match(Nil,xs) \blacktriangleright \vec{x}{\leftarrow}\bot) \\ = {} & \textbf{\textit{var}}\ h{\leftarrow}head(xs) \mathbin{;} \textbf{\textit{var}}\ t{\leftarrow}tail(xs) \mathbin{;} P_1 \mathbin{;} \textbf{\textit{end}}\ h \mathbin{;} \textbf{\textit{end}}\ t \blacktriangleleft isCons(xs) \blacktriangleright (\mathbb{1} \mathbin{;} P_2 \mathbin{;} \mathbb{1} \blacktriangleleft isNil(xs) \blacktriangleright \vec{x}{\leftarrow}\bot) \\ = {} & (\textbf{\textit{let}}\ h,t{\leftarrow}head(xs),tail(xs)\ \textbf{\textit{in}}\ P_1) \blacktriangleleft isCons(xs) \blacktriangleright P_2\ , \end{aligned}$$

because the observer *isCons* is strict. Using the infix notation : for *Cons* and [ ] for *Nil*, we can thus compute the length of a list by

$$\begin{aligned} length(\textbf{\textit{val}}\ xs, \textbf{\textit{ref}}\ r) & = \textbf{\textit{case}}\ xs\ \textbf{\textit{of}}\ \_ : t \to length(t,r) \mathbin{;} r{\leftarrow}r{+}1 \\ & \qquad\qquad\qquad [\,] \quad \to r{\leftarrow}0 \\ & = (\textbf{\textit{let}}\ t{\leftarrow}tail(xs)\ \textbf{\textit{in}}\ length(t,r) \mathbin{;} r{\leftarrow}r{+}1) \blacktriangleleft isCons(xs) \blacktriangleright r{\leftarrow}0 \\ & = length(tail(xs),r) \mathbin{;} r{\leftarrow}r{+}1 \blacktriangleleft isCons(xs) \blacktriangleright r{\leftarrow}0\ . \end{aligned}$$

Consequently, we might introduce pattern matching for value parameters by the alternative notation

$$\begin{aligned} length(\textbf{\textit{val}}\ (\_ : xs), \textbf{\textit{ref}}\ r) & = length(xs,r) \mathbin{;} r{\leftarrow}r{+}1 \\ length(\textbf{\textit{val}}\ [\,], \textbf{\textit{ref}}\ r) & = r{\leftarrow}0 \end{aligned}$$

With some more effort and a few design decisions, pattern matching could also be applied to reference parameters. Obviously, *length* computes a defined result only if *xs* is finite, but this is no restriction as the following procedure shows, which squares every element of a list:

$$\begin{aligned} squares(\textbf{\textit{ref}}\ xs) = \textbf{\textit{case}}\ xs\ \textbf{\textit{of}}\ & h : t \to squares(t) \mathbin{;} xs{\leftarrow}h^2{:}t \\ & [\,] \quad \to \mathbb{1} \end{aligned}$$

This procedure also works for infinite *xs*. Given the background of functional programming languages, we can observe that *length* and *squares* are instances of higher-order programs, and this is the topic of the following section.

7. Higher-order Procedures

In this section we discuss the implementation of higher-order procedures and programming patterns such as map and fold. Its counterpart unfold follows in Section 9. Versions of fold and unfold in our framework are presented in [19] in a rather ad hoc manner. Using the tools of the previous sections, we can now proceed more systematically. In particular, we need to store values representing procedures in variables. This has to be considered carefully, since procedures are relations, but our type constructions only allow sum, product, function and recursive types. Due to the restriction to deterministic programming constructs, we can use Lemma 12.2 to represent our relations by values of function type.

Consider types D_I and D_J and a procedure declared by $P(\textbf{\textit{val}}\ \vec{x}_I{:}D_I, \textbf{\textit{ref}}\ \vec{x}_J{:}D_J) = R$. Its type therefore is $P[\overrightarrow{in}_{I\cup J}, \overrightarrow{out'}_J] : D_{I\cup J} \leftrightarrow D_J$. Assuming that R is composed of the constructs of Definition 1 without the choice operator, we obtain that P satisfies $\mathscr{H}_C$, $\mathscr{H}_D$ and $\mathscr{H}_E$ by Theorems 13, 9 and 6.3. Hence Lemma 12.2 yields the $\preccurlyeq$-continuous mapping lea $P : D_{I\cup J} \to D_J$. Using this conversion implicitly, we can thus pass P as a parameter to higher-order procedures. In particular, we can pass the procedure P with signature $P(\textbf{\textit{val}}\ x{:}A, \textbf{\textit{ref}}\ y{:}B)$ as the first parameter to

$$\begin{array}{rl} \mathit{foldr}(\textbf{\textit{val}}\ P, z, \mathit{xs}, \textbf{\textit{ref}}\ r) = \textbf{\textit{case}}\ \mathit{xs}\ \textbf{\textit{of}} & h : t \to \mathit{foldr}(P, z, t, r) \mathbin{;} P(h, r) \\ & [\,] \quad\ \to r{\leftarrow}z \end{array}$$

The parameters z and r have type B while the type of xs is the lists of elements with type A. Hence foldr is a relation with type $(A \times B \to B) \times B \times \mathit{AList} \times B \leftrightarrow B$, where AList is constructed similarly to $\mathit{IntList}$. This works for any choice of A and B, and we shall not be concerned with parametric polymorphism.

Example 29. Using $\mathit{addto}(\textbf{\textit{val}}\ x, \textbf{\textit{ref}}\ r) = r{\leftarrow}r{+}x$, the sum of all elements of the finite list xs is assigned to r by $\mathit{foldr}(\mathit{addto}, 0, \mathit{xs}, r)$. But foldr also works on infinite lists: we can use $\mathit{sqCons}(\textbf{\textit{val}}\ x, \textbf{\textit{ref}}\ \mathit{ys}) = \mathit{ys}{\leftarrow}x^2{:}\mathit{ys}$ as a parameter in $\mathit{foldr}(\mathit{sqCons}, [\,], \mathit{xs}, \mathit{xs})$ to obtain the effect of $\mathit{squares}$ of Example 28.

The procedure $\mathit{squares}$ is an instance of another well-known scheme, namely map. It is obtained as the following instance of foldr:

$$\begin{array}{rl} \mathit{map}(\textbf{\textit{val}}\ P, \mathit{xs}, \textbf{\textit{ref}}\ \mathit{ys}) &= \mathit{foldr}(\mathit{apCons}_{P\leftarrow P}, [\,], \mathit{xs}, \mathit{ys}) \\ \mathit{apCons}(\textbf{\textit{val}}\ P, x, \textbf{\textit{ref}}\ \mathit{ys}) &= \textbf{\textit{let}}\ y\ \textbf{\textit{in}}\ P(x, y) \mathbin{;} \mathit{ys}{\leftarrow}y{:}\mathit{ys} \end{array}$$

Partial application is used to fix the procedure P that is applied to each element. We do not define map with the signature $\mathit{map}(\textbf{\textit{val}}\ P, \textbf{\textit{ref}}\ \mathit{xs})$ because xs and ys may have different types. This is not the case for the following instance:

$$\begin{array}{rl} \mathit{filter}(\textbf{\textit{val}}\ P, \textbf{\textit{ref}}\ \mathit{xs}) &= \mathit{foldr}(\mathit{ifCons}_{P\leftarrow P}, [\,], \mathit{xs}, \mathit{xs}) \\ \mathit{ifCons}(\textbf{\textit{val}}\ P, x, \textbf{\textit{ref}}\ \mathit{xs}) &= \mathit{xs}{\leftarrow}x{:}\mathit{xs} \blacktriangleleft P(x) \blacktriangleright \mathbb{1} \end{array}$$

As usual, the condition P is a mapping to Boolean values.

