# Relations as Executable Specifications: Taming Partiality and Non-determinism Using Invariants

Nuno Macedo, Hugo Pacheco, and Alcino Cunha

HASLab — High Assurance Software Laboratory
INESC TEC & Universidade do Minho, Braga, Portugal
{nfmmacedo,hpacheco,alcino}@di.uminho.pt

**Abstract.** The calculus of relations has been widely used in program specification and reasoning. It is very tempting to use such specifications as running prototypes of the desired program, but, even considering finite domains, the inherent partiality and non-determinism of relations makes this impractical and highly inefficient. To tame partiality we prescribe the usage of invariants, represented by coreflexives, to characterize the exact domains and codomains of relational specifications. Such invariants can be used as pre-condition checkers to avoid runtime errors. Moreover, we show how such invariants can be used to narrow the non-deterministic execution of relational specifications, making it viable for a relevant class of problems. In particular, we show how the proposed techniques can be applied to execute specifications of bidirectional transformations, a domain where partiality and non-determinism are paramount.

## 1 Introduction

The *relational calculus* provides a more natural way to specify programs than purely functional formalisms: most so-called functions in computer science are actually *partial*, and *non-determinism* is many times an essential characteristic of the program. In particular, since its first axiomatization by Tarski, a *point-free* (PF) version of the calculus of relations has been used in a variety of areas of computer science [3,15,16] in order to specify and reason about programs, due to its high simplicity and ease of manipulation.

However, relational specifications are frequently not amenable for execution: with partiality the behavior of the program may become unpredictable and give rise to runtime errors, while non-determinism may produce infinite runs without returning a single valid value. For instance, consider the expression $(\mathsf{id} \vartriangle \mathsf{id})^\circ \circ (\mathsf{length}^\circ \vartriangle \mathsf{head}^\circ) : \mathsf{Nat} \to [\mathsf{Nat}]$, where $\mathsf{head}$ returns the first element of a list, $\mathsf{length}$ its length, and $^\circ$ and $\vartriangle$ are the converse and split of relations, respectively. Given a natural $n$, this expression calculates a list with length $n$, whose first element is also $n$. This is not a total relation as it is not defined for the value 0, since no list with length 0 could have the same 0 as its head. We resort to the converse from the relational calculus to generate these lists: $\mathsf{head}^\circ$ generates all lists with the input value at its head, while $\mathsf{length}^\circ$ generates all lists with the

given length; both these operations are total and non-deterministic. The expression $(\mathsf{id} \vartriangle \mathsf{id})^\circ$ is the converse of the duplication operation: it is a partial function that takes as input tuples with two copies of the same element, and returns such element. In an unbounded execution, $\mathsf{length}^\circ$ and $\mathsf{head}^\circ$ would evaluate freely until they both return the same list that could be consumed by $(\mathsf{id} \vartriangle \mathsf{id})^\circ$. Such execution may not even terminate, since, for instance, $\mathsf{head}^\circ$ could be generating all possible lists by increasing length.

If we are able to determine exactly the domain (and range)[1] of an expression, such mechanism can be used to predict the behavior of partial expressions by being used as a pre-condition checker. In this case, we are able to calculate both the domain ($n \neq 0$) and the range ($\mathsf{length}\ l = \mathsf{head}\ l$, which also implies that $l$ is not empty) of this expression. Moreover, these domains can also be propagated down the expression to the inner combinators, avoiding unnecessary computations. In this case, due to $(\mathsf{id} \vartriangle \mathsf{id})^\circ$, $\mathsf{length}^\circ$ and $\mathsf{head}^\circ$ must generate the same list, and this information can be used to narrow their executions. In particular, given an input $n$, we can either restrict the values generated by $\mathsf{length}^\circ$ to those lists whose head is $n$ or, dually, restrict $\mathsf{head}^\circ$ to produce lists with length $n$. This will result in an efficient and complete (in the sense that all values will eventually be produced) non-deterministic evaluation.

In this paper, we propose a PF relational framework whose type system is enhanced with the introduction of *invariants* (represented by coreflexives), allowing the definition of more refined data-types, in order to address the above-mentioned issues. To carry this development, a powerful and simple calculus of invariants based on the relational PF notation [16] is harnessed into a type-inference and type-checking algorithm that works for many practical examples. The inferred invariants are also used to optimize the execution of a relational expression, making them viable as running prototypes of the specified program.

Our framework proves to be particularly useful in the area of bidirectional transformations (BX), where partiality and non-determinism play an important role. In particular, we put it to use in the specification of *lenses* [5], one of the most successful BX approaches. Using invariants to precisely characterize the domain and range of a lens, we can safely extend the class of expressible transformations, namely by allowing unrestricted usage of duplication, a well-known problematic feature in such frameworks. Also, propagation of such invariants allows us to efficiently execute the non-deterministic update propagation function for a wider class of transformations than before [5,17].

Section 2 introduces the PF relational calculus that is at the core of our framework. Section 3 presents our optimizations on invariant calculation and non-deterministic evaluation. Section 4 shows how standard recursion patterns can also be supported. In Sect. 5 we apply our framework in the specification of BX, obtaining non-deterministic lenses enhanced with invariants. Finally, Sect. 6 discusses related work and Sect. 7 draws the final conclusions and points directions for future work.

[1] By domain and range we refer to the exact set of values which a relation consumes and produces, respectively.

$$
\begin{array}{ll}
\cdot \circ \cdot & : (B \to C) \to (A \to B) \to (A \to C) \\
\cdot \cap \cdot & : (A \to B) \to (A \to B) \to (A \to B) \\
\cdot \cup \cdot & : (A \to B) \to (A \to B) \to (A \to B) \\
\cdot \vartriangle \cdot & : (A \to B) \to (A \to C) \to (A \to B \times C) \\
\cdot \triangledown \cdot & : (B \to A) \to (C \to A) \to (B + C \to A) \\
\top & : A \to B \\
\bot & : A \to B \\
\cdot^{\circ} & : (A \to B) \to (B \to A)
\end{array}
\qquad
\begin{array}{ll}
\mathsf{id} & : A \to A \\
\pi_1 & : A \times B \to A \\
\pi_2 & : A \times B \to B \\
i_1 & : A \to (A + B) \\
i_2 & : B \to (A + B) \\
! & : A \to 1 \\
\underline{\cdot} & : B \to (A \to B)
\end{array}
$$

**Fig. 1.** PF relational combinators.

## 2 Point-free Relational Calculus

Relation algebra [3,16,19] is a key ingredient in the formalization of our framework. It generalizes the well-known PF functional calculus, allowing us to reason about partiality and non-determinism using a powerful set of algebraic laws.

### 2.1 Syntax and Semantics

A *relation* $R$ is said to have type $A \to B$ if it is the subset of the Cartesian product $A \times B$. We write $b\ R\ a$ if the pair $(a, b)$ is in $R$. Relations can be built using the combinators presented in Fig. 1. The key combinator is *composition*, that given $R : A \to B$ and $S : B \to C$ builds a relation $S \circ R : A \to C$, which is associative and has the *identity* relation $\mathsf{id} : A \to A$ as neutral element (we thus have a category of relations). Relations $R : A \to B$ and $S : A \to B$ can be combined using the standard *intersection* and *union* operators. Every relation $R : A \to B$ also possesses a well-defined *converse* $R^{\circ} : B \to A$. For any two types $A$ and $B$, $\top : A \to B$ is the largest relation over those types (their Cartesian product) and $\bot : A \to B$ the smallest (the empty relation). A special case of $\top$ with final type 1 as range is denoted as $! : A \to 1$. For any value $b \in B$, the constant relation $\underline{b} : A \to B$ always returns $b$.

We also have *products* and *coproducts* (or *sums*). For any two relations $R : A \to B$ and $S : A \to C$, the *split* combinator is defined as $R \vartriangle S : A \to B \times C$. The left and right components of a pair can be projected with $\pi_1 : A \times B \to A$ and $\pi_2 : A \times B \to B$, respectively. Dually, for any two relations $R : B \to A$ and $S : C \to A$, the *either* combinator is defined as $R \triangledown S : B + C \to A$. Left and right tagged elements can be built with $i_1 : A \to A + B$ and $i_2 : B \to A + B$, respectively. Two derived combinators are the product and sum bifunctors, defined respectively as $R \times S = R \circ \pi_1 \vartriangle S \circ \pi_2$ and $R + S = i_1 \circ R \triangledown i_2 \circ S$. Some of the laws ruling the PF relational calculus are presented in an accompanying technical report [11].

