Lazy Relations

Walter Guttmann

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

Abstract. We present a relational model of non-strict computations in an imperative, non-deterministic context. Undefinedness is represented independently of non-termination. The relations satisfy algebraic properties known from other approaches to model imperative programs; we introduce additional laws 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. Local variables are treated by relational parallel composition.

1 Introduction

Our goal is to develop a relational model of non-strict computations in an imperative, non-deterministic context. As a simple motivation of the issues we are about to address, 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 a conventional language its execution aborts, 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 [17]. Yet also in an imperative language it can be reasonable to require that $P \,;\, x_1{:=}x_2 = x_1, x_2{:=}2, 2$ holds since the value of x_1 after the execution of P is never used. To see this, consider 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)`. Integrating non-determinism additionally is useful for program specification and development.

Let us describe our new approach which has these qualities. 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 presents a compendium of relations modelling a selection of programming constructs. We identify several algebraic properties they satisfy, starting with isotony and the left and right unit laws. In Section 3.2 we derive further properties, namely finite branching, continuity and totality. We thus obtain a theory similar to that of existing approaches, 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.

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 4. Starting from the observation that non-strict computations with defined results cannot depend on undefined inputs, we derive two additional laws. 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 a new, relational model of imperative, non-deterministic, non-strict computations and a relational description of dependence in such computations.

This paper is a condensed account of a part of the author's PhD thesis [8]. We present the key definitions and results, but omit their proofs. The work grew out of research on Hoare and He's Unifying Theories of Programming [11], however it can be discussed independently and without prior knowledge of that context. The original motivation and many connections to the Unifying Theories of Programming are included in [8].

2 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.

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. The relative complement of a subset $I \subseteq 1..m$ is denoted by $\overline{I} =_{\text{def}} 1..m \setminus I$, where m will be clear from the context. We abbreviate $\overline{\{i\}}$ as $\overline{i}$. Let $\vec{x}_I$ denote the subsequence of $\vec{x}$ comprising those x_i with $i \in I$. By writing $a{\in}\vec{x}$ or $\vec{x}{=}a$ we express that $a{=}x_i$ for some or all $i \in 1..m$, respectively. Let $D_I =_{\text{def}} \prod_{i \in I} D_i$ 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 $m \neq n$.

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 [22, 23]. We omit any notational distinction of the types of relations and their operations and assume type-correctness in their use.

We denote the zero, identity and universal relations by $\perp\!\!\!\perp$, $\mathbb{I}$ and $\top\!\!\!\top$, 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 $R\top\!\!\!\top = R$, total iff $R\top\!\!\!\top = \top\!\!\!\top$ and univalent iff $R^\smile R \subseteq \mathbb{I}$. A relation is a mapping iff it is both total and univalent.

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 any constant $\vec{c}$. Generally used to construct relational constants, infix operators without spacing have higher precedence than converse.

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 $(\exists \vec{x}'_L : R)$.

Defined in terms of the projection, we furthermore use the following relational parallel composition operator, similar to that of [1, 3]. 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 $J \cup L$). The $\|$ operator has lower precedence than meet and join.

The scope of quantifiers in a formula extends as far to the right as possible, that is, until the next unmatched closing bracket or the end of the formula. Logical quantification over the empty sequence of variables can be omitted, that is, $(\exists \vec{x}_\emptyset : A) = (\forall \vec{x}_\emptyset : A) = A$.

3 Programming Constructs

We present a relational model of non-strict computations. In particular, we give new definitions for a number of programming constructs and identify several algebraic properties they satisfy. The latter starts with isotony and the unit laws in Section 3.1, followed by boundedness, continuity and totality in Section 3.2 and two dependence conditions in Section 4.

Basic statements comprise the assignment, skip, (un)declaration of variables and alphabet extension. Control flow is provided by the conditional, sequential and parallel composition. Relations may furthermore be composed by the non-deterministic choice. Its dual, conjunction, is technically useful for the treatment of recursion which is given by the greatest fixpoint. We moreover consider its dual, the least fixpoint. This selection of programming constructs subsumes the imperative, non-deterministic core of the Unifying Theories of Programming [11].

3.1 Isotony and Neutrality

We successively define our programming constructs using relations and discuss essential algebraic properties. At first we introduce a fundamental order on the variable ranges, which is used throughout this paper.

Recall that the range D_i of a variable contains the special elements $\bot$ and ∞ modelling undefinedness and non-termination, respectively. Let $\preccurlyeq : D_i \leftrightarrow D_i$ be the flat order on D_i with ∞ as its least element, that is, $x \preccurlyeq y \Leftrightarrow_{\text{def}} x{=}\infty \vee x{=}y$. It follows that $\preccurlyeq$ is a partial order and even a meet-semilattice. A similar order, in which $\bot$ is the least element, will be introduced in Section 4.1.

Recall further that $D_I = \prod_{i \in I} D_i$. Let $\preccurlyeq : D_I \leftrightarrow D_I$ also denote the pointwise extension of that order, that is, $\vec{x}_I \preccurlyeq \vec{y}_I \Leftrightarrow_{\text{def}} \forall i \in I : x_i \preccurlyeq y_i$. Its dual order is denoted by $\succcurlyeq =_{\text{def}} \preccurlyeq^\smile$. The meet operation is obtained by pointwise extension, too. We exclusively work with finite I, indexing the variables of the current state. It is easily proved by induction on the size of the index set I, that $|C| \leq |I| + 1$ for any chain C in D_I ordered by $\preccurlyeq$. It follows that the corresponding strict order $\prec$ is regressively bounded and therefore also well-founded.

