Lazy UTP

Walter Guttmann

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

Abstract. We integrate non-strict computations into the Unifying Theories of Programming. After showing that this is not possible with designs, we develop a new relational model representing undefinedness independently of non-termination. The relations satisfy additional healthiness conditions that model dependence in computations in an elegant algebraic form using partial orders. Programs can be executed according to the principle of lazy evaluation, otherwise known from functional programming languages. We extend the theory to support infinite data structures and give examples to show their use in programs.

1 Introduction

Our goal is to extend the Unifying Theories of Programming (UTP) by non-strict computations. Consider the statement $P =_{\text{def}} (x_1, x_2 := \lightning_0, 2)$ that simultaneously assigns an undefined value to x_1 and 2 to x_2. In UTP and most conventional languages its execution fails, but we want undefined expressions to remain harmless if their value is not needed. This is standard in functional programming languages with lazy evaluation like Haskell [25], Clean [26] and Miranda [37]. Yet also in an imperative language the equation $P \,;\, (x_1 := x_2) = (x_1, x_2 := 2, 2)$ can be reasonable since the value of x_1 after the execution of P is never used. This is confirmed by the following Haskell program that implements $P \,;\, (x_1 := x_2)$ in monadic style:

```
import Data.IORef;
main = do r <- newIORef (div 1 0 , 2)
          modifyIORef r (\(x1,x2) -> (x2,x2))
          x <- readIORef r
          print x
```

It prints `(2,2)` terminating successfully, but would abort if `(x2,x2)` was changed to `(x1,x1)`. With non-strict computations available, programs can be expressed more freely since less attention has to be paid to avoid non-termination. For example, in functional programming languages they enable the use of infinite data structures. They too are not supported by UTP so far.

Regarding the statement P again, we have to address that UTP models undefinedness as non-termination [15, page 78]. In particular, $P = (\mathit{false} \vdash \mathit{true})$ holds, hence P is the never terminating program (the solution of the recursive

specification $X = X$). In consequence there is no distinction between undefinedness of individual variables; actually $P = (x_1, x_2{:=}2, {}^1\!/\!_0)$ holds. Moreover, computations are strict in the sense that $P \,;\, (x_1{:=}x_2)$ is again the endless loop.

In some contexts such a uniform treatment of non-termination and undefinedness is not appropriate. UTP's point of view is that of the specifier who does not care whether a program loops indefinitely or aborts due to an error, since in both cases it does not fulfil its objective. We can, however, argue for a differentiation between finite and infinite failure. From the users' point of view, errors can actually be observed about executions of programs whereas non-termination cannot. From the programmers' and language designers' point of view, errors might be recovered from, for example, by exception handling. From the theorists' point of view, error detection is semidecidable in contrast to *non*-termination which is not semidecidable. We therefore strive for a theory that separates undefinedness and non-termination. It is then manifest to regard variables individually to obtain an even finer distinction.

As explained in Section 2, UTP's designs are not adequate to support non-strict computations. Let us therefore describe our new approach. As usual, we represent undefinedness of individual variables by adding a special value $\bot$ to their ranges. We add another special element ∞ to distinguish non-termination from undefinedness. The difficulty is to choose the relations and operations (that model computations) such that, on the one hand, they handle these special values correctly and, on the other hand, they are continuous. The latter is required to iteratively approximate the solutions to recursive equations, which corresponds to the evaluation of recursion in practice. Furthermore, key constructs such as composition and choice should retain their familiar relational meaning to obtain nice algebraic properties. We solve this problem by introducing a partial order on the ranges of variables and states, and forming the closure of relations with respect to this order.

Section 3 gives the relational basics. A compendium of relations modelling the programming constructs known from UTP is presented in Section 4. We identify several healthiness conditions they satisfy, starting with isotony and the left and right unit laws. In Section 5 we derive further properties, namely finite branching, continuity and totality. We thus obtain a theory similar to that of designs, but describing non-strict computations, able to yield defined results in spite of undefined inputs. Moreover, it is sufficient to execute only those parts of a program necessary to calculate the final results, which can improve efficiency.

Our framework can also be applied to programs with infinite data structures. Several examples constructing and modifying infinite lists are discussed in Section 6. We also show how to express in our framework the class of fold- and unfold-computations on (finite and infinite) lists. They are well-known in functional programming languages and include such operations as *map* and *filter*, the building blocks of list comprehensions.

With lazy execution comes the need to consider dependences between individual computations. Such dependences also play a role in optimising program transformations like those performed in compilers. Their structure is investigated in Section 7. Starting from the observation that non-strict computations with

defined results cannot depend on undefined inputs, we derive two additional healthiness conditions. Using another partial order we develop an equivalent, algebraically elegant form of these properties. All our programming constructs satisfy them, but they are also applicable to relations modelling new constructs.

In short, the contributions of this paper are an extension of UTP by non-strict computations, appropriate healthiness conditions and infinite data structures.

This paper uses material obtained as a part of the author's PhD thesis [11]. A condensed account of that part is given in [12]. Substantial extensions of the present paper include the connections to UTP, a theory extended to more general orders, and programs using infinite data structures. Proofs of our results can be adapted from [11] to the present, more general setting (although some claims are considerably harder to show).

2 Designs

We have seen the need to separate undefinedness from non-termination. Already modelling non-termination, UTP's designs are obvious candidates for a modified treatment of undefinedness. In this section we show that although such an extension is possible, it leads to a fundamental problem. The conclusion is that designs cannot adequately model non-strict computations. In Section 3 we therefore introduce the relational foundations of an alternative model which is used in the remainder of this paper.

Before we investigate designs, and for further reference, recall that the healthiness conditions H1–H4 of UTP are equivalent to the following four algebraic restrictions with respect to sequential composition:

$$\text{H1a. } \mathbb{I}_\mathrm{D} \,;\, R = R \qquad \text{H3. } R = R \,;\, \mathbb{I}_\mathrm{D}$$
$$\text{H1b. } \mathbb{O}_\mathrm{D} \,;\, R = \mathbb{O}_\mathrm{D} \qquad \text{H4. } \mathbb{O}_\mathrm{D} = R \,;\, \mathbb{O}_\mathrm{D}$$

The skip design $\mathbb{I}_\mathrm{D} = (\mathit{true} \vdash \vec{x}{=}\vec{x}')$ should be left- and right-neutral and the design $\mathbb{O}_\mathrm{D} = (\mathit{false} \vdash \mathit{true})$ should be left- and right-absorbing. The design $\mathbb{O}_\mathrm{D}$ is also denoted *true* by [15] which is correct but confusing in the following discussion. We intend to explain in detail why the law H1b is incompatible with non-strictness; the reader who takes this for granted may jump to Section 3.

Consider the design $(P \vdash Q)$ where the precondition P represents the terminating states, while Q represents the possible transitions starting in those states. Let us focus on the type of the relation Q between program states. Assume for the sake of exposition that the program has two variables x_1 and x_2 ranging over the natural numbers $\mathbb{N}$. A state then is an element of $\mathbb{N}^2 =_\mathrm{def} \mathbb{N} \times \mathbb{N}$, and the transition relation Q is an element of $\mathbb{N}^2 \leftrightarrow \mathbb{N}^2$. No provisions are made to represent variables with undefined values. Indeed, there is no reason to, since undefinedness is modelled as non-termination in the component P of designs.