The formal semantics of relational expressions as membership predicates is given in Fig. 2. Notice that, apart from composition, this semantics can be directly and efficiently executed. If we assume that all types are finite, composition could also be implemented but would obviously be very inefficient. An alternative semantics as non-deterministic functions (functions returning sets of values)

$$
\begin{array}{lll}
b\ [\![S \circ R]\!]\ a & = \exists\ c.\ b\ [\![S]\!]\ c \wedge c\ [\![R]\!]\ a & a'\ [\![\mathsf{id}]\!]\ a = a \equiv a' \\
b\ [\![R \cap S]\!]\ a & = b\ [\![R]\!]\ a \wedge b\ [\![S]\!]\ a & a'\ [\![\pi_1]\!]\ (a,b) = a \equiv a' \\
b\ [\![R \cup S]\!]\ a & = b\ [\![R]\!]\ a \vee b\ [\![S]\!]\ a & b'\ [\![\pi_2]\!]\ (a,b) = b \equiv b' \\
(b,c)\ [\![R \vartriangle S]\!]\ a & = b\ [\![R]\!]\ a \wedge c\ [\![S]\!]\ a & (\mathsf{Left}\ a')\ \ [\![i_1]\!]\ a = a \equiv b \\
a\ [\![R \triangledown S]\!]\ (\mathsf{Left}\ b) & = a\ [\![R]\!]\ b & (\mathsf{Left}\ a')\ \ [\![i_2]\!]\ b = \mathsf{False} \\
a\ [\![R \triangledown S]\!]\ (\mathsf{Right}\ c) & = a\ [\![S]\!]\ c & (\mathsf{Right}\ b')\ [\![i_1]\!]\ a = \mathsf{False} \\
a\ [\![R^\circ]\!]\ b & = b\ [\![R]\!]\ a & (\mathsf{Right}\ b')\ [\![i_2]\!]\ b = a \equiv b \\
b\ [\![\bot]\!]\ a & = \mathsf{False} & 1\ [\![!]\!]\ a = \mathsf{True} \\
b\ [\![\top]\!]\ a & = \mathsf{True} & b'\ [\![\underline{b}]\!]\ a = b \equiv b'
\end{array}
$$

**Fig. 2.** Semantics as predicates.

$$
\begin{array}{lll}
[\![S \circ R]\!]\ a & = \{\, b \mid c \leftarrow [\![R]\!]\ a, b \leftarrow [\![S]\!]\ c \,\} & [\![\mathsf{id}]\!]\ a = \{\, a \,\} \\
[\![R \cap S]\!]\ a & = \{\, b \mid b \leftarrow [\![R]\!]\ a, b\ [\![S]\!]\ a \,\} & [\![\pi_1]\!]\ (a,b) = \{\, a \,\} \\
[\![R \cup S]\!]\ a & = [\![R]\!]\ a\ \cup\ [\![S]\!]\ a & [\![\pi_2]\!]\ (a,b) = \{\, b \,\} \\
[\![R \vartriangle S]\!]\ a & = \{\, (b,c) \mid b \leftarrow [\![R]\!]\ a, c \leftarrow [\![S]\!]\ a \,\} & [\![i_1]\!]\ a = \{\mathsf{Left}\ a\} \\
[\![R \triangledown S]\!]\ (\mathsf{Left}\ b) & = [\![R]\!]\ b & [\![i_2]\!]\ b = \{\mathsf{Right}\ b\} \\
[\![R \triangledown S]\!]\ (\mathsf{Right}\ c) & = [\![S]\!]\ c & [\![!]\!]\ a = \{1\} \\
[\![R^\circ]\!]\ b & = \{\, a \mid a \leftarrow A, a\ [\![R^\circ]\!]\ b \,\} & [\![\underline{b}]\!]\ a = \{\, b \,\} \\
[\![\top]\!]\ a & = B & [\![\bot]\!]\ a = \{\,\}
\end{array}
$$

**Fig. 3.** Semantics as non-deterministic functions.

is more useful if we intend to execute relational specifications. It can trivially be defined by set comprehension as $[\![R{:}A \to B]\!]\ a = \{\, b \mid b\ [\![R]\!]\ a, b \leftarrow B \,\}$, but such definition is highly inefficient and cannot be used in practice. Figure 3 presents an alternative optimized definition that avoids the exhaustive search over $B$ for all combinators but the converse. Again, this semantics can be directly implemented, for example using the non-determinism monad in a functional language like Haskell. However, given a value $a$, even if we are only interested in just one of the results of $[\![R]\!]\ a$, there are still several concerns for efficiency (besides the converse) that make such definition impractical. For example, in the left-biased implementation of intersection we still need to iterate over all results of $R$ until a suitable value that also satisfies $S$ is found.

The *kernel* of a relation is defined as $\mathsf{ker}\ R = R^\circ \circ R$, while its counterpart, the *image*, is defined as $\mathsf{img}\ R = R \circ R^\circ$. A relation $R$ is said to be *reflexive* if it is at least the identity ($\mathsf{id} \subseteq R$), and *coreflexive* if it is at most the identity ($R \subseteq \mathsf{id}$). Coreflexives will be denoted by upper-case Greek letters ($\Psi, \Phi, \Omega, ...$). Relations can be classified according to the properties of their kernel and image. A relation is said to be total or surjective if its kernel and image are reflexive, respectively, and injective or simple if its kernel or image are coreflexive, respectively. Functions arise as the particular class of relations that are total and simple. As a convention, the identifiers of relational expressions that happen to be

simple will begin with a lower-case. So, while $R, S, T, ...$ are typical identifiers for relational expressions, $f, g, h, ...$ will denote simple relations (partial functions).

### 2.2 Predicates as Coreflexives

Coreflexives act as filters of data and can be used to model predicates (and thus invariants): values $a$ for which $a \; [\![\Phi]\!] \; a$ satisfy the predicate $\Phi$. We will often see them as sets and denote predicate satisfiability using just set membership $a \in [\![\Phi]\!] \equiv a \; [\![\Phi]\!] \; a$. Coreflexives have interesting algebraic properties that simplify their manipulation like, for example, $\Phi^\circ = \Phi$, $\Phi \circ \Phi = \Phi$, and $\Phi \circ \Psi = \Phi \cap \Psi$. Evaluation of coreflexives also reduces to membership test as $[\![\Phi]\!] \; a = \{ \, a \mid a \in [\![\Phi]\!] \, \}$, meaning that its evaluation is typically efficient. The only problematic case is again composition, but as we will see shortly most of the compositions appearing in coreflexives can be evaluated efficiently. In particular, composition of coreflexives is just a conjunction of predicates.

A predicate on products can always be specified by a relation between its elements. Any relation $R : A \to B$ can be lifted to a coreflexive $[R] : A \times B \to A \times B$ defined as $[R] = (\pi_2^\circ \circ R \circ \pi_1) \cap \mathsf{id}$. Another way to put it is to say that $[R]$ is the largest coreflexive $\Phi$ such that $\pi_2 \circ \Phi \subseteq R \circ \pi_1$, since $\Phi \subseteq [R] \Leftrightarrow \pi_2 \circ \Phi \circ \pi_1^\circ \subseteq R$.

From this we can derive many interesting properties of this combinator, such as $[\top] = \mathsf{id}$, $[\bot] = \bot$, the cancellation rules $\pi_1 \circ [R] = (\mathsf{id} \vartriangle R)^\circ$ and $\pi_2 \circ [R] = (R^\circ \vartriangle \mathsf{id})^\circ$, and $[\pi_2 \circ \Phi \circ \pi_1^\circ] = \Phi$ for any coreflexive on pairs $\Phi$. For example, using this combinator we can trivially specify the predicate stating that both components of a pair are equal using the coreflexive $[\mathsf{id}] : A \times A \to A \times A$. Given coreflexives $\Phi : A \to A$ and $\Psi : B \to B$, their product is the coreflexive $\Phi \times \Psi : A \times B \to A \times B$ that holds for pairs whose left element satisfies $\Phi$ and whose right element satisfies $\Psi$. It can alternatively be specified as $\Phi \times \Psi = [\Psi \circ \top \circ \Phi]$.