Most of the time we use the partial order $\preccurlyeq$ with the index set $I = 1..m$ of all variables, as in $\vec{x} \preccurlyeq \vec{x}'$. Indeed, we take this as the definition of the new relation modelling skip, denoted also by $\mathbb{1} =_{\text{def}} \preccurlyeq$. The intention underlying the definition of $\mathbb{1}$ is to enforce an upper closure of the image of each state with respect to $\preccurlyeq$. Traces of such a procedure can be found in the healthiness condition H2 of [11] and in the $\perp$-predicates of [7]. Our definition of $\mathbb{1}$ refines this by distinguishing individual variables. As usual, skip should be a left and 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 rest of this section is devoted to giving definitions of programming constructs that satisfy or preserve these laws.

The assignment statement is usually defined as 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 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 $\top\!\!\top$ models the never terminating program.

Lemma 2. *We have $\vec{x}{\leftarrow}\infty = \top\!\!\top$ and $\vec{x}{\leftarrow}\vec{c} = \vec{x}'{=}\vec{c} = \vec{x}{:=}\vec{c}$ for any $\vec{c} \in D_{1..n}$ such that $\infty \notin \vec{c}$.*

Resuming our introductory example we now obtain $x_1, x_2 {\leftarrow} \perp, 2 ; x_1 {\leftarrow} x_2 = x_1, x_2 \leftarrow 2, 2$ and furthermore $\top\!\!\top \; ; \; x_1, x_2 \leftarrow 2, 2 = x_1, x_2, \vec{x}_{3..n} \leftarrow 2, 2, \infty$. This demonstrates that computations in our setting are indeed non-strict.

To deal with the conditional and later also with the assignment, we need to restrict the expressions that occur on the right hand side of assignments and as conditions. We assume that the expressions are isotone with respect to $\preccurlyeq$ as captured by the following condition.

Definition 3. *Let E be a partial order. The sequence of expressions $\vec{e}$ is* isotone *with respect to E iff $\mathscr{T}_E(\vec{e}) \Leftrightarrow_{\text{def}} E ; \vec{x}'{=}\vec{e} \subseteq \vec{x}'{=}\vec{e} ; E$.*

At this stage we need $\mathscr{T}_{\preccurlyeq}(\vec{e})$, that is, $\preccurlyeq ; \vec{x}'{=}\vec{e} \subseteq \vec{x}'{=}\vec{e} ; \preccurlyeq$. If the expression e is viewed as a function, then $\mathscr{T}_{\preccurlyeq}(e)$ amounts to the usual isotony in partially ordered sets, namely $\forall \vec{x}, \vec{y} : \vec{x} \preccurlyeq \vec{y} \Rightarrow e(\vec{x}) \preccurlyeq e(\vec{y})$. Its relational formulation appears, for example, in [21]. It can be shown that any expression composed of constants, variables and strict functions is isotone, thus the restriction is not too severe.

Let us elaborate the assignment $\vec{x}{\leftarrow}\vec{e}$ assuming $\mathscr{T}_{\preccurlyeq}(\vec{e})$. It 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 $\preccurlyeq$.

We treat conditions as expressions with values in $\{\infty, \bot, \mathit{true}, \mathit{false}\}$ that may depend on the input $\vec{x}$. If b is a condition, the relation $b{=}c$ is a vector for any $c \in \{\infty, \bot, \mathit{true}, \mathit{false}\}$. Recalling how relational constants are specified, and 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

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

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.

Variables are added to and removed from the current state by the projection operators. We adapt them to respect $\mathscr{H}_E$; our relations modelling variable (un)declaration are $\boldsymbol{var}\ \vec{x}_K =_{\mathrm{def}} (\exists \vec{x}_K : \mathbb{1})$ and $\boldsymbol{end}\ \vec{x}_K =_{\mathrm{def}} (\exists \vec{x}'_K : \mathbb{1})$. At this place, inhomogeneous relations enter the stage. The basic declaration can be augmented to provide initialised variable declarations.

To hide local variables from recursive calls [11] uses the *alphabet extension*. We generalise it to handle several variables and heterogeneous relations. Let $P : D_I \leftrightarrow D_J$, then our alphabet extension is $P^{+\vec{x}_K} : D_{I \cup K} \leftrightarrow D_{J \cup K}$ given by

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

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 $P^{+\vec{x}_K} = \mathbb{1} P \mathbb{1} \|_K \mathbb{1}$, which simplifies to $P \|_K \mathbb{1}$ if $\mathscr{H}_E(P)$ holds. It is typically as complex to prove a result for the more general $P \| Q$ as it is for $P^{+\vec{x}_K}$; we therefore use the former.

We have now introduced a selection of programming constructs as summarised in the following definition. This selection is inspired by [11] and rich enough to yield a basic programming and specification language.

Definition 4. *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	$(P \blacktriangleleft b \blacktriangleright Q) =_{\mathrm{def}} b{=}\infty \cup (b{=}\bot \cap \vec{x}'{=}\bot) \cup (b{=}\mathit{true} \cap P) \cup (b{=}\mathit{false} \cap Q)$
non-deterministic choice	$\bigcup_{P \in S} P$
conjunction	$\bigcap_{P \in S} P$
greatest fixpoint	$\nu f =_{\mathrm{def}} \bigcup \{P \mid f(P) = P\}$
least fixpoint	$\mu f =_{\mathrm{def}} \bigcap \{P \mid f(P) = P\}$

Composition, choice and fixpoint 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. For example, the iteration ***while*** b ***do*** P is just $\nu(\lambda X.P\,;\, X \blacktriangleleft b \blacktriangleright \mathbb{1})$.