To separate undefinedness from non-termination we have to provide means to represent undefined values in the transition relation Q of designs. This is achieved by modifying the set of states in either of two ways. Both start by

extending the range of each variable to $\mathbb{N} \cup \{\bot\}$, where the special element $\bot$ represents the undefined value.

The first approach uses the smash product of both variable ranges $\mathbb{N}^2 \cup \{\bot\}$ as the set of states. A transition relation then is an element of $(\mathbb{N}^2 \cup \{\bot\}) \leftrightarrow (\mathbb{N}^2 \cup \{\bot\})$. In this case $\bot$ models undefinedness of the state as a whole but not of its constituents, the individual variables.

To achieve the latter, we instead take the Cartesian product $(\mathbb{N} \cup \{\bot\})^2$ as the set of states (the problem we exhibit below remains also with the smash product). Thus undefined and defined variables may coexist as exemplified by

$$\begin{aligned}
&(x_1, x_2{:=}{}^1\!/_0, 2)\ ;\ (x_1{:=}x_2) \\
&= \{((x_1, x_2), (x_1', x_2')) \mid x_1'{=}\bot \wedge x_2'{=}2\}\ ;\ \{((x_1, x_2), (x_1', x_2')) \mid x_1'{=}x_2 \wedge x_2'{=}x_2\} \\
&= \{((x_1, x_2), (x_1', x_2')) \mid x_1'{=}2 \wedge x_2'{=}2\} \\
&= (x_1, x_2{:=}2, 2)\ .
\end{aligned}$$

Note that the assignment here is regarded as a plain transition relation, not as a design, because termination is not treated yet. The special element $\bot$ represents that x_1 has been assigned an expression with undefined value. However, this first assignment to x_1 has no effect since its value is never used but immediately overwritten. It is not even necessary to evaluate the corresponding right hand side. Unaffected by these considerations is the value of x_2.

The transition relations, now elements of $(\mathbb{N} \cup \{\bot\})^2 \leftrightarrow (\mathbb{N} \cup \{\bot\})^2$, are built into designs to deal with non-termination. For the following argument, we redefine the assignment as the design

$$(x_1, x_2{:=}e_1, e_2) =_{\text{def}} (\mathit{true} \vdash x_1'{=}e_1 \wedge x_2'{=}e_2)\ ,$$

reflecting the fact that an assignment always terminates as opposed to the original assignment of UTP. To complete the separation of undefinedness and non-termination, also conditional statements would have to be redefined, since their conditions are expressions and can have undefined values, too. We leave out this definition, because it does not affect the following two facts. First,

$$\begin{aligned}
(x_1{:=}{}^1\!/_0)\ ;\ (x_1, x_2{:=}2, 3) \ &=\ (\mathit{true} \vdash x_1'{=}\bot \wedge x_2'{=}x_2)\ ;\ (\mathit{true} \vdash x_1'{=}2 \wedge x_2'{=}3) \\
&=\ (\mathit{true} \vdash x_1'{=}2 \wedge x_2'{=}3) \ =\ (x_1, x_2{:=}2, 3)\ ,
\end{aligned}$$

using the composition formula of designs. The undefined value of x_1 has no effect, which is just what we expect from a non-strict computation. Second,

$$\begin{aligned}
\mathbb{O}_{\text{D}}\ ;\ (x_1, x_2{:=}2, 3) \ &=\ (\mathit{false} \vdash \mathit{true})\ ;\ (\mathit{true} \vdash x_1'{=}2 \wedge x_2'{=}3) \\
&=\ (\mathit{false} \vdash \mathit{true}) \ =\ \mathbb{O}_{\text{D}}\ ,
\end{aligned}$$

recalling that the design $\mathbb{O}_{\text{D}}$ represents non-termination. It is left absorbing, which is just what we expect from designs according to H1b. We now argue that the latter equation, although it is algebraically elegant, does not co-operate well with the first one, and hence cannot be upheld in a non-strict setting.

Consider the possible execution strategies for a program $R\ ;\ (x_1, x_2{:=}2, 3)$, assuming we do not know whether $R = (x_1 := {}^1\!/_0)$ or $R = \mathbb{O}_{\text{D}}$ holds, since

this is undecidable in general. Conventionally, one would first execute R and then $(x_1, x_2 := 2, 3)$. This leads to non-termination if $R = \mathbb{O}_\mathrm{D}$, but aborts if $R = (x_1 := 1/0)$, which is inconsistent with the first fact derived above. To avoid this error, one could alternatively start with $(x_1, x_2 := 2, 3)$, realising that the values of the variables prior to this assignment are not needed. The execution of R is thus omitted, which is inconsistent with the second fact if $R = \mathbb{O}_\mathrm{D}$.

The conflict between both facts is summarised as follows: According to the first, it is possible to recover from undefinedness, but according to the second, it is impossible to recover from non-termination. To observe the latter, otherwise unnecessary calculations have to be performed. They possibly abort due to undefined expressions, contradicting the former.

Since it is our aim to model non-strict computations, we are forced to give up an equation like $\mathbb{O}_\mathrm{D} \,;\, (x_1, x_2 := 2, 3) = \mathbb{O}_\mathrm{D}$. This is an instance of the healthiness condition $\mathbb{O}_\mathrm{D} \,;\, R = \mathbb{O}_\mathrm{D}$ that every design R satisfies, called H1b above. 'However, a lazy functional language does not satisfy this law.'[14, page 24] Although we are not specifically concerned with functional programming languages, we therefore cannot use UTP's designs for our purpose.

3 Relational Preliminaries

In this section we set up the context of the investigation of non-strictness. We describe the relational model of imperative, non-deterministic programs in detail and introduce terminology, notation and conventions used in this paper.

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 $\bot$ and ∞ with the following intuitive meaning: If the variable x_i has the value $\bot$ *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. Further structure is imposed on D_i in Sections 4.1 and 7.

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 $a \in \{\infty, \bot\}$ we express that $x_i{=}a$ for all $i \in 1..m$. 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 $x'_i \in D_i$. Composed of the output values, 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 $m \neq n$. Using UTP terminology, the input alphabet is $\{x_1, \ldots, x_m\}$ and the output alphabet is $\{x'_1, \ldots, x'_n\}$ with possibly different m and n.

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 $|R(\vec{x})| > 1$. Compared to designs, the new models get by with just one relation instead of two, and this is compensated by the additional special elements $\bot$ and ∞.

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 [31, 32]; a homogeneous model would complicate the treatment of local variables in recursive calls (by stacks) and parallel composition (by merge). We omit any notational distinction of the types of relations and their operations and assume type-correctness in their use.

We denote the identity and universal relations by $\mathbb{I}$ 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 $P\,;\,Q$ 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 $R\mathbb{T} = \mathbb{T}$ and univalent iff $R^\smile R \subseteq \mathbb{I}$. A relation is a mapping iff it is both total and univalent. Note that totality is exactly the healthiness condition H4.

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 $(\exists \vec{x}'_L : R)$. See Section 4.4 for the correspondence to variable (un)declaration.

Defined in terms of the projection, we furthermore use the following relational parallel composition operator, similar to that of [2, 3, 28]. The parallel composition of the relations $P : D_I \leftrightarrow D_J$ and $Q : D_K \leftrightarrow D_L$ is

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

If necessary, we write $P \|_K Q$ to clarify the partition of $I \cup K$ (a more detailed notation would also clarify the partition of $J \cup L$). In our theory of non-strict

computations the $\|$ operator corresponds to conjunction rather than the parallel composition of disjoint processes in [15, Section 7.1].