Coreflexives on sums are considerably simpler, since predicates on sums can always be specified using the sum combinator. The coreflexive $\Phi + \Psi : A + B \to A + B$ holds for left values that satisfy $\Phi$ and for right values that satisfy $\Psi$.

Every coreflexive has a *complement* $\overline{\Phi} : A \to A$ such that $a \; [\![\Phi]\!] \; a \Leftrightarrow \neg(a \; [\![\Phi]\!] \; a)$. A useful combinator for coreflexives is the guard $\Phi? = (\Phi \triangledown \overline{\Phi})^\circ : A \to A + A$ that tags the input as a left or right value in a sum, depending on the result of testing $\Phi$. Composed with an either, it allows the representation conditionals, i.e., $(R \triangledown S) \circ \Phi?$ applies $R$ if the input is in $\Phi$, and $S$ otherwise.

In this paper, we will use coreflexives to specify the invariants that characterize the *domain* and *range* of a relation, thus type-inference will amount to calculating the domain/range of a relation, while type-checking will consist of a membership test on those coreflexives. Given a relation $R : A \to B$, its domain, denoted as $\delta R : A \to A$, is the coreflexive $\delta R = \mathsf{ker} \; R \cap \mathsf{id}$. Dually, its range, denoted as $\rho R : B \to B$, is the coreflexive $\rho R = \mathsf{img} \; R \cap \mathsf{id}$. If $R$ is total, its kernel is larger than $\mathsf{id}$ and thus $\delta R = \mathsf{ker} \; R \cap \mathsf{id} = \mathsf{id}$, as expected, while if $R$ is simple, its image is smaller than the identity and thus $\rho R = \mathsf{img} \; R \cap \mathsf{id} = \mathsf{img} \; R$. These definitions simplify in a similar way for surjective and injective relations.

$$
\begin{array}{llll}
\delta\mathsf{id} = \mathsf{id} & \delta\Phi = \Phi & \rho\mathsf{id} = \mathsf{id} & \rho\Phi = \Phi \\
\delta\bot = \bot & \delta\top = \mathsf{id} & \rho\bot = \bot & \rho\top = \mathsf{id} \\
\delta(R \cap S) = R^\circ \circ S \cap \mathsf{id} & \delta\pi_1 = \mathsf{id} & \rho(R \cap S) = R \circ S^\circ \cap \mathsf{id} & \rho\pi_1 = \mathsf{id} \\
\delta(R \cup S) = \delta R \cup \delta S & \delta\pi_2 = \mathsf{id} & \rho(R \cup S) = \rho R \cup \rho S & \rho\pi_2 = \mathsf{id} \\
\delta(R \vartriangle S) = \delta R \cap \delta S & \delta i_1 = \mathsf{id} & \rho(R \vartriangle S) = [S \circ R^\circ] & \rho i_1 = \mathsf{id} + \bot \\
\delta(R \triangledown S) = \delta R + \delta S & \delta i_2 = \mathsf{id} & \rho(R \triangledown S) = \rho R \cup \rho S & \rho i_2 = \bot + \mathsf{id} \\
\delta(R^\circ) = \rho R & \delta! = \mathsf{id} & \rho(R^\circ) = \delta R & \rho! = \mathsf{id}_1 \\
\delta(\Phi?) = \mathsf{id} & \delta\underline{b} = \mathsf{id} & \rho(\Phi?) = \Phi + \overline{\Phi} & \rho\underline{b} = \underline{b} \cap \mathsf{id} \\
\delta(R \circ S) = \delta(\delta R \circ S) & & \rho(R \circ S) = \rho(R \circ \rho S) &
\end{array}
$$

**Fig. 4.** Domain and range of PF combinators.

A relation $R : A \to B$ that is only defined for inputs satisfying $\Phi$ and always produces outputs satisfying $\Psi$ ($R \subseteq \Psi \circ \top \circ \Phi$) will be typed as $R : A_\Phi \to B_\Psi$, or just $R : \Phi \to \Psi$ if the underlying types are irrelevant or clear from the context.

## 3 Optimizations

In the previous section, we have shown how the domain and range of a relation can be specified. However, such specifications involve relational compositions that hinder their efficient execution as pre- and pos-condition checkers of a relation. In this section we will first show how the calculation of the domain and range can be optimized to yield expressions more amenable to execution. Then, we will show how we can take advantage of such domain and range expressions to optimize the semantics defined in Fig. 3.

### 3.1 Optimizing Domain and Range Calculation

For relational programs written using the PF combinators from Fig. 1, their respective domains and ranges can be defined by induction as presented in Figs. 4 and 5. To avoid infinite reductions in compositions, the laws of Fig. 5 should be prioritized. These laws detail how the domain and range of a combinator should be further restricted in presence of a coreflexive.

The expressions resulting from these definitions are more amenable for execution (and consequently, type-checking) than the default domain and range definitions because most of the compositions are eliminated. The remaining ones (except for the range of the split combinator) fall in the special case $R^\circ \circ U \circ S$, that, as shown in [15], can be evaluated deterministically as $a \; [\![R^\circ \circ U \circ S]\!] \; b = (R \; a) \; [\![U]\!] \; (S \; b)$ if $R$ and $S$ are functions. After applying the laws of Figs. 4 and 5, we further simplify the resulting expression using a rewrite system similar to one previously developed for the optimization of PF functional expressions [4,18]. Essentially, this rewrite system applies some of the PF laws [11] as unidirectional rewrite rules oriented from left to right. This simplification phase can further eliminate problematic compositions. If the final expression still contains some of

$$
\begin{array}{ll}
\delta(\Phi \circ \Psi) = \Phi \cap \Psi & \rho(\Psi \circ \Phi) = \Psi \cap \Phi \\
\delta(\Phi \circ \mathsf{id}) = \Phi & \rho(\mathsf{id} \circ \Phi) = \Phi \\
\delta(\Phi \circ \top) = \begin{cases} \bot & \text{if } \Phi = \bot \\ (\top \circ \Phi \circ \top) \cap \mathsf{id} & \text{otherwise} \end{cases} & \rho(\top \circ \Phi) = \begin{cases} \bot & \text{if } \Phi = \bot \\ (\top \circ \Phi \circ \top) \cap \mathsf{id} & \text{otherwise} \end{cases} \\
\delta(\Phi \circ \bot) = \bot & \rho(\bot \circ \Phi) = \bot \\
\delta(\Phi \circ R^\circ) = \rho(R \circ \Phi) & \rho(R^\circ \circ \Phi) = \delta(\Phi \circ R) \\
\delta(\Phi \circ (R \cup S)) = \delta(\Phi \circ R) \cup \delta(\Phi \circ S) & \rho((R \cup S) \circ \Phi) = \rho(R \circ \Phi) \cup \rho(S \circ \Phi) \\
\delta(\Phi \circ (R \cap S)) = (R^\circ \circ \Phi \circ S) \cap \mathsf{id} & \rho((R \cap S) \circ \Phi) = (R \circ \Phi \circ S^\circ) \cap \mathsf{id} \\
\delta(\Phi \circ \pi_1) = \Phi \times \mathsf{id} & \rho(\pi_1 \circ [U]) = \delta U \\
\delta(\Phi \circ \pi_2) = \mathsf{id} \times \Phi & \rho(\pi_2 \circ [U]) = \rho U \\
\delta([U] \circ (R \vartriangle S)) = (\delta S \circ R^\circ \circ U^\circ \circ S \circ \delta R) \cap \mathsf{id} & \rho((R \vartriangle S) \circ \Phi) = [S \circ \Phi \circ R^\circ] \\
\delta((\Phi + \Psi) \circ i_1) = \Phi & \rho(i_1 \circ \Phi) = \Phi + \bot \\
\delta((\Phi + \Psi) \circ i_2) = \Psi & \rho(i_2 \circ \Phi) = \bot + \Phi \\
\delta(\Phi \circ (R \triangledown S)) = \delta(\Phi \circ R) + \delta(\Phi \circ S) & \rho((R \triangledown S) \circ (\Phi + \Psi)) = \rho(R \circ \Phi) \cup \rho(S \circ \Psi) \\
\delta(\Phi \circ \underline{b}) = \begin{cases} \mathsf{id} \text{ if } b \ [\![\Phi]\!] \ b \\ \bot \text{ otherwise} \end{cases} & \rho(\underline{b} \circ \Phi) = \begin{cases} \bot & \text{if } \Phi = \bot \\ \underline{b} \circ \Phi \circ \underline{b}^\circ & \text{otherwise} \end{cases} \\
\delta(\Phi \circ !) = \begin{cases} \mathsf{id} \text{ if } 1 \ [\![\Phi]\!] \ 1 \\ \bot \text{ otherwise} \end{cases} & \rho(! \circ \Phi) = \begin{cases} \bot & \text{if } \Phi = \bot \\ ! \circ \Phi \circ !^\circ & \text{otherwise} \end{cases} \\
\delta((\Phi + \Psi) \circ \Omega ?) = (\Phi \cap \Omega) \cup (\Psi \cap \overline{\Omega}) & \rho(\Psi \ ? \circ \Phi) = (\Psi \cap \Phi) + (\overline{\Psi} \cap \Phi)
\end{array}
$$

