Extended Static Checking by Strategic Rewriting of Pointfree Relational Expressions

Claudia Mónica Necco[1,2], José Nuno Oliveira[2], and Joost Visser[2]

[1] Dep. de Informática, Universidad de San Luis, San Luis, Argentina
[2] DI-CCTC, Universidade do Minho, Braga, Portugal

Abstract. Binary relational algebra provides semantic foundations for major areas of computing, such as database design, state-based specification, and functional programming. Remarkably, static checking support in these areas fails to exploit the full semantic content of relations. In particular, properties such as the simplicity or injectivity of relations are not statically enforced in operations that manipulate relations, such as database queries, state transitions, or composition of functional components.
We describe how a *pointfree* treatment of relations, their properties, their operators, and the laws that govern them can be captured in a type-directed strategic rewriting system for transformation of relational expressions. This rewriting tool can be used to simplify relational proof obligations and ultimately reduce them to tautologies. We demonstrate how such reductions provide extended static checking (ESC) for design contraints commonly found in software modeling and development.

Key words: Binary relations, Strategic term rewriting, Theorem proving, Extended static checking, Pointfree program transformation

1 Introduction

Software design is error-prone. The negative impact of programming errors on software productivity can be limited by catching them early. Static checkers (eg. syntax and type checkers) are tools which catch errors at compile-time, ie. before running the program. Examples of such errors are unmatched parentheses (wrong syntax) and adding integers to booleans (wrong types). Errors such as null dereferencing, division by 0, and array bound overflow, are not caught by standard static checking; detecting their presence requires extensive testing, and if their presence can not be excluded with certainty, they must be handled at run-time via exception mechanisms.

Software formalists will argue that error checking in the coding phase is too late: first a formal model should be written, queried, reasoned about, and possibly animated (using eg. a symbolic interpreter). Formal modeling relies on "rich" datatypes such as finite mappings, finite sequences, and recursive data structures, which abstract from much of the complexity found in common imperative programming languages (eg. pointers, loop boundaries). However, such

rich structures are not able to capture *all* properties, meaning that additional constraints need to be added to models such as invariants (attached to types) and pre-conditions (attached to operations). Checking such constraints is once again a process which falls outside standard static type-checking, leading to a so-called *dynamic* type checking process, typical of model animation tools such as the VDMTools system [7].

Static checking of formal models involving such constraints is a complex process, relying on generation and discharge of proof obligations [11]. While proof obligations can be generated mechanically, their discharge is in general above the *decidability ceiling* in requiring full-fledged formal verification (theorem proving) [17]. Between these two extremes of standard, cheap, decidable static checking and costly theorem proving, *extended static checking* (ESC) [18, 8] aims to catch more errors at compile-time at the relatively moderate cost of adding annotations to the code which record *design decisions* which were lost throughout the programming process (if ever explicitly recorded).

Extended static checking tools have been developed for imperative programming languages such as Modula-3 (ESC/Modula-3 [18]) and Java (ESC/Java [8]). At the heart of these tools we find a verification condition generator and the Simplify theorem prover [6]. Verification conditions are predicates in first-order logic which are computed in weakest precondition style. Theorem proving is performed by a combination of techniques, including SAT solvers, matching algorithms, and heuristics to guide proof search.

In the current paper we follow the spirit of this approach but intend to apply it much earlier in the design process: we wish to perform extended static checking for formal modeling languages such as VDM, Z, and Alloy. Since the rich data structures of these modeling languages already preclude by construction the occurrence of errors such as null pointers and array bound overflow, we will aim to catch errors higher on the semantic scale. A pertinent example is the *finite mapping* data type (also called *finite partial function*), which covers a number of interesting situations in both formal modeling (eg. Z and VDM) and programming (eg. SQL). For instance, relational tables with a primary key can be modeled as finite mappings. `UPDATE` and `INSERT` are examples of operations which in general put primary keys at risk, something that model animation tools and database systems can only check at *run-time*. The preservation of primary key relationships by these operations will be one of the targets of our extended static checks.

But the main novelty of our approach resides in the chosen method of proof construction. In our approach, first-order proof obligations are subject to the *PF-transform* [24] before they are reasoned about. This transformation eliminates quantifiers and bound variables and reduces complex formulæ to algebraic expressions which are more agile to calculate with (see Fig. 1 for details). In fact, as we will show in this paper, proof calculation can be carried out on pointfree relational algebra expressions by strategic term rewriting [26, 15, 16, 14] .

In Section 2 we will motivate our extended static checking approach with a small modeling example. In Section 3 we recapitulate binary relation theory

ϕ	$PF\ \phi$
$\langle \exists\ a\ ::\ b\ R\ a\ \wedge\ a\ S\ c \rangle$	$b(R \cdot S)c$
$\langle \forall\ a, b\ ::\ b\ R\ a \Rightarrow b\ S\ a \rangle$	$R \subseteq S$
$\langle \forall\ a\ ::\ a\ R\ a \rangle$	$id \subseteq R$
$\langle \forall\ x\ ::\ x\ R\ b \Rightarrow x\ S\ a \rangle$	$b(R \setminus S)a$
$\langle \forall\ c\ ::\ b\ R\ c \Rightarrow a\ S\ c \rangle$	$a(S\ /\ R)b$
$b\ R\ a\ \wedge\ c\ S\ a$	$(b, c)\langle R, S \rangle a$
$b\ R\ a\ \wedge\ d\ S\ c$	$(b, d)(R \times S)(a, c)$
$b\ R\ a\ \wedge\ b\ S\ a$	$b\ (R \cap S)\ a$
$b\ R\ a \vee b\ S\ a$	$b\ (R \cup S)\ a$
$(f\ b)\ R\ (g\ a)$	$b(f^\circ \cdot R \cdot g)a$
$b = a$	$b\ id\ a$
True	$b\ \top\ a$
False	$b\ \bot\ a$

In analogy to the well-known Laplace transform [12], the PF-transform takes expressions from a mathematical problem space, in this case first order logic formulæ, into a mathematical solution space, in this case relational algebra expressions [2]. The PF-transform eliminates quantifiers and bound variables (so-called *points*), resulting in a pointfree notation which is more agile to calculate with.

