Computer Aided Verification of Relational Models

Claudia M. Necco[1], José N. Oliveira[2], Joost Visser[3], and Roberto Uzal[1]

cnecco@gmail.com jno@di.uminho.pt j.visser@sig.eu ruzal@uolsinectis.com.ar

[1] Dep. de Informática, Univ. Nacional de San Luis, San Luis, Argentina
[2] HASLab/INESC TEC & University of Minho, Portugal
[3] Software Improvement Group & CWI, Amsterdam, The Netherlands

Abstract. Binary relational algebra provides semantic foundations for major areas of computing, such as database design, state-based modeling 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 such as database queries, state transitions, or composition of functional components.
When data models, their constraints and operations are represented by point-free binary relational expressions, proof obligations can be expressed as inclusions between relational expressions. We developed a type-directed, strategic term rewriting system that can be used to simplify relational proof obligations and ultimately reduce them to tautologies. Such reductions can be used to provide extended static checking for design contraints commonly found in software modeling and development. .

Keywords: Models verification, Symbolic execution; Abstract model verification; Extended static checking; Strategic term rewriting

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 (e.g. syntax and type checkers) are tools which catch errors at compile-time, i.e. before running the program. Examples of such errors are unmatched parentheses (wrong syntax) and adding integers to booleans (wrong typing). 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 e.g. 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 (e.g. 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 [8].

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) [16]. Between these two extremes of standard, cheap, decidable static checking and costly theorem proving, *extended static checking* (ESC) [9] 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 Java (ESC/Java [9]). At the heart of these tools we find a verification condition generator and the Simplify theorem prover [7]. 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 at abstract model level to catch errors higher on the semantic scale.

The main novelty of our approach resides in the chosen method of proof construction, whereby first-order proof obligations are subject to the *PF-transform* [18] before they are reasoned about. (See reference [18] for the theory behind this blending of ESC with the PF-transform, suggestively referred to as the ESC/PF proof obligation calculus.) Such a 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). As shown in [18], ESC proof obligations can be discarged at PF-level, leading to the so-called ESC/PF calculus. In the current paper we move from "paper and pencil" ESC/PF reasoning to mechanical calculation using a Haskell implementation of strategic term rewriting [20, 15, 14].

ϕ	$PF\ \phi$
$\langle \exists\, a \;::\; b\,R\,a \,\wedge\, a\,S\,c \rangle$	$b(R \cdot S)c$
$\langle \forall\, a,b \;::\; b\,R\,a \Rightarrow b\,S\,a \rangle$	$R \subseteq S$
$\langle \forall\, a \;::\; a\,R\,a \rangle$	$id \subseteq R$
$b\,R\,a \,\wedge\, c\,S\,a$	$(b,c)\langle R,S\rangle a$
$b\,R\,a \,\wedge\, b\,S\,a$	$b\,(R \cap S)\,a$
$b\,R\,a \vee b\,S\,a$	$b\,(R \cup S)\,a$
$b = a$	$b\,id\,a$
TRUE	$b \top a$
FALSE	$b \perp a$

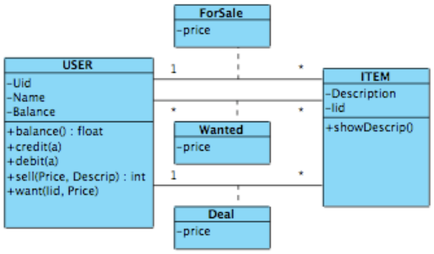
In analogy to the well-known Laplace transform [12], the PF-transform takes expressions from a mathematical problem space, in this case first order logic formulæ, into a mathematical solution space, in this case relational algebra expressions [2]. The PF-transform eliminates quantifiers and bound variables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

Fig. 1. The PF-transform.

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



$$
\begin{aligned}
\mathit{Name} \times \mathit{Balance} &\xleftarrow{\mathit{User}} \mathit{Uid}\\
\mathit{Uid} \times \mathit{Price} &\xleftarrow{\mathit{ForSale}} \mathit{Iid}\\
\mathit{Price} &\xleftarrow{\mathit{Wanted}} \mathit{Iid} \times \mathit{Uid}\\
\mathit{Uid} \times \mathit{Price} &\xleftarrow{\mathit{Deal}} \mathit{Iid}\\
\mathit{Description} &\xleftarrow{\mathit{Item}} \mathit{Iid}
\end{aligned}
$$

Fig. 2. Simplied UML model of a trading system and the corresponding binary relational model. The relations in this model are finite and simple (explained in Section 3). This is loosely based on a formal model (written in Haskell) for a real estate exchange market, which has been developed for a digital city consortium.