**Fig. 5.** Domain and range of compositions.

those, our implementation can issue a warning informing that its usage as an invariant checker may not be feasible.

This rewrite system is also used to perform the equality test $\Phi = \bot$ that occurs in some of the definitions in Fig. 5. However, since such test may not be not decidable, i.e., the rewrite system may not be able to reduce into $\bot$ an expression that is semantically equivalent to $\bot$, if we cannot show that $\Phi$ is empty, the default definitions of range and domain are applied instead. Still, for some cases when we can prove that $\Phi \neq \bot$, the range of (for instance) $! \circ \Phi$ and $\top \circ \Phi$ can be further simplified to $\mathsf{id}$.

### 3.2 Optimizing Non-deterministic Executions

The executable semantics of Fig. 3 can be optimized by propagating the domains and ranges of the outer expressions down to the inner expressions, in order to avoid the computation of intermediate values that are valid for sub-expressions but are not valid for the global expression. Figure 6 shows how this propagation can be performed ($A$ and $B$ denote the set of all elements of the respective type), where input values are assumed to have already passed the pre-condition test, i.e., for an evaluation $[\![R : \Phi \to \Psi]\!]\ a$, we assume that $a \in \Phi$. For instance, in the evaluation of $R \circ S$ we can narrow the evaluation of $S$ to return only values in the domain of $R$ (and vice-versa), thus avoiding generation of values not accepted by $R$; since the split $R \vartriangle S$ is only defined for values in the domain of both $R$ and $S$, the domain invariant of each branch takes the domain of the other, in order to disregard invalid values during execution. The converse of expressions is presented in Fig. 7, where each case is analyzed individually

$$
\begin{array}{l}
[\![R \circ S : \Phi \to \Psi]\!]\ a = \{\, b \mid c \leftarrow [\![S : \Phi \to \delta R]\!]\ a, b \leftarrow [\![R : \rho S \to \Psi]\!]\ c \,\} \\
[\![R \cap S : \Phi \to \Psi]\!]\ a = \{\, b \mid b \leftarrow [\![R : \Phi \cap \delta S \to \rho(\Psi \circ S \circ \underline{a})]\!]\ a \,\} \\
[\![R \cup S : \Phi \to \Psi]\!]\ a = [\![R : \Phi \to \Psi]\!]\ a \cup [\![S : \Phi \to \Psi]\!]\ a \\
[\![R \vartriangle S : \Phi \to [\,U\,]]\!]\ a = \{\, (b, c) \mid b \leftarrow [\![R : \Phi \cap \delta S \to \rho(U^\circ \circ S \circ \underline{a})]\!]\ a, \\
\qquad\qquad\qquad\qquad\qquad\qquad c \leftarrow [\![S : \Phi \cap \delta R \to \rho(U \circ \underline{b})]\!]\ a \,\}
\end{array}
$$

$$
\begin{array}{ll}
[\![\mathsf{id} : \Phi \to \Psi]\!]\ a = [\![\Psi]\!]\ a & [\![\pi_1 : [\,U\,] \to \Psi]\!]\ (a, b) = [\![\Psi]\!]\ a \\
[\![\Omega : \Phi \to \Psi]\!]\ a = \{\, a' \mid a' \leftarrow [\![\Omega]\!]\ a, a'\ [\![\Psi]\!]\ a' \,\} & [\![\pi_2 : [\,U\,] \to \Psi]\!]\ (a, b) = [\![\Psi]\!]\ b \\
[\![i_1 : \Phi \to \Psi + \Omega]\!]\ a = \{\, \mathsf{Left}\ a' \mid a' \leftarrow [\![\Psi]\!]\ a \,\} & [\![\bot : \Phi \to \Psi]\!]\ a = \{\,\} \\
[\![i_2 : \Phi \to \Psi + \Omega]\!]\ b = \{\, \mathsf{Right}\ b' \mid b' \leftarrow [\![\Omega]\!]\ b \,\} & [\![\top : \Phi \to \Psi]\!]\ a = \{\, b \mid b \leftarrow B, b\ [\![\Psi]\!]\ b \,\} \\
[\![R \triangledown S : \Phi + \Omega \to \Psi]\!]\ (\mathsf{Left}\ b) = [\![R : \Phi \to \Psi]\!]\ b & [\![\underline{b} : \Phi \to \Psi]\!]\ a = [\![\Psi]\!]\ b \\
[\![R \triangledown S : \Phi + \Omega \to \Psi]\!]\ (\mathsf{Right}\ c) = [\![S : \Omega \to \Psi]\!]\ c & [\![! : \Phi \to \Psi]\!]\ a = [\![\Psi]\!]\ 1
\end{array}
$$

**Fig. 6.** Optimized non-deterministic evaluation.

$$
\begin{array}{ll}
[\![!^\circ : \Phi \to \Psi]\!]\ 1 = \{\, a \mid a \leftarrow A, a\ [\![\Psi]\!]\ a \,\} & [\![\underline{b}^\circ : \Phi \to \Psi]\!]\ b = \{\, a \mid a \leftarrow A, a\ [\![\Psi]\!]\ a \,\} \\
[\![\pi_1^\circ : \Phi \to [\,U\,]]\!]\ a = \{\, (a, b) \mid b \leftarrow [\![U]\!]\ a \,\} & [\![\pi_2^\circ : \Phi \to [\,U\,]]\!]\ b = \{\, (a, b) \mid a \leftarrow [\![U^\circ]\!]\ b \,\} \\
[\![i_1^\circ : \Phi + \bot \to \Psi]\!]\ (\mathsf{Left}\ a) = [\![\Psi]\!]\ a & [\![i_2^\circ : \bot + \Phi \to \Psi]\!]\ (\mathsf{Right}\ b) = [\![\Psi]\!]\ b \\
[\![(R \vartriangle S)^\circ : [\,U\,] \to \Psi]\!]\ (b, c) = \{\, a \mid a \leftarrow [\![R^\circ : \rho U \to \rho(\Psi \circ S^\circ \circ \underline{c})]\!]\ b \,\} & \\
[\![(R \triangledown S)^\circ : \Phi \to \Psi + \Omega]\!]\ a = [\![i_1 \circ R^\circ : \Phi \to \Psi]\!]\ a \cup [\![i_2 \circ S^\circ : \Phi \to \Omega]\!]\ a &
\end{array}
$$

**Fig. 7.** Optimized non-deterministic evaluation of converses.

to achieve better efficiency. We omit the evaluation of the converse of idempotent combinators ($\mathsf{id}, \Omega, \top, \bot$) and of combinators whose converse can be easily propagated ($R \circ S, R \cap S, R \cup S$) and thus can be executed by the definitions in Fig. 6. The proof of the semantic equivalence between the two versions, in the sense that $[\![R : \Phi \to \Psi]\!] = [\![\Psi \circ R \circ \Phi]\!]$, is given in [11].

