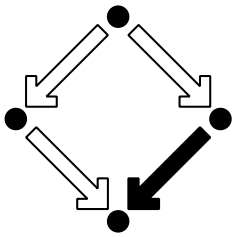
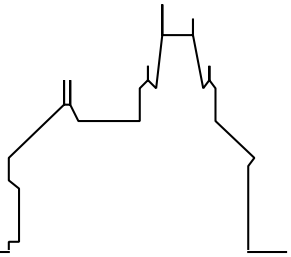


RISC-Linz
Research Institute for Symbolic Computation
Johannes Kepler University
A-4040 Linz, Austria, Europe



SCSS 2008

Austrian-Japanese Workshop on Symbolic Computation in Software Science

Bruno BUCHBERGER, Tetsuo IDA,Temur KUTSIA
(editors)

Castle of Hagenberg, Austria
July 12–13, 2008

RISC-Linz Report Series No. 08-08

Supported by: Austrian Science Fund (FWF), Japan Society for the Promotion of Science (JSPS), Software Competence Center Hagenberg (SCCH)

Preface

This proceedings, published as a RISC Technical Report, collects the papers presented at the Austrian-Japanese Workshop on Symbolic Computation in Software Science (SCSS 2008), held in the Castle of Hagenberg, Austria, in July 12–13, 2008. The workshop grew out of a couple of internal workshops of the Theorema Group at RISC (Research Institute for Symbolic Computation, Austria), the SCORE group (Symbolic Computation Research Group) at the University of Tsukuba, Japan, and the SSFG (Software Science Foundation Group) at Kyoto University, Japan, and is the first in the series which is open now for the international community. The authors of 14 contributed papers came from Austria, Belgium, Japan, The Netherlands, Romania, and Switzerland. In addition to the selected papers, the scientific program included two invited talks: Symbolic Computation in Software Science by Hoon Hong from North Carolina State University, and Some Challenges for Automated Theorem Proving by Dana Scott from Carnegie Mellon University.

We would like to thank the Program Committee members and all the referees for their careful work in the review process. The submission and programme committee work was organized through the EasyChair system. Special thanks go to the Austrian Science Fund (FWF) and the Japan Society for the Promotion of Science (JSPS) who have supported the event under the Japan-Austria Research Cooperative Program, and to Software Competence Center Hagenberg (SCCH).

The workshop has been organized by the Research Institute for Symbolic Computation (RISC) of the Johannes Kepler University Linz, in collaboration with the Software Competence Center Hagenberg (SCCH). It has been a part of the series of international scientific events RISC Summer 2008.

July 2008

Bruno Buchberger
Tetsuo Ida
Temur Kutsia

Conference Organization

General Chairs

Bruno Buchberger	RISC, Johannes Kepler University Linz
Tetsuo Ida	University of Tsukuba

Programme Chairs

Bruno Buchberger	RISC, Johannes Kepler University Linz
Tetsuo Ida	University of Tsukuba
Temur Kutsia	RISC, Johannes Kepler University Linz

Programme Committee

Dirk Draheim	Software Competence Center Hagenberg
Atsushi Igarashi	Kyoto University
Tudor Jebelean	RISC, Johannes Kepler University Linz
Yukiyoshi Kameyama	University of Tsukuba
Franz Lichtenberger	RISC, Johannes Kepler University Linz
Aart Middeldorp	University of Innsbruck
Yasuhiko Minamide	University of Tsukuba
Masahiko Sato	Kyoto University
Wolfgang Schreiner	RISC, Johannes Kepler University Linz
Gerhard Weiß	Software Competence Center Hagenberg

Local Organization

Bruno Buchberger
Betina Curtis (administration)
Temur Kutsia
Alexander Zapletal (Webmaster)

Sponsors

Austrian Science Fund (FWF)
Japan Society for the Promotion of Science (JSPS)
Software Competence Center Hagenberg (SCCH)

External Reviewers

Juergen Giesl
Florent Jacquemard
Georg Moser
Koji Nakazawa
Takafumi Sakurai

Table of Contents

Designing a Rewriting Induction Prover with an Increased Capability of Non-Orientable Theorems

Takahito Aoto

RIEC, Tohoku University, Japan
aoto@nue.riec.tohoku.ac.jp

Abstract. Rewriting induction (Reddy, 1990) is an automated proof method for inductive theorems of term rewriting systems. Reasoning by the rewriting induction is based on the noetherian induction on some reduction order and the original rewriting induction is not capable of proving theorems which are not orientable by that reduction order. To deal with such theorems, Bouhoula (1995) as well as Dershowitz & Reddy (1993) used the ordered rewriting. However, even using ordered rewriting, the weak capability of non-orientable theorems is considered one of the weakness of rewriting induction approach compared to other automated methods for proving inductive theorems. We present a refined system of rewriting induction with an increased capability of non-orientable theorems and a capability of disproving incorrect conjectures. Soundness for proving/disproving are shown and effectiveness of our system is demonstrated through some examples.

1 Introduction

Properties of programs are often proved by induction on data structures such as natural numbers or lists. Such properties are called *inductive properties* of programs. Inductive properties are indispensable in formal treatments of programs such as program verification and program transformation. For such applications, automated reasoning on inductive properties is crucial.

Comparing to the high degree of automation on automated proving of theorems, automated proving of *inductive* theorems still is considered as a very challenging problem [15]. Currently best known successful approaches to automated proving of inductive theorems are *explicit* induction methods with sophisticated heuristics [12, 17, 23] or with powerful decision procedures [27]. On the other hand, in the field of term rewriting, *implicit* induction methods for equational theories that automatically perform inductive reasoning based on implicit induction principles have been investigated for many years [5–10, 14, 16, 18, 20, 21, 24–26, 29, 31]. Although it is not among the best known successful approaches, some extensions are known to be competitive [5, 7].

Rewriting induction[1] proposed by Reddy [26] is one of such inductive theorem proving methods. Contrasted to *inductionless induction* [16, 18, 21, 25, 31], in which some kind of Church-Rosser property is needed, the basis of rewriting induction is a noetherian induction. The theorem prover SPIKE [6, 9, 10] is the best known successful induction prover based on a variant of rewriting induction *test set induction*.

Many refinements have been introduced for the rewriting induction to increase its power and efficiency of automated theorem proving. The underlying rewriting mechanism has been replaced by ordered rewriting in [13] so that rewriting induction can also handle non-orientable equations; not only ordered rewriting but also relaxed rewriting is used in the SPIKE theorem prover to get more flexible expansion and simplification rules. Modulo rewriting is also used to deal with non-orientable theorems [1]. Another refinement is to use simplification by conjectures (i.e. equations to prove) [2, 6, 9–11, 13] and a mechanism for disproving incorrect conjectures [6, 9]. A capability of theories with non-free constructors is investigated in [7, 8, 19].

Another direction for extension is to make the framework more general. The SPIKE theorem prover can handle not only equational theories but conditional ones; moreover, inductive theorems can be given not only in equations but also in clauses. Further generalization is given in [11] whose underlying logical theory is replaced with an abstract first-order deductive relation. Stratulat [29] strengthens such an abstraction further by a general abstract inference system that can be used to prove general inductive properties of any first-order deductive relation. Equations with regular constraints can be also treated in [7, 8]. Needless to say, such extensions also benefit from the enhancement of the proving power on the basic rewriting induction system.

It is well-known that an introduction of suitable lemmas often prevents automated inductive theorem proving from divergence. Thus the techniques to introduce suitable lemmas automatically in the process of proving have been investigated [22, 28, 33, 34]. Divergence critic [34] is an automated lemma discovery method for rewriting induction which finds lemmas from a divergent sequence of proofs. The SPIKE theorem prover contains a lemma discovery tool based on the divergence critic. Urso & Kounalis [33] gave a lemma discovery method *Sound Generalization* for monomorphic term rewriting systems which is sound, that is, does not generate incorrect lemmas from correct conjectures. A part of divergence critic is extend to sound one by Shimazu et.al. [28].

In this paper, we present a refined inference system of rewriting induction with an increased capability of non-orientable theorems. We also present how the system is combined with rules for adding sound lemmas and disproving incorrect conjectures. Soundness of the presented systems is shown and effectiveness is demonstrated through examples. A part of our inference system is implemented and we report a preliminary experiment.

[1] Originally, it is called "*term rewriting induction*". The terminology "*rewriting induction*" is introduced in [24].

The rest of the paper is organized as follows. After fixing basic notations (Section 2), we review rewriting induction (Section 3). In Section 4, we present our basic system of rewriting induction. Then, we incorporate a rule for adding sound lemmas (Section 5) and rules for disproving incorrect conjectures (Section 6). Section 7 introduces a modification of systems which turns out to be useful by our preliminary experiment. In Section 8, we compare our system with other rewriting induction provers. Section 9 concludes.

2 Preliminaries

We introduce notations for term rewriting used in this paper. (For details, see [3].) The sets of function symbols and variables are denoted by $\mathcal{F}$ and V, respectively. The set of terms over $\mathcal{F}, V$ is denoted by $\mathrm{T}(\mathcal{F}, V)$. We use $\equiv$ to denote the syntactical equality. We write $u \trianglelefteq t$ if u is a subterm of t. The *root symbol* of a term t is denoted by $\mathrm{root}(t)$ and the set of variables in a term t by $V(t)$.

Let $\Box$ be a constant not occurring in $\mathcal{F}$. A *context* is an element in $\mathrm{T}(\mathcal{F} \cup \{\Box\}, V)$. The constant $\Box$ is called a *hole*. If a context C has n holes in it, we denote by $C[t_1, \ldots, t_n]$ a term obtained by replacing holes with $t_1, \ldots, t_n$ from left to right. A mapping σ from V to $\mathrm{T}(\mathcal{F}, V)$ is called a *substitution*; we identify σ and its homomorphic extension. $\sigma(t)$ is also written as $t\sigma$, which is called an *instance* of the term t. The *domain* of σ is defined by $\mathrm{dom}(\sigma) = \{x \in V \mid x\sigma \not\equiv x\}$. We denote by $\mathrm{mgu}(s, t)$ the *most general unifier* of terms s, t.

A pair $\langle l, r\rangle$ of terms l, r satisfying conditions (1) $\mathrm{root}(l) \in \mathcal{F}$ and (2) $V(r) \subseteq V(l)$ is said to be a *rewrite rule*. A rewrite rule $\langle l, r\rangle$ is denoted by $l \to r$. A *term rewriting system (TRS)* is a set of rewrite rules. Let $\mathcal{R}$ be a TRS. If there exist a context C, a substitution σ, and a rewrite rule $l \to r \in \mathcal{R}$ such that $s \equiv C[l\sigma]$ and $t \equiv C[r\sigma]$, we write $s \to_{\mathcal{R}} t$. We call $s \to_{\mathcal{R}} t$ a *rewrite step*. $\to_{\mathcal{R}}$ forms a relation on $\mathrm{T}(\mathcal{F}, V)$, called the *rewrite relation* of $\mathcal{R}$. A term t is said to be *normal* when there exists no s such that $t \to_{\mathcal{R}} s$. An *equation* $l \doteq r$ is a pair $\langle l, r\rangle$ of terms. When we write $l \doteq r$, however, we do not distinguish $\langle l, r\rangle$ and $\langle r, l\rangle$. The rewrite relation of a set E of equations is defined as $s \leftrightarrow_E t$ if there exist a context C, a substitution σ and an equation $l \doteq r \in E$ satisfying $s \equiv C[l\sigma]$ and $t \equiv C[r\sigma]$. The reflexive closure and reflexive transitive closure of $\to_{\mathcal{R}}$ ($\leftrightarrow_E$) are denoted by $\xrightarrow{=}_{\mathcal{R}}$ (resp. $\xleftrightarrow{=}_E$) and $\xrightarrow{*}_{\mathcal{R}}$ (resp. $\xleftrightarrow{*}_E$). The symmetric closure of $\to_{\mathcal{R}}$ is denoted by $\leftrightarrow_{\mathcal{R}}$. The modulo rewriting relation is defined like this: $\leftrightarrow_{\mathcal{R}/E} = \xleftrightarrow{*}_E \circ \to_{\mathcal{R}} \circ \xleftrightarrow{*}_E$, where $\circ$ denotes the composition of relations. When $s \equiv C[s_1, \ldots, s_n]$, $t \equiv C[t_1, \ldots, t_n]$, and $s_i \to_{\mathcal{R}} t_i$ for all $i = 1, \ldots, n$, we write $s \Vdash_{\mathcal{R}} t$.

The set of *defined function symbols* is given by $\mathcal{D}_{\mathcal{R}} = \{\mathrm{root}(l) \mid l \to r \in \mathcal{R}\}$ and the set of *constructor symbols* by $\mathcal{C}_{\mathcal{R}} = \mathcal{F} \setminus \mathcal{D}_{\mathcal{R}}$. The set of defined symbols appearing in a term t is denoted by $\mathcal{D}_{\mathcal{R}}(t)$. When $\mathcal{R}$ is obvious from its context, we omit the subscript $_{\mathcal{R}}$ from $\mathcal{D}_{\mathcal{R}}, \mathcal{C}_{\mathcal{R}}$. Terms in $\mathrm{T}(\mathcal{C}, V)$ are said to be *constructor terms*; a substitution σ such that $x\sigma \in \mathrm{T}(\mathcal{C}, V)$ for any $x \in \mathrm{dom}(\sigma)$ is called a *constructor substitution*. A term of the form $f(c_1, \ldots, c_n)$ for some $f \in \mathcal{D}$ and

$c_1, \ldots, c_n \in \mathrm{T}(\mathcal{C}, V)$ is said to be *basic*. The set $\{u \trianglelefteq s \mid \exists f \in \mathcal{D}.\ \exists c_1, \ldots, c_n \in \mathrm{T}(\mathcal{C}, V).\ u \equiv f(c_1, \ldots, c_n)\}$ of basic subterms of s is written as $\mathcal{B}(s)$.

A term t is said to be *ground* if $V(t) = \emptyset$. The set of ground terms is denoted by $\mathrm{T}(\mathcal{F})$. If $t\sigma \in \mathrm{T}(\mathcal{F})$, $t\sigma$ is called a *ground instance* of t. The ground instance of a rewrite rule, an equation, etc. is defined similarly. A *ground substitution* is a substitution σ_g such that $x\sigma_g \in \mathrm{T}(\mathcal{F})$ for any $x \in \mathrm{dom}(\sigma_g)$. A TRS $\mathcal{R}$ is said to be *quasi-reducible* if no ground basic term is normal. Without loss of generality, we assume that $t\sigma_g$ is ground (i.e. $V(t) \subseteq \mathrm{dom}(\sigma_g)$) when we speak of an instance $t\sigma_g$ of t by a ground substitution σ_g; and so for ground instances of rewrite rules, equations, etc. An *inductive theorem* of a TRS $\mathcal{R}$ is an equation that is valid on $\mathrm{T}(\mathcal{F})$, i.e. $s \doteq t$ is an inductive theorem if $s\sigma_g \overset{*}{\leftrightarrow}_{\mathcal{R}} t\sigma_g$ holds for any ground instance $s\sigma_g \doteq t\sigma_g$. We write $\mathcal{R} \vdash_{ind} E$ then all equations in E are inductive theorems of $\mathcal{R}$.

A relation R on $\mathrm{T}(\mathcal{F}, V)$ is said to be *closed under substitutions* if $s\ R\ t$ implies $s\sigma\ R\ t\sigma$ for any substitution σ; *closed under contexts* if $s\ R\ t$ implies $C[s]\ R\ C[t]$ for any context C. A *reduction order* (*reduction quasi-order*) is a well-founded partial order (resp. quasi-order) on $\mathrm{T}(\mathcal{F}, V)$ that is closed under substitutions and contexts. For a quasi-order $\succsim$, we let $\approx = \succsim \cap \precsim$ and $\succ = \succsim \setminus \precsim$. A quasi-order $\succsim$ on $\mathrm{T}(\mathcal{F}, V)$ is said to be *ground-total* if $s_g \succsim t_g$ or $s_g \precsim t_g$ for any $s_g, t_g \in \mathrm{T}(\mathcal{F})$.

3 Rewriting Induction

Rewriting induction proposed by Reddy [26] is a method to prove inductive theorems automatically. This section reviews the rewriting induction.