Fig. 1. The PF-transform.

which can be used to capture the semantics of models with rich data structures and their operations. In Sections 4 and 5 we will demonstrate how the algebraic laws of the theory can be harnessed in a strategic term rewriting system, implemented in the functional programming language Haskell. In Section 6 we revisit the model operations of our example to show how our rewriting system is capable of generating the appropriate proof obligations and simplify or discharge them. Section 7 discusses related work and Section 8 concludes.

2 Motivating example

The UML class diagram in Fig. 2 depicts a simplified model of a system for trading non-consumable (uniquely identifiable) items. A user can put an item for sale for a given price, and other users can express their interest in these items for a price they are willing to pay. If a match between a seller and a buyer is established, this leads to a deal with an agreed price.

The specification of queries, predicates, and transformations on this model may present some pitfalls. Suppose the following operations are desired:

$$
\begin{array}{ll}
\mathit{listWantedItems} & :: \mathit{Wanted} \rightarrow \mathit{Map\ Iid\ Price} \\
\mathit{wantedItemsAreForSale} & :: \mathit{Wanted} \rightarrow \mathit{ForSale} \rightarrow \mathit{Bool} \\
\mathit{putBatchForSale} & :: (\mathit{Uid}, \mathit{Map\ Iid\ Price}) \rightarrow \mathit{ForSale} \rightarrow \mathit{ForSale} \\
\mathit{settleDeal} & :: (\mathit{Iid}, \mathit{Uid}, \mathit{Price}) \rightarrow \mathit{Deal} \rightarrow \mathit{Deal}
\end{array}
$$

The *listWantedItems* query produces a map of item identifiers together with the price that has been offered for them. The *wantedItemsAreForSale* predicate tests whether each item listed in *Wanted* is actually for sale, thus testing referential integrity. The transformation *putBatchForSale* adds a batch of items belonging to a given user to the *ForSale* relation. The *settleDeal* transformation adds an entry to the *Deal* collection.

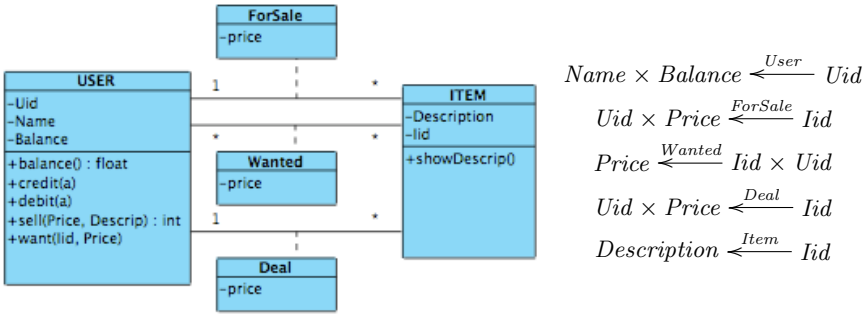

Fig. 2. Simplified UML model of a trading system and the corresponding binary relational model (explained in Section 3). This model is loosely based on a formal specification for a real estate exchange market, which has been developed for the PortoDigital Consortium.

When specifying these operations, the designer could benefit from the feedback of an extended static checker. For example, the checker should tell him that *listWantedItems* should only return a map if the *Wanted* collection contains no two offers for the same item with different prices. Rather than adding a precondition to that effect, he will likely decide to change the return type to a general relation *Rel Iid Price* or, equivalently, to $Set\ (Iid, Price)$. In case of the *settleDeal* operation, to ensure that pre-existing deals do not get lost the checker should indicate that a precondition is needed that either no deal yet exists for the given item, or that it exists with the same buyer identifier and price.

3 Overview of relation theory

In this section we provide a brief introduction to the theory of binary relations [2].

Relations. Let $B \xleftarrow{R} A$ denote a binary relation R on datatypes A (source) and B (target). We write bRa to mean that pair (b, a) is in R. The underlying partial order on relations is written $R \subseteq S$, meaning that S is more defined or less deterministic than R, that is, $R \subseteq S \equiv bRa \Rightarrow bSa$ for all a, b. $R \cup S$ denotes the union of two relations and $\top$ is the largest relation of its type. Its dual is $\bot$, the smallest such relation. The identity id relates every element to itself. Equality on relations can be established by $\subseteq$-antisymmetry: $R = S \equiv R \subseteq S \ \wedge \ S \subseteq R$.

Relations are combined by three basic operators: composition ($R \cdot S$), converse (R°) and meet ($R \cap S$). R° is such that $a(R^\circ)b$ iff bRa holds. Meet corresponds to set-theoretical intersection and composition is defined in the usual way: $b(R \cdot S)c$ holds wherever there exists some mediating a such that $bRa \ \wedge \ aSc$.