Recall that a non-empty subset S of a partially ordered set is directed iff each pair of elements of S has an upper bound in S. We apply the dual notion to the lattice of relations only: A set S of relations is *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 $R \subseteq Q$.

4 Programming Constructs

We present a relational model of non-strict computations. Since we cannot use UTP's designs, we have to reformulate the respective theory. In particular, we give new definitions for most programming constructs and identify several healthiness conditions they satisfy. The latter starts with isotony and the unit laws in Section 4.5, followed by boundedness, continuity and totality in Section 5 and two dependence conditions in Section 7.

4.1 Values

The state of an imperative program is given by the values of its variables, taken from the ranges D_i introduced above. They contain the special elements $\bot$ and ∞ modelling undefinedness and non-termination. Instead of regarding D_i as an unstructured set, we augment the ranges to partially ordered structures. This is usual, for example, in the semantics of functional programming languages. Among the various suggested structures are directed or ω-complete (pointed) partial orders [1, 30] or complete lattices [36]. We choose the *algebraic semilattices* of [6], which are complete semilattices having a basis of finite elements. They 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 and infima of non-empty 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 dual order of $\preccurlyeq$ is denoted by $\succcurlyeq =_{\text{def}} \preccurlyeq^{\smile}$. An order similar to $\preccurlyeq$, in which $\bot$ is the least element, is introduced in Section 7.

Our data types are constructed as follows. Elementary types, such as the Boolean values $\mathit{Bool} =_{\text{def}} \{\infty, \bot, \mathit{true}, \mathit{false}\}$ and the integer numbers $\mathit{Int} =_{\text{def}} \mathbb{Z} \cup \{\infty, \bot\}$, are flat partial orders, that is, $x \preccurlyeq y \Leftrightarrow_{\text{def}} x{=}\infty \vee x{=}y$. Thus $\bot$ 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 $\{\infty, \bot\} \cup \{(i, x) \mid x \in D_i\}$ ordered by $x \preccurlyeq y \Leftrightarrow_{\text{def}} x{=}\infty \vee x{=}\bot{=}y \vee (x{=}(i, x_i) \wedge y{=}(i, y_i) \wedge x_i \preccurlyeq_{D_i} y_i)$. 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, $\vec{x}_I \preccurlyeq \vec{y}_I \Leftrightarrow_{\text{def}} \forall i \in I : x_i \preccurlyeq_{D_i} y_i$. 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 [30].

Some results can be strengthened if we restrict our constructions to union and product. It is then easily proved by induction that every chain $C \subseteq D_i$ ordered by $\preccurlyeq$ is finite (a chain is a totally ordered subset). Even more, the lengths of the chains are bounded, so that the variable ranges are partial orders with finite height. Our previous work [11, 12] restricts D_i to flat orders for reasons explained in Section 5. The new extension to more general orders is indispensable for infinite data structures, see Section 6.

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 each variable x_i.

4.2 Skip

In this and the following sections, we successively define our programming constructs using relations on the state and discuss essential algebraic properties. In particular, the order $\preccurlyeq$ is a relation on states which turns out to be fundamental. Indeed, we take it as the definition of the new relation modelling skip, denoted also by $\mathbb{1} =_{\text{def}} \preccurlyeq$. While this action may appear strange, it can be compared to the redefinition of skip in [15, Section 9.1] to support procedure values. Although we do not treat such values in this paper, $\preccurlyeq$ can be interpreted as a kind of refinement [20, 22]. Further explanation of $\mathbb{1}$ is provided by the following connection to designs.

Remark. The intention underlying the definition of $\mathbb{1}$ is to enforce an upper closure of the image of each state with respect to $\preccurlyeq$. Traces of such a procedure can be found in the healthiness conditions of designs: 'The healthiness condition H2 states formally that the predicate R is upward closed in the variable ok': as ok' changes from false to true, R cannot change from true to false.'[15, page 83] Since H3 implies H2, every H3-design is upper closed in this way. For H3-designs, [10] shows how to replace the auxiliary variables ok and ok' by a special element that corresponds to ∞ in our present discussion. In particular, [10, Lemma 9.2] formulates the upper closure as $\overline{R}\mathbb{T} \cap R \subseteq \overline{V^\smile}$, where V corresponds to the vector $\vec{x}{=}\infty$. By the Schröder law of relation algebra,

$$\begin{aligned} \overline{R}\mathbb{T} \cap R \subseteq \overline{V^\smile} \;&\Leftrightarrow\; \overline{R}\mathbb{T} \cap V^\smile \subseteq \overline{R} \;\Leftrightarrow\; \overline{R}V^\smile \subseteq \overline{R} \\ &\Leftrightarrow\; RV \subseteq R \;\Leftrightarrow\; RV \cup R = R \;\Leftrightarrow\; R(V \cup \mathbb{I}) = R\,. \end{aligned}$$

If the state is a flat order, $V \cup \mathbb{I} = (x{=}\infty) \cup (x{=}x') = (x \preccurlyeq x')$, and we obtain the right unit law $R \,;\, \preccurlyeq \,=\, R$. Our definition of $\mathbb{1}$ refines this by distinguishing individual variables and non-flat orders. The refined right unit law corresponding to the healthiness condition H3 of designs is stated in the following definition.

As usual, skip should be a left and a right unit of sequential composition.

Definition 1. *$\mathscr{H}_L(P) \Leftrightarrow_{\text{def}} \mathbb{1} \,;\, P = P$ and $\mathscr{H}_R(P) \Leftrightarrow_{\text{def}} P \,;\, \mathbb{1} = P$.*

By reflexivity of $\mathbb{1}$ it suffices to demand $\subseteq$ instead of equality. We furthermore use $\mathscr{H}_E(P) \Leftrightarrow_{\text{def}} \mathscr{H}_L(P) \wedge \mathscr{H}_R(P)$. It follows that for $X \in \{E, L, R\}$ the relations satisfying $\mathscr{H}_X$ form a complete lattice. The next sections define programming constructs that satisfy or preserve these healthiness conditions.

4.3 Expressions

The assignment statement of UTP is the mapping $(\vec{x}{:=}\vec{e}) =_{\text{def}} (\vec{x}'{=}\vec{e})$, 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.

Our new relation modelling the assignment is $(\vec{x}{\leftarrow}\vec{e}) =_{\text{def}} \mathbb{1} \,;\, (\vec{x}{:=}\vec{e}) \,;\, \mathbb{1}$. We assume that each expression $e \in \vec{e}$ is $\preccurlyeq$-continuous, hence also $\preccurlyeq$-isotone. We write $(\vec{x}{\leftarrow}e)$ to assign the same expression e to all variables. The upper closure of the images perspicuously appears in the following lemma which intuitively states that $\top$ models the never terminating program.

