RAMiCS 2015

15th Int. Conf. on Relational and Algebraic Methods in Computer Science

Proceedings of the PhD/MSc Student Track

Short papers / Extended abstracts

RAMiCS 2015, Hotel do Parque, Bom Jesus,
Braga, 28 Sep–01 Oct, 2015

Local Organization by
HASLab / INESC TEC and University of Minho

Edited by

Wolfram Kahl, Michael Winter and José N. Oliveira

Preface

The RAMiCS conference series is the main forum for Relational and Algebraic Methods in Computer Science. Special focus lies on formal methods for software engineering, logics of programs and links with neighbouring disciplines.

This conference series finds its origins in the 38th Banach Semester on Algebraic Methods in Logic and their Computer Science Application in Warsaw, Poland, September and October 1991. Adapting essentially a one-and-a-half year rhythm, the first eleven RelMiCS conferences were held from 1994 to 2009 on all inhabited continents except Australia. Starting with RelMiCS 7, these were were held as joint events with Applications of Kleene Algebras (AKA) conferences. At RelMiCS 11 / AKA 6 in Doha, Qatar, it was decided to continue the series under the unifying name Relational and Algebraic Methods in Computer Science (RAMiCS). The next events, RAMiCS 12-14, were then held in Rotterdam, Netherlands, in 2011, Cambridge, UK, in 2012 and Marienstatt, Germany, in 2014.

The 15th International Conference on Relational and Algebraic Methods in Computer Science (RAMiCS 2015) was held in Braga, Portugal from September 28 to October 1, 2015. Further to invited and regular contributions, whose proceedings will be published as volume 9348 in the Lecture Notes in Computer Science (LNCS) series by Springer Verlag, the call for papers also invited researchers doing a PhD or an MSc in the areas of the RAMiCS conference to submit a short description of their ongoing work for presentation at the conference, in the form of extended abstracts not published nor submitted for publication elsewhere.

This was intended as an excellent opportunity for students to discuss their on-going work with leading experts in this field. This technical report includes the 8 contributions accepted by this "student track" of RAMiCS 2015.

September 2015

Wolfram Kahl
Michael Winter
José N. Oliveira

Acknowledgments

The RAMiCS conference was organized by *HASLab – High Assurance Software Laboratory* (http://haslab.uminho.pt), a research unit of the INESC TEC Associated Laboratory located at the University of Minho in Braga. All facilities granted by *Departamento de Informática* and *Escola de Engenharia* are gratefully acknowledged. We also thank FCT (Fundação para a Ciência e Tecnologia) for their sponsorship.

This tecnical report was type-set in LaTeX using Springer-Verlag's class package `llncs.cls`.

Table of Contents

Decision Methods for Concurrent Kleene Algebra with Tests : Based on Derivative

Yoshiki Nakamura

Tokyo Instutute of Technology, Ookayama, Meguroku, Japan,
nakamura.y.ay@m.titech.ac.jp

Abstract. *Concurrent Kleene Algebra with Tests* (**CKAT**) were introduced by Peter Jipsen[Jip14]. We give derivatives for **CKAT** to decide word problems, for example emptiness, equivalence, containment problems. These derivative methods are expanded from derivative methods for Kleene Algebra and Kleene Algebra with Tests[Brz64][Koz08][ABM12]. Additionally, we show that the equivalence problem of **CKAT** is in EXPSPACE.

Keywords: concurrent kleene algebras with tests, series-parallel strings, Brzozowski derivative, computational complexity

1 Introduction

In this paper, we assume [Jip14, theorem 1] and we use **CKAT** terms as expressions of guarded series-parallel language.

Let Σ be a set of *basic program* symbols $\mathbf{p}_1, \mathbf{p}_2, \ldots$ and T a set of *basic boolean test* symbols $\mathbf{t}_1, \mathbf{t}_2, \cdots$, where we assume that $\Sigma \cap T = \emptyset$. Each $\alpha_1, \alpha_2, \ldots$ denotes a subset of T. *Boolean term* b and **CKAT** *term* p over T and Σ are defined by the following grammar, respectively.

$$b := \mathbf{0} \mid \mathbf{1} \mid \mathbf{t} \in T \mid b_1 + b_2 \mid b_1 b_2 \mid \overline{b_1}$$

$$p := b \mid \mathbf{p} \in \Sigma \mid p_1 + p_2 \mid p_1 p_2 \mid p_1^* \mid p_1 \parallel p_2$$

The guarded series-parallel strings set $GS_{\Sigma,T}$ over Σ and T is a smallest set such that follows

- $\alpha \in GS_{\Sigma,T}$ for any $\alpha \subseteq T$
- $\alpha_1 \mathbf{p} \alpha_2 \in GS_{\Sigma,T}$ for any $\alpha_1, \alpha_2 \subseteq T$ and any basic program $\mathbf{p} \in \Sigma$
- if $w_1\alpha, \alpha w_2 \in GS_{\Sigma,T}$, then $w_1 \alpha w_2 \in GS_{\Sigma,T}$.
- if $\alpha_1 w_1 \alpha_2, \alpha_1 w_2 \alpha_2 \in GS_{\Sigma,T}$, then $\alpha_1 \{|w_1, w_2|\} \alpha_2 \in GS_{\Sigma,T}$.

Definition 1 (guarded series-parallel language). *Let $\diamond$ and $\parallel$ be binary operators over $GS_{\Sigma,T}$, respectably. They are defined as follows.*

$$w_1 \diamond w_2 = \begin{cases} w_1' \alpha w_2' & (w_1 = w_1'\alpha \text{ and } w_2 = \alpha w_2') \\ undefined & (o.w.) \end{cases}$$

In particular, if $w_1 = w_2 = \alpha$, then $w_1 \diamond w_2 = \alpha$.

$$w_1 \parallel w_2 = \begin{cases} \alpha_1\{|w_1', w_2'|\}\alpha_2 & (w_1 = \alpha_1 w_1' \alpha_2 \text{ and } w_2 = \alpha_1 w_2' \alpha_2) \\ \alpha & (w_1 = w_2 = \alpha) \\ undefined & (o.w.) \end{cases}$$

L is a map from **CKAT** *terms over Σ and T to this concrete model by*

- $L(\mathbf{0}) = \emptyset$, $L(\mathbf{1}) = 2^T$
- $L(\mathbf{t}) = \{\alpha \subseteq T \mid \mathbf{t} \in \alpha\}$ *for* $\mathbf{t} \in T$
- $L(\overline{b}) = 2^T \setminus L(b)$
- $L(\mathbf{p}) = \{\alpha_1 \mathbf{p} \alpha_2 \mid \alpha_1, \alpha_2 \subseteq T\}$ *for* $\mathbf{p} \in \Sigma$
- $L(p_1 + p_2) = L(p_1) \cup L(p_2)$
- $L(p_1 p_2) = \{w_1 \diamond w_2 \mid \ w_1 \in G(p_1)$ *and* $w_2 \in L(p_2)$ *and* $w_1 \diamond w_2$ *is defined* $\}$
- $L(p^*) = \bigcup_{n<\omega} \{\alpha_0 \diamond w_1 \diamond \cdots \diamond w_n \mid \alpha_0 \subseteq T$ *and* $w_1, \ldots, w_n \in L(p)$ *and* $\alpha_0 \diamond w_1 \cdots \diamond w_n$ *is defined* $\}$
- $L(p_1 \parallel p_2) = \{w_1 \parallel w_2 \mid w_1 \in L(p_1)$ *and* $w_2 \in L(p_2)$ *and* $\ w_1 \parallel w_2$ *is defined* $\}$

We expand L to $\overline{L}(P) = \bigcup_{p \in P} L(p)$, where P is a set of **CKAT** *terms. Furthermore, let $L_\alpha(p) = \{\alpha w \mid \alpha w \in L(p)\}$.*

In guarded series-parallel strings, $\alpha_1\{|w_1, w_2|\}\alpha_2$ has commutative(i.e. $\alpha_1\{|w_1, w_2|\}\alpha_2 = \alpha_1\{|w_2, w_1|\}\alpha_2$). We define $p_1 = p_2$ for two **CKAT** terms p_1 and p_2 as $L(p_1) = L(p_2)$ (by means of [Jip14, Theorem 1]).

2 The Brzozowski derivative for CKAT

Now, we give the naive derivative for **CKAT**. Derivative has applications to many language theoretic problems (e.g. membership problem, emptiness problem, equivalence problem, and so on).

Definition 2 (Naive Derivative). *We define E_α and D_w. They are maps from a* **CKAT** *term to a set of* **CKAT** *terms, respectively. E_α is inductively defined as follows. We expand E_α and D_w to $\overline{E}_\alpha(P) = \bigcup_{p \in P} E_\alpha(p)$ and $\overline{D}_w(P) = \bigcup_{p \in P} D_w(p)$, where P is a set of* **CKAT** *terms, respectively.*

- $E_\alpha(\mathbf{0}) = E_\alpha(\mathbf{p}) = \emptyset$
- $E_\alpha(\mathbf{1}) = E_\alpha(p_1^*) = \{\mathbf{1}\}$
- $E_\alpha(\mathbf{t}) = \begin{cases} \{\mathbf{1}\} & (\mathbf{t} \in \alpha) \\ \emptyset & (o.w.) \end{cases}$
- $E_\alpha(\overline{b}) = \{\mathbf{1}\} \setminus E_\alpha(b)$
- $E_\alpha(p_1 + p_2) = E_\alpha(p_1) \cup E_\alpha(p_2)$
- $E_\alpha(p_1 p_2) = E_\alpha(p_1 \parallel p_2) = E_\alpha(p_1) E_\alpha(p_2)$

D_w is inductively defined as follows.
For $w = \mathbf{q} \mid \{|w_1', w_2'|\}$ and any series-parallel string w',

- $D_{\alpha w \alpha' w' \alpha''}(p) = \overline{D}_{\alpha' w' \alpha''}(D_{\alpha w \alpha'}(p))$
- $D_{\alpha w \alpha'}(p_1 + p_2) = D_{\alpha w \alpha'}(p_1) \cup D_{\alpha w \alpha'}(p_2)$
- $D_{\alpha w \alpha'}(p_1 p_2) = D_{\alpha w \alpha'}(p_1)\{p_2\} \cup E_\alpha(p_1) D_{\alpha w \alpha'}(p_2)$

- $D_{\alpha w \alpha'}(p_1^*) = D_{\alpha w \alpha'}(p_1)\{p_1^*\}$
- $D_{\alpha w \alpha'}(b) = \emptyset$ *for any boolean term* b
- $D_{\alpha \mathbf{q} \alpha'}(\mathbf{p}) = \begin{cases} \{\mathbf{1}\} & (\mathbf{p} = \mathbf{q}) \\ \emptyset & (o.w.) \end{cases}$
- $D_{\alpha \mathbf{q} \alpha'}(p_1 \parallel p_2) = \emptyset$
- $D_{\alpha \{|w_1, w_2|\} \alpha'}(\mathbf{p}) = \emptyset$
- $D_{\alpha \{|w_1, w_2|\} \alpha'}(p_1 \parallel p_2) = E_{\alpha'}((D_{\alpha w_1 \alpha'}(p_1) \parallel D_{\alpha w_2 \alpha'}(p_2)) \cup (D_{\alpha w_1 \alpha'}(p_2) \parallel D_{\alpha w_2 \alpha'}(p_1)))$

The *left-quotient* of $L \subseteq GS_{\Sigma,T}$ with regard to $w \in GS_{\Sigma,T}$ is the set $w^{-1}L = \{w' \mid w \diamond w' \in L\}$.

Lemma 1. *For any series-parallel string* $\alpha w \alpha'$*,*

1. $\mathbf{1} \in E_\alpha(p) \iff \alpha \in L_\alpha(p)$
2. $(\alpha w \alpha')^{-1} L_\alpha(p) = \overline{L}_{\alpha'}(D_{\alpha w \alpha'}(p))$

Proof (Sketch). 1. is proved by induction on the size of p*.*
2. is proved by double induction on the size of w *and the size of* p*.*

We can decide whether $\alpha w \alpha' \in L(p)$ to check $\mathbf{1} \in \overline{E}_{\alpha'}(D_{\alpha w \alpha'}(p))$ by Lemma 1. We now define *efficient derivative*. This derivative is another definition of derivative for **CKAT**. This derivative is useful for giving more efficient algorithm than naive derivative in computational complexity. (In naive derivative, we should memorize w_1 and w_2 to get $D_{\alpha\{|w_1,w_2|\}\alpha'}(p)$. In particular, the size of w_1 and w_2 can be double exponential size of input size in equivalence problem.) We expand **CKAT** terms to express efficient derivative. We say these terms *intermediate* **CKAT** *terms*. *Intermediate* **CKAT** *term* is defined as following.

Definition 3 (intermediate CKAT term). Intermediate **CKAT** term *is defined by the following grammar.*

$$q := b \mid \mathbf{p} \in \Sigma \mid q_1 + q_2 \mid q_1 q_2 \mid q_1^* \mid q_1 \parallel q_2 \mid D_x(q_1)$$

We call x a *derivative variable* of $D_x(q_1)$.

The *efficient derivative* $d_{pr}(q)$ is defined in Definition 4, where q is an intermediate **CKAT** term, pr is a sequence of assignments formed $x \mathrel{+}= \alpha\mathbf{p}$ or $x \mathrel{+}= \alpha\mathcal{T}$ (The sequence of assignments pr is formed $x_1 \mathrel{+}= term_1; \ldots; x_m \mathrel{+}= term_m$.) and $\mathcal{T}$ is formed by the following grammar. $\mathcal{T} := \{|x_l\mathcal{T}_l, x_r\mathcal{T}_r|\} \mid \{|x_l\mathcal{T}_l, \mathbf{p}_r x_r|\} \mid \{|\mathbf{p}_l x_l, x_r\mathcal{T}_r|\} \mid \{|\mathbf{p}_l x_l, \mathbf{p}_r x_r|\}$. Intuitively, $d_{x+=\alpha w}(\ldots D_x(q)\ldots)$ means $(\ldots D_x(\breve{D}_{\alpha w}(\mathrm{join}_\alpha(q)))\ldots)$.

Definition 4. *The* efficient derivative $d_{pr}(q)$ *is inductively defined as follows, where we assume that any derivative variable occurred in* $\mathcal{T}$ *are different. To define* $d_{pr}(q)$*, we also define* $\breve{D}_{\alpha w}$ *and* join_α*. We expand* d_{pr} *to* $\overline{d}_{pr}(Q) = \bigcup_{q \in Q} d_{pr}(q)$*, where* Q *is a set of intermediate* **CKAT** *terms. We also expand* join_α *to* $\overline{join}_\alpha(Q) = \bigcup_{q \in Q} \overline{join}_\alpha(q)$*.*

- $d_{x+=\alpha w;pr'}(q) = \overline{d}_{pr'}(d_{x+=\alpha w}(q))$
- $d_{x+=\alpha w}(b) = \{b\}$
- $d_{x+=\alpha w}(\mathbf{p}) = \{\mathbf{p}\}$
- $d_{x+=\alpha w}(q_1 + q_2) = d_{x+=\alpha w}(q_1) \cup d_{x+=\alpha w}(q_2)$
- $d_{x+=\alpha w}(q_1 q_2) = d_{x+=\alpha w}(q_1) d_{x+=\alpha w}(q_2)$
- $d_{x+=\alpha w}(q_1^*) = d_{x+=\alpha w}(q_1)^*$
  $(= \{q'^* \mid q' \in d_{x:=\alpha w}(q_1)\})$
- $d_{x+=\alpha w}(q_1 \parallel q_2) = d_{x+=\alpha w}(q_1) \parallel d_{x+=\alpha w}(q_2)$
  $(= \{q_1' \parallel q_2' \mid q_1' \in d_{x+=\alpha w}(q_1), q_2' \in d_{x+=\alpha w}(q_2)\})$
- $d_{x+=\alpha w}(D_y(q_1)) = \overline{D}_y(d_{x+=\alpha w}(q_1))$
- $d_{x+=\alpha w}(D_x(q_1)) = \overline{D}_x(\breve{D}_{\alpha w}(\overline{join}_\alpha(q_1)))$
- $\breve{D}_{\alpha \mathbf{p}}(q) = D_{\alpha \mathbf{p}}(q)$
- $\breve{D}_{\alpha \mathcal{T}}(b) = \breve{D}_{\alpha \mathcal{T}}(\mathbf{p}) = \emptyset$
- $\breve{D}_{\alpha \mathcal{T}}(q_1 + q_2) = \breve{D}_{\alpha \mathcal{T}}(q_1) \cup \breve{D}_{\alpha \mathcal{T}}(q_2)$
- $\breve{D}_{\alpha \mathcal{T}}(q_1 q_2) = \breve{D}_{\alpha \mathcal{T}}(q_1)\{q_2\} \cup E_\alpha(q_1)\breve{D}_{\alpha \mathcal{T}}(q_2)$
- $\breve{D}_{\alpha \mathcal{T}}(q_1^*) = \breve{D}_{\alpha \mathcal{T}}(q_1)\{q_1^*\}$
- $\breve{D}_{\alpha \mathcal{T}}(q_1 \parallel q_2) = \begin{cases} (\overline{D}_{x_l}(\breve{D}_{\alpha \mathbf{p}_l}(q_1)) \parallel \overline{D}_{x_r}(\breve{D}_{\alpha \mathbf{p}_r}(q_2))) \\ \cup (\overline{D}_{x_r}(\breve{D}_{\alpha \mathbf{p}_r}(q_1)) \parallel \overline{D}_{x_l}(\breve{D}_{\alpha \mathbf{p}_l}(q_2))) & (\mathcal{T} = \{|\mathbf{p}_l x_l, \mathbf{p}_r x_r|\}) \\ (\overline{D}_{x_l}(\breve{D}_{\alpha \mathcal{T}_l}(q_1)) \parallel \overline{D}_{x_r}(\breve{D}_{\alpha \mathbf{p}_r}(q_2))) \\ \cup (\overline{D}_{x_r}(\breve{D}_{\alpha \mathbf{p}_r}(q_1)) \parallel \overline{D}_{x_l}(\breve{D}_{\alpha \mathcal{T}_l}(q_2))) & (\mathcal{T} = \{|\mathcal{T}_l x_l, \mathbf{p}_r x_r|\}) \\ (\overline{D}_{x_l}(\breve{D}_{\alpha \mathbf{p}_l}(q_1)) \parallel \overline{D}_{x_r}(\breve{D}_{\alpha \mathcal{T}_r}(q_2))) \\ \cup (\overline{D}_{x_r}(\breve{D}_{\alpha \mathcal{T}_r}(q_1)) \parallel \overline{D}_{x_l}(\breve{D}_{\alpha \mathbf{p}_l}(q_2))) & (\mathcal{T} = \{|\mathbf{p}_l x_l, \mathcal{T}_r x_r|\}) \\ (\overline{D}_{x_l}(\breve{D}_{\alpha \mathcal{T}_l}(q_1)) \parallel \overline{D}_{x_r}(\breve{D}_{\alpha \mathcal{T}_r}(q_2))) \\ \cup (\overline{D}_{x_r}(\breve{D}_{\alpha \mathcal{T}_r}(q_1)) \parallel \overline{D}_{x_l}(\breve{D}_{\alpha \mathcal{T}_l}(q_2))) & (\mathcal{T} = \{|\mathcal{T}_l x_l, \mathcal{T}_r x_r|\}) \end{cases}$
- $join_\alpha(b) = \{b\}$, $join_\alpha(\mathbf{p}) = \{\mathbf{p}\}$
- $join_\alpha(q_1 + q_2) = join_\alpha(q_1) \cup join_\alpha(q_2)$, $join_\alpha(q_1 q_2) = join_\alpha(q_1) join_\alpha(q_2)$
- $join_\alpha(q_1 \parallel q_2) = join_\alpha(q_1) \parallel join_\alpha(q_2)$
- $join_\alpha(q_1^*) = join_\alpha(q_1)^*$
- $join_\alpha(D_y(q)) = \overline{E}_\alpha(join_\alpha(q))$

Efficient derivative is essentially equal to the derivative of Definition 1. Let $sp_x(pr)$ be the string corresponded to x of pr. (For example, $sp_{x_0}(x_0 \mathrel{+}= \alpha\{\mathbf{p}_1 x_1, \mathbf{p}_2 x_2\}; x_1 \mathrel{+}= \alpha' \mathbf{p}_3; x_0 \mathrel{+}= \alpha'' \mathbf{p}_4) = \alpha\{\mathbf{p}_1 \alpha' \mathbf{p}_3, \mathbf{p}_2\}\alpha'' \mathbf{p}_4$. $sp_{x_1}(x_0 \mathrel{+}= \alpha\{\mathbf{p}_1 x_1, \mathbf{p}_2 x_2\}; x_1 \mathrel{+}= \alpha' \mathbf{p}_3; x_0 \mathrel{+}= \alpha'' \mathbf{p}_4) = \alpha \mathbf{p}_1 \alpha' \mathbf{p}_3$)

Lemma 2. $\overline{join}_{\alpha'}(\overline{d}_{pr}(D_x(p))) = \overline{E}_{\alpha'}(D_{sp_x(pr)\alpha'}(p))$

By Lemma 1 and Lemma 2, $sp_x(pr)\alpha' \in L(p) \iff \mathbf{1} \in \overline{\mathrm{join}}_{\alpha'}(\overline{d}_{pr}(D_x(p)))$. Therefore, we can use effective derivative instead of naive derivative.

Next, we define the *size* of a intermediate **CKAT** term q, denoted by $|q|$ as follows.

- $|\mathbf{0}| = |\mathbf{1}| = |\mathbf{t}| = |\mathbf{p}| = 1$
- $|\overline{b}| = 1 + |b|$
- $|q_1^*| = |D_x(q_1)| = 1 + |q_1|$

– $|q_1 + q_2| = |q_1 q_2| = |q_1 \parallel q_2| = 1 + |q_1| + |q_2|$

Definition 5 (Closure). *Cl_X is a map from a intermediate* **CKAT** *term to a set of intermediate* **CKAT** *terms, where X is a set of intersection variables. Cl_X is inductively defined as follows.*

– $Cl_X(\underline{a}) = \{\underline{a}\}$ *for* $a = \mathbf{0} \mid \mathbf{1} \mid \mathbf{t}$
– $Cl_X(\overline{b}) = \{\overline{b}\} \cup Cl_X(b)$ *for any boolean term* b
– $Cl_X(\mathbf{p}) = \{\mathbf{p}, \mathbf{1}\}$
– $Cl_X(q_1 + q_2) = \{q_1 + q_2\} \cup Cl_X(q_1) \cup Cl_X(q_2)$
– $Cl_X(q_1 q_2) = \{q_1 q_2\} \cup Cl_X(q_1)\{q_2\} \cup Cl_X(q_2)$
– $Cl_X(q_1^*) = \{q_1^*\} \cup Cl_X(q_1)\{q_1^*\}$
– $Cl_X(q_1 \parallel q_2) = \{q_1 \parallel q_2\} \cup \{D_{x_1}(q_1') \parallel D_{x_2}(q_2') \mid q_1' \in Cl_X(q_1), q_2' \in Cl_X(q_2), x_1, x_2 \in X\}$
– $Cl_X(D_x(q_1)) = \{D_x(q_1)\} \cup \overline{D}_x(Cl_X(q_1))$

We expand Cl_X to $\overline{Cl}_X(Q) = \bigcup_{q \in Q} Cl_X(q)$, where Q is a set of intermediate **CKAT** terms. $\overline{Cl}_X$ is a closed operator. In other words, $\overline{Cl}_X$ satisfies (1) $Q \subseteq \overline{Cl}_X(Q)$, (2) $Q_1 \subseteq Q_2 \Rightarrow \overline{Cl}_X(Q_1) \subseteq \overline{Cl}_X(Q_2)$ and (3) $\overline{Cl}_X(\overline{Cl}_X(Q)) = \overline{Cl}_X(Q)$. We also define the intersection width $iw(q)$ over intermediate **CKAT** terms and $iw(w)$ over $GI_{\Sigma,T}$ as follows.