Coreflexives. An endo-relation $A \xleftarrow{R} A$ is referred to as *reflexive* iff $id \subseteq R$ holds, and as *coreflexive* iff $R \subseteq id$ holds. Coreflexive relations, which we

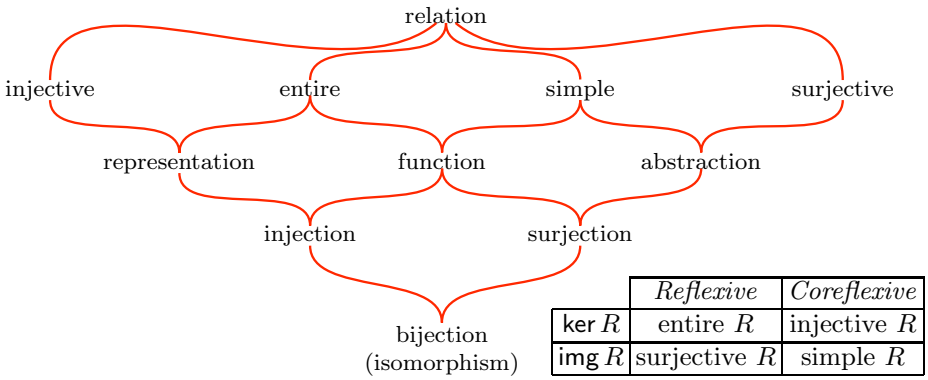

Fig. 3. Binary relation taxonomy

denote by Greek letters (Φ, Ψ, etc.), are fragments of the identity relation that model predicates or sets. A predicate p is modeled by the coreflexive $[\![p]\!]$ with $b[\![p]\!]a \equiv (b = a) \ \wedge \ (p\ a)$, that is, the relation that maps every a which satisfies p onto itself. Negation is modeled by $\neg\Phi = id - \Phi$. A set $S \subseteq A$ is modeled by $[\![\lambda a.a \in S]\!]$, that is $b[\![S]\!]a \equiv (b = a) \ \wedge \ a \in S$.

Taxonomy. To establish a fundamental taxonomy of relations (illustrated in Fig. 3), let us first define the *kernel* of a relation, $\mathsf{ker}\, R = R^\circ \cdot R$ and its dual, $\mathsf{img}\, R = \mathsf{ker}\,(R^\circ) = R \cdot R^\circ$, called the *image* of R. A relation R is said to be *entire* (or total) iff its kernel is reflexive; and *simple* (or functional) iff its image is coreflexive. Simple relations are denoted with capital letters M, N, etc. Dually, R is *surjective* iff $\mathsf{img}\, R$ is reflexive, and R is *injective* iff $\mathsf{ker}\, R$ is coreflexive.

Functions. As the taxonomy indicates, a relation is a *function* iff it is both simple and entire. Functions will be denoted by lowercase letters (f, g, etc.) and are such that bfa means $b = f\ a$. The constant function which maps every value of its domain to the value k is denoted by $\underline{k}$. We write ! (read "bang") for the constant function that targets the unit domain 1.

Algebraic properties. A rich set of algebraic properties is available for the various operators of relational algebra [2], of which a small sample is listed in Table 1. Of particular interest for the current paper are the various *shunting laws*. They allow the 'shunting' of relations (functions and simple relations in the listed cases) from one side of the inclusion to the other, similar to the shunting rules we learned in high school, such as eg. $x - y \leqslant z \equiv x \leqslant z + y$. The utility of such laws will become evident below.

4 Rewriting relational expressions and propositions

The various algebraic laws of binary relations presented in Table 1 can be harnessed into type-safe, type-directed rewriting systems for normalization of re-

Table 1. Some laws of the binary relational algebra.

$(R \cdot S) \cdot T = R \cdot (S \cdot T)$	*comp_assoc*
$(R \cdot S)^\circ = S^\circ \cdot R^\circ$	*inv_comp*
$(R^\circ)^\circ = R$	*inv_inv*
$(R \cup S)^\circ = R^\circ \cup S^\circ$	*inv_union*
$R \cdot id = R \quad \wedge \quad id \cdot R = R$	*comp_id*
$\Phi^\circ = \Phi$	*corefl_symm*
$R \cdot \bot = \bot \quad \wedge \quad \bot \cdot R = \bot$	*comp_empty*
$R \cdot \delta\ R = R$	*dom_elim*
$\underline{k} \cdot R = \underline{k} \cdot \delta\ R$	*const_fusion*
$R \cdot \neg\ (\delta\ R) = \bot$	*not_dom_cancel*
$R \cdot f^\circ \subseteq S \equiv R \subseteq S \cdot f$	*shunt_fun_inv*
$f \cdot R \subseteq S \equiv R \subseteq f^\circ \cdot S$	*shunt_fun*
$R \cdot M^\circ \subseteq S \equiv R \cdot \delta\ M \subseteq S \cdot M$	*shunt_map_inv*
$M \cdot R \subseteq S \equiv \delta\ M \cdot R \subseteq M^\circ \cdot S$	*shunt_map*
$\mathsf{img}\ R = R \cdot R^\circ$	*img_def*
$\mathsf{ker}\ R = R^\circ \cdot R$	*ker_def*

lational expressions as well as for derivation of proofs for propositions involving relational expressions. The functional programming language Haskell allows a straightforward and type-safe implementation of such rewriting systems, in particular due to its support for *generalized* abstract datatypes (GADTs) and existential types. We will explain these features as we use them.

Representation of types. We employ a representation of types which is a classical example of a GADT [9]:

```
data Type a where
  One    :: Type ()
  Int    :: Type Int
  Bool   :: Type Bool
  String :: Type String
  List   :: Type a → Type [a]
  Prod   :: Type a → Type b → Type (a, b)
  Either :: Type a → Type b → Type (Either a b)
  Rel    :: Type a → Type b → Type (Rel a b)
  ...
type Rel a b   -- abstract
```

Note that the type a that parameterizes the type-representation *Type a* is instantiated differently in each constructor. This is precisely the difference between a GADT and a common parameterized datatype, where the parameters in the result type are unrestricted in all constructors. In the definition of *Type a*, the parameter a of each constructor is restricted exactly to the type that the constructor represents. For example, *Rel* (*Prod Int String*) *One* represents the type *Rel* $(Int, String)$ $()$. Note that the type-constructor for relations can remain

abstract for the purposes of our paper. It is possible to define a class with representable types, with instances trivial to define, eg. for integers and relations:

```
class Typeable a where typeof :: Type a
instance Typeable Int where typeof = Int
instance (Typeable a, Typeable b) ⇒ Typeable (Rel a b)
  where typeof = Rel typeof typeof
```

Below we will use the *typeof* function to define smart constructors for relational algebra expressions. Type equality tests are performed by induction on type representations:

```
teq :: Type a → Type b → Maybe (Equal a b)
teq Int Int = Just Eq
teq (Rel a b) (Rel c d) = do Eq ← teq a c; Eq ← teq b d; return Eq
...
teq _ _ = Nothing

data Equal a b where Eq :: Equal a a
```

The constructor *Eq* of the GADT *Equal* can be seen as a proof token of the equality of types a and b [25].