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

$$
\begin{aligned}
&\mathit{listWantedItems} &&:: \mathit{Wanted} \to \mathit{Map}\ \mathit{Iid}\ \mathit{Price}\\
&\mathit{putBatchForSale} &&:: (\mathit{Uid}, \mathit{Map}\ \mathit{Iid}\ \mathit{Price}) \to \mathit{ForSale} \to \mathit{ForSale}\\
&\mathit{settleDeal} &&:: (\mathit{Iid}, \mathit{Uid}, \mathit{Price}) \to \mathit{Deal} \to \mathit{Deal}
\end{aligned}
$$

The $\mathit{listWantedItems}$ query produces a map of item identifiers together with the price that has been offered for them. The transformation $\mathit{putBatchForSale}$ adds a batch of items belonging to a given user to the $\mathit{ForSale}$ relation. The $\mathit{settleDeal}$ transformation adds an entry to the Deal collection.

When specifying these operations, the designer could benefit from the feedback of an extended static checker. For example, the checker should tell her/him that query $\mathit{listWantedItems}$ 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 $\mathit{Rel}\ \mathit{Iid}\ \mathit{Price}$ or, equivalently, to $\mathit{Set}\ (\mathit{Iid}, \mathit{Price})$. In case of the $\mathit{settleDeal}$ 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.

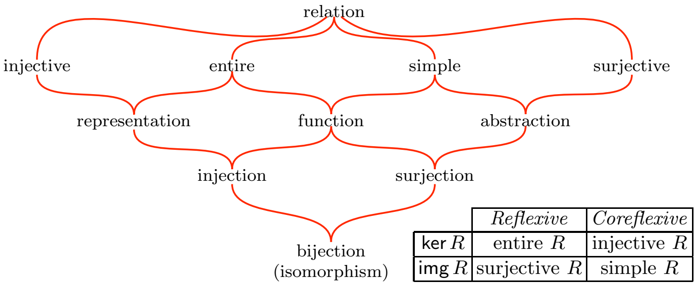
3 Overview of relation theory

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 data-types 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 $R \subseteq S$, with the usual semantics of the subset relation between sets of pairs. In relational terms, it means 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: $R = S \equiv R \subseteq S \ \wedge \ S \subseteq R$.

Three more operators are introduced to combine relations: 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 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 $[\![p]\!]$ such that $b[\![p]\!]a \equiv (b = a) \ \wedge \ (p\ a)$ holds, 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 $b[\![S]\!]a \equiv (b = a) \ \wedge \ a \in S$.



	Reflexive	*Coreflexive*
ker R	entire R	injective R
img R	surjective R	simple R

Figure 1: Binary relation taxonomy

Fig. 3. Binary relation taxonomy

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

Table 1. Some laws of the binary relational algebra.