– $iw(b) = iw(\mathbf{p}) = 1$ for any boolean term b and any basic program $\mathbf{p} \in \Sigma$
– $iw(q_1 + q_2) = iw(q_1 q_2) = \max(iw(q_1), iw(q_2))$
– $iw(q_1^*) = iw(D_x(q_1)) = iw(q_1)$
– $iw(q_1 \parallel q_2) = 1 + iw(q_1) + iw(q_2)$
– $iw(\alpha) = 1$ for any $\alpha \subseteq T$
– $iw(\alpha_1 \mathbf{p} \alpha_2) = 1$
– $iw(w_1 \alpha w_2) = \max(iw(w_1 \alpha), iw(\alpha w_2))$
– $iw(\alpha_1 \{|w_1, w_2|\} \alpha_2) = 1 + iw(w_1) + iw(w_2)$

Lemma 3 (closure is bounded). *For any intermediate* **CKAT** *term q and any sequence of program pr and any set of derivative variables X, where X contains any derivative variables in pr,*

$$|Cl_X(q)| \leq 2 * |X|^{2 * iw(q)} * |q|^{iw(q)}$$

Proof (Sketch). This is proved by induction on the structure of q. We only consider the case of $q = q_1 \parallel q_2$.

$$\begin{aligned}
|Cl_X(q_1 \parallel q_2)| &\leq 1 + |X| * |Cl_X(q_1)| * |X| * |Cl_X(q_2)| \\
&\leq 1 + |X|^2 * 2 * |q_1|^{iw(q_1)} * |X|^{2 * iw(q_1)} * 2 * |q_2|^{iw(q_2)} * |X|^{2 * iw(q_2)} \\
&= 1 + 4 * |X|^{2 * iw(q_1 \parallel q_2)} * |q_1|^{iw(q_1)} * |q_2|^{iw(q_2)} \\
&\leq 2 * |X|^{2 * iw(q_1 \parallel q_2)} * (|q_1| + |q_2|)^{iw(q_1) + iw(q_2)} \\
&\leq 2 * |X|^{2 * iw(q_1 \parallel q_2)} * |q_1 \parallel q_2|^{iw(q_1 \parallel q_2)}
\end{aligned}$$

Lemma 4 (derivative is closed). *For any intermediate* **CKAT** *term q and any sequence of program pr and any set of derivative variables X, where X contains any derivative variables in pr,*

$$d_{pr}(q) \subseteq Cl_X(q)$$

Proof (Sketch). This is proved by double induction on the size of pr and the size of q.

3 CKAT equational theory is in EXPSPACE

By Lemma 1 and Lemma 2, $L(p_1) = L(p_2)$ iff $\overline{\mathrm{join}}_{\alpha'}(\overline{d}_{pr}(D_x(p_1))) = \overline{\mathrm{join}}_{\alpha'}(\overline{d}_{pr}(D_x(p_2)))$ for any pr and any α'. Thus we find some pr such that $\overline{\mathrm{join}}_{\alpha'}(\overline{d}_{pr}(D_x(p_1))) \neq \overline{\mathrm{join}}_{\alpha'}(\overline{d}_{pr}(D_x(p_2)))$ to decide $p_1 \neq p_2$. We must consider all the patterns of pr at first glance. But, we need not to check if pr is too long. We are enough to check the cases of $iw(sp(pr)) \leq \max(iw(p_1), iw(p_2))(\leq l)$ by the following Lemma 5.

Lemma 5. *If* $iw(sp(pr)) > iw(q)$, $d_{pr}(q) = \emptyset$.

By Lemma 5, we are enough to check the case of $iw(sp(pr)) \leq \max(iw(p_1), iw(p_2)) \leq l$. By $iw(sp(pr)) \leq l$, We are enough to prepare $1 + 3 * (l - 1)$ derivative variables. By Lemma 3, $|Cl_X(q)| \leq 2 * |q|^{iw(q)} * |X|^{2 * iw(q)} \leq 2 * l^l * (1 + 3 * (l - 1))^{2 * l}$. Therefore, $|Cl_X(D_x(p_1))| = O(2^{p(l)})$ and $|Cl_X(D_x(p_2))| = O(2^{p(l)})$, where $p(l)$ is a polynomial function of l.

We can give a nondeterministic algorithm. We nondeterministically select the syntax of pr. (pr is $x \mathrel{+}= \alpha\mathbf{p}$ or $x \mathrel{+}= \alpha\mathcal{T}$.) If there exists a sequent of assignments pr and α' such that $\overline{\mathrm{join}}_{\alpha'}(\overline{d}_{pr}(D_x(p_1))) \neq \overline{\mathrm{join}}_{\alpha'}(\overline{d}_{pr}(D_x(p_2)))$, $p_1 \neq p_2$. Otherwise, $p_1 = p_2$. (See Algorithm 1 if you know more details.)

It holds the Theorem 1 by this algorithm.

Theorem 1. **CKAT** *equivalence problem is in EXPSPACE.*

Corollary 1. *if* $iw(p)$ *is a fixed parameter, then* **CKAT** *equivalence problem is PSPACE-complete.*

Note that PSPACE-hardness is derived by [Hun73].

4 Concluding Remarks

We have given the derivative for **CKAT** and shown that **CKAT** equational theory is in EXPSPACE. We finish with the following some of our future works.

- Is this equivalence problem EXPSPACE-complete? (We expect that this claim is $True$.)
- If we allow ϵ (for example, $\alpha\{|\mathbf{p}, \epsilon|\}\alpha$), can we give efficient derivative? (It become a little difficult because we have to memorize α in the case of $x \mathrel{+}= \alpha\{|\mathbf{p}_1 x_1, \epsilon|\}$. We should give another derivative to show the result like Corollary 1.)

A Pseudo Code

Algorithm 1 Decide $p_1 = p_2$, given two CKAT terms p_1 and p_2

Ensure: Whether $p_1 \neq p_2$ or not?(True or False)
  $step \Leftarrow 0$, $P_1 \Leftarrow \{D_{x_0}(p_1)\}$, $P_2 \Leftarrow \{D_{x_0}(p_2)\}$
  while $step \leq 2^{|Cl_X(D_{x_0}(p_1))|} * 2^{|Cl_X(D_{x_0}(p_2))|}$ **do**
    Let α be a subset of T, which is picked up nondeterministically.
    if $\overline{\text{join}}_\alpha(P_1) \neq \overline{\text{join}}_\alpha(P_2)$ **then**
      return $True$
    end if
    Let pr be $x \mathrel{+}= \alpha\mathbf{p}$ or $x \mathrel{+}= \alpha\mathcal{T}$, which is picked up nondeterministically, where $iw(pr) \leq \max(iw(p_1), iw(p_2))$.
    $step \Leftarrow step + 1$, $P_1 \Leftarrow d_{pr}(P_1)$, $P_2 \Leftarrow d_{pr}(P_2)$
  end while
  return $False$

References

[ABM12] Ricardo Almeida, Sabine Broda, and Nelma Moreira. "Deciding KAT and Hoare Logic with Derivatives". In: *Proceedings Third International Symposium on Games, Automata, Logics and Formal Verification, GandALF 2012, Napoli, Italy, September 6-8, 2012*. 2012, pp. 127–140.

[Brz64] Janusz A Brzozowski. "Derivatives of regular expressions". In: *Journal of the ACM (JACM)* 11.4 (1964), pp. 481–494.

[Hun73] Harry B Hunt III. "On the time and tape complexity of languages I". In: *Proceedings of the fifth annual ACM symposium on Theory of computing*. ACM. 1973, pp. 10–19.

[Jip14] Peter Jipsen. "Concurrent Kleene algebra with tests". In: *Relational and Algebraic Methods in Computer Science*. Springer, 2014, pp. 37–48.

[Koz08] Dexter Kozen. *On the Coalgebraic Theory of Kleene Algebra with Tests*. Tech. rep. `http://hdl.handle.net/1813/10173`. Computing and Information Science, Cornell University, Mar. 2008.

RLE-based Algorithm for Testing Biorders

Oliver Lanzerath

Department of Computer Sciences, Bonn-Rhein-Sieg University of Applied Sciences,
Grantham-Allee 20, 53757 Sankt Augustin, Germany
oliver.lanzerath@smail.inf.hochschule-bonn-rhein-sieg.de
http://www.inf.h-brs.de

Abstract. Binary relations with certain properties such as biorders, equivalences or difunctional relations can be represented as particular matrices. In order for these properties to be identified usually a rearrangement of rows and columns is required in order to reshape it into a recognisable normal form. Most algorithms performing these transformations are working on binary matrix representations of the underlying relations. This paper presents an approach to use the RLE-compressed matrix representation as a data structure for storing relations to test whether they are biorders in a hopefully more efficient way.

Keywords: RLE-XOR · RLE-permutation · biorder

1 Introduction

The matrix representation of a binary relation can be interpreted as a bitmap image, that is, a bit sequence. In many cases the usage of a *run length encoding* (RLE) technique results in a smaller representation of such pictures by shorter codes for lengthy bit strings. Hence, algorithms which use RLE-compressed binary matrices as input may have better runtime complexity on average. A bitvector $\boldsymbol{x}$ consists of alternating series of **0**- and **1**-sequences. Referring to [3], a bitvector can be represented by the lengths of the single sequences as follows.

Run Length Encoding. Let $seq_i \in \left\{\mathbf{0}^j | j \in \mathbb{N}\right\} \cup \left\{\mathbf{1}^j | j \in \mathbb{N}\right\}, i \in \mathbb{N}$, be a sequence with $value(seq_i) = 0, seq_i \in \left\{\mathbf{0}^j | j \in \mathbb{N}\right\}$ and $value(seq_i) = 1, seq_i \in \left\{\mathbf{1}^j | j \in \mathbb{N}\right\}$. Then, a bitvector $\boldsymbol{x} = x_0 ... x_{n-1} \in \{0,1\}^n$ can be represented as $\boldsymbol{x} = seq_1 ... seq_k$, $1 \leq k \leq n, value(seq_i) \neq value(seq_{i+1}), \sum_{i=1}^{k} |seq_i| = n$. The RLE-compression of a vector $\boldsymbol{x}$ is given by the vector

$$\boldsymbol{x}^{rle} = x_0 \left[|seq_1|, ..., |seq_k|\right] \tag{1}$$

Figure 1 shows an example for the RLE-compression of a bitvector. Furthermore we make use of a notation similar to arrays for accessing elements of the RLE-compressed vector, where $\boldsymbol{x}^{rle}[0]$ refers to the leading element x_0 and $\boldsymbol{x}^{rle}[i], 1 \leq i \leq k$, to the following length specifications.[1]

[1] E.g. the RLE-compression of the vector $\boldsymbol{x} = 1100001$ is given by $\boldsymbol{x}^{rle} = 1\,[2,4,1]$ with $\boldsymbol{x}^{rle}[0] = 1, \boldsymbol{x}^{rle}[1] = 2, \boldsymbol{x}^{rle}[2] = 4$ and $\boldsymbol{x}^{rle}[3] = 1$.

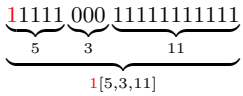


Fig. 1. Example for the RLE-compression of a bitvector.

Biorders. *Biorders* can be defined in different ways. First we introduce the formal definition[4]: Let $R \subseteq X \times X$ be a homogeneous binary relation. R is called biorder (or: Ferrers relation in heterogeneous cases), iff

$$aRb \wedge cRd \wedge \neg aRd \rightarrow cRb$$

holds $\forall\, a, b, c, d \in X$.

In the context of this paper the following definiton is helpful. A binary relation is called a biorder, iff the matrix can be represented in (upper left) *echelon block form*[2] by rearranging rows and columns independently [4]. Thus, it is sufficient to test relations for being biorders by using an algorithm that computes the echelon block form if possible and returns an error otherwise. Figure 2 shows the results of the group stage of group B during the recent FIFA World Cup, where ARB, iff A won the match against B, $A, B \in \{ESP, CHL, NLD, AUS\}$, $A \neq B$. By rearranging rows and then columns the matrix for the relation R

R	*ESP*	*CHL*	*NLD*	*AUS*
ESP	0	0	0	1
CHL	1	0	0	1
NLD	1	1	0	1
AUS	0	0	0	0

R	*ESP*	*CHL*	*NLD*	*AUS*
NLD	1	1	0	1
CHL	1	0	0	1
ESP	0	0	0	1
AUS	0	0	0	0

R	*AUS*	*ESP*	*CHL*	*NLD*
NLD	1	1	1	0
CHL	1	1	0	0
ESP	1	0	0	0
AUS	0	0	0	0

Fig. 2. Group stage FIFA World Cup 2014 (group B).

is transformed into echelon block form proving that R is a biorder.[3] If a given relation is a biorder, the echelon block form can be achieved in two steps [1]:

1. Sort the rows by their *Hamming weight*[4] in descending order.
2. Sort the columns by their Hamming weight in descending order.

A corresponding algorithm would perform these two steps and check if the result is in echelon block form. The time complexity of such an algorithm is in $\mathcal{O}(2n \log n + n^2) \subseteq \mathcal{O}(n^2)$. The following lemma states, that with respect to biorder tests the second step is not needed.

[2] Visually speaking, a binary matrix is in upper left echelon block form, if all 1-entries are placed in the upper left corner and the cardinality of the 1-entries monotonically decreases from top to bottom.

[3] Soccer results do not induce biorders in general. The example is well-chosen here.

[4] We use the notation $\|\boldsymbol{x}\| = \sum_{i=0}^{n-1} x_i$ to note the Hamming weight of a bitvector $\boldsymbol{x}$ (cf. [2]).

Lemma 1. *Let $\boldsymbol{A}$ be a binary $n \times n$-matrix, and let $\boldsymbol{A'}$ be the result of sorting the rows of $\boldsymbol{A}$ by their Hamming weights in descending order. $\boldsymbol{A}$ is a biorder if and only if $\boldsymbol{A'}$ is exclusively composed of column vectors $\boldsymbol{x}_{*i} \in \{0^n, 1^n\} \cup \left\{1^j 0^{n-j} | 1 \leq j \leq n\right\}$.*

Proof. First, we show the if-part by contradiction. Assume that a given $n \times n$-matrix resp. binary relation is a biorder, and after having sorted the rows by their Hamming weights there exists a column vector that does not fulfil the condition. Then there exists at least one column vector in which a **1**-sequence follows a **0**-sequence, i.e. there exists at least one i, $1 \leq i \leq n$, such that $\boldsymbol{x}_{*i} \in \left\{\upsilon 01\omega | \upsilon \in \{0,1\}^j, \omega \in \{0,1\}^{n-j-2}, 0 \leq j \leq n-2\right\}$. Now we sort the columns by their Hamming weights. Let $x_{ki} = 0$ and $x_{li} = 1$ with $k < l$, then $\|\boldsymbol{x}_{k*}\| \geq \|\boldsymbol{x}_{l*}\|$. After having sorted rows and columns the matrix should be in echelon block form because it is a biorder. Then $x_{l0} = x_{l1} = \cdots = x_{li} = 1$ and $x_{k0} = x_{k1} = \cdots = x_{ki} = 1$ because of $\|\boldsymbol{x}_{k*}\| \geq \|\boldsymbol{x}_{l*}\|$. This contradicts the assumption.

The other direction is obvious. If the rows are ordered by Hamming weight and all column vectors are of the desired type, there must be an ordering $\langle \boldsymbol{x}_{*j_1}, \boldsymbol{x}_{*j_2}, ..., \boldsymbol{x}_{*j_n} \rangle$, $j_i \in \{0, ..., n-1\}$ of the column vectors with $\|\boldsymbol{x}_{*j_1}\| \geq \|\boldsymbol{x}_{*j_2}\| \geq ... \geq \|\boldsymbol{x}_{*j_n}\|$. The echelon block form is accomplished by sorting the columns according to this ordering.

2 Algorithm

The complexity of checking all column vectors of a binary matrix for a certain form is still in $\mathcal{O}(n^2)$. If the column vectors are RLE-compressed, the checking can be done in linear time. We only need to verify that all RLE-compressed column vectors have a length of 1 or 2 and start with a **1**-sequence. We assume the relation is represented in RLE-compressed form, i.e. column vectors as well as row vectors are RLE-compressed (c.f. running example in Fig. 3). A biorder-

	ESP	*CHL*	*NLD*	*AUS*
ESP	0	0	0	1
CHL	1	0	0	1
NLD	1	1	0	1
AUS	0	0	0	0

ESP	0[3,1]
CHL	1[1,2,1]
NLD	1[2,1,1]
AUS	0[4]

ESP	*CHL*	*NLD*	*AUS*
0[1,2,1]	0[2,1,1]	0[4]	1[3,1]

Fig. 3. RLE-compressed rows and columns

checking algorithm is defined as follows. First one adjusts the definition of the Hamming weight for RLE-compressed vectors with respect to the new sorting algorithm. If the leading element $\boldsymbol{x}^{rle}[0]$ is 1, all odd positions in the following array refer to 1-entries in the corresponding binary matrix, and vice versa.

Hence, the Hamming weight for RLE-compressed vectors can be defined as:[5]

$$\|\boldsymbol{x}^{rle}\| = \begin{cases} \sum\limits_{i=1, i\in\mathbb{O}_+}^{|\boldsymbol{x}^{rle}|} \boldsymbol{x}^{rle}[i], & \boldsymbol{x}^{rle}[0] = 1 \\ \sum\limits_{i=2, i\in\mathbb{E}_+}^{|\boldsymbol{x}^{rle}|} \boldsymbol{x}^{rle}[i], & \boldsymbol{x}^{rle}[0] = 0 \end{cases} \tag{2}$$

where $\mathbb{O}_+$ and $\mathbb{E}_+$ are the positive odd and even numbers, respectively.

Sorting RLE-compressed row vectors by the Hamming weight causes changes in the column vectors, too. In binary-coded cases this is no point of interest because this change occurs automatically. But, if the vectors are RLE-compressed, it is much more complicated. A permutation of the row vectors $\boldsymbol{x}^{rle}_{j*}$ can have effects on all column vectors $\boldsymbol{x}^{rle}_{*i}$. Assume $\boldsymbol{x}_{j*}$ and $\boldsymbol{x}_{k*}$ are swapped then x_{ji} and x_{ki} must be inverted iff $x_{ji} \neq x_{ki}$. Algorithm 1 implements a bit-compare function for RLE-compressed vectors.

Data: A RLE-compressed n-bitvector $\boldsymbol{x}^{rle}$ and two integers $j, k, 1 \leq j \leq n, 1 \leq k \leq n, j < k$
Result: Boolean: $x_k \neq x_j$
$dist := 0$;
$counter := \boldsymbol{x}^{rle}[1]$;
for $i := 2$ **to** $|\boldsymbol{x}^{rle}|$ **do**
  if $counter \geq j \ \&\& \ counter \leq k$ **then**
    $dist++$;
  end
  $counter += \boldsymbol{x}^{rle}[i]$;
end
if $dist == 1 \ mod \ 2$ **then**
  return $true$;
else
  return $false$;
end

Algorithm 1: $bitCompare$-function for RLE-compressed n-bitvectors

To ensure that in case $x_{ji} \neq x_{ki}$ the corresponding bits must be switched, we use an XOR-operation with a special bit mask:

$\boldsymbol{x}$	x_1	$\cdots$	x_{j-1}	x_j	x_{j+1}	$\cdots$	x_{k-1}	x_k	x_{k+1}	$\cdots$	x_n
mask	0	$\cdots$	0	1	0	$\cdots$	0	1	0	$\cdots$	0
XOR	x_1	$\cdots$	x_{j-1}	$\overline{x_j}$	x_{j+1}	$\cdots$	x_{k-1}	$\overline{x_k}$	x_{k+1}	$\cdots$	x_n

The logical XOR is the essential part of the whole procedure. Algorithm 2 shows how the XOR-operation can be computed on RLE-compressed vectors.

[5] $|\boldsymbol{x}^{rle}|$ is the length of the array or the RLE-compressed vector, respectively.

First the biorder checking algorithm sorts the row vectors $\boldsymbol{x}_{j*}^{rle}$ by their Hamming weights. For each pairwise permutation of rows all column vectors $\boldsymbol{x}_{*i}^{rle}$ must be adjusted if necessary. After that the resulting column vectors must be checked for meeting the conditions of Lemma 1. If each column vector meets the conditions, the underlying relation is a biorder.

3 Examination

As is well known, sorting row vectors of a binary matrix by the Hamming weight is in $\mathcal{O}(n \log n)$, as well as sorting RLE-compressed vectors. But, to check the conditions of Lemma 1 in later process, the column vectors need to be adjusted, too. In the worst case all n column vectors must be modified for each change of rows and, therefore, n XOR-operations have to be computed. The required time for each XOR-operation depends on the length of the RLE-compressed vectors. Hence, the time complexity of the sorting procedure is bounded from above by $\mathcal{O}(n^3 \log n)$. Now, the test for meeting the conditions of Lemma 1 can be done in $\mathcal{O}(n)$.

Data: Two RLE-compressed copies of n-bitvectors $\boldsymbol{x}^{rle},\ \boldsymbol{y}^{rle}$
Result: Result of the XOR $\boldsymbol{z}^{rle}$
$x_{pointer} := 1;$
$y_{pointer} := 1;$
$z_{pointer} := 1;$
$\boldsymbol{z}^{rle}[0] := |\boldsymbol{x}^{rle}[0] - \boldsymbol{y}^{rle}[0]|;$
while $\sum_{i:=1}^{z_{pointer}} \boldsymbol{z}^{rle}[i] < n$ **do**
  if $\boldsymbol{x}^{rle}[x_{pointer}] == \boldsymbol{y}^{rle}[y_{pointer}]$ **then**
    $\boldsymbol{z}^{rle}[z_{pointer}] += \boldsymbol{x}^{rle}[x_{pointer}];$
    $x_{pointer}++;$
    $y_{pointer}++;$
  else if $\boldsymbol{x}^{rle}[x_{pointer}] < \boldsymbol{y}^{rle}[y_{pointer}]$ **then**
    $\boldsymbol{z}^{rle}[z_{pointer}] += \boldsymbol{x}^{rle}[x_{pointer}];$
    $\boldsymbol{y}^{rle}[y_{pointer}] -= \boldsymbol{x}^{rle}[x_{pointer}];$
    $x_{pointer}++;$
    $z_{pointer}++;$
  else
    $\boldsymbol{z}^{rle}[z_{pointer}] += \boldsymbol{y}^{rle}[y_{pointer}];$
    $\boldsymbol{x}^{rle}[x_{pointer}] -= \boldsymbol{y}^{rle}[y_{pointer}];$
    $y_{pointer}++;$
    $z_{pointer}++;$
end

Algorithm 2: XOR on RLE-compressed n-bitvectors

To speed up the runtime of the algorithm one could use a more specific operation than the XOR. Furthermore, generating customized bitmasks for each column during the sorting process could reduce the number of necessary XOR-operations to n. In this paper the logical XOR on RLE-compressed vectors is focused because using it in context of checking relations for being biorders is only one possible use case. An other one could be a simple compare function that returns $true$ if two RLE-compressed vectors are equal and $false$ otherwise. Premising two n-bitvectors $\boldsymbol{x}^{rle}, \boldsymbol{y}^{rle}$ are equal, the bitwise XOR requires n steps and the RLE-XOR requires $|\boldsymbol{x}^{rle}| \cdot 32$ steps (assuming that natural numbers are saved as 32-bit integers). Hence, such an algorithm can reach much better runtimes for sparsely populated or sorted vectors than a bitwise one. A conceivable use case for such an algorithm could be an automated error correction for particular kinds of databases.

4 Conclusion

Advantages and disadvantages of using RLE codes in the context of relational algebraic methods have not been well studied yet. The main contribution of this paper is to motivate further research and analysis of RLE encodings for binary relations. Actually, the presented algorithm is in $\mathcal{O}(n^3)$ but clearly conveys the advantages of using RLE-codes. Our current work focuses on the development of further efficient algorithms for logical operators on RLE codes. Once a sufficient repository is available, these operations can be used for more efficient row-to-row and column-to-column comparison and, hence, sorting. Together with more test procedures for difunctionality and transitivity, we hope to find an algorithm that comes close to the "magic" runtime complexity of $\mathcal{O}(n^{2.3})$.

Acknowledgements. The author wish to express his thanks to Dr. Martin E. Müller and Professor Dr. Kurt-Ulrich Witt for careful reading and useful comments.

References

1. Müller, M.E.: Towards Finding Maximal Subrelations with Desired Properties. In: Höfner, P., Jipsen, P., Kahl, W., Müller, M.E. (eds.) RAMiCS 2014. LNCS, vol. 8428, pp. 344–361. Springer, Heidelberg (2014)
2. Reed, I.: A class of multiple-error-correcting codes and the decoding scheme. Transactions of the IRE Professional Group on Information Theory, vol. 4, no. 4, pp. 38–49 (1954)
3. Salomon, D.: Data compression - The Complete Reference, 4th Edition. Springer, London (2007)
4. Schmidt, G.: Relational Mathematics, Encyclopedia of Mathematics and its Applications, vol. 132. Cambridge University Press, Cambridge (2011)

Relational Equality in the Intensional Theory of Types

Victor Cacciari Miraldo

University of Minho
University of Utrecht

1 Introduction

Relational Algebra [BdM97] has already proven to be a very expressive formalism for calculating with programs. In particular, relational shrinking can be used to derive a program as an optimization of its specification [MO12]. Nevertheless, there is still lack of computer support for relational calculus. Our approach is basically *piggybacking* on Agda[Nor09], an emerging language with a dependent type system, instead of building such a support system from scratch. Agda has a series of features that make it a very interesting target for such system. One such feature is being able to define mix-fix operators, from where its *Equational Reasoning* framework arises. This framework can be modified to closely resemble what a *squiggolist* would write on paper.