Let $\mathcal{R}$ be a TRS and $>$ a reduction order. We list inference rules of rewriting induction in **Fig.1**. In the figure, the relation $\uplus$ expresses the disjoint union and the ternary operation Expd is defined as:

$$\mathrm{Expd}_u(s, t) = \{C[r]\sigma \doteq t\sigma \mid s \equiv C[u], \sigma = \mathrm{mgu}(u, l), l \to r \in \mathcal{R}, l{:}\mathrm{basic}\}$$

A rewriting induction procedure starts from a pair $\langle E_0, \emptyset \rangle$ where E_0 is the set of conjectures to prove. It successively applies the inference rules to a pair $\langle E, H \rangle$. Intuitively, E is a set of equations to be proved and H is a set of induction hypotheses and theorems already proved. If a derivation eventually reaches the form $\langle \emptyset, H' \rangle$ then $\mathcal{R} \vdash_{ind} E_0$. On the other hand, when none of the rules are applicable for $\langle E, H \rangle$ with $E \neq \emptyset$, the procedure reports "failure" and the procedure may also run forever ("divergence")—in these cases, rewriting induction fails to prove $\mathcal{R} \vdash_{ind} E_0$. We use $\rightsquigarrow$ to denote one step application of an inference rule possibly with a superscript indicating which inference rule is used. $\overset{*}{\rightsquigarrow}$ is the reflexive transitive closure of $\rightsquigarrow$.

*Example 1 (*RI*).* Let

$$\mathcal{R} = \left\{ \begin{array}{ll} \mathsf{plus}(\mathsf{0}, y) & \to y \\ \mathsf{plus}(\mathsf{s}(x), y) & \to \mathsf{s}(\mathsf{plus}(x, y)) \end{array} \right\},$$

Expand

$$\frac{\langle E \uplus \{s \doteq t\},\ H\rangle}{\langle E \cup \mathrm{Expd}_u(s,t),\ H \cup \{s \to t\}\rangle}\ u \in \mathcal{B}(s), s > t$$

Simplify

$$\frac{\langle E \uplus \{s \doteq t\},\ H\rangle}{\langle E \cup \{s' \doteq t\},\ H\rangle}\ s \to_{\mathcal{R}\cup H} s'$$

Delete

$$\frac{\langle E \uplus \{s \doteq s\},\ H\rangle}{\langle E,\ H\rangle}$$

Fig. 1. Inference rules of RI

$E = \{\mathsf{plus}(\mathsf{plus}(x,y),z) \doteq \mathsf{plus}(x,\mathsf{plus}(y,z))\}$, $>$ a lexicographic path order based on the precedence $\mathsf{plus} > \mathsf{s} > \mathsf{0}$. The following is a successful derivation of RI:

$$\begin{aligned}
&\left\langle \left\{ \mathsf{plus}(\mathsf{plus}(x,y),z) \doteq \mathsf{plus}(x,\mathsf{plus}(y,z)) \right\}, \left\{\right\} \right\rangle \\
\rightsquigarrow^{e}_{\mathrm{RI}} &\left\langle \begin{Bmatrix} \mathsf{plus}(y_0,z) \doteq \mathsf{plus}(\mathsf{0},\mathsf{plus}(y_0,z)) \\ \mathsf{plus}(\mathsf{s}(\mathsf{plus}(x_1,y_1)),z) \doteq \mathsf{plus}(\mathsf{s}(x_1),\mathsf{plus}(y_1,z)) \end{Bmatrix}, \left\{ \mathsf{plus}(\mathsf{plus}(x,y),z) \to \mathsf{plus}(x,\mathsf{plus}(y,z)) \right\} \right\rangle \\
\overset{*}{\rightsquigarrow}^{s}_{\mathrm{RI}} &\left\langle \begin{Bmatrix} \mathsf{plus}(y_0,z) \doteq \mathsf{plus}(y_0,z) \\ \mathsf{s}(\mathsf{plus}(x_1,\mathsf{plus}(y_1,z))) \doteq \mathsf{s}(\mathsf{plus}(x_1,\mathsf{plus}(y_1,z))) \end{Bmatrix}, \left\{ \mathsf{plus}(\mathsf{plus}(x,y),z) \to \mathsf{plus}(x,\mathsf{plus}(y,z)) \right\} \right\rangle \\
\overset{*}{\rightsquigarrow}^{d}_{\mathrm{RI}} &\left\langle \{\}, \left\{ \mathsf{plus}(\mathsf{plus}(x,y),z) \to \mathsf{plus}(x,\mathsf{plus}(y,z)) \right\} \right\rangle
\end{aligned}$$

4 Proving Non-Orientable Theorems

In this section, we present the basic system BRI of our rewriting induction extended with a capability of proving non-orientable theorems.

The inference rules of BRI are presented in **Fig.2**. The system is based on a TRS $\mathcal{R}$ and a reduction quasi-order $\succsim$. Elements of H are rewriting rules, and those of K are equations $l \doteq r$ such that $l \not\succ r$ nor $r \not\succ l$. $K^{\succ}$ and $K^{\approx}$ are instantiations of equations in K whose sides are orientable or equivalent (c.f. [4]):

$$K^{\succ} = \{l\sigma \to r\sigma \mid l \doteq r \in K, l\sigma \succ r\sigma\}$$
$$K^{\approx} = \{l\sigma = r\sigma \mid l \doteq r \in K, l\sigma \approx r\sigma\}$$

Expd2 is an operation introduced in [1] that expands not only the larger side of an equation but both sides of the equation:

$$\mathrm{Expd2}_{u,v}(s,t) = \bigcup \left\{ \mathrm{Expd}_{v\sigma}(t\sigma, s') \mid s' \doteq t\sigma \in \mathrm{Expd}_u(s,t) \right\}$$

Example 2 (Expd2). Let $\mathcal{R}$ be as in *Example 1*, $s \equiv \mathsf{plus}(x,\mathsf{plus}(y,z))$, and $t \equiv \mathsf{plus}(y,\mathsf{plus}(x,z))$. Then $u \equiv \mathsf{plus}(y,z)$ is the only basic subterm of s and $v \equiv \mathsf{plus}(x,z)$ is the only basic subterm of t. We have

$$\mathrm{Expd2}_{u,v}(s,t) = \left\{ \begin{array}{l} \mathsf{plus}(\mathsf{0},z) \doteq \mathsf{plus}(\mathsf{0},z), \\ \mathsf{plus}(\mathsf{s}(x_1),z) \doteq \mathsf{plus}(\mathsf{0},\mathsf{s}(\mathsf{plus}(x_1,z))), \\ \mathsf{plus}(\mathsf{s}(x_2),\mathsf{s}(\mathsf{plus}(y_2,z))) \doteq \mathsf{plus}(\mathsf{s}(y_2),\mathsf{s}(\mathsf{plus}(x_2,z))) \end{array} \right\}.$$

Expand

$$\frac{\langle E \uplus \{s \doteq t\},\ H,\ K\rangle}{\langle E \cup \mathrm{Expd}_u(s,t),\ H \cup \{s \to t\},\ K\rangle}\ u \in \mathcal{B}(s), s \succ t$$

Expand2

$$\frac{\langle E \uplus \{s \doteq t\},\ H,\ K\rangle}{\langle E \cup \mathrm{Expd2}_{u,v}(s,t),\ H,\ K \cup \{s \doteq t\}\rangle}\ u \in \mathcal{B}(s), v \in \mathcal{B}(t), s \not\succ t \wedge t \not\succ s$$

Simplify

$$\frac{\langle E \uplus \{s \doteq t\},\ H,\ K\rangle}{\langle E \cup \{s' \doteq t\},\ H,\ K\rangle}\ s \to_{(\mathcal{R} \cup H \cup K^{\succ})/K^{\approx}} s'$$

Simplify-C

$$\frac{\langle E \uplus \{s \doteq t\},\ H,\ K\rangle}{\langle E \cup \{s' \doteq t\},\ H,\ K\rangle}\ s \overset{*}{\leftrightarrow}_{K^{\approx}} \circ \Vdash_E \circ \overset{*}{\leftrightarrow}_{K^{\approx}} s', s \succsim s' \vee t \succsim s'$$

Delete

$$\frac{\langle E \uplus \{s \doteq t\},\ H,\ K\rangle}{\langle E,\ H,\ K\rangle}\ s \overset{*}{\leftrightarrow}_{K^{\approx}} \circ \Vdash_K \circ \overset{*}{\leftrightarrow}_{K^{\approx}} t$$

Fig. 2. Inference rules of BRI

The system BRI is an extension of eRI of [1] and cRI of [2]. Inference rules *Expand2* and *Simplify-C* are extended (instead, we will impose the ground-totality of $\succsim$ as we will see) from the corresponding rules of eRI and cRI. The biggest difference is that, the system includes not only a capability of conjectures with equivalent sides but also that of conjectures with incomparable sides.

Theorem 1 (soundness of BRI). *Let $\mathcal{R}$ be a quasi-reducible TRS, E a set of equations, $\succsim$ a ground-total reduction quasi-order satisfying $\mathcal{R} \subseteq \succ$. If $\langle E, \emptyset, \emptyset\rangle \overset{*}{\rightsquigarrow}_{\mathsf{BRI}} \langle \emptyset, H, K\rangle$ for some H, K, then $\mathcal{R} \vdash_{ind} E$.*

Proof. Proved based on a method similar to the proofs of [1, 2]. Abstract principle is designed based on the ground-totality. One also needs the commutativity of rewrite steps at parallel positions to show the property of $s_g \Vdash_E t_g$. □

*Example 3 (*BRI*).* Let $\mathcal{R}$ be as in *Example 1* and

$$E = \left\{ \mathsf{plus}(x, \mathsf{plus}(y, z)) \doteq \mathsf{plus}(y, \mathsf{plus}(x, z)) \right\}$$

Let $\succsim$ be a multiset path order based on the precedence $\mathsf{plus} \succ \mathsf{s} \succ \mathsf{0}$. The following is a successful derivation of BRI:

$$\left\langle \left\{ \mathsf{plus}(x, \mathsf{plus}(y, z)) \doteq \mathsf{plus}(y, \mathsf{plus}(x, z)) \right\}, \{\}, \{\} \right\rangle$$

$$\rightsquigarrow^{e2} \left\langle \begin{array}{l} \left\{ \begin{array}{l} \mathsf{plus}(\mathsf{0}, z) \doteq \mathsf{plus}(\mathsf{0}, z), \\ \mathsf{plus}(\mathsf{s}(x_1), z) \doteq \mathsf{plus}(\mathsf{0}, \mathsf{s}(\mathsf{plus}(x_1, z))), \\ \mathsf{plus}(\mathsf{s}(x_2), \mathsf{s}(\mathsf{plus}(y_2, z))) \doteq \mathsf{plus}(\mathsf{s}(y_2), \mathsf{s}(\mathsf{plus}(x_2, z))) \end{array} \right\} \\ \{\}, \left\{ \mathsf{plus}(x, \mathsf{plus}(y, z)) \doteq \mathsf{plus}(y, \mathsf{plus}(x, z)) \right\} \end{array} \right\rangle$$

$$\overset{*}{\rightsquigarrow}_s \overset{*}{\rightsquigarrow}_d \left\langle \begin{array}{l} \{\, \mathsf{s}(\mathsf{plus}(x_2, \mathsf{s}(\mathsf{plus}(y_2, z)))) \doteq \mathsf{s}(\mathsf{plus}(y_2, \mathsf{s}(\mathsf{plus}(x_2, z)))) \,\} \\ \{\,\}, \{\, \mathsf{plus}(x, \mathsf{plus}(y, z)) \doteq \mathsf{plus}(y, \mathsf{plus}(x, z)) \,\} \end{array} \right\rangle$$

$$\rightsquigarrow_{e2} \left\langle \begin{array}{l} \left\{ \begin{array}{l} \mathsf{s}(\mathsf{plus}(0, \mathsf{s}(z))) \doteq \mathsf{s}(\mathsf{plus}(0, \mathsf{s}(z))) \\ \mathsf{s}(\mathsf{plus}(\mathsf{s}(x_3), \mathsf{s}(z))) \doteq \mathsf{s}(\mathsf{plus}(0, \mathsf{s}(\mathsf{s}(\mathsf{plus}(x_3, z))))) \\ \mathsf{s}(\mathsf{plus}(\mathsf{s}(x_3), \mathsf{s}(\mathsf{s}(\mathsf{plus}(y_3, z))))) \doteq \mathsf{s}(\mathsf{plus}(\mathsf{s}(y_3), \mathsf{s}(\mathsf{s}(\mathsf{plus}(x_3, z))))) \end{array} \right\} \\ \{\,\}, \left\{ \begin{array}{l} \mathsf{s}(\mathsf{plus}(x_2, \mathsf{s}(\mathsf{plus}(y_2, z)))) \doteq \mathsf{s}(\mathsf{plus}(y_2, \mathsf{s}(\mathsf{plus}(x_2, z)))) \\ \mathsf{plus}(x, \mathsf{plus}(y, z)) \doteq \mathsf{plus}(y, \mathsf{plus}(x, z)) \end{array} \right\} \end{array} \right\rangle$$

$$\overset{*}{\rightsquigarrow}_s \overset{*}{\rightsquigarrow}_d \left\langle \begin{array}{l} \left\{ \begin{array}{l} \mathsf{s}(\mathsf{s}(\mathsf{plus}(x_3, \mathsf{s}(z)))) \doteq \mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{plus}(x_3, z)))) \\ \mathsf{s}(\mathsf{s}(\mathsf{plus}(x_3, \mathsf{s}(\mathsf{s}(\mathsf{plus}(y_3, z)))))) \doteq \mathsf{s}(\mathsf{s}(\mathsf{plus}(y_3, \mathsf{s}(\mathsf{s}(\mathsf{plus}(x_3, z))))))) \end{array} \right\} \\ \{\,\}, \left\{ \begin{array}{l} \mathsf{s}(\mathsf{plus}(x_2, \mathsf{s}(\mathsf{plus}(y_2, z)))) \doteq \mathsf{s}(\mathsf{plus}(y_2, \mathsf{s}(\mathsf{plus}(x_2, z)))) \\ \mathsf{plus}(x, \mathsf{plus}(y, z)) \doteq \mathsf{plus}(y, \mathsf{plus}(x, z)) \end{array} \right\} \end{array} \right\rangle$$

$$\rightsquigarrow_{e1} \left\langle \begin{array}{l} \left\{ \begin{array}{l} \mathsf{s}(\mathsf{s}(\mathsf{s}(z))) \doteq \mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{plus}(0, z)))) \\ \mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{plus}(x_4, \mathsf{s}(z))))) \doteq \mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{plus}(\mathsf{s}(x_4), z)))) \\ \mathsf{s}(\mathsf{s}(\mathsf{plus}(x_3, \mathsf{s}(\mathsf{s}(\mathsf{plus}(y_3, z)))))) \doteq \mathsf{s}(\mathsf{s}(\mathsf{plus}(y_3, \mathsf{s}(\mathsf{s}(\mathsf{plus}(x_3, z))))))) \end{array} \right\} \\ \{\, \mathsf{s}(\mathsf{s}(\mathsf{plus}(x_3, \mathsf{s}(z)))) \to \mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{plus}(x_3, z)))) \,\} \\ \left\{ \begin{array}{l} \mathsf{s}(\mathsf{plus}(x_2, \mathsf{s}(\mathsf{plus}(y_2, z)))) \doteq \mathsf{s}(\mathsf{plus}(y_2, \mathsf{s}(\mathsf{plus}(x_2, z)))) \\ \mathsf{plus}(x, \mathsf{plus}(y, z)) \doteq \mathsf{plus}(y, \mathsf{plus}(x, z)) \end{array} \right\} \end{array} \right\rangle$$

$$\overset{*}{\rightsquigarrow}_s \overset{*}{\rightsquigarrow}_d \left\langle \begin{array}{l} \{\,\}, \{\, \mathsf{s}(\mathsf{s}(\mathsf{plus}(x_3, \mathsf{s}(z)))) \to \mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{plus}(x_3, z)))) \,\} \\ \left\{ \begin{array}{l} \mathsf{s}(\mathsf{plus}(x_2, \mathsf{s}(\mathsf{plus}(y_2, z)))) \doteq \mathsf{s}(\mathsf{plus}(y_2, \mathsf{s}(\mathsf{plus}(x_2, z)))) \\ \mathsf{plus}(x, \mathsf{plus}(y, z)) \doteq \mathsf{plus}(y, \mathsf{plus}(x, z)) \end{array} \right\} \end{array} \right\rangle$$

We note that the system eRI [1] is not capable of this proof, because the conjecture $\mathsf{plus}(x, \mathsf{plus}(y, z)) \doteq \mathsf{plus}(y, \mathsf{plus}(x, z))$ has incomparable sides.

5 Adding Sound Lemmas

It is well-known that an introduction of suitable lemmas often prevents automated inductive theorem proving from divergence. Thus it is very helpful for the success of derivations to add a suitable lemma automatically in the process of proving.

Divergence critic [34] is an automated lemma discovery method for rewriting induction which finds lemmas from a divergent sequence of proofs. The SPIKE theorem prover contains a lemma discovery tool based on the divergence critic. However, the divergence critic may introduce a lemma that is not an inductive theorem. This fact complicates the design of a rewriting induction prover with an automated introduction of lemmas.