Example 30. We obtain $\mathit{squares}$ as $\mathit{map}(\mathit{square}, \mathit{xs}, \mathit{xs})$ with the procedure $\mathit{square}(\textbf{\textit{val}}\ x, \textbf{\textit{ref}}\ y) = y{\leftarrow}x^2$. Assuming that the condition $\mathit{even} : \mathit{Int} \to \mathit{Bool}$ decides if its argument is divisible by 2, we obtain that $\mathit{enumFrom}(2, \mathit{xs}) \mathbin{;} \mathit{filter}(\mathit{even}, \mathit{xs}) = \mathit{xs}{\leftarrow}[2, 4\,..]$ where $[2, 4\,..]$ denotes the infinite list $2 : 4 : 6 : \ldots$ of even numbers starting with 2.

As a further generalisation, we can add a preprocessing step to foldr. Assuming that the procedure Q has the signature $Q(\textbf{\textit{val}}\ x, \textbf{\textit{ref}}\ \mathit{xs})$, we can pass it to the procedure fold defined by

$$\begin{array}{rl} \mathit{fold}(\textbf{\textit{val}}\ Q, P, z, \mathit{xs}, \textbf{\textit{ref}}\ r) = \textbf{\textit{case}}\ \mathit{xs}\ \textbf{\textit{of}} & h : t \to Q(h, t) \mathbin{;} \mathit{fold}(Q, P, z, t, r) \mathbin{;} P(h, r) \\ & [\,] \quad\ \to r{\leftarrow}z \end{array}$$

The parameter Q describes how to modify the list under iteration xs for the next recursive call. Thus, $\mathit{foldr}(P, z, \mathit{xs}, r) = \mathit{fold}(\mathit{skip}, P, z, \mathit{xs}, r)$ with $\mathit{skip}(\textbf{\textit{val}}\ x, \textbf{\textit{ref}}\ \mathit{xs}) = \mathbb{1}$. Let us furthermore define the useful $\mathit{cons}(\textbf{\textit{val}}\ x, \textbf{\textit{ref}}\ \mathit{xs}) = \mathit{xs}{\leftarrow}x{:}\mathit{xs}$.

Example 31. We can now reconsider the prime number sieve computation and obtain:

$$\begin{array}{rl} \mathit{primes}(\textbf{\textit{ref}}\ \mathit{xs}) &= \mathit{enumFrom}(2, \mathit{xs}) \mathbin{;} \mathit{sieve}(\mathit{xs}) \\ \mathit{sieve}(\textbf{\textit{ref}}\ \mathit{xs}) &= \mathit{fold}(\mathit{remove}, \mathit{cons}, [\,], \mathit{xs}, \mathit{xs}) \\ \mathit{remove}(\textbf{\textit{val}}\ x, \textbf{\textit{ref}}\ \mathit{xs}) &= \mathit{foldr}(\mathit{div}_{p\leftarrow x}, [\,], \mathit{xs}, \mathit{xs}) \end{array}$$

The procedures *enumFrom* and *div* have been declared in Examples 20 and 25, respectively. The procedure *cons* can furthermore be used in instances of *foldr* to realise the concatenation of lists:

$$\begin{aligned} \mathit{prepend}(\textbf{\textit{val}}\ \mathit{xs}, \textbf{\textit{ref}}\ \mathit{ys}) &= \mathit{foldr}(\mathit{cons}, \mathit{ys}, \mathit{xs}, \mathit{ys}) \\ \mathit{concat}(\textbf{\textit{val}}\ \mathit{xss}, \textbf{\textit{ref}}\ \mathit{ys}) &= \mathit{foldr}(\mathit{prepend}, [\,], \mathit{xss}, \mathit{ys}) \end{aligned}$$

We use the latter in $\mathit{concatMap}(\textbf{\textit{val}}\ P, \mathit{xs}, \textbf{\textit{ref}}\ \mathit{ys}) = \textbf{\textit{let}}\ \mathit{xss}\ \textbf{\textit{in}}\ \mathit{map}(P, \mathit{xs}, \mathit{xss}) \ ;\ \mathit{concat}(\mathit{xss}, \mathit{ys})$.

The procedure *concatMap* is defined as the composition of calls to *map* and to *foldr*. It is well-known from functional programming languages that such a composition can be transformed to a single *foldr* with the advantage of having to traverse the argument list only once instead of twice [6]. The following theorem shows that such a fold-map fusion law also holds in our framework.

Theorem 32. *Let* $P(\textbf{\textit{val}}\ x, \textbf{\textit{ref}}\ y)$ *and* $Q(\textbf{\textit{val}}\ x, \textbf{\textit{ref}}\ y)$ *be procedures. Then*

$$\textbf{\textit{let}}\ \mathit{ys}\ \textbf{\textit{in}}\ \mathit{map}(P, \mathit{xs}, \mathit{ys}) \ ;\ \mathit{foldr}(Q, z, \mathit{ys}, r) = \mathit{foldr}(R, z, \mathit{xs}, r) \ ,$$

where $R(\textbf{\textit{val}}\ x, \textbf{\textit{ref}}\ y) = \textbf{\textit{let}}\ z\ \textbf{\textit{in}}\ P(x, z) \ ;\ Q(z, y)$.

If P or Q are partial applications and the supplied values are given as expressions of the current state, they must be passed to R also by partial application because of our convention to evaluate these expressions in the original state.

Proof. We first prove the claim by induction for finite and partial xs, using Theorem 23 to expand procedure calls.

1. If $\mathit{xs} = [\,]$, then

$$\begin{aligned} &\textbf{\textit{let}}\ \mathit{ys}\ \textbf{\textit{in}}\ \mathit{map}(P, \mathit{xs}, \mathit{ys}) \ ;\ \mathit{foldr}(Q, z, \mathit{ys}, r) \\ =\ &\textbf{\textit{let}}\ \mathit{ys}\ \textbf{\textit{in}}\ \mathit{foldr}(\mathit{apCons}_{P \leftarrow P}, [\,], [\,], \mathit{ys}) \ ;\ \mathit{foldr}(Q, z, \mathit{ys}, r) \\ =\ &\textbf{\textit{let}}\ \mathit{ys}\ \textbf{\textit{in}}\ \mathit{ys} \leftarrow [\,] \ ;\ \mathit{foldr}(Q, z, \mathit{ys}, r) \\ =\ &\mathit{foldr}(Q, z, [\,], r) \\ =\ &r \leftarrow z \\ =\ &\mathit{foldr}(R, z, \mathit{xs}, r) \ . \end{aligned}$$

2. If $\mathit{xs} = c \in \{\infty, \bot\}$, then

$$\begin{aligned} &\textbf{\textit{let}}\ \mathit{ys}\ \textbf{\textit{in}}\ \mathit{map}(P, \mathit{xs}, \mathit{ys}) \ ;\ \mathit{foldr}(Q, z, \mathit{ys}, r) \\ =\ &\textbf{\textit{let}}\ \mathit{ys}\ \textbf{\textit{in}}\ \mathit{foldr}(\mathit{apCons}_{P \leftarrow P}, [\,], c, \mathit{ys}) \ ;\ \mathit{foldr}(Q, z, \mathit{ys}, r) \\ =\ &\textbf{\textit{let}}\ \mathit{ys}\ \textbf{\textit{in}}\ \mathit{ys} \leftarrow c \ ;\ \mathit{foldr}(Q, z, \mathit{ys}, r) \\ =\ &\mathit{foldr}(Q, z, c, r) \\ =\ &r \leftarrow c \\ =\ &\mathit{foldr}(R, z, \mathit{xs}, r) \ . \end{aligned}$$

   Only the reference parameters ys of the first call to *foldr* and r of the second call are affected by the undefined condition which arises from pattern matching against c in the body of *foldr*. The remaining variables of the state retain their values.