The evaluation of the primitive combinators is, for most cases, fairly obvious, since it consists in their standard definition, with a membership test for the desired invariant. Note however that all invariant tests occur at the primitives, meaning that infeasible values are not passed through higher-order combinators. Nevertheless, redundant values can still be generated, even if they produce a valid output. For instance, in the expression $\underline{b} \circ \top$, $\top$ will generate all possible values, even though they will all be transformed into the same value by $\underline{b}$. In many of such cases, the rewrite system already presented can be used to remove redundant value generation. In this case, $\underline{b} \circ \top$ would be reduced to $\underline{b}$.

The most interesting narrowing cases are those of the meet and the converse of split (which is itself a meet). For these cases, with the definition from Fig. 3, the $R$ branch would execute independently of the invariants of $S$ and its output would be tested in $S$. Naturally, the unconstrained evaluation of $R$ can be very inefficient and may process and generate infeasible values that are not in the domain or range of $S$, respectively. Using invariants, we restrict $R$ to the domain of $S$ and constrain the values generated by $R$ to only those that would also be produced by $S$. For instance, in the execution of the converse of the split

$[\![(R \vartriangle S)^\circ]\!]\ (b, c)$, instead of having $[\![R^\circ]\!]\ b$ running freely, it is restricted to produce values that would also be produced by $[\![S^\circ]\!]\ c$, as specified by its post-condition $\rho(\Psi \circ S^\circ \circ \underline{c})$. Again, a right-biased implementation would be equivalent.

## 4 Recursive Relations with Invariants

In this section, we investigate the construction of expressions and the calculation of invariants for recursive types. Most user-defined data types can be defined as fixed points of regular functors. Given a base functor, the inductive type generated by its least fixed point will be denoted by $\mu\, F$. A regular functor is either the identity functor $\mathsf{Id}$ (denoting recursive invocation), the constant functor $\underline{A}$, the lifting of the sum $\oplus$ and product bifunctors $\otimes$, or the composition of functors $\odot$. For example, for lists we have $[A] = \mu\mathsf{L}$, where $\mathsf{L} = \underline{1} \oplus (\underline{A} \otimes \mathsf{Id})$, and for naturals $\mathbb{N} = \mu\mathsf{N}$, where $\mathsf{N} = \underline{1} \oplus \mathsf{Id}$. Associated with each data type $\mu F$ we have also two unique functions $\mathsf{in}_F : F\ \mu F \to \mu F$ and $\mathsf{out}_F : \mu F \to F\ \mu F$, that are each other's inverse. The list constructors can be defined as $\mathsf{nil} = \mathsf{in}_\mathsf{L} \circ i_1$ and $\mathsf{cons} = \mathsf{in}_\mathsf{L} \circ i_2$, (thus $\mathsf{nil} \triangledown \mathsf{cons} = \mathsf{in}_\mathsf{L}$), and for naturals as $\mathsf{zero} = \mathsf{in}_\mathsf{N} \circ i_1$ and $\mathsf{succ} = \mathsf{in}_\mathsf{N} \circ i_2$. They allow us to encode and inspect values of the given type, respectively. The application of $\mathsf{out}$ results on a one-level unfolding to a sums-of-products representation capable of being processed with PF combinators. For a functor $F$ and a function $f : A \to B$, the functor mapping $F\ f : F\ A \to F\ B$ is a function that maps $f$ over the instances of the type argument, and can be defined inductively over the structure of the functor. Since $\mathsf{in}$ and $\mathsf{out}$ are bijections, they are total and surjective, and for $\mathsf{in}$ (and dually $\mathsf{out}$) we have that:

$$\delta(\Phi \circ \mathsf{in}) = \mathsf{out} \circ \Phi \circ \mathsf{in} \qquad\qquad \rho(\mathsf{in} \circ \Phi) = \mathsf{in} \circ \Phi \circ \mathsf{out}$$

Instead of defining expressions by general recursion, we resort to well-known recursion patterns, namely folds (catamorphisms) and unfolds (anamorphisms), that encode the recursion patterns of iteration and coiteration, respectively. The fold $(\!|R|\!)_F : \mu F \to A$ consumes values of a recursive type $\mu F$ according to an algebra $R : F\ A \to A$, while the dual unfold $[\![S]\!]_F : A \to \mu F$ produces elements of a recursive type $\mu F$ according to a coalgebra $S : A \to F\ A$, and are the unique relations that make the hereunder diagrams commute:

$$\begin{array}{ccc} \mu F & \xrightarrow{\mathsf{out}_F} & F\ \mu F \\ {\scriptstyle (\!|R|\!)_F}\downarrow & & \downarrow{\scriptstyle F\ (\!|R|\!)_F} \\ A & \xleftarrow[R]{} & F\ A \end{array} \qquad\qquad \begin{array}{ccc} \mu F & \xleftarrow{\mathsf{in}_F} & F\ \mu F \\ {\scriptstyle [\![S]\!]_F}\uparrow & & \uparrow{\scriptstyle F\ [\![S]\!]_F} \\ A & \xrightarrow[S]{} & F\ A \end{array}$$

As expected, these recursion patterns preserve the simplicity of their argument algebras or coalgebras [2]. Forward and backward (converse) evaluation is not problematic, because we can proceed recursively by unfolding their definitions:

$$(\!|R|\!)_F = R \circ F\ (\!|R|\!)_F \circ \mathsf{out}_F \qquad\qquad [\![S]\!]_F = \mathsf{in}_F \circ F\ [\![S]\!]_F \circ S$$

The main problem, however, is the optimization of domain/range calculation for

folds and unfolds due to the nonexistence of a normal form to express invariants over recursive types. For some simple cases, we can rely on the following laws [2]:

$$F\ \rho R \subseteq \delta R \Rightarrow \delta(\!|R|\!) = \mathsf{id} \qquad F\ \delta S \subseteq \rho S \Rightarrow \rho[\!(S)\!] = \mathsf{id}$$
$$R : F\ \Phi \to \Phi \Rightarrow (\!|R|\!)_F : \mathsf{id} \to \Phi \qquad S : \Phi \to F\ \Phi \Rightarrow [\!(S)\!]_F : \Phi \to \mathsf{id}$$

Focusing on the left column (the other is dual for unfolds), the first law states that a fold is total if the range of its algebra is contained in its own domain (in particular, total algebras yield total folds); the second law states a simple consistency condition needed to establish the range of a fold. Whenever these laws do not apply, we resort to the general definitions of domain and range presented in Sect. 2.2, and then apply the rewrite system briefly presented in Sect. 3.1, enriched with laws to handle recursive patterns, namely fusion[2]:

$$S \circ (\!|R|\!)_F = (\!|T|\!)_F \Leftarrow S \circ R = T \circ F\ S \qquad [\!(S)\!]_F \circ R = [\!(T)\!]_F \Leftarrow S \circ R = F\ R \circ T$$

Fusion laws transform the composition of a relation with a recursion pattern into a single recursion pattern. As explained before, we issue a warning if the rewrite system yields expressions whose evaluation may be problematic.

