Generating Implementations of Data Type Operations

Anthony Hunt[1][0009−0005−1085−3343] and Emil Sekerinski[1][0000−0001−9788−5842]

Department of Computing and Software, McMaster University, Hamilton, ON L8S4L8, Canada {hunta12,emil}@mcmaster.ca

1 Introduction

System specifications in modelling languages (like UML, Event-B, and Alloy) and specification languages (such as Dafny and Frama-C) rely on abstract collection data types to describe properties and expected behaviour. Sets, sequences, relations, maps, and bags are all indispensable, each accompanied by an expressive set of operators. Conversely, modern programming languages (like Python and Haskell) support only a subset of the those data types and operators, instead encouraging the use of implementation-aware types like arrays and classes. In ongoing work, we investigate how programming languages can be extended with more operations on data types. We seek to synthesize efficient implementations of data type operations with guaranteed bounds for runtime and memory consumption. Our focus is on those of Table 1, which are a subset of Event-B data types [1]. Relations are sets of pairs, and functions are functional relations.

2 Motivating Example: A Visitor Information System

Consider a service system that tracks all attendees and workshops within a conference. One visitor may attend at most one workshop per day. Only one workshop may be held in a room throughout the day. An appropriate model is:

$$location : Workshop \rightarrowtail Room \tag{1}$$

Table 1: Selected operators on sets and relations.

Syntax	Label/Description	Syntax	Label/Description
$set(T)$	Unordered, unique collection	$S \nrightarrow T$	Partial function
$S \leftrightarrow T$	Relation, $set(S \times T)$	$S \rightarrowtail T$	Total injection
$\varnothing$	Empty set	$x \mapsto y$	Pair (relational element)
$\{x, y, ...\}$	Set enumeration	$dom(S)$	Domain
$\{ x \in S \mid P \cdot E \}$	Set comprehension	$ran(S)$	Range
$S \cup T$	Union	$R[S]$	Relational image
$S \cap T$	Intersection	$R \lhd\!\!+ Q$	Relational overriding
$S \setminus T$	Difference	$R \circ Q$	Relational composition
$S \times T$	Cartesian Product	$S \lhd R$	Domain restriction
$S \subseteq T$	Subset	R^{-1}	Relational inverse

$$attends : Visitor \nrightarrow Workshop \tag{2}$$

If a workshop organizer needs to order meals for each attendee in *room*, their number is given by:

$$card((location^{-1} \circ attends^{-1})[\{room\}]) \tag{3}$$

We envisage that programmers write such expressions, and efficient code is generated. A naive implementation constructs intermediate relations for $location^{-1}$ and $attends^{-1}$. We show how this can be avoided. The implementation in Fig. 1 (a) assumes that both functions are implemented as sequences, $location : seq(Workshop \times Room)$ and $attends : seq(Visitor \times Workshop)$. The implementation in Fig. 1 (b) assumes that *location* is a hashmap. Both implementations need only constant memory.

```
card := 0
for p, r in location do
    if r = room then
        for v, p' in attends do
            if p = p' then
                card := card + 1
```

(a) The running time is proportional to $len(location) \times len(attends)$

```
card := 0
for v, p in attends do
    if location[p] = room then
        card := card + 1
```

(b) Assuming *location* is a hashmap with constant time lookup $location[p]$, the running time is proportional to $len(attends)$.

Fig. 1: Two implementations of query (3).

3 Term Rewriting Rules

The code generation is implemented by a term rewriting system. We give sufficiently many rewrite rules for transforming (3) into:

$$\sum p \mapsto r \in location \wedge v \mapsto p' \in attends \mid p = p' \wedge r = room \cdot 1 \tag{4}$$

The basic strategy for set and relational expressions is to first normalize the expression as a set comprehension and then simplify conjuncts of the set comprehension body. Necessary rewrite rules are:

$$x \in \{E\} \rightsquigarrow x = E \qquad \text{(Singleton membership)}$$

$$x \in \{\, y \in S \mid P \cdot E(y) \,\} \cdot F(x) \rightsquigarrow y \in S \mid P \cdot F(E(y)) \qquad \text{(Membership collapse)}$$

$$card(S) \rightsquigarrow \sum x \in S \cdot 1 \qquad \text{(Cardinality)}$$

$$R[S] \rightsquigarrow \{\, x \mapsto y \in R \mid x \in S \cdot y \,\} \qquad \text{(Image)}$$

$$x \mapsto y \in R^{-1} \rightsquigarrow y \mapsto x \in R \qquad \text{(Inverse)}$$

$$x \mapsto y \in (Q \circ R) \rightsquigarrow x \mapsto z \in Q \wedge z' \mapsto y \in R \wedge z = z' \qquad \text{(Composition)}$$

Once the entire expression is transformed into a set or relational comprehension, repeated application of rules below generates the algorithm in Fig. 1 (a).

$$\sum x \in S \mid P \cdot E \rightsquigarrow \begin{array}{l} c := 0 \\ \textbf{for } x \in S \textbf{ do} \\ \quad \textbf{if } P \textbf{ then } c := c + E \end{array} \qquad \text{(Summation)}$$

$$\begin{array}{l} \textbf{if } P \wedge x \in S \wedge Q \textbf{ then } p \\ \text{where } x \text{ not free in } P \end{array} \rightsquigarrow \begin{array}{l} \textbf{if } P \textbf{ then} \\ \quad \textbf{for } x \in S \textbf{ do} \\ \quad\quad \textbf{if } Q \textbf{ then } p \end{array} \qquad \text{(Composed conditional)}$$

4 Discussion

Term rewriting has been used for optimization and code generation: [4] devises a tactic language for applying rules, and [2] allows programmers to specify rewrite rules for GHC. In our work, programmers need only specify the data type implementation; a fixed set of rewrite rules guarantees predictable run times and memory consumption. Rewrite rules have also been used for source-to-source transformation, like refactoring and transpiling. The rules aim to generate human-readable code. Here, the rules aim at efficiency without the need for the result to be comprehensible.

In Specware, algorithmic and data refinement transformations can be applied to axiomatic specifications; in [3], this is extended to inductively and co-inductively specified data types. Here, the data types are fixed, and we only consider automatically generating implementations of complex expressions. Relational databases provide query optimizations for the relational data model. Here, we support a wider range of modelling data types with an arbitrary combination of operations. The goal of this project is to bring programming languages closer to modelling languages while remaining competitive at runtime.

Acknowledgments. We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC), [grant RGPIN-2024-06779].

References

[1] Jean-Raymond Abrial. *Modeling in Event-B: system and software engineering*. Cambridge University Press, 2010.

[2] Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. "Playing by the rules: rewriting as a practical optimisation technique in GHC". In: *2001 Haskell Workshop*. ACM SIGPLAN. Sept. 2001.

[3] Douglas R. Smith and Stephen J. Westfold. "Transformations for Generating Type Refinements". In: *Formal Methods. FM 2019 International Workshops*. Ed. by Emil Sekerinski et al. Cham: Springer International Publishing, 2020, pp. 371–387. DOI: 10.1007/978-3-030-54997-8_24.

[4] Eelco Visser, Zine-el-Abidine Benaissa, and Andrew Tolmach. "Building program optimizers with rewriting strategies". In: *SIGPLAN Not.* 34.1 (Sept. 1998), pp. 13–26. DOI: 10.1145/291251.289425.