comp_assoc	$(R \cdot S) \cdot T = R \cdot (S \cdot T)$	*inv_comp*	$(R \cdot S)^{\circ} = S^{\circ} \cdot R^{\circ}$
comp_id	$R \cdot id = R \;;\; id \cdot R = R$	*inv_inv*	$(R^{\circ})^{\circ} = R$
comp_empty	$R \cdot \bot = \bot \;;\; \bot \cdot R = \bot$	*inv_union*	$(R \cup S)^{\circ} = R^{\circ} \cup S^{\circ}$
union_fusion	$(R \cup S) \cdot T = (R \cdot T) \cup (S \cdot T)$	*corefl_symm*	$\Phi^{\circ} = \Phi$
	$T \cdot (R \cup S) = (T \cdot R) \cup (T \cdot S)$	*corefl_trans*	$\Phi \cdot \Phi = \Phi$
dom_elim	$R \cdot \delta\, R = R$	*const_fusion*	$\underline{k} \cdot R = \underline{k} \cdot \delta\, R$
neg_co_comp	$\neg\, \Phi \cdot \Phi = \bot \;;\; \Phi \cdot \neg\, \Phi = \bot$	*not_dom_cancel*	$R \cdot \neg\, (\delta\, R) = \bot$
incl_empty	$\bot \subseteq R \Leftrightarrow \mathit{True}$	*incl_refl*	$R \subseteq R \Leftrightarrow \mathit{True}$
monotonicity	$R \cdot \Phi \cdot S \subseteq T \Leftarrow R \cdot S \subseteq T$		
union_univ	$R \cup S \subseteq T \Leftrightarrow (R \subseteq T \wedge S \subseteq T)$		
shunt_fun	$f \cdot R \subseteq S \Leftrightarrow R \subseteq f^{\circ} \cdot S$		
shunt_fun_inv	$R \cdot f^{\circ} \subseteq S \Leftrightarrow R \subseteq S \cdot f$		
shunt_map_inv	$R \cdot M^{\circ} \subseteq S \Leftrightarrow R \cdot \delta\, M \subseteq S \cdot M$		
shunt_map	$M \cdot R \subseteq S \Leftrightarrow \delta\, M \cdot R \subseteq M^{\circ} \cdot S$		
img_def	$\mathsf{img}\, R = R \cdot R^{\circ}$	*ker_def*	$\mathsf{ker}\, R = R^{\circ} \cdot R$

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. This terminology is recorded in the summary table in Fig. 3. The coreflexive fragments of kernel and image are named *domain* (δ) and *range* (ρ).

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 $\underline{k}$.

Algebraic properties. A rich set of algebraic properties is a-vailable 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 $x - y \leqslant z \Leftrightarrow x \leqslant z + y$. The utility of such laws will become evident below.

4 Rewriting relational expressions and propositions

We developed a type-safe, type-directed rewriting system for normalization of relational expressions that harnessed the various algebraic laws of binary relations presented in Table 1 in the functional programming language Haskell. In this section and the next, we will provide a high-level description independent of the programming language.

Terms. The terms to be rewriten by our term rewriting system will be the expression of binary relational calculus with some additional annotations. The following outlines the grammar:

$$P := R \subseteq_T R \mid \mathit{True} \mid P \wedge P$$
$$R := id \mid R \cdot_T R \mid R^{\circ} \mid V_L \mid \underline{A}_T \mid ...$$

$T := 1 \mid Int \mid Bool \mid String \mid [\,T\,] \mid T \times T \mid ...$
$L \; := \textsf{entire} \mid \textsf{simple} \mid \textsf{injective} \mid \textsf{surjective} \mid \textsf{reflexive} \mid \textsf{coreflexive}$
$V := variable\ names$
$A \; := values$

Thus, some of the relation operators are annotated with type information (shown as subscript). Relation variables are annotated with properties as they appear in the leafs of the taxonomy of Figure 3. For example, a function f is an entire and simple relation and is therefore annotated as $f_{[\textsf{entire},\textsf{simple}]}$. Also, endo-relations can be annotated to be (co-)reflexive.

Predicates on relations. The first ingredient into our rewriting system are predicates for testing the various properties that relations may have, such as simplicity, surjectivity, etc. 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:

$isSimple(id) \quad\;\; = True$
$isSimple(r_l) \quad\;\; = \textsf{simple} \in l$
$isSimple(r^\circ) \quad\;\, = isInjective(r)$
$isSimple(s \cdot_b r) = isSimple(r) \wedge isSimple(s)$
$...$
$isSimple(r) \qquad = False$

Similar predicates are supplied for the remaining properties. These predicates test for properties by induction over the structure of relational expressions, but do not attempt to derive proofs for the properties. In this sense, they are approximations and may fail to discover that a certain relational expression enjoys particular properties.

Type-directed and property-aware rewriting rules. The predicates above are used in the definition of rewrite rules. Each rewrite rule encodes particular laws of the relational calculus. Since our rewrite system is type-directed, rewrite rules are annotated with types. Here is an encoding of the inv_comp law, applied in the left-to-right direction:

$$inv\_comp : (r \cdot_b s)^\circ \longmapsto_{(c \leftarrow a)} s^\circ \cdot_b r^\circ$$

Pattern matching is performed on a relational expression and, on successful match, a resulting expression is returned.