The task of encoding Relational Algebra in Martin-Löf's theory of types [ML84], however, is not as straightforward as one might think. There exists two separate efforts in such a direction. One is due to Mu et al., where a library targeted at program refinement is presented [MKJ09]; the other, more general approach, is due to Kahl [Kah14] and provides a complete categorical library for Agda, where the category of relations arises as a specific instantiation. Our goal is somewhat different, making both approaches unsuitable for us.

This *student-track* paper will focus on the difficulties we encountered when encoding relational equality in Agda in way suitable for (automatic) rewriting. We start with a (very) small introduction to Agda, section 2, aimed at readers without any Agda background whatsoever. We, then, explain how to perform syntactical rewrites in Agda, section 3. This should give a fair understanding of whats going to happen on sections 4, 5 and 6. Where we explain the encoding of relations, more specifically relational equality, in Agda and introduces some concepts from Homotopy Type Theory that allows one to fully formalize our model. We conclude on section 7 providing a summary of what was done and an example that illustrates the application of the tool we developed.

2 Agda Basics

In languages such as Haskell or ML, where a Hindley-Milner based algorithm is used for type-checking, values and types are clearly separated. Values are the objects being computed and types are simply tags to *categorize* them. In Agda,

however, this story changes. There is no distinction between types and values, which gives a whole new level of expressiveness to the programmer.

The Agda language is based on the intensional theory of types by Martin-Löf [ML84]. Great advantages arise from doing so, more specifically in the programs-as-proofs mindset. Within the Intensional Theory of Types, we gain the ability to formally state the meaning of quantifiers in a type. We refer the interested reader to [NPS90].

Datatype definitions in Agda resemble Haskell's GADTs syntax [VWPJ06]. Let us illustrate the language by defining fixed-size Vectors. For this, we need natural numbers and a notion of sum first.

```
data Nat : Set where
  Z : Nat
  S : Nat → Nat

_+_ : Nat → Nat → Nat
Z + n = n
(S m) + n = S (m + n)
```

The $Nat : Set$ statement is read as *Nat is of kind* $*$, for the Haskell enthusiast. In a nutshell, Set, or, Set_0, is the first universe, the type of small types. For consistency reasons, Agda has an infinite number of non-cumulative universes inside one another. That is, $\mathsf{Set}_i \nsubseteq \mathsf{Set}_i$ but $\mathsf{Set}_i \subseteq \mathsf{Set}_{i+1}$.

The underscores in the definition of $+$ indicate where each parameter goes. This is how we define mix-fix operators. We can use underscores virtually anywhere in a declaration, as long as the number of underscores coincide with the number of parameters the function expects.

Leaving details aside, and jumping to Vectors, the following a possible declaration of fixed-size vectors in Agda.

```
data Vec (A : Set) : Nat → Set where
  Nil : Vec Z
  Cons : A → Vec A n → Vec A (S n)
```

Here we can already see something different from Nat. The type Vec takes one parameter, which must be a small type, we called it A and it is indexed by a natural number. Given an $A : Set$, $Vec\ A$ has *kind* $Nat \to *$. This correctly models the idea of an inductive type family. For every natural number n, there is one type $Vec\ A\ n$.

Intuitively, if we concatenate a $Vec\ A\ n$ to a $Vec\ A\ m$, we will obtain a $Vec\ A\ (n + m)$. This is exactly how vector concatenation goes in Agda.

```
_++_ : {A : Set} {n m : Nat} → Vec A n → Vec A m → Vec A (n + m)
Nil ++ v = v
(Cons a as) ++ v = Cons a (as ++ v)
```

The parameters enclosed in curly brackets are known as *implicit parameters*. These are values that the compiler can figure out during type-checking, so we need not to worry.

3 Rewriting in Agda

The steps of mathematical reasoning one usually writes on a paper have a fair amount of implicit rewrites. Yet, we cannot skip these steps in a proof assistant. We need to really convince Agda that two things are equal, by Agda's equality notion, before it can rewrite.

In Agda, writing $x \equiv y$ means that x and y *evaluate* to the *same* value. This can be seen from the definition of propositional equality, where we only allow one to construct an equality type using reflexivity:

```
data _≡_ {A : Set} (x : A) : A → Set where
  refl : x ≡ x
```

Having a proof $p : x \equiv y$ convinces Agda that x and y will *evaluate* to the same value. Whenever this is the case, we can rewrite x for y in a predicate. The canonical way to do so is using the *subst* function:

```
subst : {A : Set} (P : A → Set) {x y : A} → x ≡ y → P x → P y
subst P refl p = p
```

Here, the predicate P can be seen as a context where the rewrite will happen. From a programming point of view, Agda's equality notion makes perfect sense! Yet, whenever we are working with more abstract concepts, we might need a finer notion of equality. However, this new equality must agree with Agda's equality if we wish to perform syntactical rewrites. As we will see in the next section, this is not always the case.

It is worth mentioning a subtle detail on the definition of $subst$. Note that, on the left hand side, the pattern p has type $P\ x$, according to the type signature. Still, Agda accepts this same p to finish up the proof of $P\ y$. What happens here is that upon pattern matching on *refl*, Agda knows that x and y evaluate to the same value. Therefore it basically substitutes, in the current goal, every y for x. As we can see here, pattern-matching in Agda actually allows it to infer additional information during type-checking.

4 Relations and Equality in Agda

In order to have a Relational Reasoning framework, we first need to have relations. We follow the same powerset encoding of [MKJ09], and encode a subset of a given set by:

```
ℙ : Set → Set1
ℙ A = A → Set
```

In Agda, Set is the type of types, which allows us to encode a subset of a set A as a function $f : \mathbb{P}\ A$. Such subset is defined by $\{a \in A \mid f\ a \text{ is inhabited}\}$. A simple calculation will let one infer $B \overset{R}{\leftarrow} A = B \to A \to Set$ from $\mathbb{P}\ (A \times B)$. The subrelation notion is intuitively defined by:

$$
\begin{aligned}
&\_ \subseteq \_ : \{A\ B : Set\} \to (B \overset{R}{\leftarrow} A) \to (B \overset{S}{\leftarrow} A) \to Set \\
&R \subseteq S = \forall\ a\ b \to R\ b\ a \to S\ b\ a
\end{aligned}
$$

This makes sense because we are saying that $B \overset{R}{\leftarrow} A$ is a subrelation of $B \overset{S}{\leftarrow} A$ whenever the set $R\ b\ a$ being inhabited implies that the set $S\ b\ a$ is also inhabited, for all $b\ a$. For this matter, a set S being inhabited means that there exist some $s : S$. This follows from the inclusion of (mathematical) sets.

The relation $\subseteq$ is an order: it is reflexive, transitive and anti-symmetric. Reflexivity and transitivity are straight-forward to prove, but there is a catch in anti-symmetry. Remember that relational equality is defined by mutual inclusion:

$$
\begin{aligned}
&\equiv_r : \{A\ B : Set\} \to (B \overset{R}{\leftarrow} A) \to (B \overset{S}{\leftarrow} A) \to Set \\
&R \equiv_r S = R \subseteq S \times S \subseteq R
\end{aligned}
$$

On paper, anti-symmetry follows by construction. But for rewriting purposes in Agda, we need to find a way to prove $R \equiv S$ from $R \equiv_r S$. Unfortunately, this is not possible without additional machinery. We would like to have:

$$
\frac{\begin{array}{l}\forall\ a\ b \to R\ b\ a \to S\ b\ a \\ \forall\ a\ b \to S\ b\ a \to R\ b\ a\end{array}}{R \equiv S}\ \equiv_r \text{ promote}
$$

We could postulate function extensionality[1] and, if the functions are isomorphisms, this would finish the proof. But this is somewhat cheating. By pinpointing the problem one can give a better shot at solving it.

Well, if $R \equiv S$, then $R\ b\ a \equiv S\ b\ a$ at least propositionally. On the other hand, if $R \equiv_r S$ then we might have propositionally different relations. Consider the following relations (where $\mathbb{1}$ is the unit type and $+$ is the coproduct).

$$
\begin{aligned}
&Top : Rel\ \mathbb{N}\ \mathbb{N} \\
&Top\ \_\ \_ = \mathbb{1}
\end{aligned}
\qquad\qquad
\begin{aligned}
&Top' : Rel\ \mathbb{N}\ \mathbb{N} \\
&Top'\ \_\ \_ = \mathbb{1} + \mathbb{1}
\end{aligned}
$$

Although they are equivalent, it is clear that $Top\ b\ a = \mathbb{1} \not\equiv \mathbb{1} + \mathbb{1} = Top'\ b\ a$. To prove propositional equality from relational equality we depend

[1] Function extensionality is expressed by point-wise equality. It is known not to introduce any inconsistency and it is considered to be a safe postulate. In short, given $f, g : A \to B, f \equiv g$ only if $\forall\ x \to f\ x \equiv g\ x$.

on the user not making stupid decisions. We are thus facing a subtle encoding problem.

5 Homotopy Type Theory

Luckily people from the Univalent Foundations have thought of this problem. In this section we will borrow a few concepts from Homotopy Type Theory [Uni13] (HoTT) and discuss how these concepts can contribute to a solution for our encoding. There is no *final* solution, though, since they will boil down to design decisions.

Recalling our problem, given $R \equiv_r S$, $R\ b\ a$ and $S\ b\ a$ might evaluate to different types. But we do not care to which values they evaluate to, as long as one is inhabited iff the other is so. This notion is called *proof-irrelevance* in HoTT jargon, and the sets which are proof irrelevant are called *mere propositions*.

$$
\begin{aligned}
&isProp : Set \to Set \\
&isProp\ P = (p1\ p1 : P) \to p1 \equiv p2
\end{aligned}
$$

Let us denote the set of all *proof irrelevant* types, or $(\Sigma\ Set\ isProp)$, by $\mathbb{MP}$. Should we have defined our relations as $B \to A \to \mathbb{MP}$, our problem would be almost done. The drawback of such a decision is the evident loss of expressiveness, for instance, coproducts are already not proof-irrelevant. Not to mention that users would have to be familiar with such notions before encoding their relations. We chose to encode this as a *typeclass* and, for using a fully formal definition of anti-symmetry, both relations must belong into that typeclass.

There is a very useful result we exploit. Given P a mere proposition. If we find an inhabitant $p : P$, then $P \approx \mathbb{1}$. If we find a contradiction $P \to \bot$, then $P \approx \bot$, where $\approx$ means univalence. For the unfamiliar reader, univalence can be thought of as some sort of isomorphism. The concept is too deep to be introduced in detail here. We refer the reader to [Uni13].

6 Anti-Symmetry of Relational Inclusion

Being a mere proposition is of no interest if we cannot compute the set $R\ a\ b$, for a given R , a and b. Yet, we can also make this explicit in Agda, by means of another *typeclass*. We require our relations to be decidable.

$$
\begin{aligned}
&isDec : \{A\ B : Set\}\ (B \xleftarrow{R} A) \to Set \\
&isDec\ R = (a : A)\ (b : B) \to (R\ b\ a) + (R\ b\ a \to \bot)
\end{aligned}
$$

On these terms, together with function extensionality, we are able to provide a proof of anti-symmetry in Agda's terms. The type[2] is:

[2] The double braces $\{\{\_\}\}$ work almost like Haskell's type context syntax $(Fa) \Rightarrow$.

$$\begin{aligned}&\subseteq\text{−antisym} : \{A\ B : Set\}\ \{R\ S : Rel\ A\ B\}\\&\quad \{\{decr : IsDec\ R\}\}\ \{\{decs : IsDec\ S\}\}\\&\quad \{\{prpr : IsProp\ R\}\}\ \{\{prps : IsProp\ S\}\}\\&\quad \to R \subseteq S \to S \subseteq R \to R \equiv S\end{aligned}$$

Yet, this model of anti-symmetry is too restrictive for the user. During development, base relations might change, which will trigger changes in all of their instances. For this reason, we provide a postulate of the $\equiv_r$-promote rule introduced in section 4. Note, however, that this postulate allows for a contradiction. It is easy to see why. Take the Top and Top' relations defined in section 4. It is easy to prove $Top \equiv_r Top'$, therefore, through $\equiv_r$-promote we have $Top \equiv Top'$, but $\mathbb{1} \not\equiv \mathbb{1} + \mathbb{1}$, and, through $cong\ (const \circ const)$, we have $\lambda \_\ \_ \to \mathbb{1} \not\equiv \lambda \_\ \_ \to \mathbb{1} + \mathbb{1}$. Therefore, $Top \not\equiv Top'$. The actual Agda code for this is trickier than one thinks, hence it is omitted here.

This is not inconsistent with our model, however. Since the definition of a relation is basically the encoding of its defining predicate in Agda, one should not need to use Sets which are not mere propositions. The $\equiv_r$-promote postulate is there to speed up development. As we proved in this section (more details on [Mir15]), if we use decidable mere-propositions in the domain of our relations, no inconsistency arises. The coproduct is not a mere-proposition.

7 Summary and Conclusions

On this very short paper we presented the problem of encoding relational equality in Agda, and gave a few pointers on how to tackle it. This is just a short summary of [Mir15], where we present our library in full. We provide standard relational constructs, including a prototype of catamorphisms, generic on their functor. Our encoding uses *W-types* and is built on top of [AAG04]. We had to build all of these constructs from scratch, since we had the goal of also providing automatic inference of rewriting context for them, which we accomplished. The full Agda code of our relational algebra library, together with the author's master dissertation is available on GitHub.

https://github.com/VictorCMiraldo/msc-agda-tactics

It is arguable that we did not really solve the problem, since our final solution still relies on the univalence axiom ($a \approx b \to a \equiv b$ [Uni13]). We reiterate that there is no final solution to this problem, since different options will lead to libraries with different designs, fit for different purposes.

Here we give a small example of the final product of this project, and how we use the lifting of relational equality to perform generic (automatic) rewrites in Agda. Note that Agda's mix-fix feature allows one to define operators such as the *squiggol* environment. The example below is one branch of the proof that Lists are functors. The `by` function was also developed by us, and it uses the

Reflection mechanism of Agda (similar to Template Haskell) to infer the substitution to be performed.

The rewrites we perform are just $subst$s, as explained in section 3. The automation happens on compile time. The *tactic* keyword gives us the meta-representation of the relevant terms, our engine then generates a meta-representation of the $subst$ that justifies the rewrite step, which is then plugged back in before the compiler resumes type-checking. It is very relevant to state that, although the lemmas justifying the rewrite steps have a $\equiv_r$ as their type, this is then converted to Agda's $\equiv$ in order for us to use $subst$. Relational equality is *not* substitutive, in general.

Let us imagine we are in doubt whether lists are a functor or not. A good start is if they distribute over composition, that is,

$$\mathrm{L}\ (f \cdot g) \equiv \mathrm{L}\ f \cdot \mathrm{L}\ g$$

The proof itself is simple to complete with the universal law for coproducts. We illustrate the i_2 branch in Agda, using our library.

$$
\begin{aligned}
&ex\ :\ (Id + (Id\ \times\ R)\ \circ\ Id + (Id\ \times\ S))\ \circ\ i_2\ \equiv_r\ i_2\ \circ\ Id\ \times\ (R\ \circ\ S)\\
&ex = \mathit{begin}\\
&\qquad (Id + (Id\ \times\ R)\ \circ\ Id + (Id\ \times\ S))\ \circ\ i_2\\
&\equiv_r \qquad \{\ \text{tactic (by (quote } +\text{-bi-functor))}\ \}\\
&\qquad (Id\ \circ\ Id) + (Id\ \times\ R\ \circ\ Id\ \times\ S)\ \circ\ i_2\\
&\equiv_r \qquad \{\ \text{tactic (by (quote } i_2\text{-natural))}\ \}\\
&\qquad i_2\ \circ\ Id\ \times\ R\ \circ\ Id\ \times\ S\\
&\equiv_r \qquad \{\ \text{tactic (by (quote } \times\text{-bi-functor))}\ \}\\
&\qquad i_2\ \circ\ (Id\ \circ\ Id)\ \times\ (R\ \circ\ S)\\
&\equiv_r \qquad \{\ \text{tactic (by (quote } \circ\text{-id-r))}\ \}\\
&\qquad i_2\ \circ\ Id\ \times\ (R\ \circ\ S)\\
&\qquad \square
\end{aligned}
$$

References

[AAG04] Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Representing nested inductive types using w-types. In *In Automata, Languages and Programming, 31st International Colloqium (ICALP), pages 59 71*, pages 59–71, 2004.

[BdM97] R. Bird and O. de Moor. *Algebra of Programming*. Prentice-Hall international series in computer science. Prentice Hall, 1997.

[Kah14] W. Kahl. Rath-agda, relational algebraic theories in agda, Dez 2014.

[Mir15] Victor Cacciari Miraldo. Proofs by rewriting in Agda. Master's thesis, Utrecht University and University of Minho, 2015. Submitted.

[MKJ09] S-C. Mu, H-S. Ko, and P. Jansson. Algebra of programming in agda. *Journal of Functional Programming*, 2009.

[ML84] P. Martin-Löf. Intuitionistic type theory, 1984.

[MO12] Shin-Cheng Mu and Jos Nuno Oliveira. Programming from galois connections. *The Journal of Logic and Algebraic Programming*, 81(6):680 – 704, 2012. 12th International Conference on Relational and Algebraic Methods in Computer Science (RAMiCS 2011).

[Nor09] Ulf Norell. Dependently typed programming in agda. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation*, TLDI '09, pages 1–2, New York, NY, USA, 2009. ACM.

[NPS90] Bengt Nordström, Kent Petersson, and Jan Smith. *Programming in Martin-Löf's Type Theory*. Oxford University Press, 1990.

[Uni13] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. http://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

[VWPJ06] Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. Boxy types: Inference for higher-rank types and impredicativity. *SIGPLAN Not.*, 41(9):251–262, September 2006.

Loop Analysis and Repair

Nafi Diallo, PhD Candidate
Department of Computer Science
Advisor: Ali Mili

New Jersey Institute of Technology

Abstract. This doctoral work proposes to use invariant relations to analyze and repair loops. We discuss how invariant relations allow us to derive loop properties such as termination, correctness and incorrectness and to generate invariant assertions. We also present a method to statically repair a loop using invariant relations, to what we refer as "Debugging without Testing".

1 Introduction

Software quality is of critical importance due to the pervasiveness of software in modern societies, its use in critical applications and its growing size and complexity. The demands on software technology are putting relentless pressure on software research to deliver ever more capable methods, tools, and processes.
For imperative programming languages,still most prevalent in software production, loops still constitute a main source of complexity and to analyze a loop, one must re-engineer the inductive argument that underlies its construction.
Due to the usual difficulty of this task, many approaches proceed by unrolling the loop a number of times, and analyzing the resulting program as a sequential code. In this work, we explore an orthogonal approach based on the concept of invariant relations, which allow approximating the function of a loop. The choice between capturing all the functional details of a loop whose iterations are bounded, and approximating its function for all possible executions, can be viewed as a trade-off between knowing everything about some executions and knowing something about all executions. We argue for the latter approach on the grounds that knowing everything is not necessary (many properties of interest can be established with partial information) and that making claims about bounded executions is not sufficient (a property may hold for bounded executions and fail to hold for unbounded executions).
This work contributes the following:

- A definition of the concept of loop convergence as the integration of termination and abort-freedom
- Proofs of correctness and incorrectness of while loops
- A method to statically prove that a fault is removed and that the resulting program is more correct.

– An automated loop analysis tool via analysis of the source code (to compute artifacts such as convergence condition, loop function, correctness verification)

In the following sections, we discuss related work, describe the proposed methods, and present a description of the plan of evaluation of this work.

2 Background and Related Work

This work is part of an ongoing project. [13] propose to use the relational approach for automated loop function computation via static analysis of the source code. In this context, they provide a definition of invariant relations and how it can be applied to compute the loop function and pre/post conditions.
This work builds on their results and investigates the use of invariant relations for loop analysis and repair with the aim to develop an automated tool for invariant relation generation and the implementation of the proposed methods.

2.1 Loop Analysis

Analysis of loop termination is a very mature area of research starting with the work of Alan Turing. This area can be characterized by a separation between two concerns: termination as a finite number of iterations and termination as abort-freedom. Termination as a finite number of iterations, which has also received the most attention, involves the discovery of well-founded ranking functions, namely transition invariants [2, 15] and size-change graphs [9]. In [2], Cook et. al. give a comprehensive survey of loop termination, in which they discuss transition invariants which are approximations of $(T \cap B)^+$ while invariant relations are approximations of $(T \cap B)^*$. This slight difference of form has a significant impact on the properties and uses of these distinct concepts. While transition invariants are used by Cook et al. to characterize the well founded property of $(T \cap B)^+$, we use invariant relations to approximate the function of a loop, and its domain. Abstract interpretation [3], Model Checking and Bounded Model Checking [6] are techniques aimed at capturing aspects of abort-freedom. Abstract interpretation is a broad scoped technique used to infer properties of programs by successive approximations of their execution traces and thus resembles most our approach. Model checking consists of exhaustively traversing all the reachable states of the system to verify the desired properties. Bounded model checking is a specialization of model checking in which the traversal is halted after a given number of iterations, in which case it is decided that no counter example exists and the property holds.
Overall, our work distinguishes itself with the approaches described above in that, with invariant relations, we can model the termination of while loops in a broad sense; we do that by merging the condition that the number of iterations is finite and the condition that every single iteration executes without causing an abort.

Traditionally, proof of correctness of loops involves two aspects: proof of partial correctness in the sense of Floyd/Hoare Logic and termination [7]. Another approach is based the weakest precondition theory [5].

2.2 Loop Repair

Most existing repair approaches rely on execution traces to identify faults. [8, 10, 11].
In [12] the authors consider an original program P and a variation P' of P, extract semantic information from P, and use it to instrument P' (by means of executable assertions). They then reason about semantic guarantees which can be inferred about the instrumented version of P' and analyze the condition under which both programs can execute without causing an abort (due to attempting an illegal operation), which they approximate by sufficient conditions and necessary conditions. They implement the VMV (*Verification Modulo Versions*) system aimed at exploiting semantic information about P in the analysis of P', and ensuring that the transition from P to P' happens without regression to decide that P' is *correct relative to* P. Their definition of relative correctness differs from the approach of this work, in several aspects: whereas [12] talk about relative correctness between an original program and a subsequent version in the context of adaptive maintenance (where P and P' may be subject to distinct requirements), we talk about relative correctness between an original (faulty) software product and a revised version of the program (possibly still faulty yet more-correct) in the context of corrective maintenance with respect to a fixed requirements specification; whereas [12] use a set of assertions inserted throughout the program as a specification, we use a relation that maps initial states to final states to specify the standards against which absolute correctness and relative correctness is defined; whereas [12] represents program executions by execution traces, we represent program executions by functions mapping initial states into final states; finally, while [12] define a successful execution as a trace that satisfies all the relevant assertions, we define a successful one as simply an initial state/ final state pair that falls with the specification (relation).
In [8] Lahiri et al. introduce *Differential Assertion Checking* for verifying the relative correctness of a program with respect to a previous version of the program. They explore applications of this technique as a tradeoff between soundness (which they concede) and lower costs (which they hope to achieve). Like the approach of the authors of [12] (from the same team), their work uses executable assertions as specifications, represents executions by execution traces, defines successful executions as traces that satisfy all the executable assertions, and targets abort-freedom as the main focus of the executable assertions. They define relative correctness between programs P and P' as the property that P' has a larger set of successful traces and a smallest set of unsuccessful traces than P; and they introduce relative specifications as specifications that capture functionality of P' that P does not have. Our approach differs from [8] in that we reason in terms of the initial and final states, characterize correct executions

by such pairs that belong to the specification, and we make no distinction between abort-freedom and normal functional properties.
In [11], the authors introduce a definition of relative correctness similar to that of [8]. Programs are modeled with trace semantics, and execution traces are compared in terms of executable assertions inserted into P and P'; in order for the comparison to make sense, programs P and P' have to have the same (or similar) structure and/or there must be a mapping from traces of P to traces of P'. When P' is obtained from P by a transformation, and when P' is provably correct relative to P, the transformation in question is called a *verified repair*. The authors introduce an algorithm specialized in deriving program repairs from a predefined catalog targeted to specific program constructs, such as: contracts, initializations, guards, floating point comparisons, etc. Similarly to ([12, 8]), the authors model programs by execution traces and distinguish between two types of failures: contract violations, when functional properties are not satisfied; and run-time errors, when the execution causes an abort; for the reasons we discuss above, we do not make this distinction, and model the two aspects with the same relational framework. They implement their approach in an automated tool based on the static analyzer cccheck, and assess their tool for effectiveness and efficiency.
In [14], Nguyen et al. present an automated repair method based on symbolic execution, constraint solving, and program synthesis; they call their method SemFix, on the grounds that it performs program repair by means of semantic analysis. This method combines three techniques: fault isolation by means of statistical analysis of the possible suspect statements; statement-level specification inference, whereby a local specification is inferred from the global specification and the product structure; and program synthesis, whereby a corrected statement is computed from the local specification inferred in the previous step. The method is organized in such a way that program synthesis is modeled as a search problem under constraints, and possible correct statements are inspected in the order of increasing complexity. When programs are repaired by SemFix, they are tested for (absolute) correctness against some predefined test data suite; as we argue throughout [4], it is not sensible to test a program for absolute correctness after a repair, unless we have reason to believe that the fault we have just repaired is the last fault of the program (how do we ever know that?). By advocating to test for relative correctness, we enable the tester to focus on one fault at a time, and ensure that other faults do not interfere with our assessment of whether the fault under consideration has or has not been repaired adequately.