Representation of relational expressions and propositions. A GADT is also used to represent the entities that will be subjected to rewriting, *viz.* relational expressions and propositions.

```
data R r where
  Rvar   :: String → Set Prop → R x
  Rid    :: R (Rel a a)
  Rempty :: R (Rel a b)
  Rtop   :: R (Rel a b)
  Rnot   :: R (Rel a b) → R (Rel a b)
  Rcomp  :: Type b → R (Rel b c) → R (Rel a b) → R (Rel a c)
  Rker   :: Type b → R (Rel a b) → R (Rel a a)
  Rimg   :: Type a → R (Rel a b) → R (Rel b b)
  Rdom   :: Type b → R (Rel a b) → R (Rel a a)
  Rrng   :: Type a → R (Rel a b) → R (Rel b b)
  Rinter :: R (Rel a b) → R (Rel a b) → R (Rel a b)
  Runion :: R (Rel a b) → R (Rel a b) → R (Rel a b)
  Rincl  :: Type r → R r → R r → R Bool
  Rand   :: R Bool → R Bool → R Bool
```

Constructors with (implicitly) existentially quantified type variables, such as b in *Rcomp*, receive an additional type representation as argument. These arguments serve to give access to the representation of the type hidden by existential quantification, such that rewriting can be made sensitive to these types.

Note that the constructor for variables (*Rvar*) has not only a *String* as argument to store the variable name, but also a set of properties of type *Prop* which serve to declare the location of the denoted relation in the relational taxonomies.

$$\textbf{data}\ \mathit{Prop} = \mathit{Simple} \mid \mathit{Entire} \mid \mathit{Injective} \mid \mathit{Surjective} \mid \mathit{Coreflexive}$$

The last property only makes sense for endorelations, and is simply ignored when dealing with relations that have different argument and result types.

The various properties declared on relational variables propagate through relational operators. For example, the composition of two surjective relations is surjective, and the inverse of an injective relation is simple. This gives rise to predicates on relations that inductively check their properties. For example:

$$\begin{aligned}
&\mathit{isSimple} :: \mathit{Type}\ (\mathit{Rel}\ a\ b) \to R\ (\mathit{Rel}\ a\ b) \to \mathit{Bool}\\
&\mathit{isSimple}\ _\ \mathit{Rid} &&= \mathit{True}\\
&\mathit{isSimple}\ _\ (\mathit{Rvar}\ _\ ps) &&= \mathit{Set.member}\ \mathit{Simple}\ ps\\
&\mathit{isSimple}\ (\mathit{Rel}\ a\ b)\ (\mathit{Rinv}\ r) &&= \mathit{isInjective}\ (\mathit{Rel}\ b\ a)\ r\\
&\mathit{isSimple}\ (\mathit{Rel}\ a\ c)\ (\mathit{Rcomp}\ b\ s\ r) &&= \mathit{isSimple}\ (\mathit{Rel}\ a\ b)\ r \wedge \mathit{isSimple}\ (\mathit{Rel}\ b\ c)\ s\\
&\ldots\\
&\mathit{isSimple}\ _\ _ &&= \mathit{False}
\end{aligned}$$

Similar predicates are supplied for the remaining properties.

Using the *typeof* function defined above, we can define some smart constructors that allow us to avoid providing type arguments manually (first column):

$$\begin{array}{lll}
\delta\ r = \mathit{Rdom}\ \mathit{typeof}\ r & r^{\circ} = \mathit{Rinv}\ r & \mathit{id} = \mathit{Rid}\\
r \cdot s = \mathit{Rcomp}\ \mathit{typeof}\ r\ s & r \cup s = \mathit{Runion}\ r\ s & \top = \mathit{Rtop}\\
r \subseteq s = \mathit{Rincl}\ \mathit{typeof}\ r\ s & p \wedge q = \mathit{Rand}\ p\ q & \neg\ r = \mathit{Rnot}\ r
\end{array}$$

The other columns contain further pretty-printed shorthands. Though type-set[3] as mathematical symbols, all these are operators in actual Haskell code.

Type-directed and property-aware rewriting rules. We build rewrite systems from rewrite combinators, following the paradigm of *strategic programming* [26, 15, 16, 14]. Individual rewrite rules as well as the rewrite systems composed from them, are represented by monadic functions of the following type:

$$\textbf{type}\ \mathit{Rule} = \forall r\ .\ \mathit{Type}\ r \to R\ r \to \mathit{Rewrite}\ (R\ r)$$

Thus, a rule takes a relational expression (or proposition) of type r into a new expression of the same type. The type-representation passed as first argument allows rules to make type-based rewriting decisions. *Rewrite* is a backtracking monad, based on the list monad, that supports the following operations:[4]

$$\begin{aligned}
&\mathit{mzero} :: \mathit{Rewrite}\ a && \text{-- zero}\\
&\mathit{mplus} :: \mathit{Rewrite}\ a \to \mathit{Rewrite}\ a \to \mathit{Rewrite}\ a && \text{-- plus (distributive)}\\
&\mathit{mcatch} :: \mathit{Rewrite}\ a \to \mathit{Rewrite}\ a \to \mathit{Rewrite}\ a && \text{-- left-catch}
\end{aligned}$$

The *mplus* combinator performs full backtracking search into its two arguments, while *mcatch* only explores the second argument if the first one fails. In addition, our *Rewrite* monad offers the capability of generating rewrite traces, of which we will see examples below.