We now give some examples of recursive expressions that are already supported in our framework. We begin with $(\mathsf{id} \vartriangle \mathsf{id})^\circ \circ (\mathsf{length}^\circ \vartriangle \mathsf{head}^\circ)$, the example from the introduction, where $\mathsf{head} = \pi_1 \circ \mathsf{cons}^\circ$ and $\mathsf{length} = (\!|\mathsf{in_N} \circ (\mathsf{id} + \pi_2)|\!)$. The domain of $\mathsf{head}$ is $\mathsf{in_L} \circ (\bot + \mathsf{id}) \circ \mathsf{out_L}$, meaning that the list cannot be empty, while $\mathsf{length} : \mathsf{id} \to \mathsf{id}$ by applying the above laws for folds, since its algebra has type $F\ \mathsf{id} \to \mathsf{id}$. The range of the whole expression can be computed as follows:

$\rho((\mathsf{id} \vartriangle \mathsf{id})^\circ \circ (\mathsf{head}^\circ \vartriangle \mathsf{length}^\circ))$
  $=$ {-Range definition: Fig. 4 -}
$\rho((\mathsf{id} \vartriangle \mathsf{id})^\circ \circ \rho(\mathsf{head}^\circ \vartriangle \mathsf{length}^\circ))$
  $=$ {-Range definition: Fig. 4 -}
$\rho((\mathsf{id} \vartriangle \mathsf{id})^\circ \circ [\mathsf{length}^\circ \circ \mathsf{head}])$
  $=$ {-Range definition: Fig. 5 -}
$\delta([\mathsf{length}^\circ \circ \mathsf{head}] \circ (\mathsf{id} \vartriangle \mathsf{id}))$
  $=$ {-Domain definition: Fig. 5, Simplifications: PF Laws [11] -}
$(\mathsf{head}^\circ \circ \mathsf{length}) \cap \mathsf{id}$

Since $l \in [\!(\mathsf{head}^\circ \circ \mathsf{length}) \cap \mathsf{id}]\!] \Leftrightarrow \mathsf{head}\ l \equiv \mathsf{length}\ n$, we have the expected invariant on the range. On the other hand, its domain is $\mathsf{in_N} \circ (\bot + \mathsf{id}) \circ \mathsf{out_N}$ (the proof can be found in [11]), and thus, $(\mathsf{id} \vartriangle \mathsf{id})^\circ \circ (\mathsf{length}^\circ \vartriangle \mathsf{head}^\circ) : \mathsf{in_N} \circ (\bot + \mathsf{id}) \circ \mathsf{out_N} \to (\mathsf{head}^\circ \circ \mathsf{length}) \cap \mathsf{id}$.

Another example of a catamorphism is the $\mathsf{unzip} : [A \times B] \to [A] \times [B]$ function, that splits a list of pairs into two lists with the left and right elements. Since the algebra of $\mathsf{unzip}$ is a total function $g = (\mathsf{nil} \vartriangle \mathsf{nil}) \bigtriangledown ((\mathsf{cons} \circ \pi_1 \times \pi_1) \vartriangle (\mathsf{cons} \circ \pi_2 \times \pi_2))$, the domain of the catamorphism is $\mathsf{id}$. As for its range, our rewrite system performs a calculation equivalent to the following:

[2] We implement the "guessing step" required for fusion using the technique from [18].

$\rho\mathsf{unzip}$
$\quad = \;$ {-Definitions: range -}
$(\mathsf{unzip} \circ \mathsf{unzip}^\circ) \cap \mathsf{id}$
$\quad = \;$ {-Simplifications: unzip is simple, Liftify: range of unzip is a product -}
$[\pi_2 \circ \mathsf{unzip} \circ \mathsf{unzip}^\circ \circ \pi_1^\circ]$
$\quad = \;$ {-Catamorphism fusion: $\pi_1 \circ g = \mathsf{nil} \triangledown (\mathsf{cons} \circ (\pi_1 \times \mathsf{id})) \circ F\ \pi_1$ -}
$[(\!|\mathsf{nil} \triangledown (\mathsf{cons} \circ (\pi_1 \times \mathsf{id}))|\!) \circ (\!|\mathsf{nil} \triangledown (\mathsf{cons} \circ (\pi_2 \times \mathsf{id}))|\!)^\circ]$
$\quad = \;$ {-Definitions: map -}
$[(\mathsf{map}\ \pi_1) \circ (\mathsf{map}\ \pi_2)^\circ]$
$\quad = \;$ {-Simplifications: map converse, map fusion (see below) -}
$[\mathsf{map}\ (\pi_1 \circ \pi_2^\circ)]$
$\quad = \;$ {-Simplifications: PF Laws [11] -}
$[\mathsf{map}\ \top]$

Here, $\mathsf{map}\ f = (\!|\mathsf{in_L} \circ (\mathsf{id} + (f \times \mathsf{id}))|\!)$ is the mapping that applies $f$ to all elements of a list, whose converse and fusion properties are defined as $(\mathsf{map}\ f)^\circ = \mathsf{map}\ f^\circ$ and $\mathsf{map}\ f \circ \mathsf{map}\ g = \mathsf{map}\ (f \circ g)$. The resulting range $[\mathsf{map}\ \top]$ means that unzip always produces lists with the same length but unrelated elements. Maps of coreflexives are themselves coreflexives, and represent a special shape of invariants over recursive types. For instance, the domain of in (and dually the range of out) over map invariants can be calculated as $\delta(\mathsf{map}\ \Phi \circ \mathsf{in_L}) = \mathsf{id} + (\Phi \times \mathsf{map}\ \Phi)$.

The reasoning about anamorphisms follows the same rationale and is omitted.

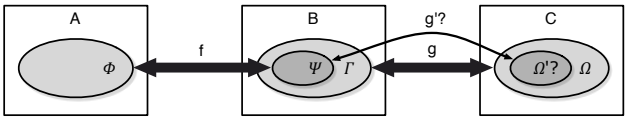
## 5 Application Scenario: Bidirectional Transformations

*Lenses* [5] are one of the most successful BX approaches. A lens, denoted by $S \trianglerighteq V$, is a bidirectional transformation between sources of type $S$ and views of type $V$ that comprises two functions: a forward transformation $\mathsf{Get} : S \to V$ that abstracts a source into a view; and a backward transformation $\mathsf{Put} : V \times S \to S$ that takes an updated view and the original source to return an updated source. A lens is *well-behaved* if it satisfies the round-tripping properties $\mathsf{Get} \circ \mathsf{Put} \subseteq \pi_1$ (denoted *acceptability* or PutGet) and $\mathsf{Put} \circ (\mathsf{Get} \vartriangle \mathsf{id}) \subseteq \mathsf{id}$ (denoted *stability* or GetPut). A lens is also said to be *total* if Get and Put are total functions. Due to these laws, the Get of a total well-behaved lenses must be a surjective function (where any value of $V$ must be the view of some source) and obviously total. For this reason, many interesting transformations (such as the split combinator) are not admissible as total well-behaved lenses since they are not surjective.

In fact, when designing a BX language there is a well-known tradeoff between the expressiveness allowed by its syntax and the robustness enforced by the totality and round-tripping laws. Some approaches [17,22] compromise the expressiveness; others ignore the totality requirement [12,20,21]; others maintain totality, but weaken the round-tripping laws [6]; some relax both totality and round-tripping laws [13,10,9,8]; finally, it also possible to avoid compromising the laws by developing a more refined type system, as proposed in the original lens framework [5]: in order to preserve totality, a powerful semantic type system with invariants was used to specify the exact domain and range of lenses,

which allowed the definition of duplication and conditional combinators as total well-behaved lenses. Unfortunately, to retain decidability in the type system, the expressiveness was still restricted by forcing composed lenses to agree not only on types but also on invariants. For example, in such a scenario, duplication could be followed by a *merge* combinator that only accepts pairs with two equal values, but not by a generic projection that works for whatever pair.

Consider the composition of two transformations $f : A_\Phi \to B_\Psi$ and $g : B_\Gamma \to C_\Omega$, where $\Psi$ is more restrictive then $\Gamma$, as depicted in the following diagram:




Since the range of $f$ and the domain of $g$ do not match, the $\mathsf{Put}$ of the composed lens would only be defined for $\Omega'$, the values in the range of $g$ for which the values produced by $\mathsf{Put}_g$ are within the range of $f$. To support such generalized composition, we wil use the techniques proposed in this paper to: 1) perform invariant inference to discover the exact range $\Omega'$ of the (global) transformation; 2) specify a non-deterministic $\mathsf{Put}$, whose optimization can be efficiently narrowed to the $\mathsf{Put}$ of a lens $g' : B_\Psi \trianglerighteq C_{\Omega'}$ that only generates values in $\Psi$.