3 Semantics

We assume the reader familiar with elementary relational mathematics and generally use the notation adapted from [1].
Given a set S defined by the values of some program variables, say x and y, elements of S are denoted by s and expressed as $s = \langle x, y \rangle$. We represent the

x-component and (resp.) y-component of s as $x(s)$ and $y(s)$. When there is no ambiguity, we refer to $x(s)$ as x and $x(s')$ as x' for elements s and s' of S. We refer to S as the state *space* of the program and to elements of S, denoted by s, as the *states* of the program.
A relation on S is a subset of the Cartesian product $S \times S$.
Of interest to us are the following constant relations on some set S:

- the *universal* relation, denoted by $L = S \times S$,
- the *identity* relation, denoted by $I = \{(s, s')|s = s'\}$,
- and the *empty* relation, denoted by ϕ.

Given that by definition relations are sets, set theoretic operations such as union($\cup$), intersection($\cap$) and complement($\overline{R}$, for a relation R) apply to them.
Operations on relations also include:

- the domain, *dom*(R), of R defined by *dom*$(R) = \{s|\exists s' : (s, s') \in R\}$,
- the *range*, *rng*(R), of R, defined *rng*$(R) = \{s'|\exists s : (s, s') \in R\}$,.
- The *converse*, denoted by $\widehat{R}$, and defined by $\widehat{R} = \{(s, s')|(s', s) \in R\}$.
- The *product* of relations, say R and R', denoted by $R \circ R'$ (or RR') and defined by $R \circ R' = \{(s, s')|\exists s'' : (s, s'') \in R \wedge (s'', s') \in R'\}$.
- The *nucleus* of relation R, denoted by $\mu(R)$ and defined by $\mu(R) = R\widehat{R}$.
- The n^{th} *power* of relation R, for natural number n, denoted by R^n and defined by $R^0 = I$, and $R^n = R \circ R^{n-1}$, for $n \geq 1$.
- The *transitive closure* of relation R, denoted by R^+ and defined by $R^+ = \{(s, s')|\exists i > 0 : (s, s') \in R^i\}$.
- The *reflexive transitive closure* of relation R, denoted by R^* and defined by $R^* = I \cup R^+$. We admit without proof that $R^*R^* = R^*$ and that $R^*R^+ = R^+R^* = R^+$.
- The *pre-restriction* (resp. *post-restriction*) of relation R to predicate t is the relation $\{(s, s')|t(s) \wedge (s, s') \in R\}$ (resp. $\{(s, s')|(s, s') \in R \wedge t(s')\}$).

Given a program p on state space S, we let P be the function of p. P is defined as:
$P = \{(s, s')|p\ starts\ execution\ on\ s,\ then\ it\ terminates\ normally\ in\ state\ s'\}$
Normal termination means that the program terminates after a finite number of operations, without causing an abort and returns a well-defined final state.
We consider while loops written in some C-like programming language, of the form `while (t) {b}` and the semantic following definition:
$[\{$`while (t) {b}`$\}] \equiv (T \cap B)^* \cap \widehat{\overline{T}}$.
B is the function of b and T is the vector defined by: $\{(s, s')|t(s)\}$

4 Invariant Relations

Informally, an invariant relation can be described as a binary relation between input states and their corresponding output states obtained by applying zero or more iterations of the loop body. More formally, we define an invariant relation as follows:

Definition 1. *Given a while loop of the form* w =`while t {b}` *on space* S*, a relation* R *on* S *is said to be an* invariant relation *for* w *if and only if it is a reflexive and transitive superset of* $(T \cap B)$.

To illustrate the concept of invariant relation, we consider the loop below on state space $S = \mathbb{N} \times \mathbb{N} \times \mathbb{N}$ where $\mathbb{N}$ is the set of natural numbers. A state $s \in S$ is defined by the integer variables n, f, and k such that a state $s =< n, f, k >$.

```
while (k!=n) {k=k+1; f=f*k;}
```

We consider the following relation: $R = \left\{(s, s') | \frac{f}{k!} = \frac{f'}{k'!}\right\}$.
This relation is reflexive and transitive, since it is the nucleus of a function; to prove that it is a superset of $(T \cap B)$ we compute the intersection $R \cap (T \cap B)$ and easily find that it equals $(T \cap B)$.
Other invariant relations include:
$R_1 = \{(s, s') | n' = n\}$ $R_2 = \{(s, s') | k \leq k'\}$.
One of the main attributes of invariant relations is that they allow us to derive invariant assertions, a key concept in the established approach for proving correctness of while loops. In [7], an invariant assertion α for a while loop w: `while t {b}` with respect to precondition ϕ and postcondition ψ is defined as a predicate on S such that the following are true:

- $\phi \Rightarrow \alpha$,
- $\{\alpha \wedge t\} b \{\alpha\}$,
- $\alpha \wedge \neg t \Rightarrow \psi$.

We consider the factorial example given above to which we add initialization of the variables involved.

```
w: n=n0;k=0;f=1; while (k!=n) {k=k+1; f=f*k;}
```

The reader can easily verify that $f = k!$ and $n = n0$ are invariant assertions for the loop with respect to the precondition $n = n0; f = 1; k = 1;$ and postcondition $f = n0!$
As expressed above, an invariant assertion depends on both the loop body and the context of the loop, namely its precondition and its post condition while invariant relations only involve the loop body.To make the comparison between invariant assertions and invariant relations meaningful, we redefine invariant assertions to be assertions that satisfy the condition $\{\alpha \wedge t\} b \{\alpha\}$ since it is the only condition that depends exclusively on the loop and does not depend on the precondition (as $\psi \Rightarrow \alpha$) nor the postcondition (as $\alpha \wedge \neg t \Rightarrow \psi$). Given a predicate α, let A be defined as a vector on S by: $A = \{(s, s') | \alpha(s)\}$

Definition 2. *Vector* A *is said to be an* invariant assertion *for the while loop* w*:* `while t {b}` *if and only if it satisfies the following condition:* $(A \cap T \cap B) \subseteq \widehat{A}$ *where* T *is a vector defined by predicate* t

The following two propositions, describe the relationships between invariants assertions and invariant relations.

Proposition 1. *Let R be an invariant relation of w:* `while t {b}` *on space S and let C be an arbitrary vector on S. Then $\widehat{R}C$ is an invariant assertion for w.*

Proposition 2. *Given an invariant assertion A, there exists an invariant relation R and a vector C such that $A = \widehat{R}C$.*

5 Loop Analysis

5.1 Convergence

In the spirit of merging the two aspects of termination described in related work, we introduce the concept of convergence as the integration of these two aspects. The following theorem provides a general framework for convergence and illustrates how we only need to model one aspect or another or both by our choice of invariant relation.

Theorem 1. *We consider a while loop w of the form* `w: while (t) {b}` *on space S, and we let R be an invariant relation for w. Then:* $WL \subseteq R\overline{T}$.

While this result provides that any invariant relation gives a necessary condition of termination, smaller invariant relations are favored as they can lead to both a necessary and sufficient condition. In [4], we present a theorem that provides means of capturing aspects of abort-freedom by generating invariant relations of this form : $R = \{(s, s')|\forall u : (s, u) \in B'^{*} \wedge (u, s') \in B'^{+} \Rightarrow u \in dom(B)\}$, where B' is a superset of B. In practice, B' is an approximation of B, derived by focusing on the variables that are involved in abort-prone statements and recording how B transforms them while $dom(B)$ is modeled using the abort condition of interest. For example, to model

- the condition that arithmetic operations in B does not cause overflow, $dom(B)$ will express a clause to the effect that all operations produce a result within the range of representable values.
- the condition of non-zero division in the execution of B, a condition is added in $dom(B)$ that ensures that all divisors in B are non-zero;

To illustrate the concept of convergence, we consider the following example where we assume that we want to avoid a division by zero (variable j).

```
while (i!=0) {i=i-1; j=j+1; k=k-k/j;}
```

We find the following invariant relations:

- $R_1 = \{(s, s')|j \leq j'\}$
- $R_2 = \{(s, s')|i \geq i'\}$
- $R_3 = \{(s, s')|i + j == iP + jP\}$
- $R_4 = \{(s, s')|\forall h : j \leq h < j', 1 + h \neq 0\}$

Using these relations, we compute the following convergence condition:
$(i = 0 \vee (i \geq 1 \wedge j \geq 0) \vee (i > 0 \wedge 1 + i + j \leq 0))$

5.2 Correctness Verification

In [4], two propositions are introduced for computing necessary and sufficient conditions of correctness using invariant relations. We use them to derive an algorithm which generates successive invariant relations, and tests a necessary condition of correctness and a sufficient condition of correctness, until one of three conditions arises:

- either the sufficient condition is true, then we diagnose the loop as *correct*.
- or the necessary condition is false, then we diagnose the loop as *incorrect*.
- or we run out of invariant relations before we reach the conclusions above; in which case we exit with the conclusion that we do not know enough about the loop to rule on its correctness.

6 Loop Repair

Simply put, loop repair consists of removing a fault and proving that the fault has been removed. Our proposed method for loop repair consists of the following.

1. Observation of failure (loop is diagnosed as incorrect by finding an invariant relation that violates the necessary condition of correctness)
2. fault diagnosis (statements involving variables of the invariant relation above are targeted)
3. fault removal (deriving a new invariant that verifies the necessary condition of correctness)
4. proof of relative correctness (defined below)

6.1 Relative Correctness

Definition 3. *Let R be a specification on state space S and let p and p' be two programs on space S whose functions are respectively P and P'.*

- *We say that program p' is* more-correct *than program p with respect to specification R (abbreviated by: $P' \sqsupseteq_R P$) if and only if: $(R \cap P')L \supseteq (R \cap P)L$.*
- *Also, we say that program p' is* strictly more-correct *than program p with respect to specification R (abbreviated by: $P' \sqsupset_R P$) if and only if $(R \cap P')L \supset (R \cap P)L$.*

where L be the universal relation on S

$(R \cap P)L$ is referred to as the *competence domain* of program p; it represents the set of initial states for which the program agrees with the specification R. Thus to be more-correct means to have a larger competence domain. Note for program p' to be more-correct than program p, it does not have to duplicate the behavior of p over the competence domain of p: it may have a different behavior (since R is potentially non-deterministic) provided this behavior is also correct with respect to R; In [4], We illustrate the concept with an example for which $(R \cap P)L = \{1, 2, 3, 4\} \times S$, $(R \cap P')L = \{1, 2, 3, 4, 5\} \times S$, where $S = \{0, 1, 2, 3, 4, 5, 6\}$. Hence p' is more-correct than p with respect to R.

7 Proving Relative Correctness for Loops

In [4], we propose a theorem and an algorithm to the effect that we can achieve the four steps mentioned above for loop repair. To illustrate loop repair, we consider the following loop, where all the variables except `t` are of type `double`, and where a and b are positive constants.

```
w:  while (abs(r-p)>upsilon)
      {t=t+1;n=n+x; m=m-1;l=l*(1+b);k=k+1000;y=n+k;
       w=w+z;z=(1+a)+z;v=w+k;r=(v-y)/y;u=(m-n)/n;d=r-u;}
```

We consider the following specification:

$$R = \{(s,s') | b < a < 1 \wedge x' = x \wedge w' = w - z \times \frac{1-(1+a)^{t'-t}}{a}$$
$$\wedge k' = k + 1000 \times (t'-t) \wedge t \leq t' \wedge 0 < l \leq l' \wedge z > 0 \wedge l \times (1+b)^{-t} = l' \times (1+b)^{-t'}\}.$$

Analysis of this loop by our invariant relation generator derives fourteen invariant relations, five of which are found to be incompatible with the specification. We select the following incompatible invariant relation for remediation: $Q = \{(s,s') | l \times (1+b)^{-\frac{z}{1+a}} = l' \times (1+b)^{-\frac{z'}{1+a}}\}$. To fix this incompatibility, we must alter variable z or variable l. We compute the condition on z and l under which a change in these variables does not alter any of the relevant compatible relations, and we find: $z' \geq z \wedge (l = l' \vee l \times (l' - l) > 0)$. z is chosen and common mutation operators are applied to the statement $\{$`z=(1+a)+z`$\}$ while preserving the condition $z' \geq z$; for each mutant of this statement, we recompute the new invariant relation that substitutes for Q and check whether it is compatible with R. We find that the statement $\{$`z=(1+a)*z`$\}$ produces a compatible invariant relation, and conclude that the loop w' obtained when we replace $\{$`z=(1+a)+z`$\}$ by $\{$`z=(1+a)*z`$\}$ is more-correct than w with respect to R. Running the invariant relations generator on the new loop produces fourteen invariant relations, one of which is incompatible with R (hence w' is indeed incorrect); it seems that by removing the earlier fault we have remedied four invariant relations at once. Applying the same process to w', we find the following loop, which is correct with respect to R:

```
wc: while (abs(r-p)>upsilon)
{t=t+1;n=n+x; m=m+1;l=l*(1+b);
k=k+1000;y=n+k;w=w+z;z=(1+a)*z;
v=w+k;r=(v-y)/y;u=(m-n)/n;d=r-u;}
```

8 Conclusions and Prospects

Presently, we are developing and evolving the tool to automate the methods described above. It currently covers numeric data types and provide artifacts related to convergence and correctness verification. The results of the experiments are promising and encouraging. We intend to assess the effectiveness of

the proposed approach and tool by comparing the tool with tools from other approaches[Astree, Genprog],using relevant benchmarks. While evolving the tool to include more application domains, we plan to further explore the implications and applications of relative correctness, to further derive techniques for proving relative correctness by static analysis of the source code. We also intend to create a new benchmark for convergence and analysis of program repair with relative correctness. This should all contribute as evidence of the significance of this work.

References

1. C. Brink, W. Kahl, and G. Schmidt. *Relational Methods in Computer Science*. Springer Verlag, New York, NY and Heidelberg, Germany, 1997.
2. B. Cook, A. Podelski, and A. Rybalchenko. Proving program termination. *Communications of the ACM*, 54(5), 2011.
3. P. Cousot. Abstract interpretation. Technical Report www.di.ens.fr/čousot/AI/, Ecole Normale Superieure, Paris, France, August 2008.
4. N. Diallo, W. Ghardallou, and A. Mili. Correctness and relative correctness. In *Proceedings, 37th International Conference on Software Engineering*, Firenze, Italy, May 20–22 2015.
5. E. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.
6. S. Falke, D. Kapur, and C. Sinz. Termination analysis of imperative programs using bitvector arithmetic. In *VSTTE*, pages 261–277, 2012.
7. C. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576 – 583, Oct. 1969.
8. S. K. Lahiri, K. L. McMillan, R. Sharma, and C. Hawblitzel. Differential assertion checking. In *Proceedings, ESEC/ SIGSOFT FSE*, pages 345–455, 2013.
9. C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. In *ACM SIGPLAN Notices*, volume 36, pages 81–92. ACM, 2001.
10. C. LeGoues, S. Forrest, and W. Weimer. Current challenges in automatic software repair. *Software Quality Journal*, 21(3):421–443, 2013.
11. F. Logozzo and T. Ball. Modular and verified automatic program repair. In *Proceedings, OOPSLA*, pages 133–146, 2012.
12. F. Logozzo, S. Lahiri, M. Faehndrich, and S. Blackshear. Verification modulo versions: Towards usable verification. In *Proceedings, PLDI*, 2014.
13. A. Mili, S. Aharon, and C. Nadkarni. Mathematics for reasoning about loop. *Science of Computer Programming*, pages 989–1020, 2009.
14. H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: Program repair via semantic analysis. In *Proceedings, ICSE*, pages 772–781, 2013.
15. A. Podelski and A. Rybalchenko. Transition invariants. In *Proceedings, 19th Annual Symposium on Logic in Computer Science*, pages 132–144, 2004.

Monoid Modules and Structured Document Algebra
(Extendend Abstract)

Andreas Zelend

Institut für Informatik, Universität Augsburg, Germany
zelend@informatik.uni-augsburg.de

1 Introduction

Feature Oriented Software Development (e.g. [3]) has been established in computer science as a general programming paradigm that provides formalisms, methods, languages, and tools for building maintainable, customisable, and extensible *software product lines (SPLs)* [8]. An SPL is a collection of programs that share a common part, e.g., functionality or code fragments. To encode an SPL, one can use *variation points (VPs)* in the source code. A VP is a location in a program whose content, called a *fragment*, can vary among different members of the SPL. In [2] a *Structured Document Algebra (SDA)* is used to algebraically describe modules that include VPs and their composition. In [4] we showed that we can reason about SDA in a more general way using a so called *relational predomain monoid module (RMM)*. In this paper we present the following extensions and results: an investigation of the structure of transformations, e.g., a condition when transformations commute, insights into the pre-order of modules, and new properties of predomain monoid modules.

2 Structured Document Algebra

VPs and Fragments. Let $\mathtt{V}$ denote a set of VPs at which fragments may be inserted and $\mathtt{F}(\mathtt{V})$ be the set of *fragments* which may, among other things, contain VPs from $\mathtt{V}$. Elements of $\mathtt{F}(\mathtt{V})$ are denoted by $\mathtt{f}_1, \mathtt{f}_2, \ldots$. There are two special elements, a default fragment $\mathtt{0}$ and an error $\lightning$. An error signals an attempt to assign two or more non-default fragments to the same VP within one module. The addition, or supremum operator $+$ on fragments obeys the following rules:

$$\begin{aligned} \mathtt{0} + \mathtt{x} &= \mathtt{x}\,, & \lightning + \mathtt{x} &= \lightning\,, \\ \mathtt{x} + \mathtt{x} &= \mathtt{x}\,, & \mathtt{f}_\mathtt{i} + \mathtt{f}_\mathtt{j} &= \lightning \ (\mathtt{i} \neq \mathtt{j})\,, \end{aligned}$$

where $\mathtt{x} \in \{\mathtt{0}, \mathtt{f}_\mathtt{i}, \lightning\}$. This structure forms a flat lattice with least element $\mathtt{0}$ and greatest element $\lightning$ and pairwise incomparable $\mathtt{f}_\mathtt{i}$. By standard lattice theory $+$ is commutative, associative and idempotent and has $\mathtt{0}$ as its neutral element.

Modules. A *module* is a partial function $\mathtt{m} : \mathtt{V} \rightsquigarrow \mathtt{F}(\mathtt{V})$. A VP $\mathtt{v}$ is *assigned* in $\mathtt{m}$ if $\mathtt{v} \in \mathtt{dom}(\mathtt{m})$, otherwise *unassigned* or *external*. By using partial functions rather

than relations, a VP can be filled with at most one fragment *(uniqueness)*. The simplest module is the *empty module* $\mathbf{0}$, i.e., the empty partial map.

Module Addition. The main goal of feature oriented programming is to construct programs step by step from reusable modules. In the algebra this is done by module addition ($+$). Addition fuses two modules while maintaining uniqueness (and signaling an error upon a conflict). Desirable properties for $+$ are commutativity and associativity. For module addition, $+$ on fragments is lifted to partial functions:

$$(\mathtt{m}+\mathtt{n})(\mathtt{v}) =_{df} \begin{cases} \mathtt{m}(\mathtt{v}) & \text{if } \mathtt{v} \in \mathtt{dom}(\mathtt{m}) - \mathtt{dom}(\mathtt{n}) \,, \\ \mathtt{n}(\mathtt{v}) & \text{if } \mathtt{v} \in \mathtt{dom}(\mathtt{n}) - \mathtt{dom}(\mathtt{m}) \,, \\ \mathtt{m}(\mathtt{v}) + \mathtt{n}(\mathtt{v}) & \text{if } \mathtt{v} \in \mathtt{dom}(\mathtt{m}) \cap \mathtt{dom}(\mathtt{n}) \,, \\ \text{undefined} & \text{if } \mathtt{v} \notin \mathtt{dom}(\mathtt{m}) \cup \mathtt{dom}(\mathtt{n}) \,. \end{cases}$$

If in the third case $\mathtt{m}(\mathtt{v}) \neq \mathtt{n}(\mathtt{v})$ and $\mathtt{m}(\mathtt{v}), \mathtt{n}(\mathtt{v}) \neq \mathtt{0}$ then $(\mathtt{m}+\mathtt{n})(\mathtt{v}) = \lightning$, thus signalling an error.

The set of modules forms a commutative monoid under $+$ with the neutral element $\mathbf{0}$.

Deletion and Subtraction. For modules $\mathtt{m}$ and $\mathtt{n}$ the *subtraction* $\mathtt{m}-\mathtt{n}$ is defined as

$$(\mathtt{m}-\mathtt{n})(\mathtt{v}) =_{df} \begin{cases} \mathtt{m}(\mathtt{v}) & \text{if } \mathtt{v} \in \mathtt{dom}(\mathtt{m}) - \mathtt{dom}(\mathtt{n}) \,, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Overriding. To allow *overriding*, an operation $\mathbin{\mathtt{+\!\!>}}$ can be defined in terms of subtraction and addition. Module $\mathtt{m}$ overrides $\mathtt{n}$, written $\mathtt{m} \mathbin{\mathtt{+\!\!>}} \mathtt{n}$, if

$$\mathtt{m} \mathbin{\mathtt{+\!\!>}} \mathtt{n} \;=\; \mathtt{m} + (\mathtt{n} - \mathtt{m})$$

This replaces all assignments in $\mathtt{n}$ for which $\mathtt{m}$ also provides a value. $\mathbin{\mathtt{+\!\!>}}$ is associative and idempotent with neutral element $\mathbf{0}$.

Modules $\mathtt{m}$ and $\mathtt{n}$ are called *compatible*, in signs $\mathtt{m} \downarrow \mathtt{n}$, if their fragments coincide on their shared domains, i.e.,

$$\mathtt{m} \downarrow \mathtt{n} \;\Leftrightarrow_{df}\; \forall\, \mathtt{v} \in \mathtt{dom}(\mathtt{m}) \cap \mathtt{dom}(\mathtt{n}) :\; \mathtt{m}(\mathtt{v}) = \mathtt{n}(\mathtt{v}) \,.$$

We have chosen this characterization of compatibility with regard to the non-relational abstract approach, to be pursued in Section 3, which especially needs no converse operation. For a relational treatment see [9]. All submodules of a module are pairwise compatible with each other.

2.1 Transformations

In this section we sketch an extension of SDA, intended to cope with some standard techniques in software refactoring (e.g., [1, 7]). Examples of such techniques are consistent renaming of methods or classes in a large software system. To stay at the same level of abstraction as before, we realize this by a mechanism for generally modifying the fragments in SDA modules.

By a *transformation* or *modification* or *refactoring* we mean a total function $\mathtt{T} : \mathtt{F}(\mathtt{V}) \to \mathtt{F}(\mathtt{V})$. By $\mathtt{T} \cdot \mathtt{m}$ we denote the *application* of $\mathtt{T}$ to a module $\mathtt{m}$. It yields a new module defined by

$$(\mathtt{T} \cdot \mathtt{m})(\mathtt{v}) =_{df} \begin{cases} \mathtt{T}(\mathtt{m}(\mathtt{v})) & \text{if } \mathtt{v} \in \mathtt{dom}(\mathtt{m}) \\ \text{undefined otherwise} . \end{cases}$$

We need to handle the special case of transforming the error fragment $\lightning$. Since we don't want to allow transformations to mask errors that are related to module addition, we add the requirement

$$\mathtt{T}(\lightning) = \lightning \ .$$

We use the convention that $\cdot$ binds stronger than all other operators. Note that, although $\mathtt{T}$ is supposed to be a total function on all fragments, it might well leave many of those unchanged, i.e., act as the identity on them.

2.2 Structure of Transformations

Definition 2.1 A *monoid of transformations* is a structure $\mathtt{F} = (\mathtt{F}, \circ, \mathbb{1})$, where $\mathtt{F}$ is a set of total functions $\mathtt{f} : \mathtt{X} \to \mathtt{X}$ over some set $\mathtt{X}$, closed under function composition $\circ$, and $\mathbb{1}$ the identity function. The pair $(\mathtt{X}, \mathtt{F})$ is called *transformation monoid* of $\mathtt{X}$. By $\mathtt{T}|_{\mathtt{A}}$ we denote the transformation $\mathtt{T}$ restricted to the set $\mathtt{A}$.