We conclude our compendium of programming constructs by two useful results. The first states isotony, which is important for the existence of fixpoints needed to solve recursive equations. The second establishes $\mathbb{1}$ as a left and right unit of sequential composition, which is useful to terminate iterations and to obtain a one-sided conditional. Necessary restrictions of the theorems in this paper are summarised in Table 1 in Section 5.

Theorem 5. *Functions composed of the constructs of Definition 4 with the restrictions stated in Table 1 are isotone.*

Theorem 6. *Relations composed of the constructs of Definition 4 with the restrictions stated in Table 1 satisfy $\mathscr{H}_R$ and $\mathscr{H}_L$. The latter requires $\mathscr{T}_{\preccurlyeq}(b)$ for all conditions b.*

3.2 Finite Branching

From the computational perspective, it is necessary to regard the greatest fixpoint not as the supremum of all fixpoints but as the infimum of a certain chain. Not all properties, however, are preserved by infima of chains. It occasionally helps to restrict the attention to infima of chains of relations that model a finite degree of non-determinism. Such relations represent what are sometimes called *boundedly non-deterministic* programs, see [6, 10, 27]. In graph theory, taking states as nodes and transitions as edges, one speaks of a finite outdegree. As elaborated below, the pure condition of finite branching is not appropriate. We therefore provide a new, relaxed condition. Finite branching is necessary to show the continuity of functions and the totality of relations, which we do afterwards.

To prepare our definition of finite branching, we have to discuss minimal elements of the set $D_{1..n}$ ordered by $\preccurlyeq$. Since many results also hold in more general orders, we abstract to a set S partially ordered by $\preceq$. The *minimal elements* of $A \subseteq S$ are $\min A =_{\text{def}} \{x \mid x \in A \wedge \forall y : (y \in A \wedge y \preceq x) \Rightarrow y = x\}$. We call S *well-founded* iff $\min A \neq \emptyset$ for all $\emptyset \neq A \subseteq S$. The *upper closure* of $A \subseteq S$ is $\uparrow A =_{\text{def}} \{y \mid y \in S \wedge \exists x \in A : x \preceq y\}$ and A is an *upper set* iff $A = \uparrow A$.

These concepts are connected to computations by applying them to the image set of each state with $\preccurlyeq$ as the partial order. We have already observed that $D_{1..n}$ is well-founded and the following lemma establishes these images as upper sets provided the computation satisfies $\mathscr{H}_R$.

Lemma 7. *If S is well-founded and $A \subseteq S$ is an upper set, then $A = \uparrow \min A$. Furthermore, $\mathscr{H}_R(P)$ holds for a relation P iff $P(\vec{x})$ is an upper set for all $\vec{x}$.*

This provides the link between the relation-algebraic viewpoint of $\mathscr{H}_R$ and the pointwise upper sets. One can represent and calculate with minima as relations, see [23] and Section 4.3, but the proof of Lemma 7 remains essentially pointwise.

We are ready to state the condition for boundedly non-deterministic computations. Traditional finite branching cannot be used since we need $\mathbb{T}$ to represent the never terminating program. This is due to the demonic interpretation of non-deterministic choice. The condition that each state $\vec{x}$ has only a finite number of successor states can be relaxed by allowing additionally the case that every state in $D_{1..n}$ is a successor of $\vec{x}$ [10]. This solves the problem with $\mathbb{T}$, that satisfies the relaxed condition, but is not fine enough for our purposes. We further need to distinguish the individual variables, which is done by the condition $\mathscr{H}_B$ using the pointwise minima with respect to $\preccurlyeq$.

Definition 8. $\mathscr{H}_B(P) \Leftrightarrow_{\text{def}} \forall \vec{x} : |\min P(\vec{x})| \in \mathbb{N}$.

The intention of using min is the following: $\mathscr{H}_B$ will be applied to relations that satisfy $\mathscr{H}_R$. By Lemma 7 the image sets of such relations are in a one-to-one correspondence with their minimal elements. Indeed, it is the minimal elements that actually represent the successor states, and their upper closure is formed to satisfy $\mathscr{H}_R$ and to avoid unboundedness. Thus $\mathscr{H}_B$ accounts for the proper successor states, excluding those that have been added for technical reasons. We can show that many relations from our compendium satisfy $\mathscr{H}_B$.

Theorem 9. *Relations composed of the constructs of Definition 4 with the restrictions stated in Table 1 satisfy $\mathscr{H}_B$. In particular, $\mathscr{T}_{\preccurlyeq}(\vec{e})$ is required for all expressions $\vec{e}$.*

The proof uses the fact that $D_{1..n}$ ordered by $\preccurlyeq$ is a meet-semilattice having finite height. Finite height (which implies well-foundedness) is guaranteed since there are only a finite number of variables and the ranges D_i are flat orders. The latter suffices for data structures with strict constructors, but excludes infinite data structures which are modelled by non-flat orders. However, the problem is not caused by the infinite data structures themselves, but by having non-determinism at the same time. A more general investigation using powerdomains with finitely generable elements [18, 20, 26] is postponed to future work.