3. If $\mathit{xs} = x{:}\mathit{xs}'$, then, using the induction hypothesis,

$$\begin{aligned} &\textbf{\textit{let}}\ \mathit{ys}\ \textbf{\textit{in}}\ \mathit{map}(P, \mathit{xs}, \mathit{ys}) \ ;\ \mathit{foldr}(Q, z, \mathit{ys}, r) \\ =\ &\textbf{\textit{let}}\ \mathit{ys}\ \textbf{\textit{in}}\ \mathit{foldr}(\mathit{apCons}_{P \leftarrow P}, [\,], x{:}\mathit{xs}', \mathit{ys}) \ ;\ \mathit{foldr}(Q, z, \mathit{ys}, r) \\ =\ &\textbf{\textit{let}}\ \mathit{ys}\ \textbf{\textit{in}}\ \mathit{foldr}(\mathit{apCons}_{P \leftarrow P}, [\,], \mathit{xs}', \mathit{ys}) \ ;\ \mathit{apCons}_{P \leftarrow P}(x, \mathit{ys}) \ ;\ \mathit{foldr}(Q, z, \mathit{ys}, r) \\ =\ &\textbf{\textit{let}}\ \mathit{ys}\ \textbf{\textit{in}}\ \mathit{map}(P, \mathit{xs}', \mathit{ys}) \ ;\ \mathit{apCons}(P, x, \mathit{ys}) \ ;\ \mathit{foldr}(Q, z, \mathit{ys}, r) \\ =\ &\textbf{\textit{let}}\ \mathit{ys}\ \textbf{\textit{in}}\ \mathit{map}(P, \mathit{xs}', \mathit{ys}) \ ;\ (\textbf{\textit{let}}\ y\ \textbf{\textit{in}}\ P(x, y) \ ;\ \mathit{ys} \leftarrow y{:}\mathit{ys}) \ ;\ \mathit{foldr}(Q, z, \mathit{ys}, r) \\ =\ &\textbf{\textit{let}}\ \mathit{ys}\ \textbf{\textit{in}}\ \mathit{map}(P, \mathit{xs}', \mathit{ys}) \ ;\ \textbf{\textit{let}}\ y\ \textbf{\textit{in}}\ P(x, y) \ ;\ \mathit{foldr}(Q, z, y{:}\mathit{ys}, r) \ ;\ \mathit{ys} \leftarrow y{:}\mathit{ys} \\ =\ &\textbf{\textit{let}}\ \mathit{ys}\ \textbf{\textit{in}}\ \mathit{map}(P, \mathit{xs}', \mathit{ys}) \ ;\ \textbf{\textit{let}}\ y\ \textbf{\textit{in}}\ P(x, y) \ ;\ \mathit{foldr}(Q, z, \mathit{ys}, r) \ ;\ Q(y, r) \\ =\ &(\textbf{\textit{let}}\ \mathit{ys}\ \textbf{\textit{in}}\ \mathit{map}(P, \mathit{xs}', \mathit{ys}) \ ;\ \mathit{foldr}(Q, z, \mathit{ys}, r)) \ ;\ \textbf{\textit{let}}\ y\ \textbf{\textit{in}}\ P(x, y) \ ;\ Q(y, r) \\ =\ &\mathit{foldr}(R, z, \mathit{xs}', r) \ ;\ R(x, r) \\ =\ &\mathit{foldr}(R, z, \mathit{xs}, r) \ . \end{aligned}$$

If xs is infinite, it has been generated by a recursively specified computation S. In the following, we argue for the case where S is a simple recursion; nested recursions can be treated based on this. Thus, let $S = \nu f$ for some function f mapping computations to computations without using recursion. The function f is co-continuous by Theorem 14 and therefore $S = \nu f = \bigsqcap_{n\in\mathbb{N}} f^n(\top\!\!\!\top)$ by Kleene's theorem. Moreover, the value of xs is partial or finite after each of the computations $f^n(\top\!\!\!\top)$, hence we can apply our inductively proved claim. The applicability conditions of Lemma 12.3 are satisfied by Theorems 13, 9 and 6.3. We use it twice in

$$\begin{aligned}
&S \,;\, \textbf{\textit{let}}\ ys\ \textbf{\textit{in}}\ map(P, xs, ys) \,;\, foldr(Q, z, ys, r)\\
&= (\textstyle\bigsqcap_{n\in\mathbb{N}} f^n(\top\!\!\!\top)) \,;\, \textbf{\textit{let}}\ ys\ \textbf{\textit{in}}\ map(P, xs, ys) \,;\, foldr(Q, z, ys, r)\\
&= \textstyle\bigsqcap_{n\in\mathbb{N}} f^n(\top\!\!\!\top) \,;\, \textbf{\textit{let}}\ ys\ \textbf{\textit{in}}\ map(P, xs, ys) \,;\, foldr(Q, z, ys, r)\\
&= \textstyle\bigsqcap_{n\in\mathbb{N}} f^n(\top\!\!\!\top) \,;\, foldr(R, z, xs, r)\\
&= (\textstyle\bigsqcap_{n\in\mathbb{N}} f^n(\top\!\!\!\top)) \,;\, foldr(R, z, xs, r)\\
&= S \,;\, foldr(R, z, xs, r)\ .
\end{aligned}$$

□

The case of infinite xs in the previous theorem can alternatively be proved by using the algebraic semilattice structure to represent xs as the supremum of the compact elements below xs. In the present case, every compact element is a partial or finite list.

Catamorphisms such as fold and map are also investigated by [7] in a relational context, however, in a strict setting.

8. List Comprehensions

Using the functions introduced in Section 7 we can deal with list comprehensions as known from Haskell [26]. In particular, we treat generators, filters and local declarations.

Definition 33. A list comprehension is an assignment $ys \leftarrow [\ e \mid Q\ ]$ where e is an expression and Q a sequence of generators $pat \leftarrow xs$, Boolean expressions b, and local variable declarations $\textbf{\textit{let}}\ \vec{x}_I \leftarrow \vec{e}_I$. The variables in the pattern pat and $\vec{x}_I$ must be new in the state and can be used in subsequent parts of the comprehension and in e. Each generator $pat \leftarrow xs$ can itself be a list comprehension, or xs is given directly as a list. The semantics is given recursively by

$$\begin{aligned}
ys\leftarrow[\ e \mid pat\leftarrow xs,\ Q\ ] &=_{\mathrm{def}} \textbf{\textit{let}}\ ts\leftarrow xs\ \textbf{\textit{in}}\ concatMap(T_{\vec{v}\leftarrow\vec{v}}, ts, ys)\\
&\qquad T(\textbf{\textit{val}}\ \vec{v}, t, \textbf{\textit{ref}}\ xs) = \textbf{\textit{case}}\ t\ \textbf{\textit{of}}\ pat \to xs\leftarrow[\ e \mid Q\ ]\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad _ \ \ \to xs\leftarrow[\,]\\
ys\leftarrow[\ e \mid b,\ Q\ ] &=_{\mathrm{def}} ys\leftarrow[\ e \mid true\leftarrow[b],\ Q\ ]\\
ys\leftarrow[\ e \mid \textbf{\textit{let}}\ \vec{x}_I\leftarrow\vec{e}_I,\ Q\ ] &=_{\mathrm{def}} \textbf{\textit{let}}\ \vec{x}_I\leftarrow\vec{e}_I\ \textbf{\textit{in}}\ ys\leftarrow[\ e \mid Q\ ]\\
ys\leftarrow[\ e \mid \varepsilon\ ] &=_{\mathrm{def}} ys\leftarrow[e]
\end{aligned}$$

In the first line, the vector $\vec{v}$ comprises the variables that are free in e and Q without those in pat. Their values are passed to the procedure T by partial application. Each element of xs that does not match pat is mapped to the empty list that disappears during concatenation. The same happens if the filter condition b is not satisfied.

Only the value of the variable ys may be modified by the list comprehension $ys\leftarrow[\ e \mid Q\ ]$ which preserves the other variables of the state. We obtain $ys\leftarrow[\ e \mid b,\ Q\ ] = (ys\leftarrow[\ e \mid Q\ ] \blacktriangleleft b \blacktriangleright ys\leftarrow[\,])$ if the condition b is defined.