We call the set of transformations on fragments Γ. Then, by the above definition, $(\mathtt{F}(\mathtt{V}), \Gamma)$ is the transformation monoid of fragments $\mathtt{F}(\mathtt{V})$ which we abbreviate to Γ. Since these transformations are not necessarily invertible, in general Γ is not a transformation group. We now can extend the list of properties given in [2].

(6) $\mathtt{T} \cdot (\mathtt{m} + \mathtt{n}) = \mathtt{T} \cdot \mathtt{m} + \mathtt{n} \quad \Leftarrow \quad \mathtt{T}|_{\mathtt{ran}(\mathtt{n})} = \mathbb{1}|_{\mathtt{ran}(\mathtt{n})} \ \wedge \ \mathtt{m} \downarrow \mathtt{n}$,
(7) $\mathbb{1} \cdot \mathtt{m} = \mathtt{m}$,
(8) $\mathtt{T} \cdot \mathbf{0} = \mathbf{0}$.

$\mathbf{0}$ being an annihilator, (Property (8)) means that transformations can only change existing fragments rather than create new ones.

Furthermore we can define the *application equivalence* $\approx$ of two transformations $\mathtt{S}, \mathtt{T}$ by $\mathtt{S} \approx \mathtt{T} \Leftrightarrow_{df} \forall \mathtt{m} : \mathtt{S} \cdot \mathtt{m} = \mathtt{T} \cdot \mathtt{m}$.

It is common to undo refactorings, e.g., undo a renaming of a variable. This can be modelled by inverse transformations, denoted by $^{-1}$. When it exists, the inverse of a *stacked*, or composed transformation, is given by $(\mathtt{T} \circ \mathtt{S})^{-1} = \mathtt{S}^{-1} \circ \mathtt{T}^{-1}$. Of course the inverse of $\mathtt{T}$ or $\mathtt{S}$ might not exist, e.g., if $\mathtt{T}$ is not injective.

As stated above transformations are total functions. Since they act as the identity for fragments that should not be modified we define the set of fragments they transform as follows.

Definition 2.2 Let $\mathtt{T} : \mathtt{F}(\mathtt{V}) \to \mathtt{F}(\mathtt{V})$ be a transformation. Then we call $\mathtt{T}_{\mathtt{m}} =_{df} \{\mathtt{f} \in \mathtt{F}(\mathtt{V}) : \mathtt{T}(\mathtt{f}) \neq \mathtt{f}\}$ the *modified fragments of* $\mathtt{T}$ and $\mathtt{T}_{\mathtt{v}} =_{df} \{\mathtt{T}(\mathtt{f}) \in \mathtt{F}(\mathtt{V}) : \mathtt{T}(\mathtt{f}) \neq \mathtt{f}\} = \mathtt{ran}(\mathtt{T}|_{\mathtt{T}_{\mathtt{m}}})$ the *value set* of $\mathtt{T}$.

Restricting transformations to their modified fragments allows us to state situations in which transformations can be omitted or commute.

Lemma 2.3

1. $\mathtt{T} \cdot (\mathtt{S} \cdot \mathtt{m}) = \mathtt{S} \cdot \mathtt{m}$ *if* $\mathtt{T}_\mathtt{m} \subseteq \mathtt{S}_\mathtt{m} \wedge \mathtt{T}_\mathtt{m} \cap \mathtt{S}_\mathtt{v} = \emptyset$.
2. $\mathtt{T}$ *and* $\mathtt{S}$ *commute if* $\mathtt{T}_\mathtt{m} \cap \mathtt{S}_\mathtt{m} = \emptyset \wedge \mathtt{T}_\mathtt{m} \cap \mathtt{S}_\mathtt{v} = \emptyset \wedge \mathtt{T}_\mathtt{v} \cap \mathtt{S}_\mathtt{m} = \emptyset$.

A proof can be found in Appendix A.1.

3 Abstracting from SDA

The set $\mathtt{M}$ of modules, i.e., partial maps $\mathtt{m} : \mathtt{V} \rightsquigarrow \mathtt{F}(\mathtt{V})$, with $+$ and $-$, defined as in subsection 2, forms an algebraic structure $\mathtt{SDA} =_{df} (\mathtt{M}, +, -, 0)$ such that $(\mathtt{M}, +, 0)$ is an idempotent and commutative monoid and which satisfies the following laws for all $\mathtt{l}, \mathtt{m}, \mathtt{n} \in \mathtt{M}$:

1. $(\mathtt{l} - \mathtt{m}) - \mathtt{n} = \mathtt{l} - (\mathtt{m} + \mathtt{n})$,
2. $(\mathtt{l} + \mathtt{m}) - \mathtt{n} = (\mathtt{l} - \mathtt{n}) + (\mathtt{m} - \mathtt{l})$,
3. $0 - \mathtt{l} = 0$,
4. $\mathtt{l} - 0 = \mathtt{l}$.

Definition 3.1 A *monoid module* (m-module) is an algebraic structure $(\mathtt{B}, \mathtt{M}, :)$ where $(\mathtt{M}, +, 0)$ is an idempotent and commutative monoid and $(\mathtt{B}, +, \cdot, 0, 1, \neg)$ is a Boolean algebra in which 0 and 1 are the least and greatest element and $\cdot$ and $+$ denote meet and join. Note that 0 and $+$ are overloaded, like in classical modules or vector spaces. The restriction, or scalar product, $:$ is a mapping $\mathtt{B} \times \mathtt{M} \to \mathtt{M}$ satisfying for all $\mathtt{p}, \mathtt{q} \in \mathtt{B}$ and $\mathtt{m}, \mathtt{n} \in \mathtt{M}$:

$$(\mathtt{p} + \mathtt{q}) : \mathtt{m} = \mathtt{p} : \mathtt{m} + \mathtt{q} : \mathtt{m} \,, \quad (1) \qquad\qquad (\mathtt{p} \cdot \mathtt{q}) : \mathtt{m} = \mathtt{p} : (\mathtt{q} : \mathtt{m}) \,, \quad (4)$$

$$\mathtt{p} : (\mathtt{m} + \mathtt{n}) = \mathtt{p} : \mathtt{m} + \mathtt{p} : \mathtt{n} \,, \quad (2) \qquad\qquad 1 : \mathtt{m} = \mathtt{m} \,, \quad (5)$$

$$0 : \mathtt{m} = 0 \,, \quad (3) \qquad\qquad \mathtt{p} : 0 = 0 \,. \quad (6)$$

We define the natural pre-order on $(\mathtt{M}, +, 0)$ by $\mathtt{m} \leq \mathtt{n} \Leftrightarrow_{df} \mathtt{m} + \mathtt{n} = \mathtt{n}$. Therefore $+$ is isotone in both arguments.

We have choosen the name monoid module following the notion of a module over a ring and because SDA's modules form an idempotent and commutative monoid.

Lemma 3.2

1. *Restriction* $:$ *is isotone in both arguments.*
2. $\mathtt{p} : \mathtt{m} \leq \mathtt{m}$.
3. $\mathtt{p} : (\mathtt{q} : \mathtt{m}) = \mathtt{q} : (\mathtt{p} : \mathtt{m})$

The first claim follows by distributivity, the second by isotony and (5) and the third by (4) and Boolean algebra.

The structure $\mathtt{RMM} = (\mathcal{P}(\mathtt{M}), \mathcal{P}(\mathtt{M} \times \mathtt{N}), :)$, where $:$ is restriction, i.e., $\mathtt{p} : \mathtt{m} = \{(\mathtt{x}, \mathtt{y}) \mid \mathtt{x} \in \mathtt{p} \wedge (\mathtt{x}, \mathtt{y}) \in \mathtt{m}\}$, forms a mono module. To model subtraction we extend mono modules with the predomain operator $^\ulcorner : \mathtt{M} \to \mathtt{B}$.

Definition 3.3 A *predomain monoid module* (predomain m-module) is a structure $(\mathtt{B}, \mathtt{M}, :, ^\ulcorner)$ such that$(\mathtt{B}, \mathtt{M}, :)$ is a m-module and $^\ulcorner : \mathtt{M} \to \mathtt{B}$ satisfies for all $\mathtt{p} \in \mathtt{B}$ and $\mathtt{m} \in \mathtt{M}$:

(d1) $\mathtt{m} \leq \lceil\mathtt{m} : \mathtt{m}$, (d2) $\lceil(\mathtt{p} : \mathtt{m}) \leq \mathtt{p}$.

In a predomain m-module $\lceil\mathtt{m}$ is the least left preserver of $\mathtt{m}$ and $\neg\lceil\mathtt{a}$ is the greatest left annihilator. To justify this we present the following lemma.

Lemma 3.4 *In a predomain m-module* $(\mathtt{B}, \mathtt{M}, :, \lceil\ )$ *for all* $\mathtt{p} \in \mathtt{B}$ *and* $\mathtt{m} \in \mathtt{M}$*:*

(llp) $\lceil\mathtt{m} \leq \mathtt{p} \Leftrightarrow \mathtt{m} \leq \mathtt{p} : \mathtt{m}$, *(gla)* $\mathtt{p} \leq \neg\lceil\mathtt{m} \Leftrightarrow \mathtt{p} : \mathtt{m} \leq 0$.

Note that (llp) does not establish a Galois connection, since there is no greatest element in the monoid $(\mathtt{M}, +, 0)$ per se, cf. [5] A proof can be found in Appendix A.2.

Now we can give more useful properties of the predomain function like isotony or strictness.

Lemma 3.5 *In a predomain m-module* $(\mathtt{B}, \mathtt{M}, :, \lceil\ )$ *for all* $\mathtt{p} \in \mathtt{B}$ *and* $\mathtt{m}, \mathtt{n} \in \mathtt{M}$*:*

1. $\mathtt{m} = 0 \Leftrightarrow \lceil\mathtt{m} = 0$,
2. $\mathtt{m} \leq \mathtt{n} \Rightarrow \lceil\mathtt{m} \leq \lceil\mathtt{n}$,
3. $\mathtt{m} = \lceil\mathtt{m} : \mathtt{m}$,
4. $\lceil(\mathtt{m} + \mathtt{n}) = \lceil\mathtt{m} + \lceil\mathtt{n}$,
5. $\lceil(\mathtt{p} : \mathtt{m}) : \mathtt{m} = \mathtt{p} : \mathtt{m}$,
6. $\lceil(\mathtt{p} : \mathtt{m}) = \mathtt{p} \cdot \lceil\mathtt{m}$.

A proof can be found in Appendix A.3.

Now it is easy to verify that the SDA laws for subtractions also hold in a predomain m-module. Note that the sides change, e.g., *right* distributivity becomes *left* distributivity.

Lemma 3.6 *Assume a predomain m-module* $(\mathtt{B}, \mathtt{M}, :, \lceil\ )$*. Then for all* $\mathtt{l}, \mathtt{m}, \mathtt{n} \in \mathtt{M}$*:*

1. $\lceil(\neg\lceil\mathtt{n} : \mathtt{m}) = \lceil\mathtt{m} \cdot \neg\lceil\mathtt{n}$,
2. $(\neg\lceil\mathtt{n} : 0 = 0)$,
3. $\neg\lceil\mathtt{l} : (\mathtt{m} + \mathtt{n}) = \neg\lceil\mathtt{l} : \mathtt{m} + \neg\lceil\mathtt{l} : \mathtt{n}$,
4. $\neg\lceil(\mathtt{m} + \mathtt{n}) : \mathtt{l} = \neg\lceil\mathtt{n} : (\neg\lceil\mathtt{m} : \mathtt{l})$,
5. $\neg\lceil 0 : \mathtt{m} = \mathtt{m}$,
6. $\neg\lceil\mathtt{m} : \mathtt{m} = 0$,
7. $\neg\lceil\mathtt{n} : \mathtt{m} \leq \mathtt{m}$,
8. $\mathtt{m} \leq \mathtt{n} \Rightarrow \neg\lceil\mathtt{n} : \mathtt{m} = 0$.

A proof can be found in Appendix A.4.

By defining $\lceil\mathtt{m} =_{df} \{\mathtt{x} \mid (\mathtt{x}, \mathtt{y}) \in \mathtt{m}\}$ RMM becomes a predomain m-module and using an RMM over binary functional relations $\mathtt{R} \subseteq \mathtt{V} \times \mathtt{F}(\mathtt{V})$, i.e., $\mathtt{R}^{\smile}; \mathtt{R} \subseteq \mathsf{id}(\mathtt{F}(\mathtt{V}))$, allows us to reason about SDA. As a result, SDA's subtraction $\mathtt{m} - \mathtt{n}$ of modules is equivalent to $\neg\lceil\mathtt{n} : \mathtt{m}$ in the corresponding RMM.

Using SDA's module addition, cf. Section 2, we can investigate the induced natural pre-order.

$$
\begin{aligned}
\mathtt{m} \leq \mathtt{n} &\Leftrightarrow_{df} \mathtt{m} + \mathtt{n} = \mathtt{n} \\
&\Leftrightarrow \left\{ \begin{array}{ll} \mathtt{m}(\mathtt{v}) & \text{if } \mathtt{v} \in \lceil \mathtt{m} - \lceil \mathtt{n} \\ \mathtt{n}(\mathtt{v}) & \text{if } \mathtt{v} \in \lceil \mathtt{n} - \lceil \mathtt{m} \\ \mathtt{m}(\mathtt{v}) + \mathtt{n}(\mathtt{v}) & \text{if } \mathtt{v} \in \lceil \mathtt{m} \cap \lceil \mathtt{n} \\ \mathsf{undefined} & \text{if } \mathtt{v} \notin \lceil \mathtt{m} \cup \lceil \mathtt{n} \end{array} \right\} = \mathtt{n}(\mathtt{v}) \\
&\Leftrightarrow \left\{ \begin{array}{ll} \mathtt{m}(\mathtt{v}) = \mathtt{n}(\mathtt{v}) & \text{if } \mathtt{v} \in \lceil \mathtt{m} - \lceil \mathtt{n} \\ \mathtt{n}(\mathtt{v}) = \mathtt{n}(\mathtt{v}) & \text{if } \mathtt{v} \in \lceil \mathtt{n} - \lceil \mathtt{m} \\ \mathtt{m}(\mathtt{v}) + \mathtt{n}(\mathtt{v}) = \mathtt{n}(\mathtt{v}) & \text{if } \mathtt{v} \in \lceil \mathtt{m} \cap \lceil \mathtt{n} \\ \mathsf{true} & \text{if } \mathtt{v} \notin \lceil \mathtt{m} \cup \lceil \mathtt{n} \end{array} \right. \\
&\Leftrightarrow \left\{ \begin{array}{ll} \mathsf{false} & \text{if } \mathtt{v} \in \lceil \mathtt{m} - \lceil \mathtt{n} \\ \mathsf{true} & \text{if } \mathtt{v} \in \lceil \mathtt{n} - \lceil \mathtt{m} \\ \mathtt{m}(\mathtt{v}) \leq \mathtt{n}(\mathtt{v}) & \text{if } \mathtt{v} \in \lceil \mathtt{m} \cap \lceil \mathtt{n} \\ \mathsf{true} & \text{if } \mathtt{v} \notin \lceil \mathtt{m} \cup \lceil \mathtt{n} \end{array} \right. \\
&\Leftrightarrow \mathtt{v} \in \lceil \mathtt{n} - \lceil \mathtt{m} \vee (\mathtt{v} \in \lceil \mathtt{m} \cap \lceil \mathtt{n} \wedge \mathtt{m}(\mathtt{v}) \leq \mathtt{n}(\mathtt{v})) \vee \mathtt{v} \notin \lceil \mathtt{m} \cup \lceil \mathtt{n} \text{ for any } \mathtt{v} \in \mathtt{V}.
\end{aligned}
$$

Therefore the least element w.r.t. $\leq$ is the empty module $\mathtt{0}$ and the top element is the module $\mathtt{t}$ with $\mathtt{t}(\mathtt{v}) = \lightning$ for any $\mathtt{v} \in \mathtt{V}$.

SDA's overriding operator $\mathtt{m} \rightarrow\!\!\triangleright\, \mathtt{n}$ can also be defined in a predomain m-module: $\mathtt{m} \rightarrow\!\!\triangleright\, \mathtt{n} =_{df} \mathtt{m} + \neg\lceil \mathtt{m} : \mathtt{n}$. In [6] this operator, embedded into a Kleene algebra, is used to update links in pointer structures.

Lemma 3.7 *In a predomain m-module* $(\mathtt{B}, \mathtt{M}, :, \lceil\ )$ *for all* $\mathtt{p} \in \mathtt{B}$ *and* $\mathtt{l}, \mathtt{m}, \mathtt{n} \in \mathtt{M}$*:*

1. $\mathtt{0} \rightarrow\!\!\triangleright\, \mathtt{n} = \mathtt{n}$,
2. $\mathtt{m} \rightarrow\!\!\triangleright\, \mathtt{0} = \mathtt{m}$,
3. $\mathtt{m} \leq \mathtt{m} \rightarrow\!\!\triangleright\, \mathtt{n}$,
4. $\mathtt{m} = \lceil \mathtt{m} : (\mathtt{m} \rightarrow\!\!\triangleright\, \mathtt{n})$,
5. $\lceil (\mathtt{m} \rightarrow\!\!\triangleright\, \mathtt{n}) = \lceil \mathtt{m} + \lceil \mathtt{n}$,
6. $\lceil \mathtt{m} \geq \lceil \mathtt{n} \Rightarrow \mathtt{m} \rightarrow\!\!\triangleright\, \mathtt{n} = \mathtt{m}$,
7. $\mathtt{l} \rightarrow\!\!\triangleright\, (\mathtt{m} + \mathtt{n}) = \mathtt{l} \rightarrow\!\!\triangleright\, \mathtt{m} + \mathtt{l} \rightarrow\!\!\triangleright\, \mathtt{n}$.

A proof can be found in Appendix A.5.

Future work will focus on further properties of transformations and their incorporation into the abstract framework of predomain m-modules.

Acknowledgments I am very grateful to the anonymous referees for helpful comments and suggestions and I thank B. Möller for fruitful discussions and valuable comments.

References

1. Batory, D.: Program refactorings, program synthesis, and model-driven design. In: Krishnamurthi, S., Odersky, M. (eds.) Compiler Construction. LNCS, vol. 4420, pp. 156–171. Springer (2007)
2. Batory, D., Höfner, P., Möller, B., Zelend, A.: Features, modularity, and variation points. Tech. Rep. CS-TR-13-14, The University of Texas at Austin (2013)
3. Batory, D., O'Malley, S.: The design and implementation of hierarchical software systems with reusable components. ACM Transactions Software Engineering and Methodology 1(4), 355–398 (1992)
4. Dang, H.H., Glück, R., Möller, B., Roocks, P., Zelend, A.: Exploring modal worlds. Journal of Logical and Algebraic Methods in Programming 83(2), 135 – 153 (2014), http://www.sciencedirect.com/science/article/pii/S1567832614000058, festschrift in Honour of Gunther Schmidt on the Occasion of his 75th Birthday
5. Desharnais, J., Möller, B., Struth, G.: Kleene algebra with domain. ACM Trans. Comput. Log. 7(4), 798–833 (2006), http://doi.acm.org/10.1145/1183278.1183285
6. Ehm, T.: The Kleene Algebra of Nested Pointer Structures: Theory and Applications. Ph.D. thesis, Universität Augsburg (2005)
7. Kuhlemann, M., Batory, D., Apel, S.: Refactoring feature modules. In: Edwards, S., Kulczycki, G. (eds.) Formal Foundations of Reuse and Domain Engineering. LNCS, vol. 5791, pp. 106–115. Springer (2009)
8. Lopez-Herrejon, R., Batory, D.: A standard problem for evaluating product-line methodologies. In: Bosch, J. (ed.) GCSE '01: Generative and Component-Based Software Engineering. LNCS, vol. 2186, pp. 10–24. Springer (2001)
9. Möller, B.: Towards pointer algebra. Sci. Comput. Program. 21(1), 57–90 (1993), http://dx.doi.org/10.1016/0167-6423(93)90008-D

A Proofs for Subsection 2.2 and Section 3

A.1 Proofs for Lemma 2.3

1. $\mathtt{T}_\mathtt{m} \cap \mathtt{S}_\mathtt{v} = \emptyset$ implies that $\mathtt{T}$ acts as the identity on the set $\mathtt{S}_\mathtt{v}$. Since $\mathtt{T}_\mathtt{m} \subseteq \mathtt{S}_\mathtt{m}$, we obtain that $\mathtt{T}$ also is the identity on the set $\mathtt{ran}(\mathtt{m}) - \mathtt{S}_\mathtt{m}$. Therefore we have $\mathtt{T} \cdot (\mathtt{S} \cdot \mathtt{m})(\mathtt{v}) = \mathtt{T} \cdot (\mathtt{S}(\mathtt{m}(\mathtt{v}))) = \mathtt{S}(\mathtt{m}(\mathtt{v}))$.
2. We have $\mathtt{x} \in \mathtt{S}_\mathtt{m} \Rightarrow \mathtt{T}(\mathtt{x}) = \mathtt{x} \wedge \mathtt{x} \in \mathtt{T}_\mathtt{m} \Rightarrow \mathtt{S}(\mathtt{x}) = \mathtt{x}$ because $\mathtt{T}_\mathtt{m} \cap \mathtt{S}_\mathtt{m} = \emptyset$. Therefore, for an arbitrary $\mathtt{x} = \mathtt{m}(\mathtt{v})$, we get

$$(\mathtt{T} \circ \mathtt{S})(\mathtt{x}) = \mathtt{T}(\mathtt{S}(\mathtt{x})) = \begin{cases} \mathtt{S}(\mathtt{x}) \text{ if } \mathtt{x} \in \mathtt{S}_\mathtt{m} \text{ since } \mathtt{T}_\mathtt{m} \cap \mathtt{S}_\mathtt{v} = \emptyset\,, \\ \mathtt{T}(\mathtt{x}) \text{ if } \mathtt{x} \in \mathtt{T}_\mathtt{m}\,, \\ \mathtt{x} \quad \text{if } \mathtt{x} \in \overline{\mathtt{S}_\mathtt{m}} \cap \overline{\mathtt{T}_\mathtt{m}}\,. \end{cases}$$

   On the other hand we get

$$(\mathtt{S} \circ \mathtt{T})(\mathtt{x}) = \mathtt{S}(\mathtt{T}(\mathtt{x})) = \begin{cases} \mathtt{T}(\mathtt{x}) \text{ if } \mathtt{x} \in \mathtt{T}_\mathtt{m} \text{ since } \mathtt{S}_\mathtt{m} \cap \mathtt{T}_\mathtt{v} = \emptyset\,, \\ \mathtt{S}(\mathtt{x}) \text{ if } \mathtt{x} \in \mathtt{S}_\mathtt{m}\,, \\ \mathtt{x} \quad \text{if } \mathtt{x} \in \overline{\mathtt{T}_\mathtt{m}} \cap \overline{\mathtt{S}_\mathtt{m}}\,. \end{cases}$$

A.2 Proofs for Lemma 3.4

(llp) $\Rightarrow$: $\mathtt{m} \leq \lceil \mathtt{m} : \mathtt{m} \leq \mathtt{p} : \mathtt{m}$ by (d1) and isotony of restriction ($\lceil \mathtt{m} \leq \mathtt{p}$).
$\Leftarrow$: $\mathtt{m} \leq \mathtt{p} : \mathtt{m} \Rightarrow \mathtt{m} = \mathtt{p} : \mathtt{m}$ by Lemma 3.2.2 and therefore $\lceil \mathtt{m} = \lceil(\mathtt{p} : \mathtt{m}) \leq \mathtt{p}$ by (d2).

(gla) $\Rightarrow$: $\mathtt{p} : \mathtt{m} \leq \mathtt{p} : (\lceil \mathtt{m} : \mathtt{m}) = (\mathtt{p} \cdot \lceil \mathtt{m}) : \mathtt{m}$ by (d1), isotony of restriction and (4). Since $\mathtt{p} \leq \neg\lceil \mathtt{m}$, $\mathtt{p} \cdot \lceil \mathtt{m} = 0$ holds and therefore $(\mathtt{p} \cdot \lceil \mathtt{m}) : \mathtt{m} = 0 : \mathtt{m} = 0$ by (3).
$\Leftarrow$: $\mathtt{m} = 1 : \mathtt{m} = (\mathtt{p} + \neg\mathtt{p}) : \mathtt{m} = \mathtt{p} : \mathtt{m} + \neg\mathtt{p} : \mathtt{m} = 0 + \neg\mathtt{p} : \mathtt{m} = \neg\mathtt{p} : \mathtt{m}$ by (5), Boolean algebra, (1) and the assumption ($\mathtt{p} : \mathtt{m} \leq 0$). Using (llp) we have $\lceil \mathtt{m} \leq \neg\mathtt{p}$ which is equivalent to $\mathtt{p} \leq \neg\lceil \mathtt{m}$ by shunting.