Using the relational calculus, it is quite simple to specify a generic non-deterministic $\mathsf{Put}$ that is the largest relation that satisfies the round-tripping properties. Any transformation (i.e., a simple relation) $f : A \to B$ can be lifted to a total, well-behaved non-deterministic lens $\lfloor f \rfloor : \delta f \trianglerighteq \rho f$, with $\mathsf{Get}_f = f$ and $\mathsf{Put}_f = (\pi_2 \triangledown (f^\circ \circ \pi_1)) \circ [f^\circ]?$. This specification of $\mathsf{Put}_f$ trivially satisfies the round-tripping laws (a formal proof can be found in [11]), because it explicitly tests if the view was modified using the coreflexive $[f^\circ]$, as $(v, s) \in [\![[f^\circ]]\!] \Leftrightarrow v \equiv f\ s$. If so, it returns the original source; otherwise, it runs $\mathsf{Get}$ backwards to recover all possible sources that could have originated that view. Thus, $\mathsf{Put}_f$ is also the largest non-deterministic relation that keeps the lens well-behaved. Although trivial, the lens resulting from this lifting cannot be used in a practical BX framework. Of course, we could use the semantics of Fig. 2 to evaluate the invariants $\delta f$ and $\rho f$ and perform type-checking, but as explained in Sect. 2.1, due to composition the resulting algorithm would be undecidable. Similarly for $\mathsf{Put}_f$, we could use the semantics of Fig. 3 to perform evaluation. Even (reasonably) assuming that the user only wants a single updated source, and relying on lazy evaluation, the efficiency problem would be even worse, due to the central role played by the converse in the definition.

Both these problems can be handled by the optimizations presented in the previous sections. Our lens language allows any simple (or simplicity-preserving) PF combinator to be used to specify the forward transformation. Although unconstrained converse is not allowed (since it is not simple in general), we include the converses of the injections that are partial functions useful for "destructing sums". Thus, the domain and range of the transformations can be trivially calculated, and except particular ranges of splits, type checking is decidable.

As for the backward transformation, by applying the rules already presented in Fig. 6, the generic definition can be efficiently executed. Our language supports transformations including splits, conditionals, and converses of injections, that are not supported by most existing lens frameworks. In particular, the duplication operator $\mathsf{id} \vartriangle \mathsf{id} : A \to A \times A$ yields a lens $\lfloor \mathsf{id} \vartriangle \mathsf{id} \rfloor : \mathsf{id} \trianglerighteq [\mathsf{id}]$ (whose backward transformation only accepts pairs with equal components) that can be freely composed with other lenses irrespective of their invariants. Recursive expressions are also supported as they preserve the simplicity of their algebras.

To give an example of the performed optimizations, consider the transformation $f = \pi_1 \vartriangle \mathsf{id}$. Using the algorithm of Sect. 3.1, we can infer its range and domain and lift it to the lens $\lfloor \pi_1 \vartriangle \mathsf{id} \rfloor : \mathsf{id} \trianglerighteq [\pi_1^\circ]$ that only accepts views $(x,(y,z))$ where $x \equiv y$. Should the duplicated value be updated, the optimized backward transformation would execute as follows:

$$
\begin{aligned}
&[\![(\pi_2 \triangledown (f^\circ \circ \pi_1)) \circ [f^\circ]\,?:[\pi_1^\circ] \times \mathsf{id} \to \mathsf{id}]\!]\ ((a,(a,y)),(x,y)) \\
&\quad = \ \{\text{-Optimized semantics: } R \circ S \text{ (Fig. 6); Domain/Range (Fig. 4) -}\} \\
&\{\, c \mid b \leftarrow [\![[f^\circ]\,?:[\pi_1^\circ] \times \mathsf{id} \to \mathsf{id} + [\pi_1^\circ] \times \mathsf{id}]\!]\ ((a,(a,y)),(x,y)), \\
&\qquad c \leftarrow [\![\pi_2 \triangledown (f^\circ \circ \pi_1) : [f^\circ] + \overline{[f^\circ]} \to \mathsf{id}]\!]\ b \,\} \\
&\quad = \ \{\text{-Optimized semantics: } \Phi?;\ ((a,(a,y)),(x,y)) \in [\![\overline{[f^\circ]}]\!] \text{ -}\} \\
&\{\, c \mid c \leftarrow [\![(\pi_2 \triangledown (f^\circ \circ \pi_1)) : [f^\circ] + \overline{[f^\circ]} \to \mathsf{id}]\!]\ (\mathsf{Right}\ ((a,(a,y)),(x,y)))\,\} \\
&\quad = \ \{\text{-Optimized semantics: } R \triangledown S \text{ (Fig. 6) -}\} \\
&\{\, c \mid c \leftarrow [\![(f^\circ \circ \pi_1) : \overline{[f^\circ]} \to \mathsf{id}]\!]\ ((a,(a,y)),(x,y))\,\} \\
&\quad = \ \{\text{-Optimized semantics: } R \circ S,\ \pi_1 \text{ (Fig. 6); Domain/Range (Fig. 4) -}\} \\
&\{\, c \mid k \leftarrow [\![[\pi_1^\circ]]\!]\ (a,(a,y)),\ c \leftarrow [\![f^\circ : \mathsf{id} \to \mathsf{id}]\!]\ k\,\} \\
&\quad = \ \{\text{-}(a,(a,y)) \in [\![[\pi_1^\circ]]\!] \text{ -}\} \\
&\{\, c \mid c \leftarrow [\![(\pi_1 \vartriangle \mathsf{id})^\circ : \mathsf{id} \to \mathsf{id}]\!]\ (a,(a,y))\,\} \\
&\quad = \ \{\text{-Optimized semantics: } (R \vartriangle S)^\circ \text{ (Fig. 6); Domain/Range (Fig. 4) -}\} \\
&\{\, c \mid c \leftarrow [\![\pi_1^\circ : \mathsf{id} \to \rho\underline{(a,y)}]\!]\ a\,\} \\
&\quad = \ \{\text{-}\rho\underline{(a,y)} = \rho\underline{a} \times \rho\underline{y} = [\rho\underline{a} \circ \top \circ \rho\underline{y}]\text{; Optimized semantics: } \pi_1^\circ \text{ (Fig. 6) -}\} \\
&\{\, (a,l) \mid l \leftarrow [\![\rho\underline{a} \circ \top \circ \rho\underline{y}]\!]\ a\,\} \\
&\quad = \ \{\text{-Simplifications: PF Laws [11]; Semantics: } R \circ S,\ \Phi \text{ (Fig. 3); } a \in [\![\rho\underline{a}]\!] \text{ -}\} \\
&\{\, (a,y)\,\}
\end{aligned}
$$

Note how the invariants only need to be evaluated for the primitives. Although the semantics of $\pi_1^\circ$ is non-deterministic, $\mathsf{id}$ forces a single result. Simplifications are applied to convert the invariant over pairs into the lifted form.

Recursive specifications can also be lifted to lenses. For instance, the transformation $\mathsf{tail} \vartriangle \mathsf{length}$ can be lifted to the lens $\lfloor \mathsf{tail} \vartriangle \mathsf{length} \rfloor : \mathsf{id} \trianglerighteq [\mathsf{succ} \circ \mathsf{length}]$, whose backward transformation only accepts values such that $(l,n) \in [\![[\mathsf{succ} \circ \mathsf{length}]]\!] \Leftrightarrow \mathsf{length}\ l + 1 \equiv n$. In this case, $[\![\mathsf{Put}]\!]\ (([2,3],3),[1,2,3]) = \{[1,2,3]\}$ since the view did not change, while $[\![\mathsf{Put}]\!]\ (([2,0],3),[1,2,3]) = \{[0,2,0],[1,2,0],[2,2,0],\ldots\}$, generating all possible lists with $[2,0]$ as tail. Since $\lfloor \mathsf{unzip} \rfloor : \mathsf{id} \trianglerighteq [\mathsf{map}\ \top]$ is an injective relation, its backward transformation is simple; therefore, even if the view lists are updated, $\mathsf{Put}_{\mathsf{unzip}}$ always returns a single result that is the $\mathsf{zip}$ of the view pair.