Another approach is to add only lemmas that are guaranteed to be inductive theorems (when the initial conjectures are inductive theorems). Urso & Kounalis [33] gave a lemma discovery method called *Sound Generalization* for monomorphic TRSs which is sound, that is, does not generate incorrect lemmas from inductive theorems. A part of divergence critic is extend to sound one by Shimazu et.al. [28].

We incorporate an inference rule for adding sound lemmas (**Fig.3**). We denote by BRIL the obtained system.

$$\textit{Lemma}\quad \frac{\langle E,\ H,\ K\rangle}{\langle E \cup L,\ H,\ K\rangle}\ \mathcal{R} \vdash_{ind} E \Longrightarrow \mathcal{R} \vdash_{ind} L$$

Fig. 3. Additional inference rules of BRIL

Theorem 2 (soundness of BRIL). *Let $\mathcal{R}$ be a quasi-reducible TRS, E a set of equations, $\succsim$ a ground-total reduction quasi-order satisfying $\mathcal{R} \subseteq \succ$. If $\langle E, \emptyset, \emptyset\rangle \overset{*}{\rightsquigarrow}_{\mathsf{BRIL}} \langle \emptyset, H, K\rangle$ for some H, K, then $\mathcal{R} \vdash_{ind} E$.*

Proof. The case for the application of *Lemma* inference rule is safely incorporated into the proof of **Theorem 1** without modifying the abstract principle. □

*Example 4 (*BRIL*).* Let $\mathcal{R}$ and E be as follows:

$$\mathcal{R} = \left\{ \begin{array}{ll} \mathsf{plus}(\mathsf{0}, y) & \to y \\ \mathsf{plus}(\mathsf{s}(x), y) & \to \mathsf{s}(\mathsf{plus}(x, y)) \\ \mathsf{times}(\mathsf{0}, y) & \to \mathsf{0} \\ \mathsf{times}(\mathsf{s}(x), y) & \to \mathsf{plus}(y, \mathsf{times}(x, y)) \end{array} \right\}$$

$$E = \left\{ \mathsf{times}(x, \mathsf{s}(y)) \doteq \mathsf{plus}(x, \mathsf{times}(x, y)) \right\}$$

Then $\mathcal{R}$ is monomorphic. Let $\succsim$ be a multiset path order based on the precedence $\mathsf{plus} \succ \mathsf{s} \succ \mathsf{0}$. The following is a successful derivation of BRIL using the *Lemma* inference with the sound generalization [33].

$$\left\langle \left\{ \mathsf{times}(x, \mathsf{s}(y)) \doteq \mathsf{plus}(x, \mathsf{times}(x, y)) \right\}, \{\}, \{\} \right\rangle$$

$$\rightsquigarrow^{e} \quad \left\langle \begin{array}{l} \left\{ \begin{array}{l} \mathsf{0} \doteq \mathsf{plus}(\mathsf{0}, \mathsf{times}(\mathsf{0}, y)) \\ \mathsf{plus}(\mathsf{s}(y), \mathsf{times}(x_1, \mathsf{s}(y))) \doteq \mathsf{plus}(\mathsf{s}(x_1), \mathsf{times}(\mathsf{s}(x_1), y)) \end{array} \right\} \\ \left\{ \mathsf{times}(x, \mathsf{s}(y)) \to \mathsf{plus}(x, \mathsf{times}(x, y)) \right\}, \{\} \end{array} \right\rangle$$

$$\overset{*}{\rightsquigarrow}^{s}\ \overset{*}{\rightsquigarrow}^{d} \left\langle \begin{array}{l} \left\{ \mathsf{plus}(y, \mathsf{plus}(x_1, \mathsf{times}(x_1, y))) \doteq \mathsf{plus}(x_1, \mathsf{plus}(y, \mathsf{times}(x_1, y))) \right\} \\ \left\{ \mathsf{times}(x, \mathsf{s}(y)) \to \mathsf{plus}(x, \mathsf{times}(x, y)) \right\}, \{\} \end{array} \right\rangle$$

$$\rightsquigarrow^{l} \rightsquigarrow^{sc} \left\langle \begin{array}{l} \left\{ \begin{array}{l} \mathsf{plus}(x_1, \mathsf{plus}(y, \mathsf{times}(x_1, y))) \doteq \mathsf{plus}(x_1, \mathsf{plus}(y, \mathsf{times}(x_1, y))) \\ \mathsf{plus}(y, \mathsf{plus}(x_1, z))) \doteq \mathsf{plus}(x_1, \mathsf{plus}(y, z)) \end{array} \right\} \\ \left\{ \mathsf{times}(x, \mathsf{s}(y)) \to \mathsf{plus}(x, \mathsf{times}(x, y)) \right\}, \{\} \end{array} \right\rangle$$

$$\overset{*}{\rightsquigarrow} \quad \left\langle \begin{array}{l} \{\}, \left\{ \begin{array}{l} \mathsf{s}(\mathsf{s}(\mathsf{plus}(x_3, \mathsf{s}(z)))) \to \mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{plus}(x_3, z)))) \\ \mathsf{times}(x, \mathsf{s}(y)) \to \mathsf{plus}(x, \mathsf{times}(x, y)) \end{array} \right\} \\ \left\{ \begin{array}{l} \mathsf{s}(\mathsf{plus}(x_2, \mathsf{s}(\mathsf{plus}(y_2, z)))) \doteq \mathsf{s}(\mathsf{plus}(y_2, \mathsf{s}(\mathsf{plus}(x_2, z)))) \\ \mathsf{plus}(y, \mathsf{plus}(x_1, z)) \doteq \mathsf{plus}(x_1, \mathsf{plus}(y, z)) \end{array} \right\} \end{array} \right\rangle$$

In the derivation, the generalization from the equation $\mathsf{plus}(y, \mathsf{plus}(x_1, \mathsf{times}(x_1, y))) \doteq \mathsf{plus}(x_1, \mathsf{plus}(y, \mathsf{times}(x_1, y)))$ to $\mathsf{plus}(y, \mathsf{plus}(x_1, z))) \doteq \mathsf{plus}(x_1, \mathsf{plus}(y, z))$ is obtained by the sound generalization.

6 Disproving Incorrect Conjectures

Rewriting induction with inference rules for disproving incorrect conjectures has been introduced in [6, 9]. Usefulness of a mechanism for detecting incorrect conjectures is clear.

Our system BRIL is extended to the system BRILD with inference rules for disproving incorrect conjectures as in **Fig.4**.

Decompose

$$\frac{\langle E \uplus \{f(s_1,\ldots,s_n) \doteq f(t_1,\ldots,t_n)\},\ H,\ K\rangle}{\langle E \cup \{s_i \doteq t_i \mid 1 \le i \le n\},\ H,\ K\rangle}\ f \in \mathcal{C}$$

Disproof1

$$\frac{\langle E \uplus \{s \doteq x\},\ H,\ K\rangle}{\bot}\ x \in V \setminus V(s)$$

Disproof2

$$\frac{\langle E \uplus \{f(s_1,\ldots,s_n) \doteq x\},\ H,\ K\rangle}{\bot}\ f \in \mathcal{C}, x \in V$$

Disproof3

$$\frac{\langle E \uplus \{f(s_1,\ldots,s_m) \doteq g(t_1,\ldots,t_n)\},\ H,\ K\rangle}{\bot}\ f \neq g, f, g \in \mathcal{C}$$

Fig. 4. Additional inference rules of BRILD

The next lemma follows easily from the definition.

Lemma 1. *Let $\mathcal{R}$ be a quasi-reducible TRS. Then (1) $\mathcal{R} \vdash_{ind} s \doteq t$ implies $\mathcal{R} \vdash_{ind} \mathrm{Expd}_u(s,t)$ for any $u \in \mathcal{B}(s)$. (2) $\mathcal{R} \vdash_{ind} s \doteq t$ implies $\mathcal{R} \vdash_{ind} \mathrm{Expd2}_{u,v}(s,t)$ for any $u \in \mathcal{B}(s)$ and $v \in \mathcal{B}(t)$.*

Theorem 3 (soundness of BRILD). *Let $\mathcal{R}$ be a quasi-reducible confluent TRS, E a set of equations, $\succsim$ a ground-total reduction quasi-order satisfying $\mathcal{R} \subseteq \succ$. (1) $\langle E, \emptyset, \emptyset\rangle \overset{*}{\rightsquigarrow}_{\mathsf{BRILD}} \langle \emptyset, H, K\rangle$ for some H, K, then $\mathcal{R} \vdash_{ind} E$. (2) $\langle E, \emptyset, \emptyset\rangle \overset{*}{\rightsquigarrow}_{\mathsf{BRILD}} \bot$ then $\mathcal{R} \nvdash_{ind} E$.*

Proof. (1) Since $\mathcal{R}$ is confluent, $\mathcal{R} \vdash_{ind} f(s_1,\ldots,s_n) \doteq f(t_1,\ldots,t_n)$ iff $\mathcal{R} \vdash_{ind} s_i \doteq t_i$ for all $1 \le i \le n$. Thus, any successful derivation can be modified in such a way that all inferences of *Decompose* rule are replaced by those of *Lemma* rule and *Simplify-C* rule. Then the claim follows from **Theorem 2**. (2) It can be shown by induction on the length of $\langle E, H, K\rangle \overset{*}{\rightsquigarrow}_{\mathsf{BRILD}} \bot$ that $\mathcal{R} \vdash_{ind} H \cup K$ implies $\mathcal{R} \nvdash_{ind} E$, using **Lemma 1**. □

7 Expanding Quasi-Basic Subterms

In this section, we present a small modification of our system which turned out to be useful for proving some additional theorems in our preliminary experiment.

Definition 1 (quasi-basic). *A term u is quasi-basic w.r.t. a TRS $\mathcal{R}$ if (1)* $\mathrm{root}(u) \in \mathcal{D}$ *and (2) for any $l \to r \in \mathcal{R}$ such that l is basic,* $\mathrm{cap}(u)$ *is unifiable with l then there exists $\sigma = \mathrm{mgu}(\mathrm{cap}(u), l)$ such that σ does not instantiate any variable in $V(\mathrm{cap}(u)) \setminus V(u)$. Here,* $\mathrm{cap}(f(u_1, \ldots, u_n))$ *($f \in \mathcal{D}$) is a term $f(\tilde{u}_1, \ldots, \tilde{u}_n)$ where $\tilde{u}_i$ is obtained from u_i by replacing maximal subterms with defined root symbol by fresh variables. The set of quasi-basic subterms of s is denoted by $\mathcal{QB}(s)$.*

Example 5 (quasi-basic). Quasi-basic subterms of $\mathsf{plus}(x, \mathsf{plus}(y, z))$ (w.r.t. the TRS $\mathcal{R}$ in *Example 1*) are $\mathsf{plus}(y, z)$ and $\mathsf{plus}(x, \mathsf{plus}(y, z))$.

We now extend the notions of $\mathrm{Expd}_u(s,t)$ and $\mathrm{Expd2}_{u,v}(s,t)$ for basic subterms u, v to those for quasi-basic subterms u, v. For $u \in \mathcal{QB}(s)$ and $v \in \mathcal{QB}(t)$, $\mathrm{Expd}_u(s,t)$ and $\mathrm{Expd2}_{u,v}(s,t)$ are defined like this:

$$\begin{aligned}\mathrm{Expd}_u(s,t) &= \{C[r]\sigma \doteq t\sigma \mid s \equiv C[u], \sigma = \mathrm{mgu}(u,l), l \to r \in \mathcal{R}, l{:}\mathrm{basic}\}\\ \mathrm{Expd2}_{u,v}(s,t) &= \bigcup \left\{\mathrm{Expd}_{v\sigma}(t\sigma, s') \mid s' \doteq t\sigma \in \mathrm{Expd}_u(s,t)\right\}.\end{aligned}$$

This definition generalizes those for basic terms. From the definition of quasi-basic term it follows that if v is a quasi-basic term and l is a basic lhs of a rewrite rule unifiable with v and $\sigma = \mathrm{mgu}(v, l)$ then $x\sigma \in \mathrm{T}(\mathcal{C}, V)$ for any $x \in \mathrm{dom}(\sigma) \cap V(v)$. Then the well-definedness of $\mathrm{Expd2}_{u,v}(s,t)$ follows from the next lemma.

Lemma 2. *Let $v \in \mathcal{QB}(t)$ and σ a constructor substitution. Then $v\sigma \in \mathcal{QB}(t\sigma)$.*

Since $\mathcal{B}(s) \subseteq \mathcal{QB}(s)$ for any term s, the following rules are more flexible than *Expand* and *Expand2* rules and sometimes more useful to prove theorems.

Expand′

$$\frac{\langle E \uplus \{s \doteq t\},\ H,\ K\rangle}{\langle E \cup \mathrm{Expd}_u(s,t),\ H \cup \{s \to t\},\ K\rangle}\ u \in \mathcal{QB}(s), s \succ t$$

Expand2′

$$\frac{\langle E \uplus \{s \doteq t\},\ H,\ K\rangle}{\langle E \cup \mathrm{Expd2}_{u,v}(s,t),\ H,\ K \cup \{s \doteq t\}\rangle}\ u \in \mathcal{QB}(s), v \in \mathcal{QB}(t),\ s \not\succ t \wedge t \not\succ s$$

The systems obtain by replacing *Expand* and *Expand2* rules in BRI/BRIL/BRILD by *Expand′* and *Expand2′* rules are denoted by BRI′/BRIL′/BRILD′. The soundness of these systems are obtained by using the following lemma.

Lemma 3. *Let $\mathcal{R}$ be a quasi-reducible TRS and u a quasi-basic term. For any ground constructor substitution σ_g, $u\sigma_g$ is reducible.*

8 Related Works

In this section, we compare our inference system and other closely related rewriting induction provers SPIKE and NICE.

SPIKE[2] is a well-known rewriting induction (*test set induction*) prover. The scope of SPIKE is much broader than ours—it can handle not only equational theories but conditional ones, conjectures can be given not only in equations but also in clauses. A disproving mechanism is also included in the system. The mechanisms of SPIKE for proving non-orientable theorem includes the ordered rewriting and relaxed rewriting [6, 9, 10]. Apart from the original version, recently the new version[3] has became available. The new version of SPIKE is based on 'Descente Infinie' induction [5, 29] and an extended mechanism for dealing with non-orientable theorems has been incorporated [30]. We denote by SPIKE/B and SPIKE/S for the original version and new version of SPIKE respectively.

The most notable difference between SPIKE and BRILD′ is the capability of non-orientable theorems by *Expand2* rule. The inference rule of simplification by conjecture (*Simplify-C*) is essentially from [2] and we refer [2] for the comparison to simplification rules of SPIKE. The inference rule for disproving incorrect conjectures of SPIKE is as follows ([9, 10]):

$$\frac{\langle E \cup \{C\}, H\rangle}{\bot}\ C : \text{quasi-inconsistent}$$

We refer [10] for the definition of quasi-inconsistency (which is too complex to present here). The *Decompose* inference rule is from [10] and the *Disproof1-3* inference rules are from [28]. Our inference rules for disproving incorrect conjectures are simplified by making use of the fact that the underlying TRSs are limited to quasi-reducible TRSs.

NICE[4] is a rewriting induction prover that incorporates two extensions for monomorphic TRSs—namely, term partition techniques [32] and a sound generalization technique [33]. The proofs by rewriting induction with these two new mechanisms run independently. We denote by $\mathsf{NICE_P}$ and $\mathsf{NICE_G}$ the proof with the term partition and the proof with the sound generalization. We also note that NICE is capable of conditional theories. NICE does not have a special mechanism for proving non-orientable theorems but the underlying rewriting is performed by the ordered rewriting. It does not incorpolates mechanisms for disproving incorrect conjectures nor simplification by conjectures.