A.3 Proofs for Lemma 3.5

1. $\Rightarrow$: $\lceil \mathtt{m} = \lceil 0 = \lceil(0 : 0) \leq 0$ by (3) and (d2).
$\Leftarrow$: $\mathtt{m} \leq \lceil \mathtt{m} : \mathtt{m}$ by (d1) and therefore $\mathtt{m} \leq 0 : \mathtt{m} = 0$ by (3).
2. $\neg\lceil \mathtt{n} \leq \neg\lceil \mathtt{n} \Rightarrow \neg\lceil \mathtt{n} : \mathtt{n} \leq 0 \Rightarrow \neg\lceil \mathtt{n} : \mathtt{m} \leq 0 \Rightarrow \neg\lceil \mathtt{n} \leq \neg\lceil \mathtt{m} \Rightarrow \lceil \mathtt{m} \leq \lceil \mathtt{n}$ by (gla), isotony of restriction, (gla) again and shunting (2 times).
3. $\mathtt{p} \leq \neg\lceil(\mathtt{m} + \mathtt{n}) \Leftrightarrow \mathtt{p} : (\mathtt{m} + \mathtt{n}) \leq 0 \Leftrightarrow \mathtt{p} : \mathtt{m} + \mathtt{p} : \mathtt{n} \leq 0 \Leftrightarrow \mathtt{p} : \mathtt{m} \leq 0 \wedge \mathtt{p} : \mathtt{n} \leq 0 \Leftrightarrow \mathtt{p} \leq \neg\lceil \mathtt{m} \wedge \mathtt{p} \leq \neg\lceil \mathtt{n} \Leftrightarrow \mathtt{p} \leq \neg\lceil \mathtt{m} \cdot \neg\lceil \mathtt{n} \Leftrightarrow \mathtt{p} \leq \neg(\lceil \mathtt{m} + \lceil \mathtt{n})$. Using indirect equality we get $\neg\lceil(\mathtt{m} + \mathtt{n}) = \neg(\lceil \mathtt{m} + \lceil \mathtt{m}) \Leftrightarrow \lceil(\mathtt{m} + \mathtt{n}) = \lceil \mathtt{m} + \lceil \mathtt{m}$.
4. $\mathtt{m} \leq \lceil \mathtt{m} : \mathtt{m}$ holds by (d1) and $\lceil \mathtt{m} : \mathtt{m} \leq \mathtt{m}$ holds by Lemma 3.2.2.
5. $\leq$: $\lceil(\mathtt{p} : \mathtt{m}) : \mathtt{m} \leq \mathtt{p} : \mathtt{m}$ holds by (d2) and isotony of restriction.
$\geq$: $\mathtt{p} : \mathtt{m} \leq \lceil(\mathtt{p} : \mathtt{m}) : (\mathtt{p} : \mathtt{m}) = (\lceil(\mathtt{p} : \mathtt{m}) \cdot \mathtt{p}) : \mathtt{m}$ by (d1) and (4). Since $\lceil(\mathtt{p} : \mathtt{m}) \leq \mathtt{p}$ by (d2) it follows that $\lceil(\mathtt{p} : \mathtt{m}) \cdot \mathtt{p} = \lceil(\mathtt{p} : \mathtt{m})$. In sum we have $\mathtt{p} : \mathtt{m} \leq \lceil(\mathtt{p} : \mathtt{m}) : \mathtt{m}$.
6. First we have $\lceil \mathtt{m} = \lceil(1 : \mathtt{m}) = \lceil((\mathtt{p} + \neg\mathtt{p}) : \mathtt{m}) = \lceil(\mathtt{p} : \mathtt{m} + \neg\mathtt{p} : \mathtt{m}) = \lceil(\mathtt{p} : \mathtt{m}) + \lceil(\neg\mathtt{p} : \mathtt{m})$ by (5), Boolean algebra, and Part 4. By (d2) it holds that $\lceil(\mathtt{p} : \mathtt{m}) \leq \mathtt{p} \wedge \lceil(\neg\mathtt{p} : \mathtt{m}) \leq \neg\mathtt{p}$. And therefore by Boolean algebra: $\mathtt{p} \cdot \lceil(\mathtt{p} : \mathtt{m}) = \lceil(\mathtt{p} : \mathtt{m})$ and $\mathtt{p} \cdot \lceil(\neg\mathtt{p} : \mathtt{m}) = 0$. In sum we conclude: $\mathtt{p} \cdot \lceil \mathtt{m} = \mathtt{p} \cdot (\lceil(\mathtt{p} : \mathtt{m}) + \lceil(\neg\mathtt{p} : \mathtt{m})) = \mathtt{p} \cdot \lceil(\mathtt{p} : \mathtt{m}) + \mathtt{p} \cdot \lceil(\neg\mathtt{p} : \mathtt{m}) = \mathtt{p} \cdot \lceil(\mathtt{p} : \mathtt{m}) + 0 = \lceil(\mathtt{p} : \mathtt{m})$.

A.4 Proofs for Lemma 3.6

1. $\lceil(\neg\lceil \mathtt{n} : \mathtt{m}) = \lceil \mathtt{m} \cdot \neg\lceil \mathtt{n}$ by Lemma 3.5.6 and Boolean algebra.
2. $(\neg\lceil \mathtt{n} : 0 = 0)$ by (6).
3. $\neg\lceil 1 : (\mathtt{m} + \mathtt{n}) = \neg\lceil 1 : \mathtt{m} + \neg\lceil 1 : \mathtt{n}$ by (5).
4. $\neg\lceil(\mathtt{m} + \mathtt{n}) : 1 = \neg(\lceil \mathtt{m} + \lceil \mathtt{n}) : 1 = (\neg\lceil \mathtt{m} \cdot \neg\lceil \mathtt{n}) = \neg\lceil \mathtt{m} : (\neg\lceil \mathtt{n} : 1) = \neg\lceil \mathtt{m} : (\neg\lceil \mathtt{n} : 1)$ by Lemma 3.5.4, Boolean algebra, (4) and Lemma 3.2.3.
5. $\neg\lceil 0 : \mathtt{m} = \neg 0 : \mathtt{m} = 1 : \mathtt{m} = \mathtt{m}$ by (5) and $\neg\lceil 0 = \neg 0 = 1$ Lemma 3.5.1, Boolean algebra and (5).
6. $\neg\lceil \mathtt{m} \leq \neg\lceil \mathtt{m} \Rightarrow \neg\lceil \mathtt{m} : \mathtt{m} \leq 0$ by (gla).
7. $\neg\lceil \mathtt{n} : \mathtt{m} \leq \mathtt{m}$ by Lemma 3.2.2.
8. $\mathtt{m} \leq \mathtt{n} \Rightarrow \lceil \mathtt{m} \leq \lceil \mathtt{n} \Rightarrow \neg\lceil \mathtt{n} \leq \neg\lceil \mathtt{m} \Rightarrow \neg\lceil \mathtt{n} : \mathtt{m} \leq 0$ by Lemma 3.5.2, Boolean algebra and (gla).

A.5 Proofs for Lemma 3.7

1. $0 \rightarrow\!\triangleright\, n = 0 + \neg\lceil 0 : n = \neg 0 : n = 1 : n = n$ by Lemma 3.5.1 and (5).
2. $m \rightarrow\!\triangleright\, 0 = m + \neg\lceil m : 0 = m + 0 = m$ by (6).
3. $m \leq m + \neg\lceil m : n = m \rightarrow\!\triangleright\, n$ by isotony of $+$.
4. $\lceil m : (m \rightarrow\!\triangleright\, n) = \lceil m : (m + \neg\lceil m : n) = \lceil m : m + \lceil m : (\neg\lceil m : n) = m + (\lceil m \cdot \neg\lceil m) : n = m + 0 : n = m + 0 = m$ by (2), (4) and (3).
5. $\lceil (m \rightarrow\!\triangleright\, n) = \lceil (m + \neg\lceil m : n) = \lceil m + \lceil (\neg\lceil m : n) = \lceil m + \neg\lceil m \cdot \lceil n = \lceil m + \lceil n$, by Lemma 3.5.4, 3.5.6 and Boolean algebra.
6. First we conclude $\lceil m \geq \lceil n \Rightarrow \neg\lceil m \leq \neg\lceil n \Rightarrow \neg\lceil m : n \leq 0$ by (gla). Therefore $m \rightarrow\!\triangleright\, n = m + \neg\lceil m : n = m + 0 = m$.
7. $l \rightarrow\!\triangleright\, (m+n) = l + \neg\lceil l : (m+n) = l + \neg\lceil l : m + \neg\lceil l : n = l + \neg\lceil l : m + l + \neg\lceil l : n = l \rightarrow\!\triangleright\, m + l \rightarrow\!\triangleright\, n$ by (2) and idempotence.

On a Monadic Encoding of Continuous Behaviour (extended abstract)

Renato Neves

INESC TEC (HASLab) & University of Minho, Portugal
rjneves@inescporto.pt

Abstract. The original purpose of component–based development was to provide techniques to master complex software, through composition, reuse, and parametrisation. However, such systems are rapidly moving towards a level in which they become prevalently intertwined with (continuous) physical processes. A possible way to accommodate the latter in component calculi relies on a suitable encoding of continuous behaviour as (yet another) computational effect.
This paper reports such an encoding through a monad which, in the compositional development of hybrid systems, may play a role similar to the one played by the maybe, powerset, and distribution monads in the characterisation of partial, non deterministic and probabilistic components, respectively.

Keywords: Monads, components, hybrid systems, control theory

1 Introduction

Component-based software development is often explained through a visual metaphor: a palette of computational units, and a blank canvas in which they are dropped and interconnected by drawing wires abstracting different composition and synchronisation mechanisms. More and more, however, components are not limited to traditional information processing units, but encapsulate some form of interaction with physical processes. The resulting systems, referred to as *hybrid*, exhibit a complex dynamics in which *loci* of computation, coordination, and control of physical processes interact, become mutually constrained, and cooperate to achieve specific goals.

One way of looking at components, proposed in [1, 2], emphasises an *observational* semantics, through a signature of observers and methods, making them amenable to a *coalgebraic* [3] characterisation as (generalisations of) abstract Mealy machines. The resulting calculus is parametric on whatever behavioural model underlies a component specification. This captures, for example, partial, non deterministic or probabilistic evolution of a component's dynamics by encoding such behavioural effects as *strong monads* [4, 5].

This paper summarises a number of results developed in the context of the author's PhD project [6]. Namely, the introduction of a strong monad $\mathcal{H}$ [7] that

subsumes *continuous* behaviour and the study of the corresponding Kleisli category [8] as the mathematical space in which the underlying behaviour can be isolated and its effect over different forms of composition studied. This work may pave the way to the development of a *coalgebraic* calculus of *hybrid components*.

Related work. A few categorial models for hybrid systems have been proposed. For example, document [9] introduced an institution – in essence, a categorial rendering of logic – for hybrid systems and provided basic forms of composition. Around the same time, Jacobs [10] suggested a coalgebraic framework where hybrid systems are viewed as coalgebras equipped with a monoid action. Some years later Haghverdi *et. al* [11] provided a formalisation of hybrid systems using a conceptual framework that is closer to the coalgebraic perspective.

The monad reported in this paper captures the typical continuous behaviour of hybrid systems. Actually, there is a close relationship between the work reported here and Peter Höfner's algebra of hybrid systems [12]: the latter's main operator and its laws are embedded in the (sequential) composition of $Kl\mathcal{H}$, the Kleisli category for monad $\mathcal{H}$.

Since our approach, differently from Höfner's calculus, is structured around a monad that encodes continuous evolution, a number of canonical constructions come for free. Moreover, the integration with other behavioural effects, such as non determinism or probabilistic evolution, becomes more systematic.

Roadmap. After a brief detour on preliminaries and notation in Section 2, monad $\mathcal{H}$ is described in Section 3. Section 4 gives some details about the corresponding Kleisli category $Kl\,\mathcal{H}$, characterising composition and some (co)limits. Finally, conclusions and possible future research directions are discussed in Section 5. In this paper many calculations adopt a pointfree style in the spirit of the Bird-Meertens formalism [13].

2 Preliminaries

2.1 The category of topological spaces

The typical *continuous* behaviour of hybrid systems suggests the category $\mathbb{T}op$ of topological spaces and continuous functions as a suitable working environment for developing the aforementioned results. In the sequel, if the context is clear, a topological space will be denoted just by its underlying set. Also, assume that spaces $X \times Y$, $X + Y$ correspond to the canonical product and coproduct of X, Y, respectively, and that whenever Y is *core-compact*, space X^Y comes with the *exponential* topology [14]. In this context, given a continuous function $f : X \times Y \to Z$ where Y is core–compact, we denote its curried version by

$\lambda f : X \to Z^Y$. Moreover, we will use the following isomorphisms in $\mathbb{T}op$:

$$\begin{aligned}
&\alpha_l : (X \times Y) \times Z \cong X \times (Y \times Z)\\
&sw : X \times Y \cong Y \times X\\
&i : (X \times Y)^{\mathbb{R}_0} \cong X^{\mathbb{R}_0} \times Y^{\mathbb{R}_0}
\end{aligned}$$

2.2 Notation

Arrows $X \to 1$ to the final object in $\mathbb{T}op$ will be denoted by !, and a function constantly yielding a value x by $\underline{x}$. Given two functions $f, g : X \to Y$, and a predicate p, conditional expression $f \lhd p \rhd g : X \to Y$ is defined by

$$(f \lhd p \rhd g)\, x = (f\, x \lhd p\, x \rhd g\, x) = \begin{cases} f\, x & p\, x \\ g\, x & \text{otherwise} \end{cases}$$

The continuous functions *minimum* $\curlywedge : \mathbb{R} \times (\mathbb{R}+1) \to \mathbb{R}$ and *truncated subtraction* $\ominus : \mathbb{R} \times (\mathbb{R}+1) \to \mathbb{R}$ play a key role in the sequel. They are defined as follows

$$\begin{aligned}
&r \curlywedge (i_1\, s) = (\pi_1 \lhd (\leq) \rhd \pi_2)\, (r, s) && r \ominus (i_1\, s) = ((-) \lhd (>) \rhd \underline{0})\, (r, s)\\
&r \curlywedge (i_2\, \star) = r && r \ominus (i_2\, \star) = 0
\end{aligned}$$

where $\leq, >$ are the usual ordering relations over the reals, and 1 introduces *infinity*. Set $\mathbb{R}_0$ denotes the non–negative real numbers. Then, we have $(\curlywedge_d)\, r = r \curlywedge d$ and $(\ominus_d)\, r = r \ominus d$. Finally, for any category $\mathbb{C}$, $|\mathbb{C}|$ denotes the corresponding class of objects.

3 A Monad for Continuity

Formally, we see *continuous systems* as arrows of the type

$$I \to O^{\mathbb{R}_0} \times D$$

where $D = \mathbb{R}_0 + 1$ and I, O are the input and output spaces, respectively. The intuition is that outputs of such systems are *continuous evolutions* (also known as *trajectories*) with a specific (possibly infinite) duration $d \in D$.

Definition 1 *$\mathcal{H} : \mathbb{T}op \to \mathbb{T}op$ is a functor such that, for any objects $X, Y \in |\mathbb{T}op|$ and any continuous function $g : X \to Y$,*

$$\begin{aligned}
&\mathcal{H}X = \{\, (f, d) \in X^{\mathbb{R}_0} \times D \mid f \cdot \curlywedge_d = f \,\}\\
&\mathcal{H}g = g \cdot \times id
\end{aligned}$$

where $(g \cdot)\, h = g \cdot h$. Condition $f \cdot \curlywedge_d = f$ tells that function f must become constant after reaching its duration; more formally, for any $r \in \mathbb{R}_0$ such that $r > d$, $f\, r = f\, d$. Hence, continuous systems become arrows of the type

$$I \to \mathcal{H}O$$

also denoted as $I \nrightarrow O$.

The crucial step now is to equip $\mathcal{H}$ with a monad structure, *i.e.* with natural transformations

$$\eta : Id \dot{\rightarrow} \mathcal{H}, \;\; \mu : \mathcal{H}\mathcal{H} \dot{\rightarrow} \mathcal{H}.$$

First,

Definition 2 *Given any $X \in |\mathbb{T}op|$, define $\eta_X : X \rightarrow \mathcal{H}X$ such that*

$$\eta_X \; x = (\underline{x}, i_1 \; 0)$$

in pointfree notation $\eta_X = \langle \lambda \pi_1, i_1 \cdot \underline{0} \rangle$.

The definition of μ is more demanding.

Definition 3 *Define the continuous functions $g : \mathcal{H}\mathcal{H}X \times \mathbb{R}_0 \rightarrow X^{\mathbb{R}_0}$, $h : \mathcal{H}\mathcal{H}X \times \mathbb{R}_0 \rightarrow \mathbb{R}_0$ such that*

$$\begin{aligned} g((f,d),r) &= (\pi_1 \cdot f) \; (r \mathbin{\curlywedge} d), \\ h((f,d),r) &= r \ominus d \end{aligned}$$

Next, we have $fl_1 : \mathcal{H}\mathcal{H}X \rightarrow X^{\mathbb{R}_0}$ where $fl_1 = \lambda(ev \cdot \langle g, h \rangle)$. In pointwise notation, fl_1 is defined as

$$fl_1(f,d) = ev \cdot \langle \pi_1 \cdot f \cdot \curlywedge_d, \ominus_d \rangle$$

Then, define function $fl_2 : \mathcal{H}^2 X \rightarrow D$ such that

$$fl_2 \; (f,d) = ((\pi_2 \cdot f) \; d \; \lhd (d \notin 1) \; \rhd \; i_2 \star) + d$$

Finally, we define for any $X \in |\mathbb{T}op|$, $\mu_X = \langle fl_1, fl_2 \rangle$.

Intuitively, operation μ_X 'concatenates' functions: given a pair $(f,d) \in \mathcal{H}\mathcal{H}X$, μ_X concatenates function $(\pi_1 \cdot f) \; \text{-} \; 0 : [0,d] \rightarrow X$ with $(\pi_1 \cdot f) \; d \; \text{-} \; : [0,d'] \rightarrow X$, and sums the corresponding durations.

Theorem 1 *The triple $\langle \mathcal{H}, \eta, \mu \rangle$ forms a monad.*

Proof. In document [7].

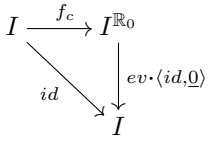
4 The Category of Continuous Behaviours

4.1 Kleisli Composition

The Kleisli category for $\mathcal{H}$ ($Kl\mathcal{H}$) provides an interesting setting to study the requirements placed by continuity over different forms of composition; actually, the envisaged component calculus for hybrid systems is essentially its calculus. This motivates the study of $Kl\,\mathcal{H}$, summmarised in the current section.

The definition of Kleisli composition in $Kl\,\mathcal{H}$ suggests a relevant distinction between continuous systems.

Definition 4 *A continuous system $c : I \to \mathcal{H}I$ is* passive *if the following diagram commutes*



where $f_c = \pi_1 \cdot c$. It is active *otherwise.*

Intuitively, the diagram tells that any evolution triggered by c 'starts' at the point given as input. To see why such a distinction is relevant, let us consider two continuous systems $c_1 : I \to \mathcal{H}K$, $c_2 : K \to \mathcal{H}O$. Through Kleisli composition we obtain component $c_2 \bullet c_1 : I \to \mathcal{H}O$ whose behaviour is computed as follows:

$$
\begin{array}{ll}
 & \pi_1 \cdot (c_2 \bullet c_1)\, x \\
= & \quad \{ \text{ Kleisli composition } \} \\
 & \pi_1 \cdot \mu \cdot \mathcal{H}c_2 \cdot c_1\, x \\
= & \quad \{ \text{ Cancellation } \times \} \\
 & fl_1 \cdot \mathcal{H}c_2 \cdot c_1\, x \\
= & \quad \{ \text{ Definition of } \mathcal{H} \text{ (taking } d = (\pi_2 \cdot c_1)\, x \text{) } \} \\
 & fl_1\, (c_2 \cdot (f_{c_1}\, x), d) \\
= & \quad \{ \text{ Application } \} \\
 & ev \cdot \langle \pi_1 \cdot c_2 \cdot (f_{c_1}\, x) \cdot \curlywedge_d, \ominus_d \rangle \\
= & \quad \{ \text{ Notation } \} \\
 & ev \cdot \langle f_{c_2} \cdot (f_{c_1}\, x) \cdot \curlywedge_d, \ominus_d \rangle
\end{array}
$$

Going pointwise,

$$
\begin{array}{ll}
 & ev \cdot \langle f_{c_2} \cdot (f_{c_1}\, x) \cdot \curlywedge_d, \ominus_d \rangle\, t \\
= & \quad \{ \text{ Application } \} \\
 & f_{c_2}\, \big(\, f_{c_1}\, x\, (t \curlywedge d)\, \big)\, (t \ominus d) \\
= & \quad \{ \text{ Notation } \} \\
 & f_{c_2}\, \big(\, f_{c_1}\, x\, t\, \big)\, 0 \lhd (\, t \leq d\,) \rhd f_{c_2}\, \big(\, f_{c_1}\, x\, d\, \big)\, (t - d) \\
= & \quad \{ \text{ If } c_2 \text{ is passive } \} \\
 & f_{c_1}\, x\, t \lhd (\, t \leq d\,) \rhd f_{c_2}\, \big(\, f_{c_1}\, x\, d\, \big)\, (t - d)
\end{array}
$$

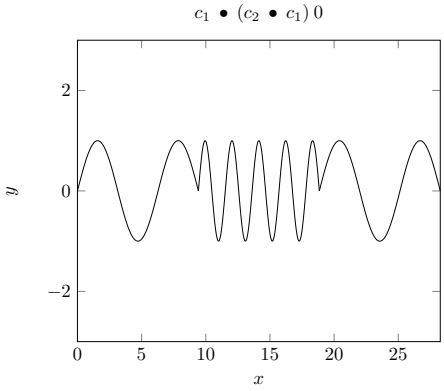
Assuming that c_2 is passive, the last expression tells that given an input $i \in I$ the resulting evolution corresponds to the evolution of the first component

$f_{c_1}\ i$ ensued by the evolution of the second, which receives as input the 'last' point of evolution $f_{c_1}\ i$. Therefore, when c_2 is passive Kleisli composition may be alternatively called *sequential* composition or concatenation. On the other hand if c_2 is active, Kleisli composition tells that c_2 can alter the evolution of c_1 and then proceed with its own evolution. This is illustrated in the following examples.

Example 1. *Given two signal generators $c_1, c_2 : \mathbb{R} \to \mathcal{H}\mathbb{R}$ defined as*

$$\begin{aligned} c_1\ r &= (r + sin\ \text{-}\ , 3\pi), \\ c_2\ r &= (r + sin\ (3 \times\ \text{-}\ ), 3\pi) \end{aligned}$$

the signal given by $c_1 \bullet (c_2 \bullet c_1)\ 0$ yields the plot below

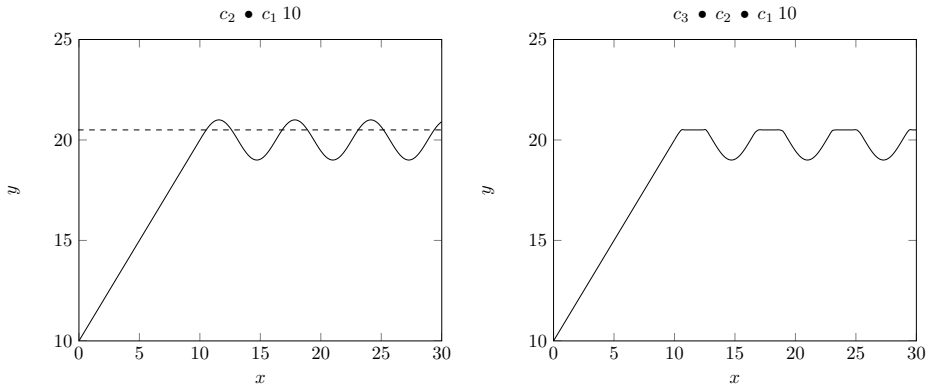


This type of signal is common in the domain of *frequency modulation*, where the varying frequency is used to encode information for electromagnetic transmission.

Example 2. *Suppose the temperature of a room is to be regulated according to the following discipline: starting at $10\,°C$, seek to reach and maintain $20\,°C$, but in no case surpass $20.5\,°C$. To realise such a system, three components have to work together: c_1 to raise the temperature to $20\,°C$, component c_2 to maintain a given temperature, and component c_3 to ensure the temperature never goes over $20.5\,°C$. Formally,*

$$\begin{aligned} c_1\ x &= (\ (x +\ \text{-}\ ),\ 20 \ominus x\ ), \\ c_2\ x &= (\ x + (\sin\ \text{-}\ ),\ \infty\ ), \\ c_3\ x &= (\ \underline{x}\ \lhd\ (x \leq 20.5)\ \rhd\ \underline{20.5}\ , 0\ ) \end{aligned}$$

One may then compose c_2, c_1 into $c_2 \bullet c_1$, which results in a component able to read the current temperature, raise it to $20\,°C$, and then keep it stable, as shown by the plot below on the left. If, however, temperatures over $20.5\,°C$ occur, composition $c_3 \bullet c_2 \bullet c_1$ puts the system back into the right track as the plot below on the right illustrates.