We call a function f *continuous* iff it distributes over infima of non-empty chains of relations, formally $f(\bigcap C) = \bigcap_{P \in C} f(P)$ for each chain $C \neq \emptyset$. The importance of continuity comes from the permission to represent the greatest fixpoint νf by the constructive $\bigcap_{n \in \mathbb{N}} f^n(\mathbb{T})$. This enables the approximation of νf by repeatedly unfolding f, which simulates recursive calls of the modelled computation. That infinite branching or unbounded non-determinism breaks continuity is shown, for example, in [6, Chapter 9] and [5, Section 5.7]. We use the finite branching property $\mathscr{H}_B$ to establish the continuity of functions composed in our framework.

Theorem 10. *Functions composed of the constructs of Definition 4 with the restrictions stated in Table 1 are continuous, that is, distribute over infima of non-empty chains of relations satisfying $\mathscr{H}_E$ and $\mathscr{H}_B$.*

The proof uses the following two distribution results.

1. Let C be a non-empty chain such that $\mathscr{H}_R(P)$ and $\mathscr{H}_B(P)$ for all $P \in C$. Then $(\bigcap_{P\in C} PQ) = (\bigcap C)Q$.
2. Let C be a non-empty chain such that $\mathscr{H}_L(Q)$ for all $Q \in C$, and let P be such that $\mathscr{H}_R(P)$ and $\mathscr{H}_B(P)$. Then $(\bigcap_{Q\in C} PQ) = P(\bigcap C)$.

Besides finite branching, another reasonable condition for computation purposes is totality, or non-empty branching. Consider the usual interpretation of relations as programs and specifications. Then $\subseteq$ models refinement: $P \subseteq Q$ states that the program P implements the specification Q, because any observation of the execution of P is admitted by Q. But since the empty relation $\perp\!\!\!\perp$ is the least element with respect to $\subseteq$, it implements *any* specification. More generally, the refinement interpretation of P fails if some state has no successors under P. This is prevented by requiring totality of relations.

Theorem 11. *Let* $\mathscr{H}_T(P) \Leftrightarrow_{\text{def}} P \,;\, \top\!\!\top = \top\!\!\top$. *Relations composed of the constructs of Definition 4 with the restrictions stated in Table 1 satisfy* $\mathscr{H}_T$.

4 Dependence

We now have a relational theory of computations where undefined and defined variables coexist. In this section we discuss two aspects of non-strictness that can be described in terms of dependence of variables. The first gives conditions in case the computation has non-strict parts, and the second gives conditions if it has no strict parts. Let us illustrate the distinction in the case $m = n = 1$, that is, a single input and output variable.

The relation R has a non-strict part if there is an $x_1' \neq \bot$ such that $(\bot, x_1') \in R$. For this part, the value of x_1' 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. This essentially reflects that one cannot test for undefinedness: If the value of a variable is undefined, such a test is undefined, too.

The relation R has no strict part if $(\bot, \bot) \notin R$. Then the value of x_1' must not depend on the value of x_1 for any part. Hence the above condition is not sufficient because we must assure 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 $(\bot, \bot) \in R$.

In the following two sections, each of these conditions is generalised to arbitrary m and n, then expressed relationally and in order-theoretic terms, and finally applied to our programming constructs.

For a sequence $\vec{x}$ of length n let $\vec{x}_{i\mapsto a}$ denote $x_1, \ldots, x_{i-1}, a, x_{i+1}, \ldots, x_n$, that is, the replacement of x_i by a. If $I \subseteq 1..n$, let $\vec{x}_{I\mapsto a}$ denote the replacement of x_i by a in $\vec{x}$ for all $i \in I$.

4.1 Non-strict Parts

We first deal with the non-strict parts of a relation. Let us formalise the case $m = n = 1$. As stated above, a non-strict part of the relation R is given by an outcome $x_1' \neq \bot$ for $x_1 = \bot$. Then x_1' must be an outcome for all x_1. We thus have

$$\forall x_1' : (x_1' \neq \bot \wedge (\bot, x_1') \in R) \Rightarrow \forall x_1 : (x_1, x_1') \in R.$$

By a series of generalisations we obtain the following predicate for arbitrary m and n (choose $m = n = i = 1$ and $J = \emptyset$ to recover the special case, observing that $\overline{i} = \emptyset$ and $\overline{J} = \{1\}$ and $(\vec{x}_{i \mapsto \bot}, \vec{x}'_{J \mapsto \bot}) = (\bot, x_1')$ hold):

$$\begin{aligned} &\forall i \in 1..m : \forall J \subseteq 1..n : \forall \vec{x}_{\overline{i}} : \forall \vec{x}'_{\overline{J}} : \\ &\quad (\bot \notin \vec{x}'_{\overline{J}} \wedge (\vec{x}_{i \mapsto \bot}, \vec{x}'_{J \mapsto \bot}) \in R) \Rightarrow \forall x_i : \exists \vec{x}'_J : (\vec{x}, \vec{x}') \in R. \end{aligned}$$

Intuitively, the antecedent states that for $x_i = \bot$ there is an outcome such that $x'_j \neq \bot$ if and only if $j \notin J$. Then all such x'_j must not depend on x_i. This means that there must be an outcome with these values of x'_j for all values of x_i. The general condition can be equivalently transformed into relational terms:

$$\forall i \in 1..m : \forall J \subseteq 1..n : x_i := \bot \,;\, R \,\cap\, \vec{x}'_J = \bot \;\subseteq\; R \,;\, \vec{x}_J := \bot \,\cup\, \bot \in \vec{x}'_{\overline{J}}. \qquad (1)$$