Lemma 2. *We have* $(\vec{x}{\leftarrow}\infty) = \top$ *and* $(\vec{x}{\leftarrow}\vec{c}) = (\vec{x}'{=}\vec{c}) = (\vec{x}{:=}\vec{c})$ *for every* $\preccurlyeq$*-maximal* $\vec{c} \in D_{1..n}$*. Moreover,* $(\vec{x}{\leftarrow}\vec{e}) \,;\, (\vec{x}{\leftarrow}f(\vec{x})) = (\vec{x}{\leftarrow}f(\vec{e}))$ *holds.*

Resuming our introductory example we now obtain $(x_1, x_2{\leftarrow}\bot, 2) \,;\, (x_1{\leftarrow}x_2) = (x_1, x_2 \leftarrow 2, 2)$ and furthermore $\top \,;\, (x_1, x_2 \leftarrow 2, 2) = (x_1, x_2, \vec{x}_{3..n} \leftarrow 2, 2, \infty)$. If all expressions $\vec{e}$ are constant we have $\top \,;\, (\vec{x}{\leftarrow}\vec{e}) = (\vec{x}{\leftarrow}\vec{e})$. These properties hold instead of the healthiness condition H1b of designs, and demonstrate that computations in our setting are indeed non-strict.

Let us elaborate the assignment $(\vec{x}{\leftarrow}\vec{e})$ using $\preccurlyeq \,;\, (\vec{x}'{=}\vec{e}) \subseteq (\vec{x}'{=}\vec{e}) \,;\, \preccurlyeq$ which relationally states that the expressions $\vec{e}$ are $\preccurlyeq$-isotone [20]. The assignment then simplifies to $(\vec{x}{\leftarrow}\vec{e}) = (\vec{x}{:=}\vec{e}) \,;\, \mathbb{1}$ since

$$\mathbb{1} \,;\, (\vec{x}'{=}\vec{e}) \,;\, \mathbb{1} \subseteq (\vec{x}'{=}\vec{e}) \,;\, \mathbb{1} \,;\, \mathbb{1} = (\vec{x}'{=}\vec{e}) \,;\, \mathbb{1} \subseteq \mathbb{1} \,;\, (\vec{x}'{=}\vec{e}) \,;\, \mathbb{1} \,.$$

Hence $(\vec{x}{\leftarrow}\vec{e}) = (\vec{x}'{=}\vec{e}) \,;\, \mathbb{1} = \{(\vec{x}, \vec{x}') \mid \exists \vec{y} : \vec{y}{=}\vec{e}(\vec{x}) \wedge \vec{y} \preccurlyeq \vec{x}'\} = \{(\vec{x}, \vec{x}') \mid \vec{e}(\vec{x}) \preccurlyeq \vec{x}'\}$. This means that the successor states of $\vec{x}$ under this assignment comprise the usual successor $\vec{e}(\vec{x})$ and its upper closure with respect to $\preccurlyeq$.

Consider the conditional statement $(P \lhd b \rhd Q) = (b \cap P) \cup (\bar{b} \cap Q)$ of UTP, where the condition b is treated as a vector. In common terms this reads as 'if b then P else Q' but the definition does not take into account the possibility of b being undefined. Its extension to designs $(P \lhd b \rhd Q) = (\mathcal{D}b \Rightarrow (b \cap P) \cup (\bar{b} \cap Q))$ does, but yields non-termination whenever the condition b is undefined. We therefore have to adapt the definition.

To this end, we no longer treat conditions as vectors but as $\preccurlyeq$-continuous expressions with values in *Bool* that may depend on the input $\vec{x}$. Nevertheless, if b is a condition, the relation $b{=}c$ is a vector for each $c \in \mathit{Bool}$. Using $\vec{x}_{1..m}$ as input variables, we obtain 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 new relation modelling the conditional 'if b then P else Q' is

$$(P \blacktriangleleft b \blacktriangleright Q) =_{\text{def}} b{=}\infty \cup (b{=}\bot \cap \vec{x}'{=}\bot) \cup (b{=}\mathit{true} \cap P) \cup (b{=}\mathit{false} \cap Q) \,.$$

The effect of an undefined condition in a conditional statement is to set all variables of the current state undefined. By Lemma 2 we can indeed replace $b{=}\infty \cup (b{=}\bot \cap \vec{x}'{=}\bot)$ with $(b{=}\infty \cap \vec{x}{\leftarrow}\infty) \cup (b{=}\bot \cap \vec{x}{\leftarrow}\bot)$. This models the fact that the evaluation of b is always necessary if the execution of the conditional is. Any non-termination or undefinedness is thus propagated.

As in UTP, the law $(P \lhd b \rhd P) = P$ holds if b is defined, but not in general since an implementation cannot check if both branches of a conditional are equal. The conditional shall have lower precedence than sequential composition.

4.4 Variables

Variables are added to and removed from the current state by UTP's variable declaration $\mathbf{var}\ x_i = (\exists x_i : \mathbb{I})$ and undeclaration $\mathbf{end}\ x_i = (\exists x_i' : \mathbb{I})$. These relations are not homogeneous: The declaration includes x_i' in its range but not x_i in its domain, and the undeclaration the other way round.

Again we have to adapt the statements to respect the healthiness conditions $\mathscr{H}_L$ and $\mathscr{H}_R$. The new relations modelling the simultaneous (un)declaration of the variables $\vec{x}_K$ are $\boldsymbol{var}\ \vec{x}_K =_{\text{def}} (\exists \vec{x}_K : \mathbb{1})$ and $\boldsymbol{end}\ \vec{x}_K =_{\text{def}} (\exists \vec{x}'_K : \mathbb{1})$.

Since $\boldsymbol{var}\ \vec{x}_K = \mathbb{1}\ ;\ (\exists \vec{x}_K : \mathbb{I})$ can be shown, the declaration itself does not impose any restriction on the new variables. This means that accessing a declared but uninitialised variable results in non-termination. A more appropriate statement that yields undefinedness instead can be obtained by using $\boldsymbol{var}\ \vec{x}_K\ ;\ (\vec{x}_K{\leftarrow}\bot)$. Alternatively, the language designer may opt to allow only initialised variable declarations $(\boldsymbol{var}\ \vec{x}_K{\leftarrow}\vec{e}_K) =_{\text{def}} \boldsymbol{var}\ \vec{x}_K\ ;\ (\vec{x}_K{\leftarrow}\vec{e}_K)$. The expressions $\vec{e}_K$ must not refer to the new variables $\vec{x}_K$ in this case.

The alphabet extension is UTP's mechanism to hide local variables from recursive calls. It is given by $P_{+x_i} = (x_i'{=}x_i) \cap \mathbf{end}\ x_i\ ;\ P\ ;\ \mathbf{var}\ x_i$, making explicit the change of P's type. The domain of P is extended by x_i and the range by x_i', and both are equated.

To adapt the alphabet extension to our setting, let $P : D_I \leftrightarrow D_J$ be a (possibly heterogeneous) relation and K such that $I \cap K = J \cap K = \emptyset$. The new alphabet extension of P by the variables $\vec{x}_K$ is $P^{+\vec{x}_K} : D_{I \cup K} \leftrightarrow D_{J \cup K}$ given by

$$P^{+\vec{x}_K} =_{\text{def}} \boldsymbol{end}\ \vec{x}_I\ ;\ \boldsymbol{var}\ \vec{x}_J \cap \boldsymbol{end}\ \vec{x}_K\ ;\ P\ ;\ \boldsymbol{var}\ \vec{x}_K\ .$$

Intuitively, the part $\boldsymbol{end}\ \vec{x}_I\ ;\ \boldsymbol{var}\ \vec{x}_J$ preserves the values of $\vec{x}_K$ and the part $\boldsymbol{end}\ \vec{x}_K\ ;\ P\ ;\ \boldsymbol{var}\ \vec{x}_K$ applies P to $\vec{x}_I$ to obtain $\vec{x}_J$. Just as the variable undeclaration may be seen as a projection, the alphabet extension is an instance of relational parallel composition. This follows since $P^{+\vec{x}_K} = (\mathbb{1}P\mathbb{1})_I\|_K \mathbb{1}$, which simplifies to $P_I\|_K \mathbb{1}$ if $\mathscr{H}_E(P)$ holds. While this resembles [15, Definition 9.1.3], the parallel composition of designs is different as regards termination. It is typically as complex to prove a result for the more general $P\|Q$ as it is for $P^{+\vec{x}_K}$.