## 6 Related Work

Although our calculus of invariants was inspired in [16], our typing rules impose a stronger restriction. In our case, a relation $R : \Phi \to \Psi$ is exactly defined only for values of $\Phi$ and only produces values in $\Psi$, while in [16] invariants represent pre- and post-conditions, i.e., $R \circ \Phi \subseteq \Psi \circ R$, meaning that there may exist values outside $\Phi$ for which $R$ is defined but whose behavior is unpredictable. It follows that all typing rules of [16] are applicable to our framework.

Functional logic programming languages like Curry [7] focus on the non-deterministic evaluation of specifications written in a functional programming style. While such languages focus on the evaluation of the specifications, our approach provides a better understanding of the program and its behavior during executing, resorting to a calculus of invariants.

The universal resolving algorithm (URA) [1] has been developed to compute the inverses of functional programs. Like our evaluation algorithm, it is complete (it lazily enumerates all possible values) but not always terminating (since recursive types may admit infinitely many values). Nevertheless, unlike in URA, we are able to optimize expressions before evaluation using the relational calculus. This allows to cut many intermediate infeasible values, making value generation for most invariants much more efficient.

Regarding BX, our framework can be seen as a domain-specific language over inductive types similar to the one for lenses over generalized trees first developed by Foster *et al* [5]. They devise a complex set-based type system with invariants to precisely define the domains for which their combinators are well-behaved. However, combining lenses requires matching on invariants rather than on types, which is too restrictive. A dual approach is followed in [6], where composition requires matching on equivalence relations that relax the lens domains.

Our application of the relational PF calculus to BX builds up from [17,18], where we have developed a language of functional PF combinators allowing only surjective transformations over inductive types. In this paper, we extend such language to support typical non-surjective combinators such as splits and injections. Unlike the data abstraction approach from [22], our lens language allows arbitrary type constructors and deconstructors without extending the language with ad-hoc primitives and surjectivity tests.

Most BX approaches rely on more standard and decidable type systems, at the cost of a more limited expressiveness [17,22], by allowing partial transformations [12,20,21] or by assuming both partiality and weaker round-tripping laws [13,10,9,8]. More closely related to our approach, some frameworks derive the backward transformations by calculation, but are less expressive than ours. In [13], Put is derived by inverting *injective* forward transformations through algebraic reasoning, while [12] bidirectionalizes a restricted first-order language (namely, without duplication) based on a notion of view-update under constant complement. They also calculate an automata that matches the exact domain of the transformations, and acts similarly to our invariants. The lens language for graph transformations proposed in [8] processes view insertions using URA, exploring all possible right inverses for the forward transformation.

## 7 Conclusion

In this paper, we have presented mechanisms for the efficient execution of expressions in a PF relational language over data-types with invariants. By defining a careful semantics that uses invariants to narrow evaluation, we attain a viable non-deterministic implementation. In retrospect, our handling of product invariants in lifted form made the difference from previous approaches to domain and range calculation, and ended up being a key component of our framework.

In the context of BX, we identify an open problem in the composition of lenses with (explicit or implicit) invariants that is responsible for the latent partiality found in most practical BX frameworks. We have proposed to alleviate this problem by modeling lenses using the relational calculus and their particular domains using invariants. Applying our proposed non-deterministic calculus and semantics, we were able to implement an expressive PF BX language that supports duplication, conditional choice and recursion patterns, whose backward transformations emerge naturally from the lens laws.

Although we are already able to handle many interesting recursive transformations, there is still a lot of room for improvement in the algorithm for recursive invariant inference. Namely, likewise the lifted form for products, we are currently researching possible normal forms for such invariants that are more amenable for calculation and optimization.

We also intend to explore mechanisms for a better control of the non-determinism through user-defined quality measures as additional invariants on the domains. In particular, we are studying ways to take advantage of the *shrink* operator proposed in [14], which narrows the output of non-deterministic PF relations, by selecting the "best" values defined by a given order.

## Acknowledgements

This work is funded by ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT - Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project FCOMP-01-0124-FEDER-020532. Nuno Macedo is sponsored by the FCT under grant SFRH/BD/69585/2010. Exchange of ideas with J. N. Oliveira (HASLab) is gratefully acknowledged.

## References

1. Abramov, S., Glück, R.: The universal resolving algorithm: Inverse computation in a functional language. In: MPC'00, LNCS, vol. 1837, pp. 187–212. Springer (2000)
2. Backhouse, R., Hoogendijk, P., Voermans, E., van der Woude, J.: A relational theory of datatypes (December 1992), draft of book in preparation, available at `http://www.cs.nott.ac.uk/~rcb/MPC/papers`

3. Bird, R., de Moor, O.: Algebra of Programming, International Series in Computer Science, vol. 100. Prentice-Hall (1996)
4. Cunha, A., Visser, J.: Transformation of structure-shy programs with application to XPath queries and strategic functions. Sci. Comput. Program. 76(6), 512–539 (2011)
5. Foster, N., Greenwald, M., Moore, J., Pierce, B., Schmitt, A.: Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. TOPLAS'07 29(3) (2007)
6. Foster, N., Pilkiewicz, A., Pierce, B.: Quotient lenses. In: ICFP'08. pp. 383–396. ACM (2008)
7. Hanus, M.: Multi-paradigm declarative languages. In: ICLP'07. LNCS, vol. 4760, pp. 45–75. Springer (2007)
8. Hidaka, S., Hu, Z., Inaba, K., Kato, H., Matsuda, K., Nakano, K.: Bidirectionalizing graph transformations. In: ICFP'10. pp. 205–216. ACM (2010)
9. Hu, Z., Mu, S.C., Takeichi, M.: A programmable editor for developing structured documents based on bidirectional transformations. Higher-Order and Symbolic Computation 21(1–2), 89–118 (2008)
10. Liu, D., Hu, Z., Takeichi, M.: Bidirectional interpretation of XQuery. In: PEPM'07. pp. 21–30. ACM (2007)
11. Macedo, N., Pacheco, H., Cunha, A.: Relations as executable specifications: Taming partiality and non-determinism using invariants. Technical Report TR-HASLab:03:2012, University of Minho (Jul 2012), available at `http://www.di.uminho.pt/~nfmmacedo/publications/invariants-tr.pdf`
12. Matsuda, K., Hu, Z., Nakano, K., Hamana, M., Takeichi, M.: Bidirectionalization transformation based on automatic derivation of view complement functions. In: ICFP'07. pp. 47–58. ACM (2007)
13. Mu, S.C., Hu, Z., Takeichi, M.: An algebraic approach to bi-directional updating. In: APLAS'04. LNCS, vol. 3302, pp. 2–20. Springer (2004)
14. Mu, S.C., Oliveira, J.N.: Programming from galois connections. In: RAMiCS'11. LNCS, vol. 6663, pp. 294–313. Springer (2011)
15. Oliveira, J.N.: Transforming data by calculation. In: GTTSE'07. LNCS, vol. 5235, pp. 134–195. Springer (2008)
16. Oliveira, J.N.: Extended static checking by calculation using the pointfree transform. In: LerNet ALFA Summer School 2008, LNCS, vol. 5520, pp. 195–251. Springer (2009)
17. Pacheco, H., Cunha, A.: Generic point-free lenses. In: MPC'10. LNCS, vol. 6120, pp. 331–352. Springer (2010)
18. Pacheco, H., Cunha, A.: Calculating with lenses: Optimising bidirectional transformations. In: PEPM'11. pp. 91–100. ACM (2011)
19. Schmidt, G.: Relational Mathematics. No. 132 in Encyclopedia of Mathematics and its Applications, Cambridge University Press (2010)
20. Voigtländer, J.: Bidirectionalization for free! (Pearl). In: POPL'09. pp. 165–176. ACM (2009)
21. Voigtländer, J., Hu, Z., Matsuda, K., Wang, M.: Combining syntactic and semantic bidirectionalization. In: ICFP'10. pp. 181–192. ACM (2010)
22. Wang, M., Gibbons, J., Matsuda, K., Hu, Z.: Gradual refinement: Blending pattern matching with data abstraction. In: MPC'10. pp. 397–425. LNCS, Springer (2010)