Here is an encoding of the *inv_comp* law, applied in the left-to-right direction:

[3] We use the excellent `lhs2TeX` preprocessor for LaTeX by Andres Loeh and Ralf Hinze.
[4] See also: http://haskell.org/hawiki/MonadPlus.

```
inv_comp :: Rule
inv_comp _ (Rinv (Rcomp t r r')) = return (Rcomp t (Rinv r') (Rinv r))
inv_comp _ _                     = mzero   -- catch all
```

This simple rule is not type-directed, so the first argument is ignored (indicated by _). Pattern matching is performed on a relational expression and, on successful match, a resulting expression is returned. Otherwise failure of the rule is indicated by *mzero*. For brevity, we omit such catch-all cases in the rules below.

The *const_fusion* rule provides an example of rewriting directed by properties:

```
const_fusion :: Rule
const_fusion t@(Rel a c) (Rcomp b s@(Rvar n p) r)
  | isEntire (Rel b c) s ∧ isConstant s ∧ (¬ (isCoreflexive a b r))
  = success "const_fusion" t (Rcomp a (Rvar n p) (Rdom b r))
```

The rule works on a composition and, if the first argument s is entire and constant as required by the guarding predicate, then it replaces the second argument r by its domain. When r is coreflexive, the rule does not trigger, because the domain of a coreflexive is that relation itself.

An example of a rewrite rule on the level of relational propositions is offered by the shunting rule for functions:

```
shunt_fun_inv :: Rule
shunt_fun_inv _ (Rincl (Rel a c) (Rcomp b x (Rinv f)) y)
  | isEntire (Rel b a) f ∧ isSimple (Rel b a) f
  = return (Rincl (Rel b c) x (Rcomp a y f))
```

Note the use of a guarding predicate that tests whether the relation f is indeed a function (entire and simple).

Combinators for strategic rewriting. To compose rewriting systems out of individual rewrite rules, we employ the following set of rule combinators known from strategic term rewriting:

```
nop :: Rule                                -- identity rule
(▷) :: Rule → Rule → Rule                  -- sequential composition
(⊕) :: Rule → Rule → Rule                  -- choice (based on mplus)
(⊘) :: Rule → Rule → Rule                  -- choice (based on mcatch)
all :: Rule → Rule                         -- map on all children
one :: Rule → Rule                         -- map on one child
run :: Rule → R r → (R r, Derivation)      -- top-level application
```

The implementation of each of these combinators is straightforward, and omitted here for brevity. The top-level application function *run* takes the result of rewriting and the derivation (proof trace) out of the *Rewrite* monad; in case of failure it returns the original term and an empty derivation.

Using the basic rule combinators, more sophisticated ones can be defined:

```
many r      = (r ▷ (many r)) ⊘ nop   -- repeat until failure
once r      = r ⊘ one (once r)       -- apply once, at any depth
innermost r = all (innermost r) ▷ ((r ▷ innermost r) ⊘ nop)
```

The derived combinator *innermost* performs exhaustive rewrite rule application according to the leftmost innermost rewriting strategy.

5 Rewriting strategies

Having defined individual rules and rule combinators, we can proceed to the composition of rewrite systems for various purposes.

Normalization of relational expressions. The following definitions express that a relational expression can be normalized by exhaustive application of individual association, desugaring, and normalization rules:

$$
\begin{array}{ll}
\mathit{normalize} :: \mathit{Rule} & \text{-- until fixpoint is reached} \\
\mathit{normalize} = \mathit{innermost}\ (\mathit{comp\_assoc} \oslash \mathit{normalize1}) & \\
\mathit{normalize1}, \mathit{desugar1}, \mathit{simplify1} :: \mathit{Rule} & \text{-- single step} \\
\mathit{normalize1} = \mathit{desugar1} \oslash \mathit{simplify1} & \\
\mathit{desugar1} \quad = \mathit{ker\_def} \oslash \mathit{img\_def} \oslash \ldots & \\
\mathit{simplify1} \quad = \mathit{inv\_comp} \oslash \mathit{inv\_inv} \oslash \mathit{comp\_id} \oslash \mathit{comp\_empty} \oslash \mathit{dom\_elim} \oslash & \\
\qquad \mathit{corefl\_symm} \oslash \mathit{const\_fusion} \oslash \mathit{not\_dom\_cancel} \oslash \ldots &
\end{array}
$$

We use the convention of postfixing the names of single-step rule combinations with 1 in order to distinguish them from rule combinations that rewrite repetitively until a fixpoint is reached. Note that the *comp_assoc* rule is employed to bring relational compositions into left-associative form. Since the normalization rules together form a confluent and terminating rewrite system, the left-catching combinator $\oslash$ is sufficient to combine them — no need for backtracking.

For example, the following derivation is constructed when applying the *normalize* strategy to $(N \cdot (\neg\ (\delta\ N))^\circ \cdot M^\circ)^\circ$, where N and M are simple relations:

$$
\begin{array}{ll}
 & (N \cdot (\neg\ (\delta\ N))^\circ \cdot M^\circ)^\circ \\
= & \{\, \mathit{corefl\_symm} \,\} \\
 & (N \cdot (\neg\ (\delta\ N)) \cdot M^\circ)^\circ \\
= & \{\, \mathit{not\_dom\_cancel} \,\} \\
 & (\bot \cdot M^\circ)^\circ \\
= & \{\, \mathit{comp\_empty} \,\} \\
 & \bot^\circ \\
= & \{\, \mathit{corefl\_symm} \,\} \\
 & \bot
\end{array}
$$

This normalization proof trace demonstrates that the original expression is equal to $\bot$. (Recall that proof traces are generated by our *Rewrite* monad.)