Example 34. Consider the list comprehension $zs \leftarrow [\ xy{+}z \mid x\leftarrow[0..3],\ y\leftarrow[x..3],\ \textbf{\textit{let}}\ z\leftarrow x{+}y,\ even(xz)\ ]$. This elaborates to $concatMap(T, [0..3], zs)$ where

$$\begin{aligned}
T(\textbf{\textit{val}}\ x, \textbf{\textit{ref}}\ ys) &= ys\leftarrow[\ xy{+}z \mid y\leftarrow[x..3],\ \textbf{\textit{let}}\ z\leftarrow x{+}y,\ even(xz)\ ]\\
&= \textbf{\textit{let}}\ ts\leftarrow[x..3]\ \textbf{\textit{in}}\ concatMap(S_{x\leftarrow x}, ts, ys)\\
S(\textbf{\textit{val}}\ x, y, \textbf{\textit{ref}}\ ys) &= ys\leftarrow[\ xy{+}z \mid \textbf{\textit{let}}\ z\leftarrow x{+}y,\ even(xz)\ ]\\
&= \textbf{\textit{let}}\ z\leftarrow x{+}y\ \textbf{\textit{in}}\ ys\leftarrow[\ xy{+}z \mid even(xz)\ ]\\
&\cong \textbf{\textit{let}}\ z\leftarrow x{+}y\ \textbf{\textit{in}}\,(ys\leftarrow[xy{+}z] \blacktriangleleft even(xz) \blacktriangleright ys\leftarrow[\,])
\end{aligned}$$

The last step is not an equality if either x or y, and hence $even(xz)$ are undefined, since the conditional then sets x, y, z and ys undefined, whereas the list comprehension affects ys only. In the context of the procedure declaration S this has no effect because the modified variables are local to S. By $P(\textbf{\textit{val}}\ \vec{x}_I, \textbf{\textit{ref}}\ \vec{x}_J) = Q \cong R$ we express that $P(\textbf{\textit{val}}\ \vec{x}_I, \textbf{\textit{ref}}\ \vec{x}_J) = Q$ and $P'(\textbf{\textit{val}}\ \vec{x}_I, \textbf{\textit{ref}}\ \vec{x}_J) = R$ declare the same procedure $P = P'$.

Example 35. We now use fold-map fusion to express the procedure *remove* of Example 31 using list comprehensions. Let $remove'(\textbf{\textit{val}}\ p, \textbf{\textit{ref}}\ xs) = xs \leftarrow [\ x \mid x \leftarrow xs,\ p \nmid x\ ]$. Then

$$\begin{aligned}
& remove'(p, xs) \\
={}& xs \leftarrow [\ x \mid x \leftarrow xs,\ p \nmid x\ ] \\
={}& \textbf{\textit{let}}\ ts \leftarrow xs\ \textbf{\textit{in}}\ concatMap(T_{p \leftarrow p}, ts, xs) \\
={}& concatMap(T_{p \leftarrow p}, xs, xs)\ ,
\end{aligned}$$

where $T(\textbf{\textit{val}}\ p, x, \textbf{\textit{ref}}\ xs) = xs \leftarrow [\ x \mid p \nmid x\ ] \cong (xs \leftarrow [x] \blacktriangleleft p \nmid x \blacktriangleright xs \leftarrow [\,])$. Hence we continue by using Theorem 32 in

$$\begin{aligned}
& concatMap(T_{p \leftarrow p}, xs, xs) \\
={}& \textbf{\textit{let}}\ xss\ \textbf{\textit{in}}\ map(T_{p \leftarrow p}, xs, xss)\ ;\ concat(xss, xs) \\
={}& \textbf{\textit{let}}\ xss\ \textbf{\textit{in}}\ map(T_{p \leftarrow p}, xs, xss)\ ;\ foldr(prepend, [\,], xss, xs) \\
={}& foldr(R_{p \leftarrow p}, [\,], xs, xs)\ ,
\end{aligned}$$

where

$$\begin{aligned}
R(\textbf{\textit{val}}\ p, x, \textbf{\textit{ref}}\ xs) &= \textbf{\textit{let}}\ ys\ \textbf{\textit{in}}\ T_{p \leftarrow p}(x, ys)\ ;\ prepend(ys, xs) \\
&\cong \textbf{\textit{let}}\ ys\ \textbf{\textit{in}}\ (ys \leftarrow [x] \blacktriangleleft p \nmid x \blacktriangleright ys \leftarrow [\,])\ ;\ foldr(cons, xs, ys, xs) \\
&= \textbf{\textit{let}}\ ys\ \textbf{\textit{in}}\ (ys \leftarrow [x]\ ;\ foldr(cons, xs, ys, xs) \blacktriangleleft p \nmid x \blacktriangleright ys \leftarrow [\,]\ ;\ foldr(cons, xs, ys, xs)) \\
&= \textbf{\textit{let}}\ ys\ \textbf{\textit{in}}\ (ys \leftarrow [x]\ ;\ foldr(cons, xs, [\,], xs)\ ;\ cons(x, xs) \blacktriangleleft p \nmid x \blacktriangleright ys \leftarrow [\,]\ ;\ xs \leftarrow xs) \\
&= xs \leftarrow x{:}xs \blacktriangleleft p \nmid x \blacktriangleright \mathbb{1} \\
&\cong div(p, x, xs)\ .
\end{aligned}$$

Hence $R = div$ and $remove'(p, xs) = foldr(div_{p \leftarrow p}, [\,], xs, xs) = remove(p, xs)$. We could use this in our prime number sieve by defining

$$sieve(\textbf{\textit{ref}}\ xs) = \textbf{\textit{case}}\ xs\ \textbf{\textit{of}}\ h : t \rightarrow xs \leftarrow [\ x \mid x \leftarrow t,\ h \nmid x\ ]\ ;\ sieve(xs)\ ;\ xs \leftarrow h{:}xs\ .$$

9. Programming Patterns

In this section we discuss how to express further programming patterns in our framework. We start with *unfold* [14], that successively modifies a seed x according to a computation Q until it satisfies a condition P. The elements of the result xs are obtained by applying R to the current seed. We define *unfold* by

$$unfold(\textbf{\textit{val}}\ P, Q, R, x, \textbf{\textit{ref}}\ xs) = xs \leftarrow [\,] \blacktriangleleft P(x) \blacktriangleright \textbf{\textit{let}}\ t\ \textbf{\textit{in}}\ R(x, t)\ ;\ Q(x)\ ;\ unfold(P, Q, R, x, xs)\ ;\ xs \leftarrow t{:}xs\ .$$

The parameter P is a condition and the signatures of Q and R are $Q(\textbf{\textit{ref}}\ x)$ and $R(\textbf{\textit{val}}\ x, \textbf{\textit{ref}}\ r)$, respectively.

Example 36. We obtain $enumFrom(x, xs)$ as the instance of *unfold* where $P(x) = false$ and $Q(\textbf{\textit{ref}}\ x) = x \leftarrow x{+}1$ and $R(\textbf{\textit{val}}\ x, \textbf{\textit{ref}}\ r) = r \leftarrow x$. By further initialising the parameter x with 0, the computation assigns to xs the infinite list of natural numbers $[0..]$. Choosing $R(\textbf{\textit{val}}\ x, \textbf{\textit{ref}}\ r) = r \leftarrow 1$ yields the infinite list where each element is 1.

There is nothing to be said against implementing *unfold* in an imperative language with strict semantics. But in such instances, where termination is not available or not guaranteed, our program also works to construct (possibly) infinite lists. Moreover, it is not necessary to compute the result entirely, but only to the required precision.