4.5 Isotony and Neutrality

We have introduced a selection of programming constructs as summarised in the following definition. This selection subsumes the imperative, non-deterministic

core of UTP and hence is rich enough to yield a basic programming and specification language.

Definition 3. *We use the following relations and operations:*

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

No new definitions are given for sequential composition, the non-deterministic choice and the fixpoint operator. They are just the familiar operations of relation algebra. This simplifies reasoning because it enables applying familiar laws, like distribution of ; over $\cup$, also to programs. We use the *greatest* fixpoint to define the semantics of specifications given by recursive equations, and thus obtain demonic non-determinism. This is consistent with UTP, which uses the term 'weakest fixed point' and the notation μ, but with the reverse order. The specification $P = f(P)$ is resolved as $\nu(\lambda P.f(P))$ which we abbreviate as $\nu P.f(P)$. For example, the iteration $\textbf{\textit{while}}\ b\ \textbf{\textit{do}}\ P$ is just $\nu X.(P \,;\, X \lhd b \rhd \mathbb{1})$.

We conclude our compendium of programming constructs by two useful results. The first states isotony of functions on programs with respect to refinement $\subseteq$, which is important for the existence of fixpoints needed to solve recursive equations. Corresponding to the healthiness conditions H1a and H3 of designs, the second result establishes $\mathbb{1}$ as a left and a right unit of sequential composition, which is useful to terminate iterations and to obtain a one-sided conditional.

Theorem 4. *All functions composed of the constructs of Definition 3 are $\subseteq$-isotone. All relations composed of these constructs satisfy $\mathscr{H}_L$ and $\mathscr{H}_R$.*

Actually, these results hold for more constructs than those of Definition 3, for example, also for the infinite choice $\bigcup$, least fixpoints, arbitrary conjunctions and any constant relations satisfying $\mathscr{H}_L$ and $\mathscr{H}_R$, including assignments and conditionals with isotone expressions. These additional constructs are further investigated in [11] for flat D_i. The theory presented in this section is a proper generalisation of the previous results to arbitrary partial orders containing $\bot$ and a least element ∞. Most results below also apply to further constructs.

5 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 each co-directed

set S. The importance of continuity comes from the permission to represent the greatest fixpoint νf by the constructive $\bigcap_{n\in\mathbb{N}} f^n(\top)$. This enables the approximation of νf by repeatedly unfolding f, which simulates recursive calls of the modelled computation. However, unbounded non-determinism breaks continuity as shown, for example, in [7, Chapter 9] and [4, Section 5.7]. Sources of unbounded non-determinism in our theory are the use of

- unrestricted non-deterministic choice $\bigcup$ and
- finite choice $\cup$ within (recursively constructed) infinite data structures.

Considering Definition 3, we have already banned $\bigcup$ and are about to replace its use by $\bigcap$ for the greatest fixpoint. The remaining source of unbounded non-determinism can be neutralised in either of two ways: by restriction to orders with finite height or to deterministic programs.

Our previous work [11] pursues the first approach by assuming D_i to be flat orders (actually, finite height suffices). Before presenting its main result, we characterise *boundedly non-deterministic* programs, see [7, 13, 35]. Traditionally, this requires that each state $\vec{x}$ has finitely many successor states $P(\vec{x})$, given by the image under the relation P. We adapt this to our context using the pointwise minima with respect to $\preccurlyeq$.

Definition 5. $\mathscr{H}_B(P) \Leftrightarrow_{\text{def}} \forall \vec{x} : |\min P(\vec{x})| \in \mathbb{N}$*, where the minimal elements of* $A \subseteq D_{1..n}$ *are* $\min A =_{\text{def}} \{x \mid x \in A \wedge \forall y : (y \in A \wedge y \preccurlyeq x) \Rightarrow y = x\}$.

This way the condition $\mathscr{H}_B$ accounts for the proper successor states, excluding those that have been added for technical reasons by forming the upper closure. Using $\mathscr{H}_B$ we can show the following statements.

Theorem 6. *Assume that the ranges D_i have finite height.*

1. *Relations composed of the constructs of Definition 3 satisfy $\mathscr{H}_B$.*
2. *Functions composed of the constructs of Definition 3 are co-continuous, that is, they distribute over infima of co-directed sets of relations satisfying $\mathscr{H}_E$ and $\mathscr{H}_B$.*
3. *Relations composed of the constructs of Definition 3 are total.*

The former approach suffices for basic data structures, but excludes functions as values and infinite data structures. However, the problem is not caused by the orders with infinite height, but by having non-determinism at the same time, since this introduces relations with infinitely many proper successor states. Our new proposal therefore is to restrict relations to represent deterministic programs. This is sufficient to show continuity even in the presence of infinite data structures. 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. Similarly to $\mathscr{H}_B$ above, we characterise deterministic computations in our context by the following $\mathscr{H}_D$.

Definition 7. $\mathscr{H}_D(P) \Leftrightarrow_{\text{def}} (\operatorname{lea} P)\top = \top$*, where* $\operatorname{lea} P =_{\text{def}} P \cap \overline{P;\overline{\succcurlyeq}}$ *is the pointwise least elements of P with respect to $\preccurlyeq$. Moreover, 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 $\preccurlyeq$.*

By taking the pointwise least elements, also $\mathscr{H}_D$ accounts for the proper successor states. The condition $\mathscr{H}_C$ is needed to prove part 2 of the following result and generalises $\preccurlyeq$-continuity to relations. If P satisfies $\mathscr{H}_R$ and $\mathscr{H}_D$, the relation $\mathrm{lea}\, P$ is a mapping that is $\preccurlyeq$-continuous iff P satisfies $\mathscr{H}_L$ and $\mathscr{H}_C$.

Theorem 8. *Consider Definition 3 without the choice operator.*

1. *Relations composed of these constructs satisfy $\mathscr{H}_D$ and $\mathscr{H}_C$. In particular, they are total.*
2. *Functions composed of these constructs are co-continuous, that is, they distribute over infima of co-directed sets of relations satisfying $\mathscr{H}_E$ and $\mathscr{H}_D$ and $\mathscr{H}_C$.*

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 4 and 8 are the main results to guarantee that the application of our theory in the next section is meaningful.

6 Infinite Data Structures

Supporting infinite data structures in a theory is nice, but one also needs means to construct and use them in programs. In this section we focus on lists, but our discussion also applies to more general structures such as infinite trees.