We can also derive an order-theoretic representation of (1). To this end, we introduce an order similar to $\preccurlyeq$, but now with respect to $\bot$. Let $\sqsubseteq : D_i \leftrightarrow D_i$ be the flat order on D_i with $\bot$ as its least element, that is, $x \sqsubseteq y \Leftrightarrow_{\text{def}} x = \bot \vee x = y$. Again, the order is extended pointwise to D_I by $\vec{x}_I \sqsubseteq \vec{y}_I \Leftrightarrow_{\text{def}} \forall i \in I : x_i \sqsubseteq y_i$, and its dual order is denoted by $\sqsupseteq =_{\text{def}} \sqsubseteq^{\smile}$. The properties of $\preccurlyeq$ can be transferred to $\sqsubseteq$. Using this order, we obtain an algebraic characterisation.

Lemma 12. *Let* $\mathscr{H}_N(R) \Leftrightarrow_{\text{def}} \sqsupseteq \,;\, R \;\subseteq\; R \,;\, \sqsupseteq$. *Then* (1) $\Leftrightarrow \mathscr{H}_N(R)$.

If R is a mapping, the condition $\mathscr{H}_N(R)$ states that R is isotone with respect to $\sqsupseteq$ [21]. Further remarks about $\mathscr{H}_N$ are given in Section 4.2 once the second condition is established. Let us 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 [16] two orders based on wp and wlp are combined for approximation.

We can show that our programming constructs satisfy $\mathscr{H}_N$. To deal with the assignment and the conditional, we assume that the expressions are isotone with respect to $\sqsubseteq$. The proof of the following result requires $\mathscr{T}_{\preccurlyeq}(\vec{e})$ and $\mathscr{T}_{\sqsubseteq}(\vec{e})$ for all expressions $\vec{e}$. Since $\preccurlyeq$ and $\sqsubseteq$ are structurally similar, the properties of $\mathscr{T}_{\preccurlyeq}$ can be transferred to $\mathscr{T}_{\sqsubseteq}$.

Theorem 13. *Relations composed of the constructs of Definition 4 with the restrictions stated in Table 1 satisfy* $\mathscr{H}_N$.

4.2 Absent Strict Parts

We now treat the case where relations have no strict parts. Let us again start with formalising the case $m = n = 1$. As stated at the beginning of Section 4, the relation R has no strict part if $(\bot, \bot) \notin R$. We then must make sure that the value of x'_1 does not depend on the value of x_1. In other words, any outcome x'_1 must be an outcome for all x_1. We therefore have

$$(\bot, \bot) \notin R \Rightarrow \forall x_1, x'_1 : (x_1, x'_1) \in R \Rightarrow \forall \tilde{x}_1 : (\tilde{x}_1, x'_1) \in R.$$

By a series of generalisations we obtain the following predicate for arbitrary m and n (choose $m = n = i = j = 1$ to recover the special case, observing that $\overline{i} = \overline{j} = \emptyset$ and $(\vec{x}_{i \mapsto \bot}, \vec{x}'_{j \mapsto \bot}) = (\bot, \bot)$ and $(\vec{x}_{i \mapsto \tilde{x}_i}, \vec{\tilde{x}}'_{j \mapsto x'_j}) = (\tilde{x}_1, x'_1)$ hold):

$$\begin{aligned} \forall i \in 1..m : \forall j \in 1..n : \forall \vec{x}_{\overline{i}} : (\forall \vec{x}'_{\overline{j}} : (\vec{x}_{i \mapsto \bot}, \vec{x}'_{j \mapsto \bot}) \notin R) \Rightarrow \\ \forall x_i : \forall \vec{x}' : (\vec{x}, \vec{x}') \in R \Rightarrow \forall \tilde{x}_i : \exists \vec{\tilde{x}}'_{\overline{j}} : (\vec{x}_{i \mapsto \tilde{x}_i}, \vec{\tilde{x}}'_{j \mapsto x'_j}) \in R. \end{aligned}$$

Intuitively, the first antecedent states that for $x_i{=}\bot$ there is no outcome such that $x'_j{=}\bot$. Then x'_j must not depend on x_i. This means that if there is an outcome $\vec{x}'$ for some value of x_i, there must be an outcome with the same value of x'_j for all values of x_i. We can again equivalently transform to relational terms:

$$\forall i \in 1..m : \forall j \in 1..n : \vec{x}_{\overline{i}}{=}\vec{x}'_{\overline{i}} \,;\, R \;\subseteq\; x_i{:=}\bot \,;\, R \,;\, x_j{=}\bot \;\cup\; R \,;\, x_j{=}x'_j. \qquad (2)$$

It turns out that we have to strengthen this condition to be able to prove closure under sequential composition. The reason is that the two occurrences of R on the right hand side are not coupled tightly enough. Such a problem did not arise with $\mathscr{H}_N$ that is structurally simpler, but it is solved in Lemma 14. Using the order $\sqsubseteq$ introduced in Section 4.1, we can derive an algebraic characterisation. It is proved to be stronger than (2) in the presence of $\mathscr{H}_N$.