So far we have seen recursion patterns such as *foldr*, *fold* and *unfold*. They are themselves instances of a general scheme, namely, symmetric linear recursion:

$$slr(\textbf{\textit{val}}\ P,Q,R,S,x,r,\textbf{\textit{ref}}\ r) = S(x,r) \blacktriangleleft P(x) \blacktriangleright \textbf{\textit{let}}\ t{\leftarrow}x\ \textbf{\textit{in}}\ Q(x,r)\ ;\ slr(P,Q,R,S,x,r,r)\ ;\ R(t,r)\ .$$

The parameter P is a condition again, and the other signatures are $Q(\textbf{\textit{ref}}\ x,r)$ and $R(\textbf{\textit{val}}\ x,\textbf{\textit{ref}}\ r)$ and $S(\textbf{\textit{val}}\ x,\textbf{\textit{ref}}\ r)$. Note that the parameter r of *slr* is passed by value and reference. The scheme subsumes cata-, ana-, hylo- and paramorphisms [23] on lists. For example, the instances mentioned above are

$$\begin{aligned}
foldr(\textbf{\textit{val}}\ P,z,xs,\textbf{\textit{ref}}\ r) &= slr(isNil,Q,R_{P\leftarrow P},skip,xs,z,r)\\
&\quad Q(\textbf{\textit{ref}}\ x,r) = x{\leftarrow}tail(x)\\
&\quad R(\textbf{\textit{val}}\ P,x,\textbf{\textit{ref}}\ r) = P(head(x),r)\\
fold(\textbf{\textit{val}}\ Q,P,z,xs,\textbf{\textit{ref}}\ r) &= slr(isNil,Q'_{Q\leftarrow Q},R_{P\leftarrow P},skip,xs,z,r)\\
&\quad Q'(\textbf{\textit{val}}\ Q,\textbf{\textit{ref}}\ x,r) = \textbf{\textit{let}}\ h{\leftarrow}head(x)\ \textbf{\textit{in}}\ x{\leftarrow}tail(x)\ ;\ Q(h,x)\\
&\quad R(\textbf{\textit{val}}\ P,x,\textbf{\textit{ref}}\ r) = P(head(x),r)\\
unfold(\textbf{\textit{val}}\ P,Q,R,x,\textbf{\textit{ref}}\ xs) &= slr(P,Q'_{Q\leftarrow Q},apCons_{P\leftarrow R},skip,x,[\,],xs)\\
&\quad Q'(\textbf{\textit{val}}\ Q,\textbf{\textit{ref}}\ x,r) = Q(x)
\end{aligned}$$

Example 37. Another instance of *slr* is

$$\begin{aligned}
zipWith(\textbf{\textit{val}}\ R,xs,ys,\textbf{\textit{ref}}\ zs) &= slr(P,Q,R'_{R\leftarrow R},skip,(xs,ys),[\,],zs)\\
&\quad P((xs,ys)) = isNil(xs) \vee isNil(ys)\\
&\quad Q(\textbf{\textit{ref}}\ x,r) = \textbf{\textit{case}}\ x\ \textbf{\textit{of}}(_:xs,_:ys) \to x{\leftarrow}(xs,ys)\\
&\quad R'(\textbf{\textit{val}}\ R,t,\textbf{\textit{ref}}\ r) = \textbf{\textit{case}}\ t\ \textbf{\textit{of}}(x:_,y:_) \to \textbf{\textit{let}}\ z\ \textbf{\textit{in}}\ R(x,y,z)\ ;\ r{\leftarrow}z{:}r
\end{aligned}$$

where the signature of R is $R(\textbf{\textit{val}}\ x,y,\textbf{\textit{ref}}\ z)$. For instance, we can use $add(\textbf{\textit{val}}\ x,y,\textbf{\textit{ref}}\ z) = z{\leftarrow}x{+}y$ as a parameter to *zipWith* in

$$fibs(\textbf{\textit{ref}}\ xs) = fibs(xs)\ ;\ zipWith(add,xs,tail(xs),xs)\ ;\ xs{\leftarrow}1{:}1{:}xs$$

to compute the infinite list of Fibonacci numbers.

The fold-left scheme for $Q(\textbf{\textit{val}}\ x,\textbf{\textit{ref}}\ r)$ is obtained by

$$\begin{aligned}
foldl(\textbf{\textit{val}}\ Q,a,xs,\textbf{\textit{ref}}\ r) &= slr(isNil,Q'_{Q\leftarrow Q},skip,skip,xs,a,r)\\
&\quad Q'(\textbf{\textit{val}}\ Q,\textbf{\textit{ref}}\ x,r) = \textbf{\textit{case}}\ x\ \textbf{\textit{of}}\ h:t \to Q(h,r) \| x{\leftarrow}t
\end{aligned}$$

It immediately returns from its recursive calls and therefore does not work on infinite lists in general, but *scanl* does, since it produces the list of partial results:

$$\begin{aligned}
scanl(\textbf{\textit{val}}\ Q,a,xs,\textbf{\textit{ref}}\ ys) &= slr(P,Q'_{Q\leftarrow Q},R,R,(xs,a),[\,],ys)\\
&\quad P((xs,a)) = isNil(xs)\\
&\quad Q'(\textbf{\textit{val}}\ Q,\textbf{\textit{ref}}\ x,r) = \textbf{\textit{case}}\ x\ \textbf{\textit{of}}(h:t,a) \to Q(h,a)\ ;\ x{\leftarrow}(t,a)\\
&\quad R(\textbf{\textit{val}}\ x,\textbf{\textit{ref}}\ r) = \textbf{\textit{case}}\ x\ \textbf{\textit{of}}(_,a) \to r{\leftarrow}a{:}r
\end{aligned}$$

Its dual *scanr* is even an instance of *foldr*:

$$\begin{aligned}
scanr(\textbf{\textit{val}}\ P,z,xs,\textbf{\textit{ref}}\ ys) &= foldr(apScan_{P\leftarrow P},[z],xs,ys)\\
&\quad apScan(\textbf{\textit{val}}\ P,x,\textbf{\textit{ref}}\ r) = \textbf{\textit{case}}\ r\ \textbf{\textit{of}}\ h:_ \to P(x,h)\ ;\ r{\leftarrow}h{:}r
\end{aligned}$$

Linear recursions are characterised by having at most one recursive call in every branch. The prototypic scheme that allows two or more (independent) recursive calls is divide-and-conquer. Its general version divides the current task into a list of subtasks. We assume signatures $Q(\textbf{\textit{val}}\ x,\textbf{\textit{ref}}\ t,xs)$ and $R(\textbf{\textit{val}}\ t,ys,\textbf{\textit{ref}}\ r)$ and $S(\textbf{\textit{val}}\ x,\textbf{\textit{ref}}\ r)$ in

$$\begin{aligned}
&dc(\textbf{\textit{val}}\ P,Q,R,S,x,\textbf{\textit{ref}}\ r) =\\
&\qquad S(x,r) \blacktriangleleft P(x) \blacktriangleright \textbf{\textit{let}}\ t,xs,ys\ \textbf{\textit{in}}\ Q(x,t,xs)\ ;\ map(dc_{P,Q,R,S\leftarrow P,Q,R,S},xs,ys)\ ;\ R(t,ys,r)\ .
\end{aligned}$$

The procedure Q generates the list of subtasks xs from the current task x and uses t to store further information not passed to the subtasks but used in the conquer phase. The procedure R combines this information with the recursively obtained results ys for all subtasks into the result r for the current task. The procedure S computes the result in the terminating cases determined by P. Termination also occurs if the list of subtasks is empty.

The common case of two recursive calls instead uses $Q(\textbf{\textit{val}}\ x, \textbf{\textit{ref}}\ t, x_1, x_2)$ and $R(\textbf{\textit{val}}\ t, y_1, y_2, \textbf{\textit{ref}}\ r)$ in

$$\begin{aligned} dc_2(\textbf{\textit{val}}\ P, Q, R, S, x, \textbf{\textit{ref}}\ r) = {} & dc(P, Q'_{Q\leftarrow Q}, R'_{R\leftarrow R}, S, x, r) \\ & Q'(\textbf{\textit{val}}\ Q, x, \textbf{\textit{ref}}\ t, xs) = \textbf{\textit{let}}\ x_1, x_2\ \textbf{\textit{in}}\ Q(x, t, x_1, x_2)\ ;\ xs \leftarrow [x_1, x_2] \\ & R'(\textbf{\textit{val}}\ R, t, ys, \textbf{\textit{ref}}\ r) = \textbf{\textit{case}}\ ys\ \textbf{\textit{of}} [y_1, y_2] \rightarrow R(t, y_1, y_2, r) \end{aligned}$$