Deriving proofs and proof obligations. We define a more sophisticated strategy to simplify or dispatch proof obligations:

$$
\begin{array}{l}
\mathit{simplify} = \mathit{normalize} \triangleright \mathit{all\_and}\ \mathit{process\_conjunct} \triangleright \mathit{innermost}\ \mathit{and\_true} \\
\quad \textbf{where} \\
\qquad \mathit{process\_conjunct} \quad = (\mathit{shunt\_conjunct} \oplus \mathit{strengthen\_conjunct}) \oslash \mathit{nop} \\
\qquad \mathit{shunt\_conjunct} \quad = \mathit{shunt} \triangleright \mathit{simplify}
\end{array}
$$

$$
\begin{aligned}
&\quad \mathit{strengthen\_conjunct} = \mathit{strengthen} \triangleright \mathit{simplify} \triangleright \mathit{qed} \\
&\mathit{shunt} = (\mathit{shunt\_fun\_inv} \oslash \mathit{shunt\_map\_inv}) \oplus (\mathit{shunt\_fun} \oslash \mathit{shunt\_map}) \\
&\mathit{strengthen} = \mathit{corefl\_cancel} \\
&\mathit{all\_and} :: \mathit{Rule} \to \mathit{Rule} \quad \text{-- apply argument rule on all conjuncts} \\
&\mathit{qed} \quad :: \mathit{Rule} \qquad\quad \text{-- test whether the current expression is } \mathit{True}
\end{aligned}
$$

The initial application of *normalize* brings a given proposition into conjunctive normal form, where each conjunct is a normalized relational inclusion. The *all_and* combinator applies *process_conjunct* to all conjuncts. After processing each conjunct separately, *and_true* ($p \wedge \mathit{True} \equiv \mathit{True} \wedge p \equiv p$) is applied to absorb the propositions that have been rewritten to *True*. The processing of each conjunct makes a non-deterministic choice (using the the backtracking operator $\oplus$) between starting with a shunting step (*shunt_conjunct*) or starting with a strengthening step (*strengthen_conjunct*); the conjunct is left unchanged if neither is possible (*nop*). When starting with shunting, the choice between shunting a left-composed relation or shunting a right-composed converse of a relation is again made non-deterministically (*shunt*). After the shunting step, a recursive call is made to the overal *simplify* strategy. When starting with strengthening, the subsequent recursive call to *simplify* is required to lead to a full proof (*qed*), since we are interested in strengthened propositions only for the purpose of discharging proof obligations.

Selection of results. The use of backtracking entails that several results may be obtained or the same result through different derivations. Our *Rewrite* monad delivers those results and derivations in a lazy stream, due to the underlying list monad. To float the first successful derivation to the front, we employ the following helper function:

$$
\begin{aligned}
&\mathit{truthfirst} :: [(R\ \mathit{Bool}, \mathit{Derivation})] \to [(R\ \mathit{Bool}, \mathit{Derivation})] \\
&\mathit{truthfirst}\ \mathit{results} = \mathit{first\_true} +\!\!+ \mathit{none\_true} \\
&\quad \textbf{where} \\
&\qquad (\mathit{none\_true}, \mathit{first\_true}) = \mathit{break}\ \mathit{isProven}\ \mathit{results} \\
&\qquad \mathit{isProven}\ (\mathit{Rvar}\ \texttt{"True"}\ \_, \_) = \mathit{True} \\
&\qquad \mathit{isProven}\ \_ \qquad\qquad\quad\ = \mathit{False}
\end{aligned}
$$

Now, if we select the first element from the stream of reordered results, we will obtain the first derivation that results in *True*, if it exists. Otherwise the first derivation that leads to a simplified proposition is returned. Lazy evaluation ensures that we will not compute any of the derivations that occurred in the original stream after the first *True* result.

6 Application scenarios

We now explain how our rewriting system can be used in concrete scenarios, such as the ones in our motivation example (Section 2).

List wanted items. The operation *listWantedItems* can be specified in binary relational terms as $\mathit{listWantedItems} = \mathit{Wanted} \cdot \pi_1^\circ$, where π_1 is the first projection

on pairs, ie. $\pi_1\ (a, b) = a$. Since this operation is specified to produce a finite map (thus simple), it gives rise to proof obligation $\mathsf{img}\,(\mathit{Wanted} \cdot \pi_1^\circ) \subseteq \mathit{id}$, which in turn leads to the following derivation when applying our *simplify* strategy:

$$
\begin{array}{ll}
& \mathsf{img}\,(\mathit{Wanted} \cdot \pi_1^\circ) \subseteq \mathit{id} \\
\equiv & \{\, \mathit{img\_def} \,\} \\
& \mathit{Wanted} \cdot \pi_1^\circ \cdot (\mathit{Wanted} \cdot \pi_1^\circ)^\circ \subseteq \mathit{id} \\
\equiv & \{\, \mathit{inv\_comp} \,\} \\
& \mathit{Wanted} \cdot \pi_1^\circ \cdot (\pi_1^\circ)^\circ \cdot \mathit{Wanted}^\circ \subseteq \mathit{id} \\
\equiv & \{\, \mathit{inv\_inv} \,\} \\
& \mathit{Wanted} \cdot \pi_1^\circ \cdot \pi_1 \cdot \mathit{Wanted}^\circ \subseteq \mathit{id} \\
\equiv & \{\, \mathit{shunt\_map\_inv} \,\} \\
& \mathit{Wanted} \cdot \pi_1^\circ \cdot \pi_1 \cdot \delta\ \mathit{Wanted} \subseteq \mathit{id} \cdot \mathit{Wanted} \\
\equiv & \{\, \mathit{comp\_id} \,\} \\
& \mathit{Wanted} \cdot \pi_1^\circ \cdot \pi_1 \cdot \delta\ \mathit{Wanted} \subseteq \mathit{Wanted} \\
\equiv & \{\, \mathit{shunt\_map} \,\} \\
& \delta\ \mathit{Wanted} \cdot \pi_1^\circ \cdot \pi_1 \cdot \delta\ \mathit{Wanted} \subseteq \mathit{Wanted}^\circ \cdot \mathit{Wanted}
\end{array}
$$

Using the PF-transform of Fig. 1, the last expression can be written as:

$$
\begin{array}{l}
\forall x, y\ .\ x\ \in\ \delta\ \mathit{Wanted} \wedge y\ \in\ \delta\ \mathit{Wanted} \wedge \pi_1\ (x) = \pi_1\ (y) \\
\qquad \Rightarrow \mathit{Wanted}\ (x) = \mathit{Wanted}\ (y)
\end{array}
$$

This formula expresses that *listWantedItems* only returns a finite map if the *Wanted* collection contains no two offers for the same item with different prices. This feedback should lead the designer to broaden the output type of the operation to general binary relations.