In **Tab. 1** we list the result of an experiment performed by SPIKE, NICE, and our preliminary implementation of BRILD′. Our implementation incorporates the sound generalization same as $\mathsf{NICE_G}$ to add lemmas automatically. A check in each column indicates the success of proof; all successful proofs of BRILD′ are performed at most 7 (expansion) steps. Some additional conjectures needed to prove $\mathsf{times}(x, y) \doteq \mathsf{times}(y, x)$ and $\mathsf{sum}(\mathsf{app}(xs, ys)) \doteq \mathsf{sum}(\mathsf{app}(ys, xs))$ in the system iRI [1] turn out to be unnecessary in BRILD′. The system BRILD′ can prove non-orientable inductive theorems which have not been capable in other rewriting induction provers. We have also tested equational examples from Dream Cor-

[2] http://www.loria.fr/equipes/cassis/softwares/spike/
[3] http://lita.sciences.univ-metz.fr/~stratula/
[4] http://www-sop.inria.fr/coprin/urso/

(many-sorted) conjectures	SPIKE/B	$\mathrm{NICE_P}$	$\mathrm{NICE_G}$	SPIKE/S	BRILD′
$\mathsf{plus}(x,y) \doteq \mathsf{plus}(y,x)$	√	×	√	√	√
$\mathsf{plus}(x,\mathsf{plus}(y,z)) \doteq \mathsf{plus}(y,\mathsf{plus}(x,z))$	√	×	√	√	√
$\mathsf{times}(x,\mathsf{s}(y)) \doteq \mathsf{plus}(x,\mathsf{times}(x,y))$	×	×	×	×	√
$\mathsf{times}(x,\mathsf{plus}(y,z)) \doteq \mathsf{plus}(\mathsf{times}(x,y),\mathsf{times}(x,z))$	×	×	×	×	×
$\mathsf{plus}(x,y) \doteq \mathsf{plus}(y,x)$, $\mathsf{times}(x,\mathsf{plus}(y,z)) \doteq \mathsf{plus}(\mathsf{times}(x,y),\mathsf{times}(x,z))$	×	×	×	×	√
$\mathsf{times}(x,y) \doteq \mathsf{times}(y,x)$	×	×	×	×	√
$\mathsf{sum}(\mathsf{app}(xs,ys)) \doteq \mathsf{sum}(\mathsf{app}(ys,xs))$	×	×	×	×	√

Table 1. Comparison of BRILD′ and rewriting induction provers

pus[5]; our implementation proves 51 out of 69 examples with 1 sec. timeout for each, while the RI proves 33 examples. In the Appendix, we present a proof of $\{\mathsf{plus}(x,y) \doteq \mathsf{plus}(y,x),\ \mathsf{times}(x,\mathsf{plus}(y,z)) \doteq \mathsf{plus}(\mathsf{times}(x,y),\mathsf{times}(x,z))\}$ scripted from the output of our preliminary implementation. It can be seen that the proof uses the *Expand2* rule many times.

9 Conclusion

We have presented a rewriting induction system BRILD′ with an increased capability of proving non-orientable theorems and that of disproving incorrect theorems. The system is intended to amenable for automation and a part of the system is implemented. Soundness of the system for proving and disproving are shown. It was demonstrated through some examples that our inference system enables simple rewriting induction proofs for some theorems which have not been provable in known rewriting induction provers. A comparison with other rewriting induction provers shows that our approach is useful to enlarge the scope of inductive theorems which can be proved automatically based on rewriting induction. Further implementation and experiments remain as our future work.

Acknowledgments

Thanks are due to anonymous referees for comments and suggestions. This work was partially supported by a grant from JSPS, No. 20500002.

References

1. T. Aoto. Dealing with non-orientable equations in rewriting induction. In *Proc. of the 17th International Conference on Rewriting Techniques and Applications*, volume 4098 of *LNCS*, pages 242–256. Springer-Verlag, 2006.

[5] `http://dream.inf.ed.ac.uk/dc/lib.html`

2. T. Aoto. Soundness of rewriting induction based on an abstract principle. *IPSJ Transactions on Programming*, 49(SIG 1 (PRO 35)):28–38, 2008.
3. F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
4. L. Bachmair, N. Dershowitz, and D. A. Plaisted. Completion without failure. In *Resolution of Equations in Algebraic Structures*, volume 2, pages 1–30. Academic Press, 1989.
5. G. Barthe and S. Stratulat. Validation of the JavaCard Platform with implicit induction techniques. In *Proc. of the 14th International Conference on Rewriting Techniques and Applications*, volume 2706 of *LNCS*, pages 337–351. Springer-Verlag, 2003.
6. A. Bouhoula. Automated theorem proving by test set induction. *Journal of Symbolic Computation*, 23:47–77, 1997.
7. A. Bouhoula and F. Jacquemard. Automated induction with constrained tree automata. In *Proc. of the 4th International Joint Conference on Automated Reasoning*. Springer-Verlag, to appear.
8. A. Bouhoula and J.-P. Jouannaud. Automata-driven automated induction. *Information and Computation*, 169(1):1–22, 2001.
9. A. Bouhoula, E. Kounalis, and M. Rusinowitch. Automated mathematical induction. *Journal of Logic and Computation*, 5(5):631–668, 1995.
10. A. Bouhoula and M. Rusinowitch. Implicit induction in conditional theories. *Journal of Automated Reasoning*, 14:189–235, 1995.
11. R. Bronsard, U. S. Reddy, and R. W. Hasker. Induction using term orders. *Journal of Automated Reasoning*, 16:3–37, 1996.
12. A. Bundy, D. Basin, D. Hutter, and A. Ireland. *Rippling: Meta-Level Guidance for Mathematical Reasoning*. Cambridge University Press, 2005.
13. N. Dershowitz and U. S. Reddy. Deductive and inductive synthesis of equational programs. *Journal of Symbolic Computation*, 15:467–494, 1993.
14. S. Falke and D. Kapur. Inductive decidability using implicit induction. In *Proc. of the 13th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, volume 4246 of *LNAI*, pages 45–59. Springer-Verlag, 2006.
15. B. Gramlich. Strategic issues, problems and challenges in inductive theorem proving. *Electronic Notes in Theoretical Computer Science*, 125:5–43, 2005.
16. G. Huet and J.-M. Hullot. Proof by induction in equational theories with constructors. *Journal of Computer and System Sciences*, 25(2):239–266, 1982.
17. D. Hutter and C. Sengler. INKA: The next generation. In *Proc. of the 13th International Conference on Automated Deduction*, pages 288–292, 1996.
18. J.-P. Jouannaud and E. Kounalis. Automatic proofs by induction in theories without constructors. *Information and Computation*, 82:1–33, 1989.
19. D. Kapur. Constructors can be partial, too. In *Automated reasoning and its applications: essays in honor of Larry Wos*, pages 177–210. MIT Press, 1997.
20. D. Kapur, J. Giesl, and M. Subramaniam. Induction and decision procedures. *Revista de la real academia de ciencas (RACSAM) Serie A: Matematicas*, 98(1):154–180, 2004.
21. D. Kapur, P. Narendran, and H. Zhang. Automating inductionless induction using test sets. *Journal of Symbolic Computation*, 11(1–2):81–111, 1991.
22. D. Kapur and M. Subramaniam. Lemma discovery in automating induction. In *Proc. of the 13th International Conference on Automated Deduction*, volume 1104 of *LNCS*, pages 538–552. Springer-Verlag, 1996.
23. M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, 2000.

24. H. Koike and Y. Toyama. Inductionless induction and rewriting induction. *Computer Software*, 17(6):1–12, 2000. In Japanese.
25. D. R. Musser. On proving inductive properties of abstract data types. In *Proc. of the 7th Annual ACM Symposium on Principles of Programming Languages*, pages 154–162. ACM Press, 1980.
26. U. S. Reddy. Term rewriting induction. In *Proc. of the 10th International Conference on Automated Deduction*, volume 449 of *LNAI*, pages 162–177. Springer-Verlag, 1990.
27. N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Prover Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, Sept. 1999.
28. S. Shimazu, T. Aoto, and Y. Toyama. Automated lemma generation for rewriting induction with disproof. In *The 8th JSSST Workshop on Programming and Programming Languages*, pages 75–89, 2006. In Japanese.
29. S. Stratulat. A general framework to build contextual cover set induction provers. *Journal of Symbolic Computation*, 32:403–445, 2001.
30. S. Stratulat. Combining rewriting with explicit induction to reason on non-orientable equalities. In *Proc. of the 19th International Conference on Rewriting Techniques and Applications*, LNCS, to appear.
31. Y. Toyama. How to prove equivalence of term rewriting systems without induction. *Theoretical Computer Science*, 90(2):369–390, 1991.
32. P. Urso and E. Kounalis. Term partition for mathematical induction. In *Proc. of the 14th International Conference on Rewriting Techniques and Applications*, volume 2706 of *LNCS*, pages 352–366, 2003.
33. P. Urso and E. Kounalis. Sound generalizations in mathematical induction. *Theoretical Computer Science*, 323:443–471, 2004.
34. T. Walsh. A divergence critic for inductive proof. *Journal of Artificial Intelligence Research*, 4:209–235, 1996.

A A Proof for $\{+(x,y) \doteq +(y,x), *(x,+(y,z)) \doteq +(*(x,y),*(x,z))\}$

```
SPECIFICATION:
   [ Nat ]
   [ * : Nat*Nat=>Nat,
     + : Nat*Nat=>Nat,
     s : Nat=>Nat,
     0 : Nat ]
   [ +(0,y) -> y,
     +(s(x),y) -> s(+(x,y)),
     *(0,j) -> 0,
     *(s(i),j) -> +(j,*(i,j)) ]
   (Monomorphic)
      reflective positions: s/1
      downward positions: +/2
      upward positions: +/1
      down-contextual positions:
      up-contextual positions: */1
   [ +(y,x) = +(x,y),
     *(x,+(y,z)) = +(*(x,y),*(x,z)) ]
Start the rewriting induction with
   [ +(y,x) = +(x,y),
     *(x,+(y,z)) = +(*(x,y),*(x,z)) ]
   [ +(0,y) -> y,
     +(s(x),y) -> s(+(x,y)),
```

```
     *(0,j) -> 0,
     *(s(i),j) -> +(j,*(i,j)) ]
Expand2 the equation +(y,x) = +(x,y) at e(L) and e(R).
   [ 0 = 0,
     s(+(x2,0)) = s(x2),
     s(x) = s(+(x,0)),
     s(+(x2,s(x))) = s(+(x,s(x2))) ]
Simplify & Delete.
es:
   [ *(x,+(y,z)) = +(*(x,y),*(x,z)) ]
hs:
   [ ]
ks:
   [ +(y,x) = +(x,y) ]
Expand the L of the equation *(x,+(y,z)) = +(*(x,y),*(x,z)) at e
   [ 0 = +(*(0,y),*(0,z)),
     +(+(y,z),*(i4,+(y,z))) = +(*(s(i4),y),*(s(i4),z)) ]
Try SG to +(+(y,z),+(*(i4,y),*(i4,z))) = +(+(y,*(i4,y)),+(z,*(i4,z)))
Adding New Lemma (SG): +(+(x5,z),+(*(i4,y),x6)) = +(+(x5,*(i4,y)),+(z,x6))
Simplify & Delete.
es:
   [ +(+(x5,z),+(*(i4,y),x6)) = +(+(x5,*(i4,y)),+(z,x6)) ]
hs:
   [ *(x,+(y,z)) -> +(*(x,y),*(x,z)) ]
ks:
   [ +(y,x) = +(x,y) ]
Expand2 the equation +(+(x5,z),+(*(i4,y),x6)) = +(+(x5,*(i4,y)),+(z,x6)) at 1(L) and 1(R).
   [ +(y7,+(*(i4,y),x6)) = +(*(i4,y),+(y7,x6)),
     +(s(+(x10,y10)),+(*(i4,y),x6)) = +(s(+(x10,*(i4,y))),+(y10,x6)) ]
Simplify & Delete.
es:
   [ +(y7,+(*(i4,y),x6)) = +(*(i4,y),+(y7,x6)) ]
hs:
   [ *(x,+(y,z)) -> +(*(x,y),*(x,z)) ]
ks:
   [ +(+(x5,z),+(*(i4,y),x6)) = +(+(x5,*(i4,y)),+(z,x6)),
     +(y,x) = +(x,y) ]
Expand2 the equation +(y7,+(*(i4,y),x6)) = +(*(i4,y),+(y7,x6)) at e(L) and 1(R).
   [ +(*(0,j4),x) = +(0,+(0,x)),
     s(+(x6,+(*(0,j4),x))) = +(0,+(s(x6),x)),
     +(*(s(i5),j5),x) = +(+(j5,*(i5,j5)),+(0,x)),
     s(+(x7,+(*(s(i5),j5),x))) = +(+(j5,*(i5,j5)),+(s(x7),x)) ]
Simplify & Delete.
es:
   [ +(x7,+(+(j5,*(i5,j5)),x)) = +(+(x,x7),+(j5,*(i5,j5))) ]
hs:
   [ *(x,+(y,z)) -> +(*(x,y),*(x,z)) ]
ks:
   [ +(y7,+(*(i4,y),x6)) = +(*(i4,y),+(y7,x6)),
     +(+(x5,z),+(*(i4,y),x6)) = +(+(x5,*(i4,y)),+(z,x6)),
     +(y,x) = +(x,y) ]
Expand2 the equation +(x7,+(+(j5,*(i5,j5)),x)) = +(+(x,x7),+(j5,*(i5,j5)))
   at e(L) and 1(R).
   [ +(+(j,*(i,j)),0) = +(0,+(j,*(i,j))),
     s(+(x5,+(+(j,*(i,j)),0))) = +(s(x5),+(j,*(i,j))),
     +(+(j,*(i,j)),s(x4)) = +(s(+(x4,0)),+(j,*(i,j))),
     s(+(x6,+(+(j,*(i,j)),s(x4)))) = +(s(+(x4,s(x6))),+(j,*(i,j))) ]
Simplify & Delete.
es:
   [ ]
hs:
   [ *(x,+(y,z)) -> +(*(x,y),*(x,z)) ]
ks:
   [ +(x7,+(+(j5,*(i5,j5)),x)) = +(+(x,x7),+(j5,*(i5,j5))),
     +(y7,+(*(i4,y),x6)) = +(*(i4,y),+(y7,x6)),
     +(+(x5,z),+(*(i4,y),x6)) = +(+(x5,*(i4,y)),+(z,x6)),
     +(y,x) = +(x,y) ]
Success
```

An Automated Tableau Theorem Prover for FO(ID)

Stephen Bond and Marc Denecker

Katholieke Universiteit Leuven, Belgium

Abstract. Inductive definitions are frequently encountered in software, underlying many common program and algorithm components, such as recursive functions and loops. Therefore, in disciplines such as program verification or specification extraction, it is important to be able to represent and reason with inductive definitions in a formal way. Ideally our formal representation language would extend classical logic and take advantage of the powerful symbolic proof systems that exist for it. FO(ID) is a language that extends classical logic with inductive definitions, which are captured by the well-founded semantics of logic programming. In this paper we present an automated tableau theorem prover for FO(ID), which has the potential to be of much use in the application of symbolic proof techniques to software science. We describe the tableau rules and their implementation, and discuss some possible extensions and applications of the system.

1 Introduction

Inductive definitions (also known as recursive definitions) are a fundamental construct in software science. It is well known, for example, that all iterative loops may be expressed as inductive definitions, and many functional and logical programming languages eschew explicit iteration altogether in favour of recursive calls. To this end, a formal means of representing and reasoning with inductive definitions is highly desirable in any discipline that concerns the formal treatment of computer programs and algorithms, such as verification or specification extraction.

The most well-established and extensively studied formal language for knowledge representation and reasoning is classical first-order logic; the maturity and power of the many proof techniques and solvers developed for this language make it a desirable choice for use in formal software methods. However, first-order logic lacks the ability to represent inductive definitions.

Example 1. As a running example, consider the transitive closure of a graph, defined by the following simple induction

– if the edge (x, y) is in the graph, then it is in the transitive closure of the graph;

– if for a pair of vertices (x, y), there is a vertex z such that (x, z) and (z, y) are in the transitive closure of the graph, then (x, y) is in the transitive closure of the graph.

It is well known that this definition is impossible to express in first-order logic.

It was shown in [1] that the notion of 'inductive definition' is formally captured by the well-founded semantics of logic programming [10]. The language FO(ID), the first-order variant of ID-logic ([2]) uses the well-founded semantics to extend classical first-order logic with inductive definitions. It allows various inductive definitions (such as the one above) to be represented uniformly, and in a more intuitive manner than, say, fixpoint logics. It has also been shown to be a useful language for general knowledge representation, allowing a concise expression of classic KR formalisms such as the situation calculus ([3]).

Example 2. The following is the transitive closure relation from Example 1 as expressed in FO(ID). Here the predicate T denotes the set of pairs of vertices in the transitive closure of the graph, while G denotes the set of edges. The correspondence between the FO(ID) rules and the natural languages rules above should be intuitive and straightforward.

$$\left\{ \begin{array}{l} \forall x, y\ T(x, y) \leftarrow G(x, y). \\ \forall x, y\ T(x, y) \leftarrow \exists z\ T(x, z) \wedge T(z, y). \end{array} \right\}$$

Automated symbolic reasoning is an important research topic for any formal language. Reasoning on a symbolic level brings greater flexibility to an automated solver: a problem can be isolated from its particular instances, and inconsistency and other properties of a theory can be detected independently of a domain. In the software verification setting, automated symbolic reasoning is particularly important: when exhaustive model checking is not feasible, symbolic inference needs to be applied.

Existing reasoning systems for FO(ID) are either not automatable ([7]) or domain dependent ([8]), the former requiring guesswork on the part of a human reasoner, and the latter dependent on knowing the precise domain of a given problem. In this paper we present an incomplete automated symbolic reasoning system for a useful subset of FO(ID), based on the Tableau method ([5]). While incomplete, this system has proved successful in reasoning with many standard example problems, and provides a useful first step towards an automated symbolic reasoning system for FO(ID), which could be used in verification problems.