Lemma 14. *Let* $\mathscr{H}_A(R) \Leftrightarrow_{\text{def}} \sqsubseteq \,;\, R \;\subseteq\; R \,;\, \sqsubseteq$ *and consider*

$$\forall I \subseteq 1..m : \vec{x}_{\overline{I}}{=}\vec{x}'_{\overline{I}} \,;\, R \;\subseteq\; \textstyle\bigcup_{J \subseteq 1..n} \vec{x}_I{:=}\bot \,;\, R \,;\, \vec{x}_J{:=}\bot^{\smile} \;\cap\; R \,;\, \vec{x}_{\overline{J}}{=}\vec{x}'_{\overline{J}}. \qquad (3)$$

Then (2) $\Leftarrow$ (3) $\Rightarrow \mathscr{H}_A(R)$. *If* $\mathscr{H}_N(R)$ *holds, then* (3) $\Leftrightarrow \mathscr{H}_A(R)$.

This lemma suggests to use the conjunction of $\mathscr{H}_A$ and $\mathscr{H}_N$ since it is equivalent to a stronger form of the derived conditions. If R is a mapping, we have $\mathscr{H}_N(R) \Leftrightarrow \mathscr{H}_A(R)$. But also the other programming constructs satisfy $\mathscr{H}_A$.

Theorem 15. *Relations composed of the constructs of Definition 4 with the restrictions stated in Table 1 satisfy* $\mathscr{H}_A$.

A general form of the conditions $\mathscr{H}_N$ and $\mathscr{H}_A$ appears in the literature, although in another context and not in relational form. Let $E : A \leftrightarrow A$ be a partial order and $R : A \leftrightarrow A$ a relation that satisfies $ER \subseteq RE$ and $E^{\smile} R \subseteq RE^{\smile}$. Then R is called an *isotone relation* [28] and an *order preserving multifunction* [25]. In both cases, the definition is given pointwise, requiring for all $(x_1, x_2) \in E$ that

- for each $y_1 \in R(x_1)$ there is a $y_2 \in R(x_2)$ such that $(y_1, y_2) \in E$, and
- for each $y_2 \in R(x_2)$ there is a $y_1 \in R(x_1)$ such that $(y_1, y_2) \in E$.

The investigation is concerned with the question whether A ordered by E satisfies the *relational fixed point property* [24]. This is the case iff every total, isotone relation R has a fixed point $x \in A$ such that $x \in R(x)$. Such a study has the relations themselves, interpreted as orders, as its objects. This has to be contrasted with our effort to obtain fixpoints of isotone functions over relations.

The two criteria stated above express precisely what constitutes the Egli-Milner order on powerdomains built from flat domains [18, 20]. 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 but in different ways and for a different purpose. For example, in [2, 4] it orders relations, while in [5, 9, 27] it orders domains of functional programming languages. All these sources use the Egli-Milner order to define the least fixpoint of functions. In our approach, however, fixpoints are ordered by the usual subset relation and the Egli-Milner order appears merely in the conditions $\mathscr{H}_N$ and $\mathscr{H}_A$ dealing with undefinedness. As a matter of fact the Egli-Milner order models erratic non-determinism or general correctness, but our definitions model demonic non-determinism or total correctness; see [16, 27] for the difference.

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 conditions for the termination information order $\preccurlyeq$ are discussed in Section 4.3. This view fits well with the notion of *partiality* investigated in [21]: 'A treatment of possibly partial availability of information may also be seen in descriptions of eager/data-driven evaluation as opposed to lazy/demand-driven evaluation.'[ibid., page 213]

4.3 Undefinedness and Non-termination

The conditions $\mathscr{H}_N$ and $\mathscr{H}_A$ introduced in the previous sections model dependence of undefined values. This manifests itself in the use of $\sqsubseteq$ with its least element $\bot$ in their definitions. It is legitimate to ask whether analogous conditions using $\preccurlyeq$ with its least element ∞ also hold. More generally, we should elaborate on the relationship between $\sqsubseteq$ and $\preccurlyeq$. Although these orders are structurally very similar and thus share several properties, there is an essential difference in their use. It is expressed by the conditions $\mathscr{H}_L$ and $\mathscr{H}_R$ enforcing closure with respect to $\preccurlyeq$. The reason why $\preccurlyeq$ is used for closure is the chosen model of the never terminating program: This relation should be both $\vec{x}{\leftarrow}\infty$ and the solution of the recursive equation $X = X$, that is, $\nu(\lambda X.X) = \top\!\!\top$. We thus obtain the requirement $\vec{x}{\leftarrow}\infty = \top\!\!\top$ which is satisfied by upper closure as Lemma 2 shows. To achieve this closure, $\preccurlyeq$ is inherent in our programming constructs. A similar upper closure with respect to $\sqsubseteq$ is neither necessary nor advisable for this would identify non-termination with undefinedness.

This explains why we use conditions of type $\mathscr{H}_L$ and $\mathscr{H}_R$ with respect to $\preccurlyeq$ but not $\sqsubseteq$. Let us return to the question of using $\mathscr{H}_N$- and $\mathscr{H}_A$-type conditions also with respect to $\preccurlyeq$. Such conditions can indeed be stated but we must take into account that the relations are $\mathscr{H}_L$- and $\mathscr{H}_R$-closed. Otherwise, simply requiring $\succcurlyeq\ ;\ R \subseteq R\ ;\ \succcurlyeq$ does not work since already for $R = \mathbb{1} = \preccurlyeq$ we would

obtain $\top = \succcurlyeq ; \preccurlyeq \subseteq \preccurlyeq ; \succcurlyeq$ which does not hold in general. Instead, we have to undo the effects of the upper closure and state conditions analogous to $\mathscr{H}_N$ and $\mathscr{H}_A$ using the minimal elements of the images as in Section 3.2. We use the relational formulation of min, similar to a construction in [23, page 43].