A well-known instance is

$$\begin{aligned} mergesort(\textbf{\textit{ref}}\ xs) = {} & dc_2(P, Q, R, S, xs, xs) \\ & P(xs) = isNil(xs) \mathbin{\triangledown} isNil(tail(xs)) \\ & Q(\textbf{\textit{val}}\ x, \textbf{\textit{ref}}\ t, x_1, x_2) = split(x, x_1, x_2) \\ & R(\textbf{\textit{val}}\ t, y_1, y_2, \textbf{\textit{ref}}\ r) = merge(y_1, y_2, r) \\ & S(\textbf{\textit{val}}\ x, \textbf{\textit{ref}}\ r) = r \leftarrow x \end{aligned}$$

with the auxiliary procedures $split(\textbf{\textit{val}}\ xs, \textbf{\textit{ref}}\ ys, zs)$ and $merge(\textbf{\textit{val}}\ xs, ys, \textbf{\textit{ref}}\ zs)$ that halve a list and merge two sorted lists, respectively. We omit their definitions. The sequential disjunction $b \mathbin{\triangledown} c$ yields c if $b = false$, and b otherwise. Another well-known instance is

$$\begin{aligned} quicksort(\textbf{\textit{ref}}\ xs) = {} & dc_2(isNil, Q, R, S, xs, xs) \\ & Q(\textbf{\textit{val}}\ x, \textbf{\textit{ref}}\ t, x_1, x_2) = \textbf{\textit{case}}\ x\ \textbf{\textit{of}}\ y : ys \rightarrow t \leftarrow y\ ;\ partition(y, ys, x_1, x_2) \\ & R(\textbf{\textit{val}}\ t, y_1, y_2, \textbf{\textit{ref}}\ r) = r \leftarrow t{:}y_2\ ;\ prepend(y_1, r) \\ & S(\textbf{\textit{val}}\ x, \textbf{\textit{ref}}\ r) = r \leftarrow [\,] \end{aligned}$$

with the auxiliary procedure $partition(\textbf{\textit{val}}\ p, xs, \textbf{\textit{ref}}\ ys, zs)$ that assigns to ys the elements of xs having a value less than p and to zs the remaining ones. We omit its definition, too.

The sorting examples show that non-strict computations are beneficial also for finite data structures. Efficiency can be improved by executing only those parts of programs necessary to obtain the final results. For lists of length n, our *mergesort* performs at most $O(n \log n)$ comparisons, but fewer if only the initial elements of the sorted sequence are required. Similar speedups can be observed for an implementation of heap sort and, in the average case, also for *quicksort*.

Another use of the scheme dc_2 is in defining cata- and anamorphisms for binary trees, see [23]. Recall the data type $\textbf{\textit{data}}\ IntTree = Node\ IntTree\ Int\ IntTree \mid Leaf$ with the function $isLeaf : IntTree \rightarrow Bool$ that checks whether a given tree is empty. Then

$$\begin{aligned} foldt(\textbf{\textit{val}}\ P, z, t, \textbf{\textit{ref}}\ r) = {} & dc_2(isLeaf, Q, R_{P\leftarrow P}, S_{z\leftarrow z}, t, r) \\ & Q(\textbf{\textit{val}}\ x, \textbf{\textit{ref}}\ t, x_1, x_2) = \textbf{\textit{case}}\ x\ \textbf{\textit{of}}\ Node(l, v, r) \rightarrow t, x_1, x_2 \leftarrow v, l, r \\ & R(\textbf{\textit{val}}\ P, t, y_1, y_2, \textbf{\textit{ref}}\ r) = P(y_1, t, y_2, r) \\ & S(\textbf{\textit{val}}\ z, x, \textbf{\textit{ref}}\ r) = r \leftarrow z \end{aligned}$$

and

$$\begin{aligned} unfoldt(\textbf{\textit{val}}\ P, Q, R, x, \textbf{\textit{ref}}\ r) = {} & dc_2(P, Q'_{Q,R\leftarrow Q,R}, R', S, x, r) \\ & Q'(\textbf{\textit{val}}\ Q, R, x, \textbf{\textit{ref}}\ t, x_1, x_2) = Q(x, x_1, x_2)\ ;\ R(x, t) \\ & R'(\textbf{\textit{val}}\ t, y_1, y_2, \textbf{\textit{ref}}\ r) = r \leftarrow Node(y_1, t, y_2) \\ & S(\textbf{\textit{val}}\ x, \textbf{\textit{ref}}\ r) = r \leftarrow Leaf \end{aligned}$$

Either one may be used to implement the procedure *reflect* that mirrors a tree:

$$\begin{aligned} reflect(\textbf{\textit{ref}}\ t) = {} & foldt(P, Leaf, t, t) \\ & P(\textbf{\textit{val}}\ l, v, r, \textbf{\textit{ref}}\ t) = t \leftarrow Node(r, v, l) \\ reflect(\textbf{\textit{ref}}\ t) = {} & unfoldt(isLeaf, Q, R, t, t) \\ & Q(\textbf{\textit{val}}\ x, \textbf{\textit{ref}}\ x_1, x_2) = \textbf{\textit{case}}\ x\ \textbf{\textit{of}}\ Node(l, \_, r) \rightarrow x_1, x_2 \leftarrow r, l \\ & R(\textbf{\textit{val}}\ x, \textbf{\textit{ref}}\ r) = \textbf{\textit{case}}\ x\ \textbf{\textit{of}}\ Node(\_, v, \_) \rightarrow r \leftarrow v \end{aligned}$$

Further programming patterns such as greedy algorithms and dynamic programming are discussed by [7] in a relational context.

10. Conclusion

Key properties of our relational approach to define the semantics of imperative programs are the separate treatment of undefinedness and non-termination, a model of dependence in computations discussed in [18] with additional algebraic laws, and the support for non-strict computations and infinite data structures. In the present paper we have extended the language by several kinds of abstractions to make the approach more practical and to show its versatility. Many of these abstractions have their counterparts in functional programming languages, but had to be defined afresh in our relational context. Thus another step has been taken to integrate useful concepts of functional programming into an imperative language.

Other approaches related to our theory of non-strict computations in general are discussed in [17, 18, 19]. Further work shall be concerned with implementation issues and the connections to data flow networks [20, Section 8.3] and, in particular, to the algebra of stream processing functions [9].

Acknowledgement

I thank the anonymous referees for their helpful comments. In particular, I am indebted to the referee who pointed out an error in the assumptions about the value ranges and suggested its correction, which I have adopted in the condition $\mathscr{H}_D$ of Definition 7 in the present paper and in [19].

Appendices

A. Parallel Composition

Recall from Section 2.2 that the parallel composition of the relations $P : D_I \leftrightarrow D_J$ and $Q : D_K \leftrightarrow D_L$ such that $I \cap K = \emptyset = J \cap L$ is

$$P\|Q = (\exists \vec{x}'_K : \mathbb{I}) \mathbin{;} P \mathbin{;} (\exists \vec{x}_L : \mathbb{I}) \cap (\exists \vec{x}'_I : \mathbb{I}) \mathbin{;} Q \mathbin{;} (\exists \vec{x}_J : \mathbb{I}) : D_{I \cup K} \leftrightarrow D_{J \cup L} \ .$$

Immediate consequences are isotony, distribution over $\cup$ and annihilation by $\perp\!\!\!\perp$. Further properties of $\|$ are stated in the following lemma.

Lemma 38.

1. $(P\|Q) \cap (R\|S) = P \cap R\|Q \cap S$.
2. $\overline{P\|\mathbb{T}} = \overline{P}\|\mathbb{T}$ *and* $\overline{\mathbb{T}\|Q} = \mathbb{T}\|\overline{Q}$ *and* $\overline{P\|Q} = (\overline{P}\|\mathbb{T}) \cup (\mathbb{T}\|\overline{Q})$ *and* $\mathbb{T}\|\mathbb{T} = \mathbb{T}$.
3. $(P\|Q) \cup (R\|S) = (P \cup R\|Q \cup S) \cap \overline{\overline{P}\|\overline{S}} \cap \overline{\overline{R}\|\overline{Q}}$.
4. $(P\|Q) \mathbin{;} (R\|S) = PR\|QS$.
5. $\preccurlyeq\ = \ \preccurlyeq\|\preccurlyeq$ *and* $\prec\ = (\prec\|\preccurlyeq) \cup (\preccurlyeq\|\prec)$ *and analogously for other pointwise orders.*

Proof.