Settle deal. We can define $\mathit{settleDeal}\ (i, u, p) = \mathit{Deal} \cup \underline{(u, p)} \cdot \underline{i}^{\,\circ}$. Checking the simplicity of its output gives rise to the following derivation (condensed):

$$
\begin{array}{ll}
& \mathsf{img}\,(\mathit{Deal} \cup \underline{(u, p)} \cdot \underline{i}^{\,\circ}) \subseteq \mathit{id} \\
\equiv & \{\, \mathit{img\_def}, \mathit{various\ union\ laws} \,\} \\
& \mathit{Deal} \cdot \mathit{Deal}^\circ \subseteq \mathit{id} \wedge \mathit{Deal} \cdot \underline{i} \cdot \underline{(u, p)}^{\,\circ} \subseteq \mathit{id} \wedge \underline{(u, p)} \cdot \underline{i}^{\,\circ} \cdot \mathit{Deal}^\circ \subseteq \mathit{id} \\
& \qquad \wedge\ \underline{(u, p)} \cdot \top \cdot \underline{(u, p)}^{\,\circ} \subseteq \mathit{id} \\
\equiv & \{\, \mathit{various\ shunting\ laws}, \mathit{dom\_elim} \,\} \\
& \delta\ \mathit{Deal} \cdot \underline{i} \subseteq \mathit{Deal}^\circ \cdot \underline{(u, p)} \wedge \underline{i}^{\,\circ} \cdot \delta\ \mathit{Deal} \subseteq \underline{(u, p)}^{\,\circ} \cdot \mathit{Deal}
\end{array}
$$

Thus, the simplification of this proof obligation leads to an intermediate conjunction of four proof obligations, of which two are subsequently discharged. The remaining two obligations actually express the same property (they can be converted into each other by taking their inverse). Conversion back to pointwise notation gives the following precondition:

$$i\ \in\ \delta\ \mathit{Deal} \Rightarrow (u, p) = \mathit{Deal}\ (i)$$

Note that the proof obligation we derived is weaker than the over-defensive precondition that is typically added to an operation such as *settleDeal*, namely that $i \notin \delta(\mathit{Deal})$.

Batch addition of items to sell. Once PF-transformed, our last function is defined by $\mathit{putBatchForSale}\ (u, m)\ n = n \dagger x$, where $x = \mathit{withUser}\ u\ m$ and

withUser $u\ m = \langle(\underline{u}), m\rangle$. This model illustrates the use of two other useful binary operators on relations, *split* ($\langle\cdot,\cdot\rangle$) and *override* ($\cdot\dagger\cdot$) [23]. The former pairs the outputs of two relations and the latter overrides one relation by another. Checking the simplicity of the output of *putBatchForSale* leads to a 32-step derivation of which we show only the starting and closing steps, the latter condensed for space economy:

$$
\begin{array}{l}
\mathsf{img}\,(n\ \cdot\dagger\cdot\ x) \subseteq id \\
\quad \equiv \{\, override\_def \,\} \\
\mathsf{img}\,(n \cup x \cdot \neg\,(\delta\ x)) \subseteq id \\
\quad \equiv \{\, img\_def \,\} \\
(n \cup (x \cdot \neg\,(\delta\ (n)))) \cdot (n \cup (x \cdot \neg\,(\delta\ (n))))^\circ \subseteq id \\
\quad \equiv \{\, inv\_union \,\} \\
(n \cup (x \cdot \neg\,(\delta\ (n)))) \cdot (n^\circ \cup (x \cdot \neg\,(\delta\ (n)))^\circ) \subseteq id \\
\quad ... \\
((True\ \wedge\ True)\ \wedge\ (True\ \wedge\ x \cdot \neg\,(\delta\ (n)) \subseteq x)) \\
\quad \equiv \{\, and\_true, monotonicity \,\} \\
(True\ \wedge\ x \cdot id \subseteq x)) \\
\quad \equiv \{\, and\_true, comp\_id \,\} \\
x \subseteq x \\
\quad \equiv \{\, incl\_refl \,\} \\
True
\end{array}
$$

Thus the proof obligation is discharged completely. In this case extended static checking validates the user model and no changes are needed.

7 Related work

Extended static checking. Extensive progress has been achieved on extended static checking (for review see [17]), resulting in practical tools for imperative languages [18, 8]. These tools rely on theorem provers to find counter examples of verification conditions [6], using a combination of techniques such as backtracking search, matching algorithms for universally quantified formulæ, and heuristics. As alternative or supplemental technique, we have explored proof construction through rewriting of pointfree relational expressions. The absence of quantifiers and variables in these expressions promises to allow a more effective proof search and to enlarge the scope of properties that can be practically checked for, such as those arising in software modeling using rich data structures.

Relational programming (symbolic). MacLennan pioneered relational programming and proposed it as a more general substitute for functional programming [19–21]. He keeps a separation between finite relations representing data structures, and infinite relations representing operations. Cattrall and Runciman built on his work to develop compilation support for relational programming, where finite and infinite relations are mixed, and where relational expressions are made compilable by rewriting them according to algebraic properties [3].

Relation-algebraic analysis (finite). Modeling and analysis of systems based on *finite* relational representations is supported by systems such as Grok [10] and RelView [1] which are, however, very different from our approach: Grok is a calculator for *finite* relational algebra expressions and RelView uses BDDs to implement relations in an efficient way.