The outline of this paper is as follows. In the next section we introduce FO(ID) and define its syntax and semantics. Following that, we define a Tableau Calculus for FO(ID). Then we describe the implementation of this Tableau Calculus and prove the soundness of our implementation of the FO(ID)-specific rules. Finally we offer some concluding remarks.

2 Preliminaries

In this section we introduce FO(ID), define its syntax and semantics and give some examples.

ID-logic ([2]) is the name of a family of formal representation languages that consist of classical logic extended with inductive definitions. The particular branch of ID-logic we are concerned with in this paper is FO(ID): first-order logic extended with inductive definitions. In FO(ID), an inductive definition Δ (which from now on we refer to simply as a 'definition') is a set of rules of the form

$$\forall \boldsymbol{x}\ p(\boldsymbol{x}) \leftarrow \phi$$

where ϕ is a first-order formula and $\leftarrow$ is a symbol denoting *definitional implication*, which we distinguish from the *material implication* $\Rightarrow$ of first-order logic [1]. We refer to $p(\boldsymbol{x})$ as the *head* of the rule r, denoted $head(r)$, and to the formula ϕ as its *body*, denoted $body(r)$. We call the set of predicates that appear in the heads of the rules of Δ the *defined predicates* of Δ, which we denote by $Def(\Delta)$. All other symbols in the definition are called *open*; the set of open symbols is denoted by $Op(\Delta)$. The purpose of Δ is to define the predicates $Def(\Delta)$ in terms of the symbols $Op(\Delta)$.

Example 3. The FO(ID) definition Δ from Example 2 defines the transitive closure relation of a graph in terms of its edge relation.

$$\left\{ \begin{array}{l} \forall x, y\ T(x,y) \leftarrow G(x,y). \\ \forall x, y\ T(x,y) \leftarrow \exists z\ T(x,z) \wedge T(z,y). \end{array} \right\}$$

Here, $Def(\Delta) = \{T/2\}$, where $T/2$ is a predicate denoting the set of pairs of vertices of the graph that are in its transitive closure relation. $Op(\Delta) = \{G/2\}$, where $G/2$ is a predicate denoting the set of pairs of vertices that have a directed edge between them. The definition states that vertex y is reachable from vertex x if there is a directed edge from x to y, or if there is a vertex z such that z is reachable from x and y is reachable from z.

A theory in FO(ID) consists of a set of definitions and a set of assertions. Assertions are ordinary sentences of first-order logic, defined in the usual way. The meaning of these first-order sentences is standard; to define the semantics of FO(ID), we therefore only need to define the semantics of a definition.

We first introduce some semantical concepts. A *two-valued interpretation* I of a vocabulary Σ consists of a domain D, a mapping from function symbols f/n to n-ary functions on D, and a mapping of pairs $(P, \boldsymbol{d})$ of predicate symbols P/n and n-tuples $\boldsymbol{d} \subseteq D$ to the truth-values $\{\mathbf{t}, \mathbf{f}\}$. We also denote such a pair $(P, \boldsymbol{d})$ as $P(\boldsymbol{d})$ and refer to it as a *domain atom*. A *three-valued* interpretation ν is the same as a two-valued one, except that it maps domain atoms $P(\boldsymbol{d})$ to the

[1] A definitional rule can be seen as a general logic program clause.

truth values $\{\mathbf{t}, \mathbf{f}, \mathbf{u}\}$. Such a ν assigns a truth value to each logical atom $P(\boldsymbol{c})$, namely $\nu(P(c_1^\nu, \ldots, c_n^\nu))$. Using the standard Kleene truth tables for the logical connectives, this assignment can be extended to an assignment $\nu(\phi)$ of a truth value to each formula ϕ.

Let $H(\Delta, D)$ be the set of all defined domain atoms of the definition Δ in D, i.e. the set of all domain atoms of the predicates in $Def(\Delta)$ in D. We define $U(\Delta, \nu) \subseteq H(\Delta, D)$ to be the *unfounded set* of Δ with respect to a three-valued interpretation ν if each atom $P(\boldsymbol{d}) \in U(\Delta, \nu)$ satisfies the following: For each instantiated rule $r[\boldsymbol{d}]$ in Δ such that $P(\boldsymbol{d}) = head(r[\boldsymbol{d}])$, either:

- $\nu(body(r[\boldsymbol{d}])) = \mathbf{f}$
- an element of $U(\Delta, \nu)$ positively occurs in $body(r[\boldsymbol{d}])$.

If A is a set of domain atoms and $\mathbf{v}$ a truth-value, we will denote by $\nu[A/\mathbf{v}]$ the three-valued interpretation that assigns $\mathbf{v}$ to each domain atom in A and coincides with ν on all other atoms.

We now define when an interpretation I is a model of a definition Δ. Since Δ defines the predicates $Def(\Delta)$ in terms of the symbols $Op(\Delta)$, we should assume the interpretation of $Op(\Delta)$ as given and try to construct a corresponding interpretation for $Def(\Delta)$. Let O be the restriction $R(I, Op(\Delta))$ of I to the open symbols.

We are now going to construct a sequence of three-valued interpretations $(\nu_\alpha)_{0 \leq \alpha \leq \beta}$, each of which extends O; the limit of this sequence will tell us how to interpret $Def(\Delta)$.

- ν_0 assigns $O(P(\boldsymbol{d}))$ to $P(\boldsymbol{d})$ if $P \in Op(\Delta)$ and u if $P \in Def(\Delta)$;
- ν_{i+1} is related to ν_i in one of two ways:
  1. $\nu_{i+1} = \nu_i[P(\boldsymbol{d})/\mathbf{t}]$, such that Δ contains a rule $\forall \boldsymbol{x}\ P(\boldsymbol{x}) \leftarrow \phi(\boldsymbol{x})$ with $\nu_i(\phi[\boldsymbol{d}]) = \mathbf{t}$
  2. $\nu_{i+1} = \nu_i[U(\Delta, \nu_i)/\mathbf{f}]$, where $U(\Delta, \nu_i)$ is an unfounded set of Δ with respect to ν_i as defined above.
- For each limit ordinal λ, ν_λ is the *lub* w.r.t. $\leq_p$ of all ν_δ for which $\delta < \lambda$.

We call such a sequence a *well-founded induction* of Δ in O. Each such sequence eventually reaches a limit ν_β. It was shown in [4] that all sequences reach the same limit. It is now this ν_β that tell us how to interpret the defined predicates. To be more precise, we define that:

$$I \models \Delta \text{ if and only if } R(I, Def(\Delta)) = R(\nu_\beta, Def(\Delta)),$$

where $R(\nu_\beta, Def(\Delta))$ is the restriction of ν_β to the symbols $Def(\Delta)$. Note that if some domain atom is still assigned u by ν_β, the definition has no models extending O. Intuitively, this means that, for this particular interpretation of its open symbols, Δ does not manage to unambiguously define its defined predicates, due to some non well-founded use of negation.

Example 4. Consider the following FO(ID) theory containing the definition Δ:

$$\left\{\begin{array}{l}\forall x, y\ T(x,y) \leftarrow G(x,y).\\ \forall x, y\ T(x,y) \leftarrow \exists z\ T(x,z) \wedge T(z,y).\end{array}\right\}$$
$$\neg T(a,c).$$

Given $D = \{a, b, c\}$ and an interpretation I mapping $G(a,b)$ and $G(b,c)$ to $\mathbf{t}$ and all other open atoms to $\mathbf{f}$, and $T(a,b)$ to $\mathbf{f}$ and all other defined atoms to u, then O is the interpretation mapping $G(a,b)$ and $G(b,c)$ to $\mathbf{t}$ and all other open atoms to $\mathbf{f}$. Then the calculation of ν_β proceeds as follows:

- ν_0 assigns $G(a,b)$ and $G(b,c)$ to $\mathbf{t}$, all other open atoms to $\mathbf{f}$, and all defined atoms to u.
- ν_1 assigns $T(a,b)$ and $T(b,c)$ to $\mathbf{t}$, since the bodies of these atoms in the first rule of the definition are true in ν_0. It can be verified that the unfounded set $U = \{T(a,a), T(b,a), T(b,b), T(c,a), T(c,b), T(c,c)\}$; all these atoms are assigned to $\mathbf{f}$.
- ν_2 assigns $T(a,c)$ to $\mathbf{t}$, since the body of this atom in the second rule is true in ν_1. We have now reached a fixed point.

Since $T(a,c)$ is true in the model extending O, but false in I, there is no model extending I. (In fact, since $T(a,c)$ must be false in any model satisfying the constraint $\neg T(a,c)$, this theory is unsatisfiable in the given domain and interpretation of the open predicates.)

3 A Tableau Calculus for FO(ID)

In this section we introduce the tableau calculus and describe a tableau calculus for FO(ID).

3.1 The Tableau Calculus

The tableau calculus [5] is a proof procedure for logical languages, in which a binary tree known as a *tableau* is constructed. The nodes of the tableau are sets of formulas. *Tableau expansion rules* describe the ways in which branches of the tableau may be extended. Given a tableau rule and a branch containing formulas that match the rule prerequisities, the branch can be extended according to the tableau rule.

A branch is *closed* if it contains a contradiction, and *open* otherwise. Tableaux are often used to prove by refutation. Given a theory T and a formula ϕ, to show that $T \models \phi$, we apply the tableau rules on the theory $T \cup \neg\phi$ and show that every branch in the tableau so constructed can be closed. Tableau can also be used in symbolic model generation. For theories in languages for which the tableau method is a complete proof system (such as propositional logic), once all possible tableau rules have been applied, each open branch contains a model

of the theory – in the propositional case, the model is simply the set of literals in the branch. This property holds because the tableau method is essentially an algorithm for converting a theory to disjunctive normal form. At each stage of the tableau procedure, each branch is a conjunction of formulas entailed by the theory, which is equivalent to a disjunction of the branches.

For compactness, we will state our tableau rules using Smullyan's classification of first-order formulas [9]. Under this classification, first-order formulas are of type α, β, γ or δ. Types α and β are conjunctive and disjunctive formulas, respectively. Formulas of these types each have two *components*, defined as follows (where ϕ and ψ are arbitary formulas):

α	α_1	α_2	β	β_1	β_2
$\phi \wedge \psi$	ϕ	ψ	$\phi \vee \psi$	ϕ	ψ
$\neg(\phi \vee \psi)$	$\neg\phi$	$\neg\psi$	$\neg(\phi \wedge \psi)$	$\neg\phi$	$\neg\psi$
$\neg(\phi \Rightarrow \psi)$	ϕ	$\neg\psi$	$\phi \Rightarrow \psi$	$\neg\phi$	ψ

Quantified formulas are of type γ or δ. γ formulas are universally quantified; δ formulas are existentially quantified. γ and δ formulas have *instances* γ_i and δ_i, defined as follows:

γ	γ_i
$\forall x\phi(x)$	$\phi(c)$ (where c is any constant symbol in the language)
$\neg\exists x\phi(x)$	$\neg\phi(c)$ (where c is any constant symbol in the language)

δ	δ_i
$\exists x\phi(x)$	$\phi(k)$ (where k is a new skolem constant)
$\neg\forall x\phi(x)$	$\neg\phi(k)$ (where k is a new skolem constant)

Note that these rules are independent of the domain.

3.2 Tableau Rules for FO(ID)

The Tableau Calculus has proven useful for defining proof systems for expressive knowledge representation languages; we base our tableau expansion rules on a subset of those defined for ASP in ([6]). The list of rules is given in figure 1. Each rule states that if the branch contains the formulas that occur above the line, it may be extended with the formulas that occur below the line. For the β-rule, this involves splitting the tableau into two new branches.

Note that rules (f) and (h) are sound under the well-founded induction sequence defined in the previous section, and that rule (g) is simply the contraposition of rule (h). Rule (f) would also be sound if we concluded that a member p of any unfounded set of the definition were false, but specifying the greatest unfounded set improves the completeness of the expansion rule.

Fig. 1. Tableau Expansion rules for FO(ID)

a) Z-rule:
$$\frac{\neg\neg Z}{Z}$$
$$\frac{\neg \mathbf{t}}{\mathbf{f}}$$
$$\frac{\neg \mathbf{f}}{\mathbf{t}}$$

b) α-rule:
$$\frac{\alpha}{\alpha_1}$$
$$\alpha_2$$

c) β-rule:
$$\frac{\beta}{\beta_1 \mid \beta_2}$$

d) γ-rule:
$$\frac{\gamma}{\gamma_i}$$

e) δ-rule:
$$\frac{\delta}{\delta_i}$$

f) "Well-founded negation" rule:
$$\frac{\Delta}{\neg p}$$
where p is a member of the greatest unfounded set of definition Δ in the branch.

g) "Backward false" rule:
$$\frac{\forall \boldsymbol{x}\ p(\boldsymbol{x}) \leftarrow \phi \qquad \neg p(\boldsymbol{c})}{\neg \phi[\boldsymbol{x} : \boldsymbol{c}]}$$
where $\phi[\boldsymbol{x} : \boldsymbol{c}]$ is the formula ϕ with all occurrences of the variables in $\boldsymbol{x}$ replaced by the constants in $\boldsymbol{c}$.

h) "Forward true" rule:
$$\frac{\forall \boldsymbol{x}\ p(\boldsymbol{x}) \leftarrow \phi(\boldsymbol{x}) \qquad \phi(\boldsymbol{c})}{p([\boldsymbol{x} : \boldsymbol{c}])}$$
where $p([\boldsymbol{x} : \boldsymbol{c}])$ is the domain atom $p(\boldsymbol{x})$ with all occurrences of the variables in $\boldsymbol{x}$ replaced by the constants in $\boldsymbol{c}$.

3.3 Example

Consider the theory T:

$$\left\{\begin{array}{l}\forall x, y\ T(x,y) \leftarrow G(x,y).\\ \forall x, y\ T(x,y) \leftarrow \exists z\ T(x,z) \wedge T(z,y).\end{array}\right\}$$
$$\neg T(a,b).$$

The following is a possible partial tableau for T:

$$\left\{\begin{array}{l}\forall x, y\ T(x,y) \leftarrow G(x,y).\\ \forall x, y\ T(x,y) \leftarrow \exists z\ T(x,z) \wedge T(z,y).\end{array}\right\}$$
$$\neg T(a,b)$$
$$\neg \exists z\ T(a,z) \wedge T(z,b)$$
$$\neg(T(a,a) \wedge T(a,b))$$

$$\neg T(a,a) \quad \neg T(a,b)$$

The root node of the tree contains the theory T. The formula $\neg \exists z\ T(a,z) \wedge T(z,b)$ is then added to the node by applying tableau expansion rule (g) on the second rule of the definition and $\neg T(a,b)$. Rule (d) is then applied on this

formula to extend the node with $\neg(T(a,a) \wedge T(a,b))$. Finally, rule (c) is applied on this formula to produce two branches with $\neg T(a,a)$ and $\neg T(a,b)$.

4 Implementation: A Domain-Independent Tableau Theorem Prover for FO(ID)

It is clear from the rules above that the most difficult part of the implementation of a domain-independent theorem prover for FO(ID) is the automated calculation of the greatest unfounded set of a definition. Unfounded sets was defined in section 2 with respect to a given interpretation in a given domain. In this section we give an alternative definition in terms of a given tableau branch. We then describe an automated procedure for calculating an unfounded set, as the fixed point of a monotone operator.

4.1 A fixed point algorithm for an unfounded set

In this subsection we describe a subset of FO(ID) for which we can define an algorithm for computing an unfounded set of a definition in a given tableau branch, and then describe the algorithm itself in terms of calculating the fixed point of a monotone operator. Since a branch can be extended by application of tableau rules, we need to prove that the algorithm is monotone with respect to branch extensions: any member of the calculated unfounded set of a definition in a branch will also be in the unfounded set of the definition in an extension of the branch.

In the subset of FO(ID) with which we are concerned in this subsection, a definition Δ is a set of rules of the form $\forall \boldsymbol{x}\ p(\boldsymbol{x_1}) \leftarrow \phi(\boldsymbol{x_2})$, $\boldsymbol{x_1} \subseteq \boldsymbol{x}$, $\boldsymbol{x_2} \subseteq \boldsymbol{x}$, where p is a defined predicate, and each $\phi(\boldsymbol{x_2})$ is a conjunction of literals $l_1 \wedge \cdots \wedge l_n$ with free variables in $\boldsymbol{x_2}$. At times we will abuse notation and treat $\phi(\boldsymbol{x_2})$ as a multiset of literals $\{l_1, \ldots, l_n\}$.

A tableau branch B is a set of formulas and definitions; we denote by $C(B)$ the set of all ground terms in branch B.