1. Since $\exists \vec{x}' : \mathbb{I}$ is univalent and $\exists \vec{x} : \mathbb{I}$ is injective we obtain

$$\begin{aligned} P \cap R\|Q \cap S &= (\exists \vec{x}'_K : \mathbb{I})(P \cap R)(\exists \vec{x}_L : \mathbb{I}) \cap (\exists \vec{x}'_I : \mathbb{I})(Q \cap S)(\exists \vec{x}_J : \mathbb{I}) \\ &= (\exists \vec{x}'_K : \mathbb{I})P(\exists \vec{x}_L : \mathbb{I}) \cap (\exists \vec{x}'_K : \mathbb{I})R(\exists \vec{x}_L : \mathbb{I}) \cap \\ &\quad\ (\exists \vec{x}'_I : \mathbb{I})Q(\exists \vec{x}_J : \mathbb{I}) \cap (\exists \vec{x}'_I : \mathbb{I})S(\exists \vec{x}_J : \mathbb{I}) \\ &= (P\|Q) \cap (R\|S) \ . \end{aligned}$$

2. Since $\exists\vec{x}' : \mathbb{I}$ is a mapping and $\exists\vec{x} : \mathbb{I}$ is injective and surjective we obtain

$$\begin{aligned} \overline{P\|\mathbb{T}} &= \overline{(\exists\vec{x}'_K : \mathbb{I})P(\exists\vec{x}_L : \mathbb{I}) \cap (\exists\vec{x}'_I : \mathbb{I})\mathbb{T}(\exists\vec{x}_J : \mathbb{I})} = \overline{(\exists\vec{x}'_K : \mathbb{I})P(\exists\vec{x}_L : \mathbb{I})} \\ &= (\exists\vec{x}'_K : \mathbb{I})\overline{P}(\exists\vec{x}_L : \mathbb{I}) \cap (\exists\vec{x}'_I : \mathbb{I})\mathbb{T}(\exists\vec{x}_J : \mathbb{I}) = \overline{P}\|\mathbb{T} \ . \end{aligned}$$

The proof of $\overline{\mathbb{T}\|Q} = \mathbb{T}\|\overline{Q}$ is symmetrical. By these two facts and part 1,

$$\overline{P\|Q} = \overline{(P\|\mathbb{T}) \cap (\mathbb{T}\|Q)} = \overline{P\|\mathbb{T}} \cup \overline{\mathbb{T}\|Q} = (\overline{P}\|\mathbb{T}) \cup (\mathbb{T}\|\overline{Q}) \ .$$

Finally, $\mathbb{T}\|\mathbb{T} = \overline{\overline{\mathbb{T}\|\mathbb{T}}} = \overline{\overline{\mathbb{T}}\|\mathbb{T}} = \overline{\bot\|\mathbb{T}} = \overline{\bot} = \mathbb{T}$.

3. By parts 1 and 2,

$$\begin{aligned} (P\|Q) \cup (R\|S) &= ((P\|\mathbb{T}) \cap (\mathbb{T}\|Q)) \cup ((R\|\mathbb{T}) \cap (\mathbb{T}\|S)) \\ &= ((P\|\mathbb{T}) \cup (R\|\mathbb{T})) \cap ((P\|\mathbb{T}) \cup (\mathbb{T}\|S)) \cap ((\mathbb{T}\|Q) \cup (R\|\mathbb{T})) \cap ((\mathbb{T}\|Q) \cup (\mathbb{T}\|S)) \\ &= (P \cup R\|\mathbb{T}) \cap \overline{\overline{P}\|\overline{S}} \cap \overline{\overline{R}\|\overline{Q}} \cap (\mathbb{T}\|Q \cup S) \\ &= (P \cup R\|Q \cup S) \cap \overline{\overline{P}\|\overline{S}} \cap \overline{\overline{R}\|\overline{Q}} \ . \end{aligned}$$

4. Let $P : D_I \leftrightarrow D_J$ and $Q : D_K \leftrightarrow D_L$ and $R : D_J \leftrightarrow D_M$ and $S : D_L \leftrightarrow D_N$, then

$$\begin{aligned} & (\vec{x}_{I\cup K}, \vec{z}_{M\cup N}) \in PR\|QS \\ \Leftrightarrow\ & (\vec{x}_I, \vec{z}_M) \in PR \wedge (\vec{x}_K, \vec{z}_N) \in QS \\ \Leftrightarrow\ & (\exists\vec{y}_J : (\vec{x}_I, \vec{y}_J) \in P \wedge (\vec{y}_J, \vec{z}_M) \in R) \wedge (\exists\vec{y}_L : (\vec{x}_K, \vec{y}_L) \in Q \wedge (\vec{y}_L, \vec{z}_N) \in S) \\ \Leftrightarrow\ & \exists\vec{y}_{J\cup L} : (\vec{x}_I, \vec{y}_J) \in P \wedge (\vec{x}_K, \vec{y}_L) \in Q \wedge (\vec{y}_J, \vec{z}_M) \in R \wedge (\vec{y}_L, \vec{z}_N) \in S \\ \Leftrightarrow\ & \exists\vec{y}_{J\cup L} : (\vec{x}_{I\cup K}, \vec{y}_{J\cup L}) \in P\|Q \wedge (\vec{y}_{J\cup L}, \vec{z}_{M\cup N}) \in R\|S \\ \Leftrightarrow\ & (\vec{x}_{I\cup K}, \vec{z}_{M\cup N}) \in (P\|Q)(R\|S) \ . \end{aligned}$$

5. First, we have $(\vec{x}_{I\cup K}, \vec{x}'_{I\cup K}) \in {\preccurlyeq}\|{\preccurlyeq} \Leftrightarrow \vec{x}_I \preccurlyeq \vec{x}'_I \wedge \vec{x}_K \preccurlyeq \vec{x}'_K \Leftrightarrow \vec{x}_{I\cup K} \preccurlyeq \vec{x}'_{I\cup K}$. We can analogously derive $\mathbb{I} = \mathbb{I}\|\mathbb{I}$. Together with parts 2 and 1 we obtain

$$\begin{aligned} {\prec} &= {\preccurlyeq} \cap \overline{\mathbb{I}} = ({\preccurlyeq}\|{\preccurlyeq}) \cap \overline{\mathbb{I}\|\mathbb{I}} = ({\preccurlyeq}\|{\preccurlyeq}) \cap ((\overline{\mathbb{I}}\|\mathbb{T}) \cup (\mathbb{T}\|\overline{\mathbb{I}})) = (({\preccurlyeq}\|{\preccurlyeq}) \cap (\overline{\mathbb{I}}\|\mathbb{T})) \cup (({\preccurlyeq}\|{\preccurlyeq}) \cap (\mathbb{T}\|\overline{\mathbb{I}})) \\ &= ({\preccurlyeq} \cap \overline{\mathbb{I}}\|{\preccurlyeq}) \cup ({\preccurlyeq}\|{\preccurlyeq} \cap \overline{\mathbb{I}}) = ({\prec}\|{\preccurlyeq}) \cup ({\preccurlyeq}\|{\prec}) \ . \end{aligned}$$

□

B. On Partial Orders

We first discuss a property of directed sets and then fixpoints, using $\leq$ to denote the partial order. Call a partially ordered set P *complete* iff every directed set has a supremum in P.

Theorem 39. *Let P be a partial order, $S \subseteq P$ a directed set, $A \subseteq S$ and $A' = S \setminus A$. Then A is directed or A' is directed. If P is complete, then:*

* *If both A and A' are directed, then $\sup A \leq \sup A' = \sup S$ or $\sup A' \leq \sup A = \sup S$.*
* *If only A is directed, then $\sup A = \sup S$.*
* *If only A' is directed, then $\sup A' = \sup S$.*

Proof. If $A = \emptyset$ or $A = S$, all claims clearly hold. Otherwise both A and A' are not empty.