Theorem 16. *Let* $\min R =_{\text{def}} R \cap \overline{R \prec}$ *and*

$$\mathscr{H}_W(R) \Leftrightarrow_{\text{def}} \preccurlyeq ; \min R \subseteq R ; \preccurlyeq,$$
$$\mathscr{H}_M(R) \Leftrightarrow_{\text{def}} \succcurlyeq ; \min R \subseteq R ; \succcurlyeq.$$

Relations composed of the constructs of Definition 4 with the restrictions stated in Table 1 satisfy $\mathscr{H}_M$ *and* $\mathscr{H}_W$*.*

5 Summary and Adequacy

Table 1 summarises the closure properties of the conditions investigated in this paper. It lists for each condition $\mathscr{H}$ those constructs that are allowed in the construction of a relation or function R such that $\mathscr{H}(R)$ can be shown. The column $\exists$ refers to skip and (un)declaration, and the following columns refer to assignment, arbitrary constant relations, parallel composition, sequential composition, conditional, non-deterministic choice, conjunction, greatest and least fixpoints, in this sequence.

Table 1. Closure properties

Theorem	$\exists$	$\leftarrow\vec{e}$	constant	$\parallel$	;	$\blacktriangleleft b \blacktriangleright$	$\bigcup$	$\bigcap$	ν	μ
5 : isotony	–	–	–	–	–	–	–	–	–	–
6 : $\mathscr{H}_R$	–	–	$\mathscr{H}_R$	–	–	–	–	–	–	–
6 : $\mathscr{H}_L$	–	–	$\mathscr{H}_L$	–	–	$\mathscr{T}_{\preccurlyeq}$	–	–	–	–
6 : $\mathscr{H}_E$	–	–	$\mathscr{H}_E$	–	–	$\mathscr{T}_{\preccurlyeq}$	–	–	–	–
9 : $\mathscr{H}_B$	–	$\mathscr{T}_{\preccurlyeq}$	$\mathscr{H}_{EB}$	–	–	$\mathscr{T}_{\preccurlyeq}$	$\cup$	–	–	$\times$
10 : continuity	–	$\mathscr{T}_{\preccurlyeq}$	$\mathscr{H}_{EB}$	–	–	$\mathscr{T}_{\preccurlyeq}$	$\cup$	–	–	$\times$
11 : $\mathscr{H}_T$	–	$\mathscr{T}_{\preccurlyeq}$	$\mathscr{H}_{EBT}$	–	–	$\mathscr{T}_{\preccurlyeq}$	$\uplus$	$\wr$	–	$\times$
13 : $\mathscr{H}_N$	–	$\mathscr{T}_{\preccurlyeq\sqsubseteq}$	$\mathscr{H}_{EBTN}$	–	–	$\mathscr{T}_{\preccurlyeq\sqsubseteq}$	$\uplus$	$\wr$	–	$\times$
15 : $\mathscr{H}_A$	–	$\mathscr{T}_{\preccurlyeq\sqsubseteq}$	$\mathscr{H}_{EBA}$	–	–	$\mathscr{T}_{\preccurlyeq\sqsubseteq}$	$\cup$	$\wr$	–	$\times$
16 : $\mathscr{H}_M$	–	$\mathscr{T}_{\preccurlyeq}$	$\mathscr{H}_{EBTM}$	–	–	$\mathscr{T}_{\preccurlyeq}$	$\uplus$	$\wr$	–	$\times$
16 : $\mathscr{H}_W$	–	–	$\mathscr{H}_L$	–	–	$\mathscr{T}_{\preccurlyeq}$	–	–	–	–

An entry – means that construct is permitted unconditionally. An entry $\mathscr{T}_S$ means $\mathscr{T}_X(\vec{e})$ or $\mathscr{T}_X(b)$ must hold for all $X \in S$. An entry $\mathscr{H}_S$ means the constant must satisfy $\mathscr{H}_X$ for all $X \in S$. An entry $\cup$ means only finite choice is allowed, and $\uplus$ requires finite non-empty choice. An entry $\wr$ means only chains are allowed. Finally, an entry $\times$ means that construct is not permitted.

We thus obtain a theory similar to [11] but modelling non-strict computations. In particular, the left and right unit laws $\mathscr{H}_L$ and $\mathscr{H}_R$ and the right zero law $\mathscr{H}_T$ correspond to the healthiness conditions H1–H4 without the left zero law $\top ; R = \top$. Moreover, all functions composed of programming constructs are continuous and all relations composed of programming constructs are boundedly

non-deterministic. Additionally, they satisfy the conditions $\mathscr{H}_N$ and $\mathscr{H}_A$ modelling the dependence of variables. There is also a correspondence between the constructs introduced in Definition 4 and those of [11] stating that both yield the same results except that our model has better termination properties.

One can furthermore define a formal operational semantics to describe the execution of programs modelled by our constructs. Intuitively, we start with a set of variables whose final values we are interested in, and the execution proceeds backwards, evaluating only those parts actually needed to obtain the required values. Execution of assignments considers the dependences, execution of a conditional evaluates the condition first, and execution of a recursion starts by unfolding. Neither an undefined value nor a non-terminating part has an effect if it is not reached. It follows that our theory models non-strict computations.