Informally speaking, our algorithm will begin with a set $H(\Delta, B)$ of "ground" atoms of the defined predicates in Δ in B, and will then iteratively remove members of this set until we reach a fixed point, which will be a representation of an unfounded set. We now concern ourselves with defining a useful set $H(\Delta, B)$.

We could ground the defined predicates of Δ with the Herbrand universe of B, but if there are function symbols in B then this set will be infinitely large, which is obviously not desirable. We must therefore place limits on the function terms we include. In addition, to give us a measure of domain independence, we would like to include variables in $H(\Delta, B)$, so as to enable us to conclude, for example, that $p(\boldsymbol{a}, x)$ is in an unfounded set for any x. With this in mind, we now define the set of *grounding terms* for a definition Δ in a branch B.

Definition 1. *The set $G(\Delta, B)$ of grounding terms of definition Δ in B is defined as follows.*

- *Each member of the set V' of new variables $\{x_1, \ldots x_m\}$, where m is the greatest arity of the predicates in Δ, is a grounding term.*
- *A member of $C(B)$ is a grounding term.*
- *If f is a function symbol of arity n in B, and $t_1, \ldots, t_n$ are elements of $C(B)$ or V', then $f(t_1, \ldots, t_n)$ is a grounding term.*

We denote by $F(\Delta, B)$ the set of all ground atoms of the defined predicates in Δ, ground with the symbols in $G(\Delta, B)$. It is not necessarily the case that $F(\Delta, B) = H(\Delta, B)$. Care should be taken when grounding defined predicates with the terms that contain variables: it could be the case that the newly ground atom has no rule body in the definition. We allow in $H(\Delta, B)$ only those members of $F(\Delta, B)$ for which there is a valid substitution from one of the rule heads in Δ. We define a substitution $[\boldsymbol{a} : \boldsymbol{b}]$ as valid if for each a_i and b_i, either a_i and b_i are both variables, or a_i is a variable and b_i is a constant symbol.

Definition 2. *Let A be the set of rule heads in a definition Δ.*

We define $H(\Delta, B)$ as the set $\{h \mid h \in F(\Delta, B)$ and either h contains no variables, or there is an $a \in A$ and a valid substitution θ such that $a\theta = h\}$

Example 5. Given a definition $\{\forall x, y\ p(x, y) \leftarrow \phi(x, y)\}$, and $C(B) = \{a\}$, then $H(\Delta, B) = \{p(a, a), p(x_1, a), p(a, x_2), p(x_1, x_2), p(x_1, x_1)\}$.

Example 6. Given the definition

$$\left\{ \begin{array}{c} Even(0) \leftarrow \\ \forall x\ Even(s(x)) \leftarrow \neg Even(x) \end{array} \right\}$$

and $C(B) = \{0, s(0)\}$, then $G(\Delta, B) = \{x_1, 0, s(0), s(x_1), s(s(0))\}$ and $H(\Delta, B) = \{Even(0), Even(s(0)), Even(s(x_1), Even(s(s(0)))\}$. Note that $Even(x_1) \notin H(\Delta, B)$, since neither $[0 : x_1]$ nor $[s(x) : x_1]$ is a valid substitution.

We can think of the free variables in $H(\Delta, B)$ as being implicitly universally quantified. If, for example, $p(x_1)$ is a member of $H(\Delta, B)$, we consider this as a claim that $p(x_1)$ is unfounded for all x_1. If our algorithm fails to remove $p(x_1)$ from our representation of an unfounded set, we can conclude that $\exists x p(x)$ is false.

For each atom $p(\boldsymbol{a})$ of $H(\Delta, B)$, we define $body(p(\boldsymbol{a}))$ as the set of all conjunctive formulas $\phi_i(\boldsymbol{b})$, where $\forall \boldsymbol{x}\ p(\boldsymbol{x_1}) \leftarrow \phi_i(\boldsymbol{x_2})$ is a rule in Δ, and $\boldsymbol{b}$ is the result of applying the substitution $[\boldsymbol{x_1} : \boldsymbol{a}]$ on $\boldsymbol{x_2}$, if the substitution is valid. Note that the definition of $H(\Delta, B)$ ensures that $body(p(\boldsymbol{a}))$ is nonempty if $\boldsymbol{a}$ contains a variable. We assume all free variables in $body(p(\boldsymbol{a}))$ are implicitly *existentially* quantified. Note that $body(p(\boldsymbol{a}))$ is independent of and therefore constant in the size of the branch. This fact will be important in proving that our algorithm behaves monotonically with respect to repeated tableau rule applications.

We now define the notion of an unfounded set of a definition in a branch. An unfounded set for a definition Δ in branch B is a set U of ground defined atoms p in Δ for which, for all $\phi \in body(p)$, either ϕ is false in B or ϕ contains

a member of U. To refine this notion for our set $H(\Delta, B)$ of defined atoms h that may include free variables, we must define what it means for a member of $body(h)$ to be false in B, or to contain a member of U.

The former in general involves determining whether a universally quantified disjunctive formula (i.e. the negation of the existentially quantified conjunctive body) is entailed by the branch, which is a nontrivial problem that would require multi-branch tableau reasoning in itself. The latter involves determining whether an existentially quantified conjunctive formula is entailed by the unfounded set, which is also a nontrivial problem that would require some auxiliary tableau reasoning.

However, for efficient computation, we can identify some simple cases in which these conditions are satisfied. We will not identify *any* case in which the conditions are satisfied. Thus, it is possible that some atoms may not be members of the greatest fixed point of our operator, even though these atoms are in the greatest unfounded set. However, the fixed point algorithm will still compute an unfounded set. Thus, our inference rule will be sound, but not complete.

Definition 3. *A substitution $\theta = [\boldsymbol{x} : \boldsymbol{y}]$ is a variable substitution if for each x_i and y_i, both x_i and y_i are variables.*

Let ϕ be a formula $p_1(\boldsymbol{a_1}) \wedge \cdots \wedge p_n(\boldsymbol{a_n})$, where each $p_i(\boldsymbol{a_i})$ is a literal with free variables $\boldsymbol{x_i} \subseteq \boldsymbol{a_i}$. Then, assuming all free variables in ϕ are existentially quantified, ϕ is false in a branch B if (but not only if) $\neg\exists \boldsymbol{y}\ p_i(\boldsymbol{b})$ (or an equivalent formula, where $\boldsymbol{y}$ consists of the free variables in $p_i(\boldsymbol{b})$) is a member of B and there is a variable substitution θ such that $p_i(\boldsymbol{a_i})\theta = p_i(\boldsymbol{b})$. We define the function $falseformula(\phi, B)$ to be true if and only if this condition holds.

Similarly, let ϕ be a formula $p_1(\boldsymbol{a_1}) \wedge \cdots \wedge p_n(\boldsymbol{a_n})$, where each $p_i(\boldsymbol{a_i})$ is a literal with free variables $\boldsymbol{x_i}$. Then ϕ intersects with a set $H_k \subseteq H(\Delta, B)$ (for some definition Δ and some branch B) if (but not only if) there is a $p_i(\boldsymbol{b})$ with free variables $\boldsymbol{y}$ in H_k, and a variable substitution θ such that $p_i(\boldsymbol{a_i})\theta = p_i(\boldsymbol{b})$. (In other words, if we assume $p_i(\boldsymbol{b})$ is unfounded for all $\boldsymbol{y}$, then $\exists \boldsymbol{x}\ p_1(\boldsymbol{a_1}) \wedge \cdots \wedge p_i(\boldsymbol{a_i}) \wedge \ldots \wedge p_n(\boldsymbol{a_n})$ is false.) We define the function $intersects(\phi, H_k)$ to be true if and only if this condition holds.

Note that the function $falseformula(\phi, B)$ is not true *if and only if* formula ϕ is false in branch B. However, if $falseformula(\phi, B)$ is true, then ϕ is false in B (assuming all free variables in ϕ are existentially quantified). The same holds for $intersects(\phi, S)$ as defined above.

We now define the operator we will use in our algorithm.

Definition 4. *Assume S is an element of $\mathcal{P}(H(\Delta, B))$, the powerset of $H(\Delta, B)$. We define an operator Φ on $\mathcal{P}(H(\Delta, B)) \mapsto \mathcal{P}(H(\Delta, B))$ as follows:*

$\Phi =_{def} \{h \in S \mid$ *for each* $\phi \in body(h), falseformula(\phi, B)$ *or* $intersects(\phi, S)\}$

Proposition 1. *Φ is a monotonic operator on the subset relation, i.e. if $S_1 \subseteq S_2$ then $\Phi(S_1) \subseteq \Phi(S_2)$.*

Proof. Assume $S_1 \subseteq S_2$ but $\Phi(S_1) \supset \Phi(S_2)$. Then, since $falseformula(f, B)$ is independent of S_1 and S_2, some $h \in S_1$ has a body that contains an element of

S_1, but not S_2. This is a contradiction, since $body(h)$ is independent of S_1 and S_2, and $S_1 \subseteq S_2$.

Therefore, since Φ is a monotone operator on a complete lattice, it has a greatest and least fixed point. We denote by $US(\Delta, B)$ the greatest fixed point of $\Phi(S)$, $S \in \mathcal{P}(H(\Delta, B))$.

We will now show that $US(\Delta, B)$ is monotonic with respect to iterated tableau rule applications on a branch B.

Since tableau rules never delete anything from a branch, to prove monotonicity of our unfounded set estimate with respect to iterated tableau rule applications it is sufficient to prove monotonicity over the subset relation on the branch, i.e. to show that if $B \subseteq B'$, then $US(\Delta, B) \subseteq US(\Delta, B')$.

The following proposition is trivially true:

Proposition 2. *Given $B \subseteq B'$, if $falseformula(\phi, B)$ is true for some conjunction of ground literals ϕ, then $falseformula(\phi, B')$ is true.*

Theorem 1. *Given some definition Δ, if $B \subseteq B'$, then $US(\Delta, B) \subseteq US(\Delta, B')$.*

Proof. The proof is by induction on k, where k is the number of iterated applications of $\Phi(S)$ beginning with $S = H(\Delta, B)$. We denote by S_k^B the value of the argument to Φ (in the domain $\mathcal{P}(H(\Delta, B))$ at application k.

For $k = 0$, $S_0^B = H(\Delta, B)$, and $S_0^{B'} = H(\Delta, B')$. It is trivial to show that $H(\Delta, B) \subseteq H(\Delta, B')$, and so $S_0^B \subseteq S_0^{B'}$.

We assume indictively that $S_k^B \subseteq S_k^{B'}$.

Then some h is an element of S_{k+1}^B if, for all $\phi \in body(h)$, either $falseformula(\phi, B)$ is true, or $intersects(\phi, S_k^B)$ is true.

If $falseformula(\phi, B)$ is true, then from proposition 2 it is the case that $falseformula(\phi, B')$ is true.

By the induction hypothesis, if $intersects(\phi, S_k^B)$ is true, then $intersects(\phi, S_k^{B'})$ is true.

(It is important to note here that $body(h)$ is independent of B, i.e. the number of instantiated bodies for a defined atom does not increase as the branch increases. So it is sufficient to check the same $\phi \in body(h)$ in B and B'.)

Therefore, $S_{k+1}^B \subseteq S_{k+1}^{B'}$.

We now show that any instance of $US(\Delta, B)$ is a member of an unfounded set.

Theorem 2. *Given a branch B and an interpretation ν over a domain D, which satisfies the first-order formulas in B, then for a definition $\Delta \in B$, $ground(US(\Delta, B), D)$, the grounding of $US(\Delta, B)$ over D, is an unfounded set of Δ with respect to ν.*

Proof. By definition, for every $h \in US(\Delta, B)$, for each $\phi \in body(h)$, either $falseformula(\phi, B)$ is true, or $intersects(\phi, B)$ is true.

Since ν satisfies the first-order formulas in B, then for each such formula $\psi \in B$, $\nu(\psi) = \mathbf{t}$. Then it can be shown that for any $\phi \in \mathit{body}(h)$ with free variables $\boldsymbol{x}$, if $\mathit{falseformula}(\phi, B)$ is true then $\nu(\exists \boldsymbol{x} \phi) = \mathbf{f}$, and, by extension, for any ground instance ϕ' in $\mathit{ground}(\phi, D)$, $\nu(\phi') = \mathbf{f}$.

If $\mathit{intersects}(\phi, B)$ is true, then an element of ϕ positively occurs in $US(\Delta, B)$. Then it can be shown that any ground instance ϕ' in $\mathit{ground}(\phi, D)$ has an element which positively occurs in $\mathit{ground}(US(\Delta, B), D)$.

Therefore, $\mathit{ground}(US(\Delta, B), D)$ is a set of atoms h for which, for each $\phi \in \mathit{body}(h)$, either $\nu(\phi) = \mathbf{f}$ or an element of ϕ positively occurs in $\mathit{ground}(US(\Delta, B), D)$. In other words $\mathit{ground}(US(\Delta, B), D)$ is an unfounded set of Δ with respect to ν.

Because of this result, we can use the algorithm to compute unfounded sets for tableau rule (f).

4.2 Examples

In this section we present two examples of the use of our algorithm. The first is our running example of transitive closure, and the second an inductive definition of the even numbers.

Example 7. Consider the branch B:

$$\left\{ \begin{array}{l} \forall x, y\ T(x, y) \leftarrow G(x, y). \\ \forall x, y\ T(x, y) \leftarrow \exists z\ T(x, z) \wedge T(z, y). \end{array} \right\} \tag{1}$$

$$T(a, a). \tag{2}$$

$$\forall x\ \neg G(x, a). \tag{3}$$

$$\neg G(a, a). \tag{4}$$

(Note that line 4 is the result of applying tableau expansion rule (d) to line 3.)

We denote by Δ the definition in line 1. Then
$H(\Delta, B) = \{T(a, a), T(x_1, a), T(a, x_2), T(x_1, x_2), T(x_1, x_1)\}$.

Then the following are the values of $\mathit{body}(h)$ for each $h \in H(\Delta, B)$ (recall that each variable in $\mathit{body}(h)$ is implicitly existentially quantified):

$$\begin{aligned} \mathit{body}(T(a, a)) &= \{G(a, a), T(a, z) \wedge T(z, a)\} \\ \mathit{body}(T(x_1, a)) &= \{G(x_1, a), T(x_1, z) \wedge T(z, a)\} \\ \mathit{body}(T(a, x_2)) &= \{G(a, x_2), T(a, z) \wedge T(z, x_2)\} \\ \mathit{body}(T(x_1, x_2)) &= \{G(x_1, x_2), T(x_1, z) \wedge T(z, x_2)\} \\ \mathit{body}(T(x_1, x_1)) &= \{G(x_1, x_1), T(x_1, z) \wedge T(z, x_1)\} \end{aligned}$$

The calculation of $US(\Delta, B)$ proceeds as follows. The initial estimate S_0 of the unfounded set is $H(\Delta, B)$. Then we compute $\Phi(S_0)$. (For brevity, if $\mathit{intersects}(\phi, S)$ is true, we say "ϕ contains a member of S").

- $T(a,a)$ is a member of $\Phi(S_0)$, since one of its body elements ($G(a,a)$) is negated in the branch by line 4, and the other contains two members of S_0.
- $T(x_1,a)$ is a member of $\Phi(S_0)$, since one of its body elements ($G(x_1,a)$) is negated in the branch by line 3, and the other contains two members of S_0.
- $T(a,x_2)$ is not a member of $\Phi(S_0)$, since its body element $G(a,x_2)$ is neither a member of S_0 nor negated in the branch.
- $T(x_1,x_2)$ is not a member of $\Phi(S_0)$, since its body element $G(x_1,x_2)$ is neither a member of S_0 nor negated in the branch.
- $T(x_1,x_2)$ is not a member of $\Phi(S_0)$, since its body element $G(x_1,x_1)$ is neither a member of S_0 nor negated in the branch.

Then $\Phi(S_0) = \{T(a,a), T(x_1,a)\}$. We now compute $\Phi(\Phi(S_0))$.

- $T(a,a)$ is a member of $\Phi(\Phi(S_0))$, since one of its body elements ($G(a,a)$) is negated in the branch by line 4, and the other contains $T(z,a)$, a member of $\Phi(\Phi(S_0))$.
- $\exists x\, T(x_1,a)$ is a member of $\Phi(\Phi(S_0))$, since one of its body elements ($G(x_1,a)$) is negated in the branch by line 3, and the other contains $T(z,a)$, a member of $\Phi(\Phi(S_0))$.

Then $\Phi(\Phi(S_0)) = \{T(a,a), T(x_1,a)\} = \Phi(S_0)$, and we have reached a fixed point.

It can easily be verified that $\{T(a,a), T(x_1,a)\}$ describes an unfounded set for Δ in B, and we can use the tableau expansion rule (f) to conclude that both $T(a,a)$ and $\exists x T(x,a)$ are false.