The $const\_fusion$ rule provides an example of rewriting directed by properties:

$const\_fusion : (s \cdot_b r) \longmapsto_{(c \leftarrow a)} (\underline{s}_c)$
  if $isConstant(s) \wedge \neg\, (isCoreflexive(r)) \wedge isEntire(r)$
$const\_fusion : (s \cdot_b r) \longmapsto_{(c \leftarrow a)} ((\underline{s}_c) \cdot_a (\delta_b r))$
  if $isConstant(s) \wedge (\neg\, (isCoreflexive(r))$

The rule works on a composition and, if the first argument s constant as required by the guarding predicate, then it replaces the second argument r by its domain. If the second argument r is entire, $\delta\ r = id$ then the rule return just the first (constant) argument. 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:

$\mathit{shunt\_fun\_inv} : ((x \cdot_b f^\circ) \subseteq_{(c \leftarrow a)} y) \longmapsto_{(c \leftarrow a)} (x \subseteq_{(c \leftarrow b)} (y \cdot_a f))$
if $\mathit{isEntire}(f) \wedge \mathit{isSimple}(f)$

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 [4]:

$\mathit{nop} :: \mathit{Rule}$ -- identity rule
$(\triangleright) :: \mathit{Rule} \to \mathit{Rule} \to \mathit{Rule}$ -- sequential comp.
$(\oplus) :: \mathit{Rule} \to \mathit{Rule} \to \mathit{Rule}$ -- choice (based on mplus)
$(\oslash) :: \mathit{Rule} \to \mathit{Rule} \to \mathit{Rule}$ -- choice (bas. on mcatch)
$\mathit{all} :: \mathit{Rule} \to \mathit{Rule}$ -- map on all children
$\mathit{one} :: \mathit{Rule} \to \mathit{Rule}$ -- map on one child
$\mathit{run} :: \mathit{Rule} \to R\ r \to (R\ r, \mathit{Derivation})$ -- top-level app.

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 $\mathit{Rewrite}$ 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:

$\mathit{many}\ r = (r \triangleright (\mathit{many}\ r)) \oslash \mathit{nop}$ -- repeat until failure
$\mathit{once}\ r = r \oslash \mathit{one}\ (\mathit{once}\ r)$ -- apply once, at any depth
$\mathit{innermost}\ r = \mathit{all}\ (\mathit{innermost}\ r) \triangleright ((r \triangleright \mathit{innermost}\ r) \oslash \mathit{nop})$

The derived combinator $\mathit{innermost}$ 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:

$\mathit{simplify} = \mathit{innermost}\ \mathit{simplify1}$

$\mathit{simplify1} = \mathit{comp\_assocl} \oslash \mathit{desugar1} \oslash \mathit{applylaw1}$
$\mathit{desugar1} = \mathit{ker\_def} \oslash \mathit{img\_def} \oslash ...$
$\mathit{applylaw1} = \mathit{inv\_comp} \oslash \mathit{inv\_inv} \oslash \mathit{comp\_id} \oslash \mathit{comp\_empty} \oslash \mathit{dom\_elim} \oslash$
$\mathit{corefl\_symm} \oslash \mathit{const\_fusion} \oslash \mathit{not\_dom\_cancel} \oslash ...$

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 $\mathit{comp\_assoc}$ rule is employed to bring relational compositions into left-associative form. Since the normalization rules together form a

[4] These rules and our representation technique are inspired on the *2LT* system [6].

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 *simplify* strategy to $(N \cdot (\neg\ (\delta\ N))^{\circ} \cdot M^{\circ})^{\circ}$, where N and M are simple relations:

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

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:

$$
\begin{array}{l}
\mathit{derive} = \mathit{simplify} \rhd \mathit{all\_and}\ \mathit{process\_conjunct} \rhd \mathit{innermost}\ \mathit{and\_true} \\
\quad \mathbf{where} \\
\qquad \begin{array}{ll}
\mathit{process\_conjunct} & = (\mathit{shunt\_conjunct} \oplus \mathit{strengthen\_conjunct}) \oslash \mathit{nop} \\
\mathit{shunt\_conjunct} & = \mathit{shunt} \rhd \mathit{derive} \\
\mathit{strengthen\_conjunct} & = \mathit{strengthen} \rhd \mathit{derive} \rhd \mathit{qed}
\end{array} \\
\mathit{shunt} = (\mathit{shunt\_fun\_inv} \oslash \mathit{shunt\_map\_inv}) \oplus (\mathit{shunt\_fun} \oslash \mathit{shunt\_map}) \\
\mathit{strengthen} = \mathit{corefl\_cancel} \\
\mathit{all\_and} :: \mathit{Rule} \to \mathit{Rule} \quad \text{-- apply arg. rule on all conjs} \\
\mathit{qed} \quad\ \ :: \mathit{Rule} \quad \text{-- test whether the current exp. is } \mathit{True}
\end{array}
$$

The initial application of *simplify* 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* ($p\ \wedge\ \mathit{True}\ \Leftrightarrow\ \mathit{True}\ \wedge\ p\ \Leftrightarrow\ p$) is applied to absorb the propositions that have been rewritten to *True*. The processing of each conjunct makes a non-deterministic choice (using the 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 *derive* strategy. When starting with strengthening, the subsequent recursive call to *derive* is required to lead to a full proof (*qed*), since we are interested in strengthened propositions only for the purpose of discharging proof obligations.

The use of backtracking entails that several results may be obtained or the same result through different derivations. In the implementation, lazy evaluation is employed to ensure that only a single derivation is actually constructed.

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). The overall operation of the developed tool is based on transforming and rewriting PF-relational expressions using the ESC/PF calculus described in [18].

List wanted items. The operation $listWantedItems$ can be specified in binary relational terms as $listWantedItems = Wanted \cdot \pi_1^{\circ}$, where π_1 is the first projection on pairs, i.e. $\pi_1\ (a, b) = a$. Note that we leave the argument $Wanted$ implicit in the definition of the operation. Regarding $Wanted$ as a set of pairs, the definition converts to the pointwise $\{\ (p, i) \mid (p, (i, u)) \in Wanted\ \}$, where p, i, u range over $Price$, Iid and Uid, respectively. Clearly, this won't be a simple relation in general, even if $Wanted$ is so, because dropping u from the input may lead to the same i related to different p. Since this operation is specified to produce a finite map (thus simple), it gives rise to the proof obligation $\mathsf{img}\,(Wanted \cdot \pi_1^{\circ}) \subseteq id$, which in turn leads to the following derivation when applying our $derive$ strategy:

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

What does the last line above mean? We simply have to apply the rules of the PF-transform the other way round and find the corresponding, more descriptive logic expression:

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

This formula expresses that query $listWantedItems$ 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. Using singleton relation notation as decribed in Section 3, we can define $settleDeal\ (i, u, p) = Deal \cup \underline{(u, p)} \cdot \underline{i}^{\ \circ}$. (Again we leave the old value of $Deal$ implicit in the definition.) Checking the simplicity of its output gives rise to the following derivation (condensed):

$$\begin{array}{l}
\quad \mathsf{img}\,(Deal \cup \underline{(u, p)} \cdot \underline{i}^{\ \circ}) \subseteq id \\
\Leftrightarrow \{\,img\_def, various\ union\ laws\,\}
\end{array}$$

$$
\begin{aligned}
&\quad Deal \cdot Deal^{\circ} \subseteq id \wedge Deal \cdot \underline{i} \cdot \underline{(u,p)}^{\circ} \subseteq id \wedge \\
&\quad \underline{(u,p)} \cdot \underline{i}^{\circ} \cdot Deal^{\circ} \subseteq id \wedge \underline{(u,p)} \cdot \top \cdot \underline{(u,p)}^{\circ} \subseteq id \\
&\Leftrightarrow \{\, various\ shunting\ laws, dom\_elim \,\} \\
&\quad \delta\ Deal \cdot \underline{i} \subseteq Deal^{\circ} \cdot \underline{(u,p)} \wedge \\
&\quad \underline{i}^{\circ} \cdot \delta\ Deal \subseteq \underline{(u,p)}^{\circ} \cdot Deal
\end{aligned}
$$

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:

$$i \in dom\ (Deal) \Rightarrow (u,p) = Deal\ (i)$$

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 $i \notin dom(Deal)$.