6 Conclusion

We have proposed a new relational approach to define the semantics of imperative, non-deterministic programs. Let us summarise its key properties, which also differentiate our theory from related approaches such as [11].

- 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 laws of programs are stated in a compact algebraic form and can therefore be applied to new programs given as relations.
- The framework supports an operator for the parallel composition of relations. It is used to treat local variable declarations and alphabet extension adequately also in the context of non-termination. Relation algebra is used whenever possible for clear and concise arguments.
- 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. 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.

Connections to related work have been pointed out throughout this paper. The following description of further approaches is primarily focused on the similarities and differences to the present work.

Undefinedness and non-termination are addressed by [29] using the Z notation. The former is represented by a distinguished element $\perp$ that is propagated through sequential composition and thus models strict computations. Termination is treated by pairs of predicates describing pre- and postconditions. A combination of both aspects is not examined. The Z notation itself does neither deal with undefinedness nor with termination issues [12].

Instead of modelling non-strict computations in an imperative programming language, one can proceed the other way around and introduce state into a lazy functional programming language. A restricted form of state are variables which can be assigned only once as, for example, in [13]. Mutable state is provided by the Haskell I/O monad used in our introductory example. It has the property that all actions are forced, regardless of their contribution to the final result [14, 17]. This is avoided using the more general *state transformers* [15], combining lazy evaluation with stateful computation. Since the base language is functional, the semantics is given in the λ-calculus passing around environments and states. For our imperative context this is less adequate as using relations. Non-determinism is not treated and there is no distinction between undefinedness and non-termination.

A multi-paradigm language that supports lazy functions, exception handling, mutable state and non-deterministic choice points is Oz [19]. That book gives a formal operational semantics of the kernel language which, however, does not cover non-determinism. According to the reduction rule for sequential composition, the execution of statements is forced similar to the Haskell I/O monad.

Let us point out a few topics that deserve to be further investigated. One of them concerns the implementation of the presented theory. This involves a deeper study of the operational semantics and its connection to the relational model. Another thread is to explore the relational model as an intermediate for the translation of functional programming languages. The latter should be accompanied by comparing the semantics of lazy evaluation in both frameworks. A different domain is touched by applying the presented model of dependence in computations to develop optimising transformations used, for example, in compilers. Connections are anticipated to abstract interpretation and data flow analysis, where the partial availability of information also plays a role.

Acknowledgements. I am grateful to the anonymous referees for their helpful remarks and thank Bernhard Möller for his detailed comments about [8].

References

1. 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.
2. J. W. de Bakker. Semantics and termination of nondeterministic recursive programs. In S. Michaelson and R. Milner, editors, *Automata, Languages and Programming: Third International Colloquium*, pages 435–477. Edinburgh University Press, 1976.
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. R. Berghammer and H. Zierer. Relational algebraic semantics of deterministic and nondeterministic programs. *Theoretical Computer Science*, 43:123–147, 1986.

5. 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.
6. E. W. Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.
7. 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.
8. W. Guttmann. *Algebraic Foundations of the Unifying Theories of Programming.* PhD thesis, Universität Ulm, December 2007.
9. M. Hennessy and E. A. Ashcroft. The semantics of nondeterminism. In S. Michaelson and R. Milner, editors, *Automata, Languages and Programming: Third International Colloquium*, pages 478–493. Edinburgh University Press, 1976.
10. W. H. Hesselink. *Programs, Recursion and Unbounded Choice.* Cambridge University Press, 1992.
11. C. A. R. Hoare and J. He. *Unifying theories of programming.* Prentice Hall Europe, 1998.
12. ISO/IEC. Information technology: Z formal specification notation: Syntax, type system and semantics. ISO/IEC 13568:2002(E), July 2002.
13. M. B. Josephs. Functional programming with side-effects. *Science of Computer Programming*, 7:279–296, 1986.
14. 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.
15. J. Launchbury and S. Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, 8(4):293–341, December 1995.
16. G. Nelson. A generalization of Dijkstra's calculus. *ACM Transactions on Programming Languages and Systems*, 11(4):517–561, October 1989.
17. S. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report.* Cambridge University Press, 2003.
18. G. D. Plotkin. A powerdomain construction. *SIAM Journal on Computing*, 5(3):452–487, September 1976.
19. P. Van Roy and S. Haridi. *Concepts, Techniques, and Models of Computer Programming.* MIT Press, 2004.
20. D. A. Schmidt. *Denotational Semantics: A Methodology for Language Development.* William C. Brown Publishers, 1986.
21. G. Schmidt. Partiality I: Embedding relation algebras. *Journal of Logic and Algebraic Programming*, 66(2):212–238, February–March 2006.
22. 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.
23. G. Schmidt and T. Ströhlein. *Relationen und Graphen.* Springer-Verlag, 1989.
24. B. S. W. Schröder. *Ordered Sets: An Introduction.* Birkhäuser, 2003.
25. R. E. Smithson. Fixed points of order preserving multifunctions. *Proceedings of the American Mathematical Society*, 28(1):304–310, April 1971.
26. M. B. Smyth. Power domains. *Journal of Computer and System Sciences*, 16(1):23–36, February 1978.
27. H. Søndergaard and P. Sestoft. Non-determinism in functional languages. *The Computer Journal*, 35(5):514–523, October 1992.
28. J. W. Walker. Isotone relations and the fixed point property for posets. *Discrete Mathematics*, 48(2–3):275–288, February 1984.
29. J. Woodcock and J. Davies. *Using Z.* Prentice Hall, 1996.