To see the difficulties involved, let us start with a simple example, the infinite list $ones = 1 : ones$. We assume that the type of lists of integers has been defined as $IntList = Nil + (Int : IntList)$ with non-strict constructors : and Nil. Our first attempt is a program P with one variable xs whose final value should be the required list:

$$P = (xs{\leftarrow}1{:}xs)\ ;\ P\ .$$

However, its solution $\nu P.(xs{\leftarrow}1{:}xs)\ ;\ P$ equals $\mathbb{T}$ by totality of the assignment. Obviously, non-strict computations do not prohibit programs from running into endless loops. But endless loops have no effect if their results are not needed, so we might instead try

$$P = P\ ;\ (xs{\leftarrow}1{:}xs)\ .$$

And this works indeed, which we can confirm by calculating the greatest fixpoint of $f(P) = P\ ;\ (xs{\leftarrow}1{:}xs)$. Using Theorem 8.2 we obtain $\nu f = \bigcap_{n\in\mathbb{N}} f^n(\mathbb{T})$ where

$$\begin{array}{l} f^0(\mathbb{T}) = \mathbb{T} \\ f^1(\mathbb{T}) = \mathbb{T}\ ;\ (xs{\leftarrow}1{:}xs) = (xs{\leftarrow}\infty)\ ;\ (xs{\leftarrow}1{:}xs) = (xs{\leftarrow}1{:}\infty) \\ f^2(\mathbb{T}) = f(xs{\leftarrow}1{:}\infty) = (xs{\leftarrow}1{:}\infty)\ ;\ (xs{\leftarrow}1{:}xs) = (xs{\leftarrow}1{:}1{:}\infty) \\ f^3(\mathbb{T}) = f(xs{\leftarrow}1{:}1{:}\infty) = (xs{\leftarrow}1{:}1{:}\infty)\ ;\ (xs{\leftarrow}1{:}xs) = (xs{\leftarrow}1{:}1{:}1{:}\infty) \end{array}$$

Lemma 2 is applied to calculate $f^1(\mathbb{T})$. Thus $f^n(\mathbb{T}) = (xs{\leftarrow}(1{:})^n\infty)$ and we have $\nu f = (xs{\leftarrow}ones)$.

Let us try to obtain the infinite list of natural numbers $nats = 0 : 1 : 2 : 3 : \ldots$ next. Our program should have two variables xs and c to hold the result and to count, respectively. Again the obvious first try $P = (xs{\leftarrow}c{:}xs)\ ;\ (c{\leftarrow}c{+}1)\ ;\ P$, assuming the initial value 0 for c, does not work. The above trick to reverse the construction is fruitless in this case, yielding

$$P = P\ ;\ (xs{\leftarrow}c{:}xs)\ ;\ (c{\leftarrow}c{+}1)\ .$$

In fact, this program assigns the infinite list $\infty : \infty : \infty : \ldots$ to xs. For example, if we try to access the first element of xs, the computation does not terminate, because to obtain the final value of c one has to unfold P infinitely. Even if the computation terminated, two further problems would arise: The constructed list would be decreasing (for example, the first element of xs is one larger than the second), and there is no initial value of c where this decreasing sequence could start. This could be avoided by using

$$P = P\ ;\ (c{\leftarrow}c{-}1)\ ;\ (xs{\leftarrow}c{:}xs)\ ,$$

and *somehow* ensuring that the final value of c is 0. Such a procedure we do not pursue, since not every computation can be inverted (like the increment of c by its decrement). The solution is to compute the value of c before the recursive call and to construct the sequence afterwards, as in

$$P = (c{\leftarrow}c{+}1)\ ;\ P\ ;\ (xs{\leftarrow}c{:}xs)\ .$$

We only have to make sure that the value of c is saved across the recursive call, so that it can be prepended to the list. The alphabet extension comes in handy:

$$P = (\boldsymbol{var}\ t{\leftarrow}c)\ ;\ (c{\leftarrow}c{+}1)\ ;\ P^{+t}\ ;\ (xs{\leftarrow}t{:}xs)\ ;\ \boldsymbol{end}\ t\ .$$

Using $f(P) = (\boldsymbol{var}\ t{\leftarrow}c)\ ;\ (c{\leftarrow}c{+}1)\ ;\ P^{+t}\ ;\ (xs{\leftarrow}t{:}xs)\ ;\ \boldsymbol{end}\ t$, we obtain

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

Thus $f^n(\top) = (xs, c{\leftarrow}c : c{+}1 : c{+}2 : \ldots : c{+}n{-}1 : \infty, \infty)$ and we obtain $(c{\leftarrow}0)\ ;\ \nu f = (xs, c{\leftarrow}nats, \infty)$.

The above program to construct $nats$ is motivated by the recursive definition $nats(c) = c : nats(c{+}1)$ of the natural numbers from c, also called *enumFrom*

in Haskell. Its recursion pattern is the well-known symmetric linear recursion, which is sufficiently general to subsume cata-, ana-, hylo- and paramorphisms [19] or folds and unfolds [9] on lists. For example, in functional programming languages the latter are characterised by

$$\mathit{unfold}(p, f, g, x) = \mathbf{if}\ p(x)\ \mathbf{then}\ \mathit{Nil}\ \mathbf{else}\ f(x) : \mathit{unfold}(p, f, g, g(x))\ ,$$

where the parameter p represents the terminating condition, f constructs the values of the list and g modifies the seed x. Note that p, f and g are constant parameters. We may realise *unfold* by the program

$$P = (xs{\leftarrow}\mathit{Nil} \blacktriangleleft p(x) \blacktriangleright \textbf{\textit{var}}\ t{\leftarrow}f(x)\ ;\ x{\leftarrow}g(x)\ ;\ P^{+t}\ ;\ xs{\leftarrow}t{:}xs\ ;\ \textbf{\textit{end}}\ t)\ .$$

Instantiating $p(x) = \mathit{false}$, $f(x) = x$ and $g(x) = x{+}1$ we obtain the program for *nats*. Also *ones* may be recovered by $p(x) = \mathit{false}$ and $f(x) = g(x) = 1$. In such instances, where termination is not available or not guaranteed, our program P is more general than in strict UTP. Moreover, it is not necessary to compute the result entirely, but only to the required precision.

Let us now consider several further examples, starting with the list-consuming counterpart

$$\mathit{foldr}(f, z, xs) = \mathbf{if}\ \mathit{isNil}(xs)\ \mathbf{then}\ z\ \mathbf{else}\ f(\mathit{head}(xs), \mathit{foldr}(f, z, \mathit{tail}(xs)))\ .$$

We may realise *foldr* by the program

$$\begin{aligned} P = (r{\leftarrow}z \blacktriangleleft \mathit{isNil}(xs) \blacktriangleright \textbf{\textit{var}}\ t{\leftarrow}\mathit{head}(xs)\ ;\ xs{\leftarrow}\mathit{tail}(xs)\ ;\ P^{+t}\ ;\\ r{\leftarrow}f(t, r)\ ;\ \textbf{\textit{end}}\ t) \end{aligned}$$

that is able to process finite and infinite xs, provided f is non-strict. The dual *foldl* immediately returns from its recursive calls and therefore does not work on infinite lists in general, but *scanl* does. Instantiating *foldr* with $f(t, r) = g(t) : r$ and $z = \mathit{Nil}$ we obtain a program to compute $\mathit{map}(g, xs)$, leaving the result in r. Instantiating *foldr* with $f(t, r) = \mathbf{if}\ p(t)\ \mathbf{then}\ t : r\ \mathbf{else}\ r$ and $z = \mathit{Nil}$ we obtain $\mathit{filter}(p, xs)$. This shows that we can program using list comprehensions, even on infinite lists. For example, $[\ f(x) \mid x \leftarrow xs,\ p(x)\ ]$ is obtained by

$$\begin{aligned} P = (ys{\leftarrow}\mathit{Nil} \blacktriangleleft \mathit{isNil}(xs) \blacktriangleright \textbf{\textit{var}}\ t{\leftarrow}\mathit{head}(xs)\ ;\ xs{\leftarrow}\mathit{tail}(xs)\ ;\ P^{+t}\ ;\\ (ys{\leftarrow}f(t){:}ys \blacktriangleleft p(t) \blacktriangleright \mathbb{1})\ ;\ \textbf{\textit{end}}\ t)\ . \end{aligned}$$

It consumes the input list xs and produces the output list ys. We could also call the result xs, but generally its type differs from that of xs, hence P is a heterogeneous relation. Note that only the value of the variable xs is updated during the recursion, but there is no destructive update to the original list that is persistent and could be referenced by another variable.