Typed strategic rewriting. Strategic programming [14] was first supported in non-typed setting in the Stratego language [26]. A strongly-typed combinator suite was introduced as a Haskell library by the Strafunski system [15, 16] and later generalized into the so-called 'scrap-your-boilerplate' generic programming library [13]. We developed GADT-based strategic combinator suites, similar to the one presented here, for two-level data transformation [4] and transformation of pointfree and structure-shy functions [5].

8 Concluding remarks

We have implemented a type-directed strategic rewrite system for normalization of pointfree relational expressions and simplification or discharge of relational propositions. We have demonstrated the utility of the system in the context of extended static checking of common model and program properties. We intend to elaborate our approach in various directions.

So far, we have limited ourselves to rewriting of pointfree expressions, relying on manual transformation of logic formulæ into relational algebra expressions and back. We intend to also automate this pointfree transform. The suite of operators and laws currently implemented in the system can be extended further as the need arises for more expressiveness. The strategy for proof search is likely to further evolve as well, for instance to include short cut derivations for special common cases or to eliminate duplication of proof obligations due to converse inclusions. A thorough analysis of the formal properties of the rewriting system we are building is one of our current concerns.

When achieving a good degree of maturity, an assessment will be needed as to whether this approach can indeed be an alternative or supplement to existing ESC approaches based on theorem proving. Besides ESC, we envision to apply our relational algebra rewriting system to areas such as program optimization, program verification, relational programming, and more. For example we currently exploring the construction of rewriting strategies for program inversion, useful in two-level transformation [4] and bidirectional programming [22].

References

1. R. Berghammer and F. Neumann. RelView - an OBDD-based computer algebra system for relations. volume 3718 of *LNCS*, pages 40–51. Springer, 2005.
2. R. Bird and O. de Moor. *Algebra of Programming.* Series in Computer Science. Prentice-Hall International, 1997. C.A.R. Hoare, series editor.

3. D. Cattrall and C. Runciman. Widening the representation bottleneck: a functional implementation of relational programming. In *Proc. Func. Prog. Lang. and Comp. Arch.*, pages 191–200. ACM Press, 1993.
4. A. Cunha, J.N. Oliveira, and J. Visser. Type-safe two-level data transformation. Number 4085 in LNCS, pages 284–289. Springer, 2006.
5. A. Cunha and J. Visser. Transformation of structure-shy programs, applied to XPath queries and strategic functions. In *PEPM'07, ACM SIGPLAN*, 2007.
6. D. Detlefs, G. Nelson, and J.B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.
7. J. Fitzgerald and P.G. Larsen. *Modelling Systems: Practical Tools and Techniques for Software Development* . Cambridge University Press, 1st edition, 1998.
8. C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J.B. Saxe, and R. Stata. Extended static checking for java. In *PLDI*, pages 234–245, 2002.
9. R. Hinze, A. Löh, and B.C.d.S. Oliveira. "Scrap your boilerplate" reloaded. In M. Hagiya and P. Wadler, editors, *Proc. Functional and Logic Programming, 8th Int. Symp.*, volume 3945 of *LNCS*, pages 13–29. Springer, 2006.
10. R.C. Holt. Structural manipulations of software architecture using Tarski relational algebra. In *WCRE '98*, page 210. IEEE Comp. Soc., 1998.
11. C.B. Jones. *Systematic Software Development Using VDM*. Series in Computer Science. Prentice-Hall International, 1986. C.A. R. Hoare.
12. E. Kreyszig. *Advanced Engineering Mathematics*. J.Wiley & Sons, Inc., 1988.
13. R. Lämmel and S. Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. *ACM SIGPLAN Notices*, 38(3):26–37, March 2003.
14. R. Lämmel, E. Visser, and J. Visser. The Essence of Strategic Programming. Available at `http://www.cwi.nl/~ralf`, 2003.
15. R. Lämmel and J. Visser. Typed Combinators for Generic Traversal. In *PADL'02*, volume 2257 of *LNCS*, pages 137–154. Springer, January 2002.
16. R. Lämmel and J. Visser. A Strafunski Application Letter. In V. Dahl et al., editors, *PADL'03*, volume 2562 of *LNCS*, pages 357–375. Springer, 2003.
17. K.R.M. Leino. Extended static checking: A ten-year perspective. In *Informatics - 10 Years Back. 10 Years Ahead.*, pages 157–175. Springer-Verlag, 2001.
18. K.R.M. Leino and G. Nelson. An extended static checker for Modula-3. volume 1383 of *LNCS*, pages 302–305. Springer, 1998.
19. B.J. MacLennan. Introduction to relational programming. In *FPCA '81*, pages 213–220, New York, NY, USA, 1981. ACM Press.
20. B.J. MacLennan. Overview of relational programming. *SIGPLAN Not.*, 18(3):36–45, 1983.
21. B.J. MacLennan. Four relational programs. *SIGPLAN Not.*, 23(1):109–119, 1988.
22. S.-C. Mu, Z. Hu, and M. Takeichi. An algebraic approach to bi-directional updating. volume 3302 of *LNCS*, pages 2–20. Springer, 2004.
23. J.N. Oliveira and C.J. Rodrigues. Transposing relations: from *Maybe* functions to hash tables. In *MPC'04*, volume 3125 of *LNCS*, pages 334–356. Springer, 2004.
24. J.N. Oliveira and C.J. Rodrigues. Pointfree factorization of operation refinement. In *FM'06*, volume 4085 of *LNCS*, pages 236–251. Springer-Verlag, 2006.
25. S. Peyton Jones, G. Washburn, and S. Weirich. Wobbly types: type inference for generalised algebraic data types, 2004. Univ. Penn. TR MS-CIS-05-26.
26. E. Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. volume 2051 of *LNCS*, pages 357–361. Springer, 2001.