Batch addition of items to sell. Once PF-transformed, our last function is defined by $putBatchForSale\ (u,m) = ForSale \dagger x$,
where $x = withUser\ u\ m$ and $withUser\ u\ m = \langle \underline{u}, m \rangle$. This model illustrates the use of two other useful binary operators on relations, *override* $(\cdot \dagger \cdot)$ and *split* $(\langle \cdot, \cdot \rangle)$ [19]. The latter pairs the outputs of two relations (recall Fig. 1) and the former 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:

$$
\begin{aligned}
&\mathsf{img}\ (n \dagger x) \subseteq id \\
&\ \Leftrightarrow \{\, override\_def \,\} \\
&\mathsf{img}\ (n \cup x \cdot \neg\ (\delta\ x)) \subseteq id \\
&\ \Leftrightarrow \{\, img\_def \,\} \\
&(n \cup (x \cdot \neg\ (\delta\ (n)))) \cdot (n \cup (x \cdot \neg\ (\delta\ (n))))^{\circ} \subseteq id \\
&\ \ldots \\
&((True\ \wedge\ True)\ \wedge\ (True\ \wedge\ x \cdot \neg\ (\delta\ (n)) \subseteq x)) \\
&\ \Leftrightarrow \{\, and\_true, monotonicity \,\} \\
&(True\ \wedge\ x \cdot id \subseteq x)) \\
&\ \Leftrightarrow \{\, and\_true, comp\_id \,\} \\
&x \subseteq x \\
&\ \Leftrightarrow \{\, incl\_refl \,\} \\
&True
\end{aligned}
$$

Thus the proof obligation is discharged completely. In this case extended static checking validates the user model and no changes are needed. The 32-step derivation took 0.14 seconds to run with version 6.8.2 of the Haskell interpreter (GHCi) on a MacBook Pro (1.83 GHz Intel Core Duo processor).

7 Related work

Extended static checking. Extensive progress has been achieved on extended static checking (for review see [16]), resulting in practical tools for imperative languages [9].

These tools rely on theorem provers to find counter examples of verification conditions [7], 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 [17]. 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 the non-typed setting of the Stratego language [20]. A strongly-typed combinator suite was introduced as a Haskell library by the Strafunski system [15] 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 [5] and transformation of pointfree and structure-shy functions [4].

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.

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 implemented in the system is currently under study with respect to minimality, confluence and termination.

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. A good test will be to try and discharge complex ESC/PF proof obligations such as those arising from the Verified File System project [18]. Besides ESC, we envision to apply our relational algebra rewriting system to areas such as program optimization, program verification, relational programming, and more.

References

1. Rudolf Berghammer and Frank Neumann. *RelView – An OBDD-Based Computer Algebra System for Relations*, pages 40–51. Springer Berlin Heidelberg, Berlin, Heidelberg, 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 and J. Visser. Transformation of structure-shy programs, applied to XPath queries and strategic functions. In *PEPM'07, ACM SIGPLAN*, 2007.
5. Alcino Cunha, José Nuno Oliveira, and Joost Visser. *Type-Safe Two-Level Data Transformation*, pages 284–299. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.
6. Alcino Cunha and Joost Visser. Strongly typed rewriting for coupled software transformation. *Electron. Notes Theor. Comput. Sci.*, 174(1):17–34, 2007.
7. D. Detlefs, G. Nelson, and J.B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.
8. J. Fitzgerald and P.G. Larsen. *Modelling Systems: Practical Tools and Techniques for Software Development* . Cambridge University Press, 1st edition, 1998.
9. 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.
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. A Strafunski Application Letter. In V. Dahl et al., editors, *PADL'03*, volume 2562 of *LNCS*, pages 357–375. Springer, 2003.
16. 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.
17. B.J. MacLennan. Overview of relational programming. *SIGPLAN Not.*, 18(3):36–45, 1983.
18. J.N. Oliveira. Extended static checking by calculation using the PF-transform, Jul. 2008. LerNET'08 post-workshop tutorial paper submitted for publication.
19. 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.
20. Eelco Visser. *Stratego: A Language for Program Transformation Based on Rewriting Strategies System Description of Stratego 0.5*, pages 357–361. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.