On a different note, for any $X \in |\mathbb{T}op|$, arrow η_X is a *trivial* system in the sense that its evolutions always have duration zero and the only point in the trajectories is the input given. For this reason we will refer to η_X by $copy_X$, and often omit the subscript. Setting up $Kl\ \mathcal{H}$ yields the following laws

$$copy \bullet c_1 = c_1 \tag{1}$$
$$c_1 \bullet copy = c_1 \tag{2}$$
$$(c_3 \bullet c_2) \bullet c_1 = c_3 \bullet (c_2 \bullet c_1) \tag{3}$$

for any arrows c_1, c_2, c_3 in $Kl\ \mathcal{H}$.

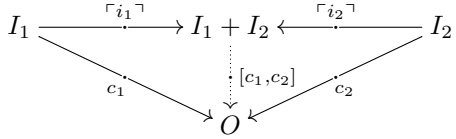
4.2 (Co)limits and Tensorial Strength

(Co)limits are a main tool to build 'new' arrows from 'old' ones, which in the case of $Kl\ \mathcal{H}$ translates to new forms of (continuous) component composition. One important colimit is the *coproduct*, which provides the *choice* operator:

Definition 5 *Given two components $c_1 : I_1 \to \mathcal{H}O$, $c_2 : I_2 \to \mathcal{H}O$ component*

$$[c_1, c_2] : I_1 + I_2 \to \mathcal{H}O$$

behaves as c_1 if input I_1 is chosen, *and as c_2 otherwise. Diagrammatically,*

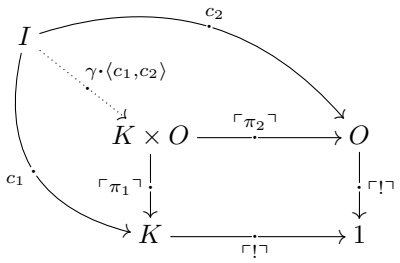


where, for any continuous function $f : X \to Y$, symbol $\ulcorner f \urcorner$ denotes $copy \cdot f$.

Since the choice operator comes from a colimit, a number of laws are given; one example is the following equation

$$c_3 \bullet [c_1, c_2] = [c_3 \bullet c_1, c_3 \bullet c_2] \tag{4}$$

An important limit of $Kl\ \mathcal{H}$ is the *pullback* below, which brings parallelism up front.



for $\gamma((f_1, d), (f_2, d)) = (\langle f_1, f_2\rangle, d)$.

Intuitively, the diagrams states that whenever two components c_1, c_2 are compatible – in the sense that for any input the duration of their evolutions coincide (commutativity of the outer square) – we can define component $\gamma \cdot \langle c_1, c_2\rangle$ whose output corresponds to the (paired) evolutions of c_1 and c_2.

Note that functions $\ulcorner \pi_1 \urcorner, \ulcorner \pi_2 \urcorner$ introduce trajectory elimination, due to their ability to remove one side of the paired evolution. Note also that $\ulcorner \triangle \urcorner : X \to \mathcal{H}(X \times X)$ duplicates trajectories, for $\triangle : X \to (X \times X)$ the diagonal function, and $\ulcorner sw \urcorner$ swaps evolutions.

Definition 6 *Given two compatible components $c_1 : I \to \mathcal{H}O_1$, $c_2 : I \to \mathcal{H}O_2$ component*

$$\langle\langle c_1, c_2\rangle\rangle = \gamma \cdot \langle c_1, c_2\rangle : I \to \mathcal{H}(O_1 \times O_2)$$

is the parallel execution of c_1, c_2.

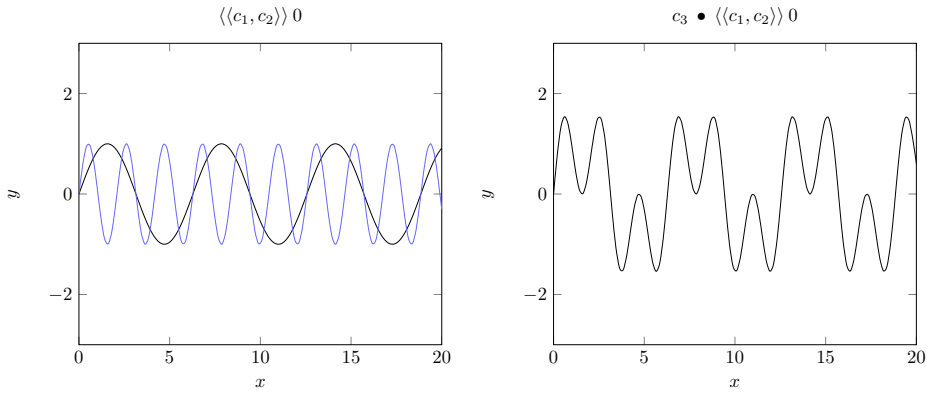
Since parallelism comes from a limit, we have again a number of laws for free; for instance

$$\langle\langle c_1, c_2\rangle\rangle \bullet d = \langle\langle c_1 \bullet d, c_2 \bullet d\rangle\rangle \tag{5}$$

Example 3. *Consider two signal generators, c_1, c_2 such that*

$$\begin{aligned} c_1\ x &= (\ x + (\sin\ \text{-}\ ), 20\ ), \\ c_2\ x &= (\ x + \sin\,(3 \times\ \text{-}\ ), 20\ ) \end{aligned}$$

For input 0, their parallel evolution $\langle\langle c_1, c_2\rangle\rangle$ is illustrated in the plot below on the left. Moreover, we can combine signals. For example, to add incoming signals, take the active component c_3, formally defined as $c_3(x, y) = (\underline{x + y}, 0)$. For input 0, the system $c_3 \bullet \langle\langle c_1, c_2\rangle\rangle$ yields the plot shown below, on the right.



We close this section introducing a tensorial strength for monad $\mathcal{H}$ — which turns out to be an essential mechanism for the generation of a calculus for hybrid components.

Definition 7 *Tensorial strength for monad $\mathcal{H}$ is a natural transformation*

$$\tau : \mathcal{H}X \times Y \stackrel{\cdot}{\rightarrow} \mathcal{H}(X \times Y)$$

defined as $\tau = \langle f_1, f_2\rangle$ where $f_1 : \mathcal{H}X \times Y \rightarrow (X \times Y)^{\mathbb{R}_0}$, $f_1((f, d), y) = \langle f, \underline{y}\rangle$, and $f_2 : \mathcal{H}X \times Y \rightarrow D$, $f_2((f, d), y) = d$.

Theorem 2 *$\langle \mathcal{H}, \eta, \mu\rangle$ is a strong monad.*

Proof. In document [7].

5 Conclusions and future work

Software systems are becoming prevalently intertwined with (continuous) physical processes. This renders their rigorous design (and analysis) a difficult challenge that calls for a wide, uniform framework where 'Continuous' Mathematics and Computer Science must work together. As a first step towards a calculus of hybrid components in the spirit of [2], this paper showed how continuous behaviour can be encoded in the form of a strong topological monad, and briefly explored the corresponding Kleisli category.

Our current research investigates how hybrid behaviour can be rendered by arrows typed as $\langle c, p\rangle : S \times I \rightarrow S \times \mathcal{H}O$, where $c : S \times I \rightarrow S$ is a discrete arrow (S comes equipped with the discrete topology) and $p : S \times I \rightarrow \mathcal{H}O$ is a continuous system. This paves the way to extending the component calculus in [2] to hybrid systems.

Acknowledgements. This work is funded by ERDF - European Regional Development Fund, through the COMPETE Programme, and by National Funds through FCT within project FCOMP-01-0124-FEDER-028923. The author is also sponsored by FCT grant SFRH/BD/52234/2013.

References

1. L. S. Barbosa, Components as coalgebras, Ph.D. thesis, DI, Minho University (2001).
2. L. S. Barbosa, Towards a calculus of state-based software components, Journal of Universal Computer Science 9 (2003) 891–909.
3. J. Rutten, Universal coalgebra: a theory of systems, Theoretical Computer Science 249 (1) (2000) 3 – 80, modern Algebra.
4. A. Kock, Strong functors and monoidal monads, Archiv der Mathematik 23 (1) (1972) 113–120.
5. E. Moggi, Notions of computation and monads, Information and computation 93 (1) (1991) 55–92.
6. R. Neves, Logics and calculi for hybrid components, Ph.D. thesis, DI, Universidade do Minho (2017).
7. R. Neves, L. S. Barbosa, D. Hofmann, M. A. Martins, Continuity as a computational effect, CoRR abs/1507.03219.
   URL http://arxiv.org/abs/1507.03219
8. R. Neves, L. S. Barbosa, M. A. Martins, A kleisli category for hybrid components, in: Formal Aspects of Component Software (FACS) 2015, submitted.
9. H. Lourenço, A. Sernadas, An institution of hybrid systems, in: D. Bert, C. Choppy, P. Mosses (Eds.), Recent Trends in Algebraic Development Techniques, Vol. 1827 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2000, pp. 219–236. doi:10.1007/978-3-540-44616-3_13.
   URL http://dx.doi.org/10.1007/978-3-540-44616-3_13
10. B. Jacobs, Object-oriented hybrid systems of coalgebras plus monoid actions, Theoretical Computer Science 239 (1) (2000) 41 – 95. doi:http://dx.doi.org/10.1016/S0304-3975(99)00213-3.
    URL http://www.sciencedirect.com/science/article/pii/S0304397599002133
11. E. Haghverdi, P. Tabuada, G. J. Pappas, Bisimulation relations for dynamical, control, and hybrid systems, Theor. Comput. Sci. 342 (2-3) (2005) 229–261. doi:10.1016/j.tcs.2005.03.045.
    URL http://dx.doi.org/10.1016/j.tcs.2005.03.045
12. P. Höfner, Algebraic calculi for hybrid systems, Ph.D. thesis, University of Augsburg (2009).
13. R. Bird, O. de Moor, The Algebra of Programming, Prentice-Hall, 1996.
    URL http://www.cs.ox.ac.uk/publications/books/algebra/
14. M. Escardó, R. Heckmann, Topologies on spaces of continuous functions, in: Topology Proceedings, Vol. 26, 2001, pp. 545–564.

Relational Approximation of Maximum Independent Sets (Extended Abstract)

Insa Stucke

Institut für Informatik, Christian-Albrechts-Universität zu Kiel, Germany
ist@informatik.uni-kiel.de

1 Introduction

Previous work has shown that relation algebra (as introduced in [14] and further developed in [9,12,13,15], for example) is well suited for computational problems on many discrete structures. In particular, adjacency or incidence relations can be used to model graphs and special relations like vectors and points to represent subsets of vertices or edges, as shown in [12].

We develop and formally verify a relational program for approximating maximum independent sets in undirected and loop-free graphs. Formal program verification means to show with mathematical rigor that the program is correct with respect to a formal problem specification, in our case we comply with the assertion-based Floyd-Hoare-approach.

At the end, the most interesting task is to prove the desired approximation bound. Of course, therefore we need knowledge about cardinalities of relations. In [7] a characterisation of a cardinality operation is developed and further consequences of it were proved. Based on this cardinality operation we not only prove the approximation bound but also facts about the cardinalities of vectors and points in a calculational, algebraic manner only.

2 Relation-Algebraic Preliminaries

In this section we recall the fundamentals of relation algebra based on the heterogeneous approach of [12,13]. Set-theoretic relations form the standard model of relation algebras. We assume the reader to be familar with the basic operations on set-theoretic relations, viz. R^T (transposition), $\overline{R}$ (complementation), $R \cup S$ (union), $R \cap S$ (intersection), $R;S$ (composition), the predicates $R \subseteq S$ (inclusion) and $R = S$ (equality), and the special relations O (empty relation), L (universal relation) and I (identity relation). The Boolean operations, the inclusion and the constants O and L form Boolean lattices.

The theoretical framework for this and many other results concerning relations is that of a (heterogeneous) *relation algebra* in the sense of [12,13], with typed relations as elements. Thus, we write $R : X \leftrightarrow Y$ to express that X is the source, Y is the target. The type of R is denoted by $X \leftrightarrow Y$. As constants and operations of a relation algebra we have those of set-theoretic relations, where we frequently overload the symbols O, L and I, i.e., avoid the binding of types

to them. If necessary we use indices as e.g., L_{XY} for L with type $X \leftrightarrow Y$. The axioms of a relation algebra are

(1) the axioms of a Boolean lattice for all relations of the same type under the Boolean operations, the inclusion, empty relation and universal relation,
(2) the associativity of composition and that identity relations are neutral elements with respect to composition,
(3) that $Q;R \subseteq S$, $Q^\mathsf{T};\overline{S} \subseteq \overline{R}$ and $\overline{S};R^\mathsf{T} \subseteq \overline{Q}$ are equivalent, for all relations Q, R and S (with appropriate types),
(4) that $R \neq \mathsf{O}$ implies $\mathsf{L};R;\mathsf{L} = \mathsf{L}$, for all relations R and all universal relations (with appropriate types).

In [12] the laws of (3) are called the *Schröder rules* and (4) is called the *Tarski rule*. In the relation-algebraic proofs of this paper we will mention only applications of (3), (4) and 'non-obvious' consequences of the axioms. Furthermore, we will assume that complementation and transposition bind stronger than composition, composition binds stronger than union and intersection and that all expressions and formulae are well-typed.

Furthermore, we call a relation R *irreflexive* if $R \subseteq \bar{\mathsf{I}}$, *symmetric* if $R = R^\mathsf{T}$ and a mapping if R is *univalent*, i.e., $R^\mathsf{T};R \subseteq \mathsf{I}$, and *total*, i.e., $R;\mathsf{L} = \mathsf{L}$ (for more details, see e.g., [12,13]). A *vector* is a relation v with $v = v;\mathsf{L}$. For $v : X \leftrightarrow Y$ the condition $v = v;\mathsf{L}$ means that v can be written in the form $v = Z \times Y$ with a subset Z of X. Then we say that v *models the subset* Z of X. Since for this purpose the target of a vector is irrelevant, we use the specific singleton set $\mathbf{1}$ as target. Moreover, a *point* p is a vector with $p;p^\mathsf{T} \subseteq \mathsf{I}$ and $\mathsf{L};p = \mathsf{L}$. In the set-theoretic case and if the point $p : X \leftrightarrow Y$ is of the specific form $p = P \times Y$ with $P \subseteq X$ these three conditions mean that p contains exactly one element.

In the remainder we use the following *point axiom* of [4] which holds for set-theoretic relations, where $\mathcal{P}_v := \{p \mid p \subseteq v \wedge p \text{ is point}\}$ for all vectors v.

Axiom 2.1 *For all sets X we have $\mathsf{L}_{X\mathbf{1}} = \bigcup_{p \in \mathcal{P}_{\mathsf{L}_{X\mathbf{1}}}} p$.*

Additionally we have the following lemma which states that this property can be generalised for arbitrary vectors (see [4]).

Lemma 2.1 *If $v : X \leftrightarrow \mathbf{1}$ is a vector, then $v = \bigcup_{p \in \mathcal{P}_v} p$.* □

3 Cardinality of Relations

In [7] Kawahara discusses the cardinality of set-theoretic relations. The main result is a characterisation of the cardinalities of relations. Considering the properties of this characterisation as axiomatic specification of the cardinality operation $|\cdot|$ this leads to the following definition:

Definition 3.1 *For all relations R we denote its cardinality by $|R|$. The following axioms specify the meaning of the cardinality operation, where Q, R and S are arbitrary relations with appropriate types:*

(C1) If R is finite, then $|R| \in \mathbb{N}$ and $|R| = 0$ iff $R = \mathsf{O}$.
(C2) $|R| = |R^\mathsf{T}|$.
(C3) If R and S are finite, then $|R \cup S| = |R| + |S| - |R \cap S|$.
(C4) If Q is univalent, then $|R \cap Q^\mathsf{T};S| \leq |Q;R \cap S|$ *and* $|Q \cap S;R^\mathsf{T}| \leq |Q;R \cap S|$.
(C5) $|\mathsf{l}_{\mathbf{11}}| = 1$.

In (C1) and (C3) the relations in question are assumed to be finite so that the cardinality $|R|$ can be regarded as a natural number, in (C2) and (C4) the notation $|R| = |S|$ (respectively $|R| \leq |S|$) is equivalent to the fact that there exists a bijection between R and S (respectively an injection from R to S) and (C5) says that the identity relation on the singleton set $\mathbf{1}$ consists of precisely one pair. Throughout this paper we assume in case of an expression $|R|$ the sets of R's type to be finite and thereby $|R| \in \mathbb{N}$.

Based on the above axioms in [7] a lot of laws for the cardinality operation are derived in a purely calculational manner. For example, from the axioms (C1) and (C3) we get the monotonocity of the cardinality operation, i.e., that $R \subseteq S$ implies $|R| \leq |S|$. Another fact we use in the remainder is following (see [7]):

Lemma 3.1 *If $R : X \leftrightarrow Y$ is univalent and $S : Y \leftrightarrow Z$ is a mapping, then* $|R;S| = |R|$. □

Next, we consider the cardinality of points of type $X \leftrightarrow \mathbf{1}$ and vectors by using only the mentioned cardinality axioms and the presented consequences of them. The next lemma states that a point in deed contains exactly one element.

Lemma 3.2 *If $p : X \leftrightarrow \mathbf{1}$ is a point, then* $|p| = 1$.

Proof. Using cardinality axioms (C2) and (C5) and Lemma 3.1 ($\mathsf{l}_{\mathbf{11}}$ is univalent and $p^\mathsf{T} : \mathbf{1} \leftrightarrow X$ is a mapping), we have the folliwng calculation:

$$|p| = |p^\mathsf{T}| = |\mathsf{l}_{\mathbf{11}};p^\mathsf{T}| = |\mathsf{l}_{\mathbf{11}}| = 1. \qquad \square$$

This lemma allows to show that the cardinality of a vector with target $\mathbf{1}$ equals the cardinality of the set of all points it contains:

Lemma 3.3 *For all $v : X \leftrightarrow \mathbf{1}$ we have* $|v| = |\mathcal{P}_v|$.

Proof. Because of Lemma 2.1, cardinality axioms (C3) and (C1) (the points of $\mathcal{P}_v$ are pair-wise disjoint) and Lemma 3.2 we obtain the claim by

$$|v| = |\bigcup_{p \in \mathcal{P}_v} p| = \sum_{p \in \mathcal{P}_v} |p| = |\mathcal{P}_v|. \qquad \square$$

4 Approximation of Maximum Independent Sets

In this section we use the notions and results of the previous sections to formally verify a relational version of the approximation algorithm of Wei for maximum independent sets (see [16]).

We assume an undirected and loop-free graph $g = (X, E)$ to be given, where the set X of vertices is non-empty and finite. We model g by an *adjacency relation* $R : X \leftrightarrow X$, that is defined by $(x, y) \in R$ iff $\{x, y\} \in E$, for all $x, y \in X$. Due to this definition R is irreflexive and symmetric. The relation R is taken as input for the relational program we want to show as correct. Since the approximation bound depends on the degrees of the vertices, we additionally assume that the maximum degree of g is $k \in \mathbb{N}$. This causes to the conjunction of the following three formulae as pre-condition $\mathit{Pre}(R, k)$:

$$R \subseteq \overline{\mathsf{I}} \qquad R = R^{\mathsf{T}} \qquad k = \max\{|R;p| \mid p \in \mathcal{P}_{\mathsf{L}_{X\mathbf{1}}}\}$$

An *independent set* (or *stable set*) of g is a set of vertices S such that $\{x, y\} \notin E$, for all $x, y \in S$. It can be easily derived that a vector $s : X \leftrightarrow \mathbf{1}$ models an independent set with respect to the adjacency relation R iff $R;s \subseteq \overline{s}$. We want to show that our program has approximation bound $k + 1$. So, the post-condition $\mathit{Post}(R, k, s)$ is the conjunction of the following two formulae:

$$R;s \subseteq \overline{s} \qquad \forall\, t : X \leftrightarrow \mathbf{1} \bullet R;t \subseteq \overline{t} \Rightarrow |t| \leq |s|(k + 1)$$

In the remainder of this section we show that with respect to these specifications the following relational program is totally correct:

$$\begin{array}{l} s, v := \mathsf{O}, \mathsf{O}_{X\mathbf{1}}; \\ \textbf{while } v \neq \mathsf{L} \textbf{ do} \\ \quad \textbf{let } p \ = \mathit{point}(\overline{v}); \\ \quad s, v := s \cup p, v \cup p \cup R;p \textbf{ od} \end{array} \tag{W}$$

We use the operation *point* that selects deterministically a point such that $\mathit{point}(v) \subseteq v$ for all non-empty vectors v. The typing rules of the relational operations in combination with the initialisation of v by $\mathsf{O}_{X\mathbf{1}}$ leads to the typing $s, v, p : X \leftrightarrow \mathbf{1}$ and also $X \leftrightarrow \mathbf{1}$ as type of the constant L of the guard of the loop. The vector v is used to collect the vertices contained in the present indepentend set, that is modeled by the vector s, and also their neighbours.

In the remainder the conjunction of the following two formulae is used as loop invariant $\mathit{Inv}(R, k, s, v)$:

$$(1)\ (R \cap v;v^{\mathsf{T}});s \subseteq \overline{s} \qquad (2)\ R;s \cup s = v$$

Here formula (1) is a generalisation of the formula $R;s \subseteq \overline{s}$ of the post-condition $\mathit{Post}(R, k, s)$ and formula (2) is simply an auxiliary formula saying that v models the union of the set modeled by s with its neighbours.

We now prove the four proof obligations of assertion-based verification with respect to the above speficied pre- and post-condition. We start with the establishment of the loop invariant by the initialisation of s and v.

Lemma 4.1 *If $R : X \leftrightarrow X$ and $k \in \mathbb{N}$ with $\mathit{Pre}(R, k)$, then $\mathit{Inv}(R, k, \mathsf{O}, \mathsf{O})$.* □

We omit the trivial proof. With the next lemma we prove the maintainence of the loop invariant.

Lemma 4.2 *Given* $R : X \leftrightarrow X$, $s, v : X \leftrightarrow \mathbf{1}$ *and* $k \in \mathbb{N}$ *such that* $\mathsf{Inv}(R, k, s, v)$ *and* $v \neq \mathsf{L}$, *we have* $\mathsf{Inv}(R, k, s \cup p, v \cup p \cup R;p)$, *for all* $p \in \mathcal{P}_{\overline{v}}$.

Proof. First, we verify that the first formula of the loop invariant holds for the new values of s and v, i.e., that $(R \cap (v \cup p \cup R;p);(v \cup p \cup R;p)^{\mathsf{T}});(s \cup p) \subseteq \overline{s \cup p}$. It is easy to see that showing the following four inclusions is sufficent:

$$(R \cap (v \cup p \cup R;p);(v \cup p \cup R;p)^{\mathsf{T}});s \subseteq \overline{s}$$
$$(R \cap (v \cup p \cup R;p);(v \cup p \cup R;p)^{\mathsf{T}});s \subseteq \overline{p}$$
$$(R \cap (v \cup p \cup R;p);(v \cup p \cup R;p)^{\mathsf{T}});p \subseteq \overline{s}$$
$$(R \cap (v \cup p \cup R;p);(v \cup p \cup R;p)^{\mathsf{T}});p \subseteq \overline{p}.$$

Because of (2) we have $R;s \subseteq v$ and $s \subseteq v$ and, moreover, because of $p \subseteq \overline{v}$ we have $R;s \subseteq \overline{p}$ and $s \subseteq \overline{p}$. Furthermore, we get

$$R;s \subseteq \overline{p} \iff R^{\mathsf{T}};p \subseteq \overline{s} \iff R;p \subseteq \overline{s}$$

using one of the Schröder rules in the first and the second formula of the pre-condition in the second step. With these auxiliary facts the second and third of the above inclusions follow immediately. Since the point p is injective and R is irreflexive due to the first formula of the pre-condition, we obtain $R;p \subseteq \overline{p}$, such that the last of the above inclusions holds. Verifying the first inclusion is more comprehensive since the following three inclusions have to be proved:

$$(R \cap (v \cup p \cup R;p);v^{\mathsf{T}});s \subseteq \overline{s}$$
$$(R \cap (v \cup p \cup R;p);p^{\mathsf{T}});s \subseteq \overline{s}$$
$$(R \cap (v \cup p \cup R;p);p^{\mathsf{T}};R^{\mathsf{T}});s \subseteq \overline{s}$$

We omit the proofs of these inclusions since they are very similar to these of the previous inclusions.

The maintenance of the second formula of the loop invariant is easy to prove, since by using (2) we get $R;(s \cup p) \cup (s \cup p) = R;s \cup R;p \cup s \cup p = v \cup p \cup R;p$. □

For the third proof obligation we verify the error-free termination of the program (W). A consequence of the guard of the loop is that each call of the partial operation *point* is defined. For this reason and the assumed finiteness of the set X, it suffices to show that the loop terminates, i.e., that v is strictly enlarged by each execution of the body of the loop.