As our final example, here is the 'unfaithful' prime number sieve [24], entirely in terms of the constructs of Section 4:

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

This may seem verbose compared to its Haskell equivalent, but it uses neither parameters and pattern matching, nor concise notations such as $[\,2..\,]$ for *from2* and $[\ t \mid t \leftarrow xs,\ p \nmid t\ ]$ for *remove* available in Haskell. Such concepts shall be added to our language in the future. Our program can be executed in such a way that only so many prime numbers are computed as actually required. But also with finite data structures a lazy execution may be advantageous. For example, we have devised versions of mergesort and heapsort in our framework which, for lists of length n, perform at most $O(n \log n)$ comparisons, but fewer if only the initial elements of the sorted sequence are required.

7 Dependence

Undefined and defined variables may coexist according to our relational theory of computations. In this section we discuss two aspects of non-strictness that can be described in terms of dependence of variables. We first illustrate the issue for the case $m = n = 1$, that is, a single input and output variable, and then present the resulting, additional healthiness conditions.

Consider a relation R with an $x'_1 \neq \bot$ such that $(\bot, x'_1) \in R$, thus R produces a defined output for an undefined input. If x'_1 is to be computed by a program, its value must not depend on the value of x_1 or else the input $x_1 = \bot$ would result in the output $x'_1 = \bot$. In other words, there must be a constant assignment to x'_1. We therefore obtain the condition $(x_1, x'_1) \in R$ for all x_1. Note that we do not conclude that R equals this constant assignment, since in general R may be composed by non-deterministic choice from the constant assignment and some non-constant computation.

Now consider a relation R with $(\bot, \bot) \notin R$, thus R does not produce an undefined output for an undefined input. Then indeed there cannot be non-constant computations and the value of x'_1 must not depend on the value of the input x_1 at all. Hence we must ensure that *only* constant assignments occur. This is achieved by requiring $(x_1, x'_1) \in R$ for all x_1, if $(x_1, x'_1) \in R$ for some x_1. Note that choosing $x_1 = \bot$ yields a special case of the first condition, while $x'_1 = \bot$ is prevented since it implies $(\bot, \bot) \in R$.

Both conditions can be generalised to arbitrary m and n, but the resulting formulae are very unwieldy. Fortunately, they have an elegant counterpart in order-theoretic terms, derived in [11] for flat orders, which we use directly. To this end, we introduce an order similar to $\preccurlyeq$, but now with respect to $\bot$. However, we have to restrict our data types by disallowing the use of functions as values.

The partial order $\sqsubseteq : D_i \leftrightarrow D_i$ with least element $\bot$ is constructed as follows. Elementary types are flat, that is, $x \sqsubseteq y \Leftrightarrow_{\text{def}} x = \bot \vee x = y$. The finite union of D_i is ordered by $x \sqsubseteq y \Leftrightarrow_{\text{def}} x = \bot \vee x = \infty = y \vee (x = (i, x_i) \wedge y = (i, y_i) \wedge x_i \sqsubseteq_{D_i} y_i)$. The finite product of types D_i is ordered by the pointwise extension of $\sqsubseteq$, that is, $\vec{x}_I \sqsubseteq \vec{y}_I \Leftrightarrow_{\text{def}} \forall i \in I : x_i \sqsubseteq_{D_i} y_i$. The constituents of the inverse limit construction for recursive data types are ordered pointwise. Using the new order, we obtain an algebraic characterisation of the healthiness conditions, where $\sqsupseteq = _{\text{def}} \sqsubseteq^{\smile}$ denotes the dual order of $\sqsubseteq$.

Definition 9. $\mathscr{H}_N(R) \Leftrightarrow_{\text{def}} \sqsupseteq ; R \subseteq R ; \sqsupseteq$ *and* $\mathscr{H}_A(R) \Leftrightarrow_{\text{def}} \sqsubseteq ; R \subseteq R ; \sqsubseteq$.

If R is a mapping, the condition $\mathscr{H}_N(R)$ is equivalent to $\mathscr{H}_A(R)$ and states that R is isotone with respect to $\sqsupseteq$. Actually, a relation R satisfying $ER \subseteq RE$ and $E^\smile R \subseteq RE^\smile$ for a partial order E is also called an 'isotone relation' [38] and an 'order preserving multifunction' [34]. These works investigate the 'relational fixed point property' [33], a property of the order E rather than of functions over relations.

Remark. The new healthiness conditions are related to the Egli-Milner order on powerdomains built from flat domains [27, 30]. Indeed, one can interpret the conjunction of $\mathscr{H}_N$ and $\mathscr{H}_A$ as imposing the Egli-Milner order on the image sets of relations. This order is frequently used in semantics to define the least fixpoint of functions. Let us therefore emphasise that $\sqsubseteq$ serves to support our reasoning about undefinedness, that is, finite failure. It is not used to approximate fixpoints, which we do by the subset order $\subseteq$ that (with closure under $\preccurlyeq$) corresponds to an order based on wp. In [23] two orders based on wp and wlp are combined for approximation. In fact the Egli-Milner order models erratic non-determinism or general correctness, but UTP's and our definitions model demonic non-determinism or total correctness. The difference is expounded in [23, 35] in more detail. A general correctness variant of UTP is explored in [8].

We can show that our programming constructs satisfy $\mathscr{H}_N$ and $\mathscr{H}_A$. To deal with the assignment and the conditional, we assume that the expressions are $\sqsubseteq$-isotone in addition to being $\preccurlyeq$-continuous.

Theorem 10. *Relations composed of the constructs of Definition 3 without the choice operator satisfy* $\mathscr{H}_N$ *and* $\mathscr{H}_A$.

The conditions $\mathscr{H}_N$ and $\mathscr{H}_A$ can also be seen as expressing an information preservation principle. In this interpretation $\sqsubseteq$ is the definedness information order and $\mathscr{H}_N$ and $\mathscr{H}_A$ convey definedness information. Corresponding healthiness conditions for the termination information order $\preccurlyeq$ are discussed in [11] and can also be generalised to the present setting of more general orders.

8 Conclusion

We have proposed a new relational approach to define the semantics of imperative programs. Let us summarise its key properties and its extensions to UTP.