Example 8. Consider the branch B, which includes a definition of the even numbers:

$$\left\{\begin{array}{c} Even(0) \leftarrow \\ \forall x\ Even(s(x)) \leftarrow \neg Even(x) \end{array}\right\} \tag{5}$$

$$Even(0). \tag{6}$$

(Note that line 6 is the result of applying tableau expansion rule (h) to the first rule in the definition Δ in line 5.)

Similarly to example 6, $H(\Delta,B) = \{Even(0), Even(s(0)), Even(s(x_1))\}$. The values of $body(h)$ for each $h \in H(\Delta,B)$ are trivial to determine and will not be printed here. To calculate $US(\Delta,B)$, we make an estimate S_0 of the unfounded set to be $H(\Delta,B)$. Then we compute $\Phi(S_0)$. It should be clear that the only member of $\Phi(S_0)$ is $Even(s(0))$, and that this is the fixed point of the calculation. Therefore $US(\Delta,B) = \{Even(s(0))\}$ and we can use the tableau expansion rule (f) to add $\neg Even(s(0))$ to the branch.

Let $B' = B \cup \neg Even(s(0))$. Then $H(\Delta,B') = \{Even(0), Even(s(0)),$ $Even(s(x_1), Even(s(s(0)))\}$. However, $US(\Delta,B') = US(\Delta,B)$ since the body of the new atom $Even(s(s(0)))$ is true in the branch, and no new atoms appear in the unfounded set.

Let $B'' = B' \cup Even(s(s(0)))$, the result of applying tableau expansion rule (h) to the second rule of Δ and the element $\neg Even(s(0))$ of B'. Now $H(\Delta, B'') = \{Even(0), Even(s(0)), Even(s(x_1), Even(s(s(0))), Even(s(s(s(0))))\}$, and, it can be verified, $US(\Delta, B'') = \{Even(s(0)), Even(s(s(s(0))))\}$. Now we can use the tableau expansion rule (f) to add $\neg Even(s(s(s(0))))$ to the branch.

This sequence can be repeated to infinity to calculate the set of all even numbers.

4.3 Implementation Details

The tableau expansion rules, and the algorithm for computing the greatest unfounded set described above, were implemented in Prolog as a tableau theorem prover for FO(ID). Our initial implementation was not designed with performance in mind, but we nevertheless we found it necessary to adopt a more optimised rule selection strategy in order to help the prover derive certain results. Since the GUS computation algorithm is more useful when there are a large number of literals in the branch (and the backward false rule requires a negated literal as 'input'), our strategy was to apply the first-order tableau rules first as far as possible. When applying first-order rules, we follow the standard tableau strategy of delaying application of the β-rule to prevent unnecessary branching, and removing non-universally-quantified formulas once a first-order rule has been applied upon them. To limit infinite looping we imposed a hard limit on the number of times the γ-rule could be applied; this limit could be specified by the user.

5 Conclusion

In this paper, we described an automated tableau theorem prover for FO(ID), a language that extends first-order logic with inductive definitions, which we claim is a useful language for representing and reasoning with problems in formal software methods. We gave an overview of the syntax and semantics of FO(ID), and described a tableau calculus for the language. We described an incomplete algorithm for implementing the most difficult expansion rule in this calculus: a rule which deduces the negation of an atom that is in the greatest unfounded set of a given definition, and briefly described how the rules and algorithm were implemented in a tableau theorem prover for FO(ID).

This research offers much opportunity for further work. Many optimisations are yet to be done on the prover itself, for example on its memory management and arithmetic performance. We are investigating the possibility of extending the tableau rules with other proving techniques for FO(ID) that are currently being researched, and looking at the feasibility of improving the 'completeness' of the unfounded set algorithm. We are also investigating applications for this theorem prover. One intended application is program verification, and the use of this theorem prover for verification has been proposed as a master thesis.

A formal investigation of the soundness and completeness of this proof system is yet to be done. Since entailment in FO(ID) is undecidable, a proof system for FO(ID) is necessarily incomplete. In particular, since our algorithm for the larger subset of FO(ID) does not calculate the greatest unfounded set of a definition in a branch, there will be atoms in unfounded sets that are not derived to be false by the system. In our example problems, we believe the algorithm struck an acceptable compromise between incompleteness and complexity; it remains to be seen how well this compromise will hold for practical problems.

The main contribution of this research is a domain-independent algorithm for calculating unfounded sets. This algorithm is not strongly bound to a tableau proof system. Further work involves investigating state-of-the-art theorem provers and proof systems for classical logic, to determine how or whether the algorithm can be adapted for and integrated into similar systems for FO(ID).

References

1. Marc Denecker. The well-founded semantics is the principle of inductive definition. In *Logics in Artificial Intelligence, Proceedings of JELIA'98 Schloss Daghstuhl, October 1998*, volume 1489 of *LNAI*, pages 1–16. Springer, 1998. URL: http://www.cs.kuleuven.ac.be/cgi-bin-dtai/publ_info.pl?id=18154.
2. Marc Denecker and Eugenia Ternovska. A logic of non-monotone inductive definitions. *ACM Transactions on Computational Logic*, 2006. URL: http://www.cs.kuleuven.ac.be/cgi-bin-dtai/publ_info.pl?id=42441.
3. Marc Denecker and Eugenia Ternovska. Inductive situation calculus. *Artificial Intelligence*, 1(171 (5-6)):332–360, April 2007.
4. Marc Denecker and Joost Vennekens. Well-founded semantics and the algebraic theory of non-monotone inductive definitions. In *Logic Programming and Nonmonotonic Reasoning, 9th International Conference, LPNMR 2007, Proceedings*, volume 4483 of *Lecture Notes in Artificial Intelligence*, pages 84–96. Springer, 2007. URL: http://www.cs.kuleuven.ac.be/cgi-bin-dtai/publ_info.pl?id=42760.
5. Melvin Fitting. *First-order logic and automated theorem proving (2nd ed.)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.
6. Martin Gebser and Torsten Schaub. Tableau calculi for answer set programming. In *ICLP*, pages 11–25, 2006.
7. Ping Hou, Johan Wittocx, and Marc Denecker. A deductive system for PC (ID). In *Logic Programming and Nonmonotonic Reasoning, 9th International Conference, LPNMR 2007, Proceedings*, volume 4483 of *Lecture Notes in Artificial Intelligence*, pages 162–174. Springer, 2007. URL: http://www.cs.kuleuven.ac.be/cgi-bin-dtai/publ_info.pl?id=42761.
8. Maarten Mariën, Johan Wittocx, and Marc Denecker. MidL: A SAT(ID) solver. In *4th Workshop on Answer Set Programming: Advances in Theory and Implementation*, pages 303–308, 2007. URL: http://www.cs.kuleuven.ac.be/cgi-bin-dtai/publ_info.pl?id=42991.
9. Raymond M. Smullyan. *First-Order Logic*. Courier Dover Publications, New York, 1995.
10. Allen van Gelder, Kenneth Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.

Lazy Thinking Synthesis of a Gröbner Bases Algorithm in *Theorema*

Adrian Crăciun[1] and Bruno Buchberger[2]

[1] e-Austria Research Institute
Timişoara, Romania
acraciun@ieat.ro
[2] Research Institute for Symbolic Computation,
Johannes Kepler University,
Linz, Austria
bruno.buchberger@jku.at

Abstract. We present a nontrivial case study in program synthesis (and more general, theory exploration), carried out in the *Theorema* system: the synthesis of a Gröbner bases algorithm.
We describe *lazy thinking*, a scheme based algorithm synthesis method, in the context of a theory exploration model (both of which were first proposed by the second author) then give an overview of the implementation in *Theorema*.
We then use this implementation to synthesize an algorithm for Gröbner bases, starting from a "critical-pair/completion" idea (algorithm scheme). The case study illustrates several exploration steps, besides applications of lazy thinking: adding new inference rules to the theory, *preprocessing* algorithm schemes, retrieval of knowledge. By applying lazy thinking we are able to re-invent the essential notion of S-polynomials.

1 Introduction

Recent developments in computer-supported reasoning, computer algebra, representation of knowledge, web technology, etc., have created a "critical mass" for addressing the issue of using computer technology to do mathematics (organize, produce, retrieve, write, calculate, reason, solve) in a way that is transparent to "working" mathematicians. Mathematical Knowledge Management (MKM), is a relatively new research field at the intersection of mathematics and computer science, see [8], that is concerned with achieving this.

One of the more (most) important aspects of MKM is the computer supported development (exploration) of mathematical theories. A *theory exploration model based on knowledge schemes* was recently proposed by the second author in [6]: use the mathematical knowledge (experience) developed over the centuries of doing mathematics in the form of *mathematical schemes* (ideas) to develop a mathematical theory. Part of this theory exploration model is *lazy thinking*, a scheme-based algorithm synthesis method, see [5],.

The exploration model was tried out and proved useful in several case studies, such as the exploration of natural numbers, see [17, 14], synthesis of sorting

algorithms, see [9, 10]. Moreover, the second author gave a "pen and paper" outline of using lazy thinking to solve the problem of Gröbner bases, i.e. how to synthesize an algorithm for the computation of Gröbner bases, see [7].

In this paper we describe how an implementation of lazy thinking in the *Theorema* system was used to solve the problem of Gröbner bases, following the general outline from [7]. The implementation work was carried out by the first author, under the guidance of the second author.

We first give an overview of the scheme based exploration model, place the method of lazy thinking in the context of that model and describe the implementation of the method in *Theorema*, in Section 2. In Section 3 we describe the problem of Gröbner bases, as formulated in *Theorema*, then describe the exploration rounds (including applications of lazy thinking) that lead to the synthesis of an algorithm that solves the problem. We compare our approach to related work in synthesis and formal Gröbner bases theory in Section 4 and present our conclusions and future work in Section 5.

2 Lazy Thinking

For the rest of the paper we use the notation conventions of the *Theorema* language, a version of (untyped) predicate logic close to the natural (textbook) style, see for example [11].

2.1 Scheme Based Theory Exploration

Let us first see how *lazy thinking*, a method for the synthesis of algorithms fits in the broader picture of (scheme-based) theory exploration.

Mathematical Theories. A *mathematical theory* is described by its:

- *Language:* the collection of symbols denoting the *functions*, *predicates* and *constants* of the theory (i.e. the *notions* of the theory).
- *Knowledge base:* the collection of formulae over the language that are true. These are either axioms or theorems.
- *Inference mechanism:* the collection of inference mechanisms available to reason in the theory. This will include rules for *predicate logic*, *rewriting* and *domain dependent* inference rules (such as *induction*).

Knowledge Schemes. Conceptually, the accumulated mathematical experience of centuries of mathematics, is available in form of higher order *knowledge schemes* that capture *mathematical ideas*.

For example the idea of a binary preorder relation r over some domain described by the unary predicate p is captured by:

$$\underset{p,r}{\forall}\,(\textit{is-preorder}[p,r] \Leftrightarrow \underset{p[x],p[y],p[z]}{\forall} \left\{ \begin{array}{l} r[x,x] \\ (r[x,y] \wedge r[y,z]) \Rightarrow r[x,z] \end{array} \right.),$$

while

$$\underset{f,g,h}{\forall}(\textit{is-rec-nat-binary-fct-1r}[f,g,h] \Leftrightarrow \underset{\textit{is-nat}[x],\textit{is-nat}[y]}{\forall} \begin{cases} f[x,0] = g[x] \\ f[x,y^+] = h[f[x,y]] \end{cases})$$

represents the idea of a binary function f defined recursively in terms of unary g, h over the natural numbers (described here by *is-nat*), see [17] for details.

As the examples above show, some knowledge schemes are completely abstract, while others may depend on some domain. Moreover, other information can be seen as part of the scheme (e.g. to reason about *is-rec-nat-binary-fct-1r* one uses structural induction over naturals).

Theory Exploration. Given an exploration situation (a theory and a collection of knowledge schemes), an exploration step in the scheme based model is represented by one of the following *exploration rounds*:

Introduce *new notions* (functions, predicates) using *definition schemes*. For instance, using *is-rec-nat-binary-fct-1r* with g being the identity function and h the successor, one can introduce a new binary function in the theory of natural numbers.

Introduce and prove (or disprove) *propositions* about a notion in the theory using *proposition schemes*. For instance, once a binary predicate has been introduced in a theory, one could use *is-preorder* to ask whether the introduced predicate is a preorder. The aim of this stage of exploration is to "saturate" the knowledge base, so other potential properties can be investigated, such as alternative definitions, interactions with other notions.

Introduce *problems* involving a notion using *problem schemes* and solve them (by lazy thinking) using *algorithm schemes*. Problems are situations where a solution is desired. For instance one could use *is-preorder* instantiated with a new symbol (not present in the language of the theory) to ask for a binary relation that should be a preorder. Algorithm schemes capture algorithmic ideas, providing the structure of potential algorithmic solutions in terms of (unknown) subalgorithms. A nontrivial example the application of lazy thinking is the main subject of this paper and is discussed in the next Section.

Introduce *new inference rules* (e.g. by lifting knowledge or using inference schemes). For instance, once a binary well founded ordering is introduced in a theory, well-founded induction can be introduced as a new inference rule.

Note that the same knowledge scheme can play different roles in the exploration (e.g. *is-preorder* can be both a proposition and a problem scheme).

2.2 Solving Problems by Lazy Thinking

Lazy thinking is a method for solving problems in the scheme based exploration model:

– *Start* with a *synthesis situation*, i.e.:

- a *problem (specification)* $\underset{d[x]}{\forall} P[x, A[x]]$ (A is a new symbol, the desired algorithmic solution of P, a predicate describing the relation between the input and the output of the desired algorithmic solution),
- the theory corresponding to the problem (the problem is well-understood, i.e. all relevant concepts and properties are known),
- a collection of algorithm schemes, containing algorithm ideas (potential structure of algorithmic solutions).

– *Select one of the algorithm schemes* available, instantiate it with A, and new symbols for the subalgorithms, and add this instantiation to the knowledge base.
– *Set up the correctness proof*, i.e. a proof of $\underset{d[x]}{\forall} P[x, A[x]]$, using the knowledge available. The proof method to be applied is suggested by the selected algorithm scheme. Note that the proof is likely to fail, due to the fact that we reason about concepts about which we have no knowledge (the subalgorithms from the algorithm scheme).

While the proof fails **do**:

– *Analyze the failed proof*, and
– *Generate a conjecture* from the failure, that will allow the proof to get over the failing situation. Add the conjecture to the knowledge base and try the proof again.

When the proof is completed, *the result of the lazy thinking exploration* round is a proof of the correctness theorem (i.e. the instantiation of the selected scheme solves the problem) provided that notions that satisfy the list of generated conjectures (if these contain symbols for the unknown algorithms) can be retrieved from the knowledge base or can be invented (synthesized, again by lazy thinking), and the conjectures that do not contain unknown symbols can be proved.

The main ingredients of the lazy thinking method are the *exploration cascade* that organizes the successive prove-analyze-conjecture exploration rounds (outlined above), the *proof failure analyzer* and the *conjecture generator*. In the following we discuss these last two.

2.3 Proof Failure Analysis

Before discussing the analysis of failed proofs, we make some remarks concerning the proof system which can be used with lazy thinking.

We use a Gentzen style proof system. *Proof situations* consist of a *goal* and available *knowledge base*. A *proof* consists of the application of a finite number of inference rules that transforms an *initial proof situation* into one or more proof situations until *trivial proof situations* (where the goal appears in the knowledge) are reached.

In *Theorema proof objects* are represented as AND-OR (deduction) tree where nodes are proof situations and (directed) edges indicate the application of an inference rule.

For the formal details of deduction trees and the implementation in *Theorema*, see [27].

A proof *fails* when a nontrivial proof situation cannot be transformed by any of the available inference rules. This means that neither the goal, nor the knowledge can be changed.

The *failure analysis component* of lazy thinking takes a failed proof objects and returns a *conjecture skeleton* $\{\mathcal{TKB}, \mathcal{G}\}$, where:

- $\mathcal{G}$ is the failing goal,
- $\mathcal{TKB}$ is obtained by collecting the temporary knowledge generated along the branch leading to the failing situation, then *filtering* it to eliminate certain formulae that are considered irrelevant.

So far we considered two strategies to filter the temporary knowledge:

- Allow only ground formulae, relevant to the failing goal (i.e. that contain notions that appear in the failed goal).
- Allow ground formulae and certain universally quantified formulae, namely those that cannot be used in knowledge rewriting (i.e. no universally quantified implications, equivalences). We assume that the proof was produced using "generalized knowledge rewriting", i.e. universally quantified equivalences, implications and equalities are used as rewrite rules. This is the PC component of the PCS method – a method for theorem proving that combines proving, computing and solving steps, see [12].

2.4 Conjecture Generation

The idea behind the *conjecture generator* is to "force" the proof to get over the failure when the conjecture generated from it is added to the knowledge base.