Lemma 4.3 *If* $v : X \leftrightarrow \mathbf{1}$ *with* $v \neq \mathsf{L}$, *we have* $v \subset v \cup p \cup R;p$, *for all* $p \in \mathcal{P}_{\overline{v}}$.

Proof. Since $v \subseteq v \cup p \cup R;p$ holds obviously, we show $v \neq v \cup p \cup R;p$ by contradiction. We start with

$$v = v \cup p \cup R;p \iff p \cup R;p \subseteq v \implies p \subseteq v.$$

The last inclusion and the assumption $p \subseteq \overline{v}$ imply $p = \mathsf{O}$, but this contradicts the fact that points are non-empty. □

Finally, we consider the last proof obligation, i.e., that if $v = \mathsf{L}$ holds, then the loop invariant implies the post-condition. Therefore, we also need the pre-condition, in particular the maximum-degree condition, for the proof.

Lemma 4.4 *Given $R : X \leftrightarrow X$, $k \in \mathbb{N}$ and $s, v : X \leftrightarrow \mathbf{1}$ such that Pre(R, k), $v = \mathsf{L}$ and Inv(R, k, s, v), we have Post(R, k, s).*

Proof. Formula (1) of the loop invariant *Inv*(R, k, s, v) and $v = \mathsf{L}$ yield $R;s \subseteq \overline{s}$, which is the first formula of *Post*(R, k, s).

To verify the second formula of *Post*(R, k, s), let $t : X \leftrightarrow \mathbf{1}$ be an arbitrary vector such that $R;t \subseteq \overline{t}$. Then we can calculate as follows:

$$
\begin{array}{ll}
|t| \leq |\mathsf{L}_{X\mathbf{1}}| & t : X \leftrightarrow \mathbf{1}, \text{monotonicity cardinality} \\
= |v| & \text{since } v = \mathsf{L}_{X\mathbf{1}} \\
= |R;s \cup s| & \text{formula (2) of } \mathit{Inv}(R, k, s, v) \\
\leq |R;s| + |s| & \text{cardinality axiom (C3)} \\
= |R;\bigcup_{p \in \mathcal{P}_s} p| + |s| & \text{by Lemma 2.1} \\
= |s| + |\bigcup_{p \in \mathcal{P}_s} R;p| & \\
\leq |s| + \sum_{p \in \mathcal{P}_s} |R;p| & \mathcal{P}_s \text{ finite, cardinality axiom (C3)} \\
\leq |s| + \sum_{p \in \mathcal{P}_s} k & \text{second formula of } \mathit{Pre}(R, k) \\
= |s| + k|s| & \text{by Lemma 3.3} \\
= (k+1)|s| &
\end{array}
$$

□

5 Conclusion and Future Work

By modelling graphs via adjacency relations we developed a relational program based on Wei's algorithm for approximating maximum independent sets in graphs. Therefore, we used vectors and points to model subsets of the vertices. Based on Kawahara's characterisation of the cardinality of relations and further consequences of it we were able to prove facts about their cardinality. In the following, we proved the correctness of the developed program by classical reasoning about the specified pre- and postconditions and loop-invariant. Especially the approximation bound was proved in a purely calculational manner by using Kawahara's and our results about the cardinality of relations.

As future work we plan an exhaustive investigation of the cardinatily operation. We hope to come upon useful laws which can be applied, for example, in the context of correctness proofs of further approximation algorithms. Due to the positive experiences we gained with theorem prover with regard to program verification, we plan an embedding of the cardinality operation in existing libraries for relation algebra as for Isabelle/HOL (see [1]) or Coq (see [10]).

Acknowledgement. I thank P. Höfner and, in particular, R. Berghammer for drawing my attention to this particular research topic and valuable discussions and comments.

References

1. Armstrong, A., Foster, S., Struth, G., Weber, T.: Relation algebra. Archive of Formal Proofs, 2014. http://afp.sf.net/entries/Relation_Algebra.shtml
2. Cormen, T.H., Leiserson, C.E., Rivest, R.L.: Introduction to algorithms. The MIT Press (1990)
3. Francez, N.: Program verification. Addison-Wesley (1992)
4. Furusawa, H.: Algebraic formalisations of fuzzy relations and their representation theorems. Ph.D. thesis, Department of Informatics, Kyushu University (1998)
5. Gries, D.: The science of programming. Springer (1981)
6. Höfner, P., Struth, G.: On automating the calculus of relations. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) Automated Reasoning. LNAI, vol. 5195, pp. 50-66. Springer (2008)
7. Kawahara, Y.: On the cardinality of relations. In: Schmidt, R.A. (ed.): Relations and Kleene Algebra in Computer Science. LNCS, vol. 4136, pp. 251-265. Springer (2006)
8. Maddux, R.: Relation algebras. In: Brink, C., Kahl, W., Schmidt, G. (eds.): Relational Methods in Computer Science. Advances in Computing Science, pp. 22-38. Springer (1997)
9. Maddux, R.: Relation algebras. Studies in Logic and the Foundations of Mathematics, vol. 150. Elsevier (2006)
10. Pous, D.: Relation algebra and KAT in Coq. http://perso.ens-lyon.fr/damien.pous/ra/
11. Schmidt, G., Ströhlein, T.: Relation algebras: Concept of points and representability. Discrete Mathematics 34, 83-97 (1985)
12. Schmidt, G., Ströhlein, T.: Relations and graphs, Discrete mathematics for computer scientists, EATCS Monographs on Theoretical Computer Science. Springer (1993)
13. Schmidt, G.: Relational mathematics. Encyclopedia of Mathematics and its Applications, vol. 132. Cambridge University Press (2010)
14. Tarski, A.: On the calculus of relations. Journal of Symbolic Logic 6, 73-89 (1941)
15. Tarski, A., Givant, S.: A formalization of set theory without variables. Colloquium Publications 41, American Mathematical Society (1987)
16. Wei, V.K.: A lower bound for the stability number of a simple graph. Bell Lab. Tech. Memor. 81-11217-9 (1981)

A Generic Matrix Manipulator

Dylan Killingbeck

Department of Computer Science,
Brock University,
St. Catharines, Ontario, Canada, L2S 3A1
dk10qt@brocku.ca

Abstract. In this paper we describe a generic matrix manipulator system that performs operations on matrices in a flexible way using a graphical user interface. A user defines allowable data entries called a basis, as well as n-ary operations defined on basis elements. These operations can be used in multiple ways to define operations on matrices. A basis and n-ary operations can be entered into the system by various ways including predefined, Java datatypes, JavaScript, and various XML formats defining certain mathematical structures.

Keywords: Allegory, Matrix Manipulation, Semiring, Sup-Semiring

1 Introduction

It is well known that matrices can be used in several key areas of science to provide meaning to data. Often these matrices can be manipulated to provide a solution, based upon the data that they hold and the operations defined between the coefficients. Relations can be represented as graphs, or more specifically as a matrices that defines the relationships between nodes and vertices [7, p. 6]. Using this method of representation an observer can verify a qualitative relationship between elements associated within the relation, and then preform further study using relation algebraic definitions and constructions. For example matrices can be used to represent a relation given as a graph and determine if a Hamiltonian cycle exists. Matrices in generalized linear algebra can also be used to represent quantitative information, and similarly this can provide further information about the meaning of the data. To provide an example, a matrix can represent an interconnected network by which the probability for success of a message traveling between two nodes is represented by a value on the unit interval.

Between the matrix representations of qualitative and quantitative information, it is complex to find meaning, and as such a generic system with user specified parameters is required to further study their behavior. A generic system that is able to flexibly manipulate matrices by user defined operations lifted from the coefficients to matrices, with the inclusion of a user defined data sets (or basis) will allow for an observer to reason between matrices more easily. This generic system will perform similar to the RelView system (see [1])

through a graphical user interface, however it will expand further upon matrix operations, and representations. Primarily, this generic system will focus on automatically inferencing types of operations, and checking the results to ensure compatibility, expanding on RelViews limitations. The remainder of this paper will discuss the mathematical preliminaries, and finally the implementation of this generic matrix manipulator system in detail.

2 Mathematical Preliminary

This section will define the basic definitions of concepts and properties associated with the quantitative and qualitative aspect of matrices. We will start with the quantitative aspect by introducing semirings as a very general approach to linear algebra.

2.1 Semirings

We want to provide a structure that capture the quantitative and qualitative information as discussed in the introduction. We summarize the theory presented in [5] and start with the definition of a semiring.

Definition 1. *$\langle R,+,*,0_R,1_R\rangle$ denotes a semiring if:*

1. *R is a commutative monoid, i.e. we have:*
   (a) *$x+0_R=x$ for all $x\in R$,* *(Identity)*
   (b) *$x+(y+z)=(x+y)+z$ for all $x,y,z\in R$,* *(Associativity)*
   (c) *$x+y=y+x$ for all $x,y\in R$,* *(Commutativity)*
2. *R is a monoid i.e., we have:*
   (a) *$x*1_R= 1_R*x = x$ for all $x\in R$,* *(Identity)*
   (b) *$x*(y*z)=(x*y)*z$ for all $x,y,z\in R$,* *(Associativity)*
3. *Multiplication will distribute over addition, from both the left and the right:*
   (a) *$x*(y+z) = x*y+x*z$,* *(Left Distributivity)*
   (b) *$(x+y)*z = x*z+y*z$,* *(Right Distributivity)*
4. *0_R is the annihilator for multiplication over R, meaning:*
   (a) *$0_R*x=0_R=0_R*x$ for all $x\in R$,* *(Annihilator Law)*

A commutative semiring is a semiring where the monoid $\langle R, *, 1_R\rangle$ is commutative. An additively idempotent semiring is a semiring so that $x + x = x$ for all $x \in R$, similarily a multiplicatively idempotent semiring is semiring so that $x * x = x^2 = x$ for all $x \in R$. We will denote the set of multiplicative idempotent elements of a semiring R by $I(R)$.

Semirings are widely used in theoretical computer science, for example in parsing formal languages [3], or if a semiring is commutative and idempotent (multiplicatively) then it forms a lower semilattice with respect to multiplication. A generalization of this property is stated in the first theorem below [5].

Theorem 1. *Let $\langle R, +, *, 0, 1\rangle$ be a commutative semiring. Then $\langle I(R), *, 0, 1\rangle$ is a lower semilattice with least element 0 and greatest element 1.*

2.2 Allegory

An allegory is the generalization of the category of binary relations between two sets. A morphism R from source A and target B is denoted as $R : A \to B$, from category $\mathcal{R}$, with all the possible morphisms denoted as $\mathcal{R}[A, B]$ [2]. Composition is denoted by ;, for example $R; S$, which reads first R and then S. $\mathbb{I}_A$ denotes the identity morphism for an object A.

Definition 2. *A category $\mathcal{R}$ is an allegory if:*

1. *The class of morphisms $\mathcal{R}[A, B]$ form a lower semilattice, with meet denoted by $\sqcap$ and the induced ordering by $\sqsubseteq$. Elements within this class are called relations.*
2. *For all relations Q, there is a converse such that R:A→B and S:B→C the following holds: $(Q; S)^\smile = S^\smile; Q^\smile$, and $(Q^\smile)^\smile = Q$.*
3. *For all relations Q: A→B, R, S:B→C, then Q;(R⊓S) ⊑ Q;R⊓Q;S.*
4. *For all relations Q:A→B, R:B→C and S:A→C, the modular law Q;R ⊓ S ⊑ Q;(R ⊓ Q$^\smile$;S) holds.*

Finally $\mathcal{R}[A, B]$ is a distributive allegory if $\mathcal{R}[A, B]$ is a distributive lattice with join $\sqcup$ and least element $\perp\!\!\!\perp_{AB}$, satisfying the additional properties:

5. *$Q; \perp\!\!\!\perp_{BC} = \perp\!\!\!\perp_{AC}$ for all relations $Q : A \to B$,*
6. *$Q; (R \sqcup S) = Q; R \sqcup Q; S$ for all relations $Q : A \to B$, $R, S : B \to C$.*

Rel, the category of binary relations between sets as well as the category L-Rel of L-valued relations between sets form a distributive allegory.

2.3 Matrices over Semirings

Matrices of size $m \times n$ will form over a semiring of equal size, induced by addition, denoted by $+$, for example if R is a semiring denoted by $\langle R, +, *, 0_R, 1_R \rangle$ and M is an $m \times n$ matrix, then $M = [a_{ij}]_{mn}$ where coefficients a_{ij} are elements from R. Regular matrix addition and multiplication is respectively defined by:

$$[a_{ij}]_{mn} + [b_{ij}]_{mn} = [a_{ij} + b_{ij}]_{mn}, \quad \text{and} \quad [a_{ij}]_{mn} * [b_{jk}]_{np} = [\sum_{j=1}^{n} a_{ij} * b_{jk}]_{mp}$$

Similarly we can define the transpose of a matrix, and the Hadamard product of two equal sized matrices, respectively defined by:

$$[a_{ij}]^\smile_{mn} = [a_{ji}]_{mn}, \quad \text{and} \quad [a_{ij}]_{mn} \cdot [b_{ij}]_{mn} = [a_{ij} * b_{ij}]_{mn}$$

0_R is defined as the zero matrix $[0]_{mn}$, and 1_R is defined at $[1]_{mn}$. $M + [0]_{mn} = M$, and $M * [0]_{mn} = [0]_{mn}$. Additionally as + is commutative then $M + M' = M' + M$, and as + is associative $M + (M' + M'') = (M + M') + M''$. A matrix that consists of only multiplicatively idempotent elements as coefficients is idempotent with respect to the Hadamard product.

It is a well-known fact that matrices over a lower semilattice form an allegory which leads to the following theorem.

Theorem 2. *Consider the category of matrices with coefficients from a commutative semiring. Then the subcategory of idempotent matrices with respect to the Hadamard product forms an allegory.*

2.4 Sup-Semiring

Recall idempotent elements from a semiring form a lower-semilattice, alternatives matrices with idempotent coefficients form an allegory. We will use these concepts to formulate a new structure, to derive a distributive lattice, and a distributive allegory.

Definition 3. $\langle D, +, *, \sqcup, 0_D, 1_D \rangle$ *denotes a sup-semiring if:*

1. $\langle D, +, *, 0_D, 1_D \rangle$ *is a commutative semiring.*
2. $\langle D, \sqcup \rangle$ *is a commutative semigroup, then*
   (a) $x \sqcup (y \sqcup z) = (x \sqcup y) \sqcup z$ *for all* $x,y,z \in D$, *(Associativity)*
   (b) $x \sqcup y = y \sqcup x$ *for all* $x,y \in D$, *(Commutativity)*
3. $(x \sqcup y) * (x \sqcup y) = x \sqcup y$ *for all* $x,y \in D$, *(Relative Idempotency)*
4. $x * (x \sqcup y) = x$ *for all* $x,y \in D$, *(Absorption)*
5. *if* $x^2 = x$, *then* $x \sqcup (x * y) = x$ *for all* $x,y \in D$, *(Relative Absorption)*
6. *if* $x^2 = x, y^2 = y$ *and* $z^2 = z$,
   then $x * (y \sqcup z) = x * y \sqcup x * z$ *for all* $x,y,z \in D$. *(Relative Distributivity)*

In the case of a sup-semiring we are able to strengthen Theorem 1:

Theorem 3. *Let* $\langle D, +, *, \sqcup, 0, 1 \rangle$ *be a sup-semiring. Then the structure* $\langle I(D), *, \sqcup, 0, 1 \rangle$ *is a distributive lattice.*

Similarly, in the case of a sup-semiring, we are able to strengthen Theorem 2 as follows.

Theorem 4. *Consider the category of matrices with coefficients from a sup-semiring. Then the subcategory of idempotent matrices with respect to the Hadamard product forms a distributive allegory.*

3 Brief Description of the System

As already mentioned in the introduction, a generic matrix manipulator system will be required to carry out various operations and provide meaning to these combined structures. This section will outline the various features, implementations and user interactions provided by the system. The matrix manipulator system is constructed using the Java environment as it offers flexibility to the developer, which can be passed on to the user. This system has previously been started by Milene Santos Teixeira as a project [6]. During this project the basic matrix operations such as matrix multiplication, addition etc have been implemented as methods that use operations on the coefficients of the matrix as parameters. The coefficients and their operations are loaded from user-defined files in XML format. Currently the system is further developed by the author as part of his MSc. thesis.

3.1 User Input

The heart of the system relies on input provided by a user. A user must supply information about two components of the system. The first, a basis to outline coefficients (elements), as well as operations between these coefficients. Secondly, operations between matrices must also be defined. Once these components are defined, a user can supply commands to the system to manipulate matrices. The manipulation possibilities includes:

1. Performing operations on coefficients (elements) within a matrix
2. Performing operations between matrices
3. Storing results, recalling results through executing equations.

Essentially a user will use the graphical user interface to facilitate the input of the basis component and any desirable operations.

Once information has been loaded into the system, the user is free to manipulate the constructed environment within the system as desired, through the graphical user interface, as well as inputting commands through an input field. This input field will accept only valid input based on stored matrices (variables), and the user defined operations between matrices. In order to facilitate such input a flexible parser must be constructed at runtime to parse user input, based on the environment. JParsec, an implementation of Haskell Parsec, is used as it is a two stage parser that provides the flexibility required to parse user input at runtime, given operators with specified operator priority [8].

3.2 User Defined Matrix

Matrices represented in this generic system can be of various forms to provide flexibility. A user may specify the values of the coefficients that correspond specifically to the source (row) and the target (column) values. To be even more general, a matrix can consists of only 1 object type A, and having morphisms that such that F is a morphism, then $F : A \to A$. This type of matrix is convenient to represent Boolean matrices, matrices with real coefficients, or even matrices with integer coefficients. To be more specific, matrices within the system can also represent homogeneous relations, or heterogeneous relations. To provide an example, the following figure below demonstrates a relation with sources along the rows, and targets along the columns.

$$\mathrm{M} = \begin{array}{c|cccc} & A & B & A & A \\ A & 1 & 3 & 2 & 1 \\ B & 2 & 4 & 3 & 1 \end{array}$$

Fig. 1. Typical matrix with sources A and B and targets A,B,A and A

Source and target objects, along with their morphisms must produce a result that is defined and allowable in the system, which is enforced by the user defined basis.

3.3 User Defined Basis

As previously outlined, a user defines a basis which specifies the allowable environment values and n-ary operations between the elements (coefficients). For example, if the basis is defined as the set $\mathbb{N}$, then possible operations could be addition, multiplication, etc. as defined for natural number. Operations defined within the basis must be closed on the given environment. In other words, the operations must accept arbitrary values from then environment as arguments and must return a value within the environment as a result. To make the system as generic as possible, these operations can be loaded into the system through various ways:

1. Java datatypes, i.e., int, double, boolean...
2. XML formats, i.e., lattice structures, graphs...

Below is a typical explicit example, whereby the user defined basis is supplied by .xml files. The first .xml file defines the allowable objects, and data types for the various source and target morphisms (integer in this case):

```
<?xml version="1.0" encoding="UTF-8"?>

<basis name="Example␣Basis" type="explicit">
    <objects type="String">
        A,B
    </objects>
    <morphisms type="Integer" symmetric="true">
        <morSet source="A" target="A">1,2</morSet>
        <morSet source="A" target="B">1,2,3</morSet>
        <morSet source="B" target="B">1,2,3,4,5</morSet>
    </morphisms>
</basis>
```

Above we can see that this basis is defined for only A and B objects, and is explicitly defined. Morphism maps are defined for each source and target object as per above, that is to say with a source A and a target A, the resulting value can only be an integer, with a value of 1 or 2. Since this explicit morphism is defined as symmetric, a source A and a target B is the same as a source B and a target A.

In addition to the user supplied basis .xml file, the user must provide operation for the matrix coefficients, as mentioned above. A typical user defined operation in .xml format is supplied below:

```
<?xml version="1.0" encoding="UTF-8"?>

<unaryOP name="My␣Op" basis="Example␣Basis"
                              code="+" notation="postfix">
    <objectMap type="explicit">
        (A,A),(B,B)
```

```
    </objectMap>
    <morphismMap type="explicit" symmetric="true">
        <morMap source="A" target="A">
            (1,2),(2,1)
        </morMap>
        <morMap source="A" target="B">
            (1,2),(2,1),(3,3)
        </morMap>
        <morMap source="B" target="B">
            (1,2),(2,4),(3,2),(4,5),(5,1)
        </morMap>
    </morphismMap>
</unaryOP>
```

Above we can see that the operation is a unary operator, with a name, code, notation type and corresponding basis. The notation maps the first value of the tuple, to the given second value in the tuple. For example, if the source is A and the target is A and the value is 1, the resulting value of the $+$ operation is 2.

To make the system more convenient, various mathematical structures can be automatically generated by the user. For example, if the user provides a Hasse diagram, then it can be used to generate the corresponding matrix representation, and basis.

3.4 User Defined Operations

Recall, a user is able to specify operations between matrices. These operations have an arbitrary number of parameters, hence the operations can theoretically be n-ary operators, keeping the concept of being generic. These operations between matrices are comprised of the operations defined within the basis. Furthermore, there are two ways that these n-ary operators can be evaluated on matrice. First, operations can be evaluated component wise as defined above, for example matrix addition defined for elements in $M(2, \mathbb{R})$ (the set of all 2×2 matrices with real coefficients), or more specifically the Hadamard product is component wise multiplication. Secondly, operations between two matrices can be defined similar to regular matrix multiplication, as defined above. This type of operation requires two binary operations, the first to provide a result between two coefficients, and the second combines the results of the first operation. An example of this can be observed through regular matrix multiplication in $M(2, \mathbb{R})$, where regular multiplication is the first operation, and the second operation is addition.

Operations between matrices must rely on the underlying operations defined in the basis. This is to ensure that the operations between matrices produce results containing elements defined by the basis. For example, using regular addition defined for matrices with real coefficients, implies that the addition between coefficients is also defined within the basis. These operations can therefore be described as higher order functions, or functions built by using previously defined functions.

Similar to a user defined basis, a user may define these operations between matrices through various methods to increase convenience and usability. A user can define an operation by several methods including:

1. XML formats, i.e. explicitly defined.
2. JavaScript, i.e., user defined functions parsed from JavaScript.
3. Java built-in operations, i.e. addition, multiplication, min, max...
4. Special mathematical structures such as lattices, additive cyclic groups $\mathbb{Z}_n^+$ modulo an integer n, or multiplicative monoids/groups $\mathbb{Z}_n^*$ modulo an integer n.

A user defined operation between matrices must require a symbol to represent the operation for user command input, this symbol is required as part of the operation definition. To further enhance the flexibility of the system, operations can either be in prefix, infix or postfix format.

Finally, operations defined by this system do not require special properties such as associativity or commutativity. The system is more general then required to investigate matrices over semirings or sup-semirings, and in fact any arbitrary matrices can be used, with corresponding user provided operations.

3.5 Graphical User Interface

Another important component of the matrix manipulator system is the graphical user interface (GUI). The GUI allows the user to have a visual representation of the data set they are working with, and also input commands into the system. Overall, the matrix manipulator system will have a comparable interface molded after the RelView system [1]. The GUI is written using Java Swing components to provide a familiar interface, rich component listing as well as reduce developer complexity. The user is able to modify the working space to modify the representations of matrices, for example they can modify the font, size and spacing of coefficients within the matrix. The GUI will provide feedback to the user in the form of a visual display, and also provide feedback be denoting working conditions such as the operations defined, or basis loaded.

4 Conclusion

In order to investigate and simplify matrix manipulation, the creation of a formal system is required. This paper has outlined the requirements, and otherwise desirable features as needed to facilitate such a system. The more flexible the generic matrix manipulator system is, the more use it will be to those requiring such a tool to investigate structures. In closing, the system is currently in development as part of the author's MSc. thesis.

References

1. Berghammer R.: Relview System. `http://www.informatik.uni-kiel.de/~progsys/relview/`: Christian-Albrechts-University of Kiel (2012).

2. Freyd P., Scedrov A.: Categories, Allegories. North-Holland (1990).
3. Goodman J.: Semiring Parsing. Computational Linguistics, 25(4), 573-605 (1999).
4. Hebisch U., & Weinert H.: Semirings. World Scientific (1998).
5. Killingbeck D., Santos Teixeira M., Winter M.: Relations among Matrices over a Semiring. In Kahl W., Oliveira J.N., Winter M. (Eds.): Relational and Algebraic Methods in Computer Science (RAMiCS 15), LNCS 9348, 96-113 (2015).
6. Santos Teixeira M.: A Generic Matrix Manipulator. Undergraduate Project (COSC 3P99), Brock University (2014).
7. Schmidt G., Ströhlein T.: Relations And Graphs. Springer (1993).
8. Yu B.: JParsec. `https://github.com/jparsec/jparsec/`: GitHub (2014).