- Undefinedness and non-termination are treated independently of each other. Finite and infinite failure can thus be distinguished, which is closer to practice and allows one to model recovery from errors. A fine distinction is offered by dealing with undefinedness separately for individual variables.
- The theory provides a relational model of dependence in computations. Additional healthiness conditions are stated in a compact algebraic form and can therefore be applied easily to new programs given as relations.

– The relations model non-strict computations in an imperative context. Efficiency can thus be improved by executing only those parts of programs necessary to obtain the final results. Programs can construct and process infinite data structures. The theory can serve as a basis to link to the semantics of functional programming languages.

The disadvantages of a possibly lazy evaluation are of course a potential overhead and reduced predictability of execution time, space and order.

We thus obtain a theory similar to that of designs but modelling non-strict computations. In particular, the left and right unit laws $\mathscr{H}_L$ and $\mathscr{H}_R$ and the totality property correspond to the healthiness conditions H1–H4 of designs without the left zero law $\mathbb{T} \mathbin{;} R = \mathbb{T}$. For elementary, sum and product types, all functions composed of programming constructs are continuous and all relations composed of programming constructs are boundedly non-deterministic. With infinite data types, continuity holds for the functions composed of deterministic programming constructs. Additionally, the relations satisfy the healthiness conditions $\mathscr{H}_N$ and $\mathscr{H}_A$ modelling the dependence of variables.

Our programming constructs introduced in Definition 3 are sufficiently similar to the original constructs of UTP to show that they yield the same results whenever the computations are defined and terminate. This correspondence is formally stated in [11] for elementary data types, but can be extended to the present, more general case. As another measure to ensure the adequacy of our framework, an operational semantics is outlined to describe the execution of programs. Future work shall extend the operational semantics to cover infinite data structures.

These observations also show the advantage of the UTP approach: We are able to compare different theories describing the semantics of programs within the same framework. Their similarities and differences are particularly apparent in the effective healthiness conditions. Such characterising properties are expressed concisely due to the fact that UTP is based on relations.

Connections to related work have been pointed out throughout this paper. In [12] we compare our work with further relational and functional approaches, including the Z notation [16, 39], Haskell's I/O monad [17, 25] and state transformers [18], and the multi-paradigm language Oz [29]. This is extended by the following notes.

Relations satisfying $\mathscr{H}_E$ are called 'ideal relations' by [20] and used to model higher order programming. The investigation aims at defining the semantics by predicate transformers rather than relations [21]. Accordingly, there is no special value to treat non-termination, which is not distinguished from undefinedness. Elementary data types have a discrete order. In [22], ideal relations are also used as 'couplings' to connect state spaces for data refinement.

Let us finally point out two topics that deserve further investigation. One of them is to explore our relational model as an intermediate for the translation of functional programming languages. The other is concerned with the connections to data flow networks [15, Section 8.3] and, in particular, to the algebra of stream processing functions [5].

Acknowledgement. I am grateful to the anonymous referees for their helpful remarks, fair criticism and interesting questions.

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. 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.
3. 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.
4. 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 *LNCS*, pages 553–592. Springer-Verlag, 1979.
5. M. Broy and G. Ştefănescu. The algebra of stream processing functions. *Theoretical Computer Science*, 258(1–2):99–129, May 2001.
6. B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, second edition, 2002.
7. E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.
8. S. Dunne. Recasting Hoare and He's Unifying Theory of Programs in the context of general correctness. In A. Butterfield, G. Strong, and C. Pahl, editors, *5th Irish Workshop on Formal Methods*, Electronic Workshops in Computing. The British Computer Society, July 2001.
9. 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.
10. W. Guttmann. Non-termination in Unifying Theories of Programming. In W. MacCaull, M. Winter, and I. Düntsch, editors, *Relational Methods in Computer Science 2005*, volume 3929 of *LNCS*, pages 108–120. Springer-Verlag, 2006.
11. W. Guttmann. *Algebraic Foundations of the Unifying Theories of Programming*. PhD thesis, Universität Ulm, December 2007.
12. W. Guttmann. Lazy relations. In R. Berghammer, B. Möller, and G. Struth, editors, *Relations and Kleene Algebra in Computer Science*, volume 4988 of *LNCS*, pages 138–154. Springer-Verlag, 2008.
13. W. H. Hesselink. *Programs, Recursion and Unbounded Choice*. Cambridge University Press, 1992.
14. C. A. R. Hoare. Theories of programming: Top-down and bottom-up and meeting in the middle. In J. M. Wing, J. Woodcock, and J. Davies, editors, *FM'99: Formal Methods*, volume 1708 of *LNCS*, pages 1–27. Springer-Verlag, 1999.
15. C. A. R. Hoare and J. He. *Unifying theories of programming*. Prentice Hall Europe, 1998.
16. ISO/IEC. Information technology: Z formal specification notation: Syntax, type system and semantics. ISO/IEC 13568:2002(E), July 2002.
17. J. Launchbury. Lazy imperative programming. In P. Hudak, editor, *Proceedings of the ACM SIGPLAN Workshop on State in Programming Languages*, Yale University Research Report YALEU/DCS/RR-968, pages 46–56, June 1993.

18. J. Launchbury and S. Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, 8(4):293–341, December 1995.
19. 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 *LNCS*, pages 124–144. Springer-Verlag, 1991.
20. D. A. Naumann. A categorical model for higher order imperative programming. *Mathematical Structures in Computer Science*, 8(4):351–399, August 1998.
21. D. A. Naumann. Predicate transformer semantics of a higher-order imperative language with record subtyping. *Science of Computer Programming*, 41(1):1–51, September 2001.
22. D. A. Naumann. Soundness of data refinement for a higher-order imperative language. *Theoretical Computer Science*, 278(1–2):271–301, May 2002.
23. G. Nelson. A generalization of Dijkstra's calculus. *ACM Transactions on Programming Languages and Systems*, 11(4):517–561, October 1989.
24. M. E. O'Neill. The genuine sieve of Eratosthenes. *Journal of Functional Programming*, 19(1):95–106, January 2009.
25. S. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.
26. R. Plasmeijer and M. van Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley, 1993.
27. G. D. Plotkin. A powerdomain construction. *SIAM Journal on Computing*, 5(3):452–487, September 1976.
28. W.-P. de Roever. *Recursive program schemes: semantics and proof theory*. Number 70 in Mathematical Centre Tracts. Mathematisch Centrum, Amsterdam, 1976.
29. P. Van Roy and S. Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, 2004.
30. D. A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. William C. Brown Publishers, 1986.
31. 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.
32. G. Schmidt and T. Ströhlein. *Relationen und Graphen*. Springer-Verlag, 1989.
33. B. S. W. Schröder. *Ordered Sets: An Introduction*. Birkhäuser, 2003.
34. R. E. Smithson. Fixed points of order preserving multifunctions. *Proceedings of the American Mathematical Society*, 28(1):304–310, April 1971.
35. H. Søndergaard and P. Sestoft. Non-determinism in functional languages. *The Computer Journal*, 35(5):514–523, October 1992.
36. J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.
37. D. A. Turner. Miranda: A non-strict functional language with polymorphic types. In J.-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume 201 of *LNCS*, pages 1–16. Springer-Verlag, 1985.
38. J. W. Walker. Isotone relations and the fixed point property for posets. *Discrete Mathematics*, 48(2–3):275–288, February 1984.
39. J. Woodcock and J. Davies. *Using Z*. Prentice Hall, 1996.