The conjecture generator takes a conjecture skeleton and produces a conjecture by applying *generalization substitutions* to the temporary knowledge and failing goals (which will become respectively the left hand side and right hand side of an implication). So far we considered two strategies, based on the shape of the conjecture skeleton.

Term and constant generalizations. Consider ground conjecture skeletons.

- *Term generalization* substitutions τ: consist of substitutions of the form $A[\ldots, aux[\ldots], \ldots] \to x$, where A is the algorithm being synthesized, aux is one of the (unknown) subalgorithms in the algorithm scheme, x is a new variable. The intuition is that applying the (still unknown) algorithm A to some term itself containing unknown operations, all we can say that the result will be some object (of the appropriate type), but nothing more about its structure.
- *Constant generalization* substitutions, σ: generalize constants to variables. These substitutions will be applied *after* term generalizations.

Using the above substitutions, the conjecture skeleton is transformed into a conjecture (in the last step we collect all the new variables in the range r):

$$\{\mathcal{TKB}, \mathcal{G}\} \to (\mathcal{TKB}_{\tau,\sigma} \Rightarrow \mathcal{G}_{\tau,\sigma}) \to \underset{r}{\forall}(\mathcal{TKB}_{\tau,\sigma} \Rightarrow \mathcal{G}_{\tau,\sigma}).$$

Semantic matching generalization. We consider conjecture skeletons that contain universally quantified formulae, i.e. $\mathcal{TKB} : \mathcal{TKB}' \cup \{\underset{s}{\forall} F\}$, such that F matches $\mathcal{G}$, yielding a substitution φ for the variables in the range s with terms from $\mathcal{G}$. The conjecture is assembled by constructing $\underset{s}{\exists} F'$ from the substitution φ, where F' is the conjunction of the "$v = t$"'s (transform the substitution rules of φ in equalities), replace the failing goal $\mathcal{G}$ with this new existential formula, and delete $\underset{s}{\forall} F$ from the temporary knowledge. Then apply term and constant generalizations τ, σ:

$$\{\mathcal{TKB}, \mathcal{G}\} \rightarrow (\mathcal{TKB}'_{\tau,\sigma} \Rightarrow \underset{s}{\exists} F'_{\tau,\sigma}) \rightarrow \underset{r}{\forall}(\mathcal{TKB}'_{\tau,\sigma} \Rightarrow \underset{s}{\exists} F'_{\tau,\sigma}).$$

2.5 Implementation in *Theorema*

The implementation of lazy thinking is organized in the following packages corresponding to the components of lazy thinking: **CascadeLT**, **FailureAnalyzer**, **ConjectureGenerator**.

In order to start a lazy thinking exploration process, the user of our implementation in *Theorema*: (a) specifies the *problem* P (correctness statement), (b) provides the *relevant knowledge base* Kb, including an instantiation of the *algorithm scheme* proposed for solving the problem, (c) indicates the *prover* Pr to be used in this process.

Automated lazy thinking exploration is then set up by calling:

```
Prove[P, using → Kb,
      by → CascadeLT[Pr, GenerateConjectures, ...],
      ProverOptions → {...}]
```

where $CascadeLT$ is the function implementing the lazy thinking exploration process, $GenerateConjectures$ is the function that implements failure analysis and conjecture generation. Arguments irrelevant to this presentation are omitted.

The above represents the usual way in which lazy thinking is set up and called from *Theorema*. However, during the various case study it has become apparent that direct access (of the user) to $FailureAnalyzer$ and $GenerateConjectures$ can be helpful tools in mathematical exploration (in understanding the reason of failure in proofs). The selection of the conjecture skeleton generation and generalization strategies are implemented as options. A detailed description of the implementation can be found in [13].

2.6 Using Lazy Thinking

Although using an algorithm scheme to provide some structure for the proposed solution certainly helps with the synthesis process, this is optional. This implies that there are other ways to use the lazy thinking:

- Provide no information on the structure of the solution (and this will make the synthesis process quite difficult).

– Instead of using an algorithm scheme directly, *preprocess* it, i.e. prove some of its properties, and use these in the exploration, instead of the scheme.

The second approach will be illustrated in the next Section.

3 Gröbner Bases Synthesis

In the following, we describe the synthesis situation consisting of the problem of Gröbner bases, the algorithm scheme that will be used for the synthesis and the knowledge base corresponding to the problem. We then present an outline of the exploration rounds that lead to the synthesis of the solution to the Gröbner bases problem and describe each of the steps in some detail.

3.1 Synthesis Situation: The Problem of Gröbner Bases

Let F, G denote sets of polynomials from some polynomial ring. We want to find an algorithm GB that satisfies the following correctness theorem (*the Gröbner bases problem*, expressed in *Theorema* syntax):

$$\boxed{\begin{array}{c} \mathit{Theorem}[\text{“Groebner bases specification”}, \mathit{any}[F], \mathit{with}[\mathit{is\text{-}finite}[F]], \\ \mathit{is\text{-}finite\text{-}Groebner\text{-}basis}[F, GB[F]] \end{array}]}$$

where

$$\boxed{\begin{array}{r} \mathit{Definition}[\text{“is finite Groebner basis”}, \mathit{any}[F, G], \mathit{with}[\mathit{is\text{-}finite}[F]], \\ \mathit{is\text{-}finite\text{-}Groebner\text{-}basis}[F, G] \Leftrightarrow \bigwedge \left\{ \begin{array}{c} \mathit{is\text{-}finite}[G] \\ \mathit{is\text{-}Groebner\text{-}basis}[G] \\ \mathit{ideal}[F] = \mathit{ideal}[G] \end{array} \right. \end{array}}$$

The above definition allows us to split the case study into three subproblems: “is-Gröbner-basis”, “ideal equality” and “termination”, each of which will be handled in a separate exploration round.

The domain in which this problem is formulated is that of polynomial rings together with the following notions (and corresponding definitions and properties):

– a *well-founded ordering on polynomials*, $\prec$,
– a *reduction relation modulo a polynomial set* F, $\rightarrow_F$, such that if $f \rightarrow_F g$, then $g \prec f$,
– a *reduction operation* modulo a polynomial, rd, such that for $g \in F$, $f \rightarrow_g rd[f, g]$,
– a *total reduction operation* modulo a polynomial set, trd, such that $f \rightarrow_F^\star trd[f, F]$, and $trd[f, F]$ is not reducible,
– *is-Church-Rosser*$[\rightarrow]$, i.e. the relation $\rightarrow$ has the Church Rosser property (confluence).

Due to lack of space, we refer the reader to [7, 13] for the full definitions and properties (such as Newman's lemma), which are standard for Gröbner bases theory. Here we only mention the most relevant results for the synthesis process.

As the inference mechanism, we use *BasicProver*, a *Theorema* user prover implementing a combination of the PCS method (knowledge rewriting), rewriting, simplification.

To complete the description of the synthesis situation, the algorithm scheme we will use, *"critical-pair/completion"* (CPC) (formulated for the polynomial domain, in *Theorema* syntax) is:

$$
\begin{array}{l}
Algorithm[\text{"CPC scheme"}, \text{any}[F, g1, g2, \overline{p}], \\
\quad GB[F] = GB[F, pairs[F]] \\
\quad GB[F, \langle\rangle] = F \\
\quad GB[F, \langle\langle g_1, g_2\rangle, \overline{p}\rangle] = \\
\quad\quad \text{where}[f = lc[g_1, g_2], h_1 = trd[rd[f, g_1], F], h_2 = trd[rd[f, g_2], F], \qquad] \\
\quad\quad \left\{ \begin{array}{ll} GB[F, \langle \overline{p} \rangle] & \Leftarrow h_1 = h_2 \\ GB[F \frown df[h_1, h_2], \langle \overline{p} \rangle \asymp \left\langle \langle F_k, df[h_1, h_2]\rangle \underset{k=1,\ldots,|F|}{|} \right\rangle] & \Leftarrow \text{otherwise} \end{array} \right.]
\end{array}
$$

The CPC idea was applied independently to solve problems in automated theorem proving (resolution [23]), polynomial ideal theory (Gröbner bases), and word problems in universal algebras (the Knuth-Bendix procedure [18]). A presentation of the common features of these algorithms (i.e. the CPC idea) is given in [3].

The CPC idea can be applied in any setting where we have a *"reduction"* relation $\rightarrow_F$, generated by a set F of finitely many objects where we want to solve a *"word problem"* on the respective domain, i.e. for any objects s, t, is (s, t) in the reflexive, symmetric, transitive closure of the reduction relation $(\leftrightarrow^\star{}_F)$? Note that this is the case for our setting.

The CPC idea is: construct a set G such that $\leftrightarrow^\star{}_F \equiv \leftrightarrow^\star{}_G$ and $\rightarrow_G$ has the Church-Rosser property (for these relations the word problem can be decided), by starting from F and a set of *"critical situations"*, for which the Church-Rosser property is checked. If the property holds, the next critical situation is checked, otherwise, the set is *"completed"* with an element that makes the property hold.

In *Algorithm*["CPC Scheme"], the new symbol GB is defined in terms of two auxiliary symbols lc and df. To satisfy the conditions of CPC, we add to the scheme the following specification:

$$
\underset{g1,g2}{\forall} \bigwedge \left\{ \begin{array}{l} rd[lc[g1, g2], g1] \prec lc[g1, g2] \\ rd[lc[g1, g2], g2] \prec lc[g1, g2] \end{array} \right. ,
$$

which immediately, from the definition of rd (see [7, 13]) implies

$$\underset{g1,g2}{\forall} \bigwedge \begin{Bmatrix} lp[g1]|lc[g1,g2] \\ lp[g2]|lc[g1,g2] \end{Bmatrix},$$

where lp is the leading power product of its argument.

3.2 Preprocessing CPC

A direct induction proof of properties of the *Algorithm*["CPC scheme"], our proposed solution for the Gröbner bases problem, is difficult to set up, due to the complexity of the recursive calls. Rather, we derive consequences of the schemes (we preprocess the scheme) and use this knowledge in the lazy thinking synthesis process.

We first introduce an inference rule to reason about the CPC scheme. Note that it is not at all clear that the algorithm terminates. In fact it is well known that the resolution and Knuth-Bendix algorithms (both instances of CPC) do not always terminate.

Informally, to prove $\mathcal{P}$: $\underset{F}{\forall}\mathcal{P}[F, GB[F]]$ of GB, take some $\mathcal{C}$ a ternary property and F_0 (playing the role of the initial argument), G_0 (playing the role of an intermediary polynomial set generated during the execution), g_1, g_2, $\overline{p_0}$ arbitrary but fixed, and prove:

- "base case": $\mathcal{C}[F_0, F_0, pairs[F_0]]$ (i.e. $\mathcal{C}$ holds for the initial call);
- "step":
  Case $h_1 = h_2$: $\mathcal{C}[F_0, G_0, \langle\langle g_1, g_2\rangle, \overline{p_0}\rangle] \Rightarrow \mathcal{C}[F_0, G_0, \langle\overline{p_0}\rangle]$,
  Case $h_1 \neq h_2$:

  $$\mathcal{C}[F_0, G_0, \langle\langle g_1, g_2\rangle, \overline{p_0}\rangle] \Rightarrow \mathcal{C}[F_0, G_0 \frown df[h_1, h_2], \langle\overline{p}\rangle \asymp \left\langle \langle F_k, df[h_1, h_2]\rangle \underset{k=1,\ldots,|F|}{|} \right\rangle]];$$

- "final step": $\mathcal{C}[F_0, G_0, \langle\rangle] \Rightarrow \mathcal{P}[F_0, G_0]$.

In the above h_1, h_2 are as in *Algorithm*["CPC Scheme"].

The above rule works only if the algorithm terminates (i.e. "final step" is reached). An extra difficulty in the use of this rule is the selection of an appropriate $\mathcal{C}$. As it turns out, a particular choice ($\mathcal{C}[F, G, B] \Leftrightarrow (\mathcal{P}[F, G] \wedge B \subseteq G \times G)$), greatly simplifies the inference rule, leaving only the "step" $h_1 \neq h_2$ for verification. This, however, limits the properties of GB that can be proved to those that are *invariant*, i.e. that hold for all intermediary sets (including the final one).

We implemented the above inference rule in *Theorema* as the $CPCInduction$ user prover, that combines the invariant rule with $BasicProver$. In effect, we enhanced the inference mechanism of the theory with this invariant proof rule for the CPC scheme.

Using $CPCInduction$ we proved invariant properties such as $\underset{F}{\forall} F \subseteq GB[F]$. Using this, it is easy to prove and then use in the exploration the following property of the CPC scheme:

$$\underset{g_1,g_2,F}{\forall} \bigvee \begin{Bmatrix} h_1 = h_2 \\ df[h_1, h_2] \in GB[F] \end{Bmatrix}.$$

3.3 Is-Gröbner-Basis

We now consider the subproblem "is-Gröbner-Basis", which can immediately be reduced to the proving that the reduction relation modulo the polynomial set has the Church-Rosser property (*Theorem*["Groebner basis specification:2"] in our *Theorema* case study). The knowledge base corresponding to this synthesis situation includes the CPC property mentioned in the previous subsection, Newman's lemma, which reduces the Church-Rosser property to local confluence, definitions and properties of the notions involved. All are collected in the *Theorema* construct *Theory*["GB knowledge"] (see [13] for details). Reasoning is done through the *BasicProver*.

The following is the call of lazy thinking, and the output of the method (the conjectures generated). Lazy thinking also produces the proof attempts, but due to the limited space we refer the reader to [7] for the "pen and paper" proof, or to [13] for the proof produced by *Theorema* (which is in fact close to textbook mathematics).

$$
\begin{array}{l}
Prove[Theorem[\text{"Groebner basis specification:2"}],\\
\quad using \rightarrow Theory[\text{"GB knowledge"}],\\
\quad by \rightarrow CascadeLT[BasicProver, GenerateConjectures, \ldots],\\
\quad ProverOptions \rightarrow \{GRWTarget \rightarrow \{"goal","kb"\},\\
\qquad\qquad RWExistentialGoal \rightarrow True\}]
\end{array}
$$

LAZY THINKING::::: The proof fails.
After analysing the failing proof, the following conjecture is added to the knowledge base:
•lf[Lemma (specification 2): 1,

$$\underset{\boldsymbol{g05},\boldsymbol{g06},\boldsymbol{p04}}{\forall}\left((\mathrm{lp}[\boldsymbol{g05}]|\boldsymbol{p04})\wedge(\mathrm{lp}[\boldsymbol{g06}]|\boldsymbol{p04})\wedge \mathrm{is-pp}[\boldsymbol{p04}]\Rightarrow\underset{\boldsymbol{a},\boldsymbol{q}}{\exists}(\boldsymbol{p04}=\boldsymbol{a}*\boldsymbol{q}*\mathrm{lc}[\boldsymbol{g05},\boldsymbol{g06}])\right),\bullet\mathrm{finfo}[]\Big]$$

Now attempt the proof with the updated knowledge base.
LAZY THINKING::::: The proof fails.
After analysing the failing proof, the following conjecture is added to the knowledge base:
•lf[Lemma (specification 4): 1,

$$\underset{\boldsymbol{F04},\boldsymbol{g09},\boldsymbol{g10}}{\forall}(\mathrm{df}[\mathrm{trd}[\mathrm{rd}[\mathrm{lc}[\boldsymbol{g09},\boldsymbol{g10}],\boldsymbol{g09}],\mathrm{GB}[\boldsymbol{F04}]],\mathrm{trd}[\mathrm{rd}[\mathrm{lc}[\boldsymbol{g09},\boldsymbol{g10}],\boldsymbol{g10}],\mathrm{GB}[\boldsymbol{F04}]]]\in\mathrm{GB}[\boldsymbol{F04}]\wedge$$

$$\boldsymbol{g09}\in\mathrm{GB}[\boldsymbol{F04}]\wedge\boldsymbol{g10}\in\mathrm{GB}[\boldsymbol{F04}]\Rightarrow$$

$$(\mathrm{trd}[\mathrm{rd}[\mathrm{lc}[\boldsymbol{g09},\boldsymbol{g10}],\boldsymbol{g09}],\mathrm{GB}[\boldsymbol{F04}]]=\mathrm{trd}[\mathrm{rd}[\mathrm{lc}[\boldsymbol{g09},\boldsymbol{g10}],\boldsymbol{g10}],\mathrm{GB}[\boldsymbol{F04}]])),\bullet\mathrm{finfo}[]]$$

Now attempt the proof with the updated knowledge base.

LAZY THINKING ::::: The proof is completed!

After some variable renaming, and together with the immediate consequence of the CPC scheme introduced in Subsection 3.1, Lemma (specification 2): 1, generated automatically by the system really is:

$$\underset{g1,g2,p}{\forall}((\mathit{is\text{-}pp}[p]\wedge lp[g1]