Assume that neither A nor A' is directed, hence there are $x, y \in A$ with no upper bound in A and $u, v \in A'$ with no upper bound in A'. Since S is directed, there is an upper bound $z \in S$ of x, y, u, v. But $z \in A$ or $z \in A'$, hence we obtain a contradiction.

For the remainder of this proof, let P be complete.

We first treat the case where both A and A' are directed, hence $\sup A$ and $\sup A'$ exist. Assume that neither $\sup A \leq \sup A'$ nor $\sup A' \leq \sup A$, hence there is $x \in A$ with $x \not\leq \sup A'$ and $u \in A'$ with $u \not\leq \sup A$. Since S is directed, there is an upper bound $z \in S$ of x, u. But $z \in A$ implies $u \leq z \leq \sup A$, and $z \in A'$

implies $x \leq z \leq \sup A'$, hence we obtain a contradiction in either case. Therefore one of $\sup A$ and $\sup A'$ is above the other, hence an upper bound of $S = A \cup A'$, but still below $\sup S$ and thus equal to $\sup S$.

We come to the case where A is directed, hence $\sup A$ exists, but A' is not directed, hence there are $u, v \in A'$ with no upper bound in A'. By the argument above, it suffices to show that $\sup A$ is an upper bound of A'. Let $w \in A'$, then there is an upper bound $z \in S$ of u, v, w since S is directed. Since $z \notin A'$, we have $z \in A$ and thus $w \leq z \leq \sup A$.

The remaining case is symmetric by swapping A with A'. ☐

The following result of Markowsky [22, Theorem 9(i)] allows us to prove closure under fixpoints. Call a partially ordered set P *chain-complete* (*chain-co-complete*) iff every chain has a supremum (infimum) in P.

Proposition 40. *Every isotone function on a chain-complete partially ordered set has a least fixpoint.*

In the following, let μf (νf) denote the least (greatest) fixpoint of f with respect to the partial order $\leq$.

Theorem 41. *Let P be a chain-complete partially ordered set, $f : P \to P$ isotone, $S \subseteq P$ closed under f and suprema of chains. Then $\mu f \in S$.*

PROOF. The least fixpoint μf exists by Proposition 40. We first show that $A =_{\text{def}} \{x \mid x \in S \wedge x \leq \mu f\}$ is chain-complete. Let C be a chain in A, then $\sup C \in S$ by closure of S under suprema of chains and $\sup C \leq \mu f$ by the join property. It is essential that the previous statement includes the empty chain. We next show that f is a function on A. Let $x \in A$, then $x \in S$ and $x \leq \mu f$, hence $f(x) \in S$ since S is closed under f and $f(x) \leq f(\mu f) = \mu f$ by isotony and the fixpoint property, thus $f(x) \in A$. We finally conclude by Proposition 40 that f has a least fixpoint $a \in A$, hence $a = \mu f$, and therefore $\mu f \in S$. ☐

Corollary 42. *Let P be a chain-co-complete partially ordered set, $f : P \to P$ isotone, $S \subseteq P$ closed under f and infima of chains. Then $\nu f \in S$.*

PROOF. Apply Theorem 41 to the dual of P. ☐

Applications of Corollary 42 in this paper instantiate P by the complete lattice of relations and S by the relations satisfying certain conditions, which not always form a complete lattice.

References

[1] S. Abramsky and A. Jung. Domain theory. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 3: Semantic Structures, chapter 1, pages 1–168. Clarendon Press, 1994.

[2] R.-J. Back and V. Preoteasa. An algebraic treatment of procedure refinement to support mechanical verification. *Formal Aspects of Computing*, 17(1):69–90, May 2005.

[3] R.-J. Back and J. von Wright. *Refinement Calculus.* Springer-Verlag, New York, 1998.

[4] R. C. Backhouse, P. J. de Bruin, P. Hoogendijk, G. Malcolm, E. Voermans, and J. van der Woude. Polynomial relators (extended abstract). In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Algebraic Methodology and Software Technology*, pages 303–326. Springer-Verlag, 1992.

[5] R. Berghammer and B. von Karger. Relational semantics of functional programs. In C. Brink, W. Kahl, and G. Schmidt, editors, *Relational Methods in Computer Science*, chapter 8, pages 115–130. Springer-Verlag, Wien, 1997.

[6] R. Bird. *Introduction to Functional Programming using Haskell.* Prentice Hall, second edition, 1998.

[7] R. Bird and O. de Moor. *Algebra of Programming.* Prentice Hall, 1997.

[8] M. Broy, R. Gnatz, and M. Wirsing. Semantics of nondeterministic and noncontinuous constructs. In F. L. Bauer and M. Broy, editors, *Program Construction*, volume 69 of *Lecture Notes in Computer Science*, pages 553–592. Springer-Verlag, 1979.

[9] M. Broy and G. Ştefănescu. The algebra of stream processing functions. *Theoretical Computer Science*, 258(1–2):99–129, May 2001.

[10] A. Cavalcanti and J. Woodcock. ZRC – a refinement calculus for Z. *Formal Aspects of Computing*, 10(3):267–289, March 1998.

[11] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order.* Cambridge University Press, second edition, 2002.

[12] J. Desharnais and B. Möller. Least reflexive points of relations. *Higher-Order and Symbolic Computation*, 18(1–2):51–77, June 2005.

[13] E. W. Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.

[14] J. Gibbons and G. Jones. The under-appreciated unfold. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming*, pages 273–279. ACM Press, 1998.

[15] M. J. C. Gordon. *The denotational description of programming languages.* Springer-Verlag, New York, 1979.

[16] T. F. Gritzner and R. Berghammer. A relation algebraic model of robust correctness. *Theoretical Computer Science*, 159(2):245–270, June 1996.

[17] W. Guttmann. *Algebraic Foundations of the Unifying Theories of Programming.* PhD thesis, Universität Ulm, December 2007.

[18] W. Guttmann. Lazy relations. In R. Berghammer, B. Möller, and G. Struth, editors, *Relations and Kleene Algebra in Computer Science*, volume 4988 of *Lecture Notes in Computer Science*, pages 138–154. Springer-Verlag, 2008.

[19] W. Guttmann. Lazy UTP. In A. Butterfield, editor, *Second International Symposium on Unifying Theories of Programming*, volume 5713 of *Lecture Notes in Computer Science*, pages 82–101. Springer-Verlag, 2010.

[20] C. A. R. Hoare and J. He. *Unifying theories of programming.* Prentice Hall Europe, 1998.

[21] J. Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, April 1989.

[22] G. Markowsky. Chain-complete posets and directed sets with applications. *Algebra Universalis*, 6(1):53–68, 1976.

[23] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer-Verlag, 1991.

[24] D. A. Naumann. A categorical model for higher order imperative programming. *Mathematical Structures in Computer Science*, 8(4):351–399, August 1998.

[25] M. E. O'Neill. The genuine sieve of Eratosthenes. *Journal of Functional Programming*, 19(1):95–106, January 2009.

[26] S. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report.* Cambridge University Press, 2003.

[27] W.-P. de Roever. *Recursive program schemes: semantics and proof theory.* Number 70 in Mathematical Centre Tracts. Mathematisch Centrum, Amsterdam, 1976.

[28] D. A. Schmidt. *Denotational Semantics: A Methodology for Language Development.* William C. Brown Publishers, 1986.

[29] G. Schmidt. Partiality I: Embedding relation algebras. *Journal of Logic and Algebraic Programming*, 66(2):212–238, February–March 2006.

[30] G. Schmidt, C. Hattensperger, and M. Winter. Heterogeneous relation algebra. In C. Brink, W. Kahl, and G. Schmidt, editors, *Relational Methods in Computer Science*, chapter 3, pages 39–53. Springer-Verlag, Wien, 1997.

[31] G. Schmidt and T. Ströhlein. *Relationen und Graphen.* Springer-Verlag, 1989.

[32] M. B. Smyth. Power domains. *Journal of Computer and System Sciences*, 16(1):23–36, February 1978.

[33] H. Søndergaard and P. Sestoft. Non-determinism in functional languages. *The Computer Journal*, 35(5):514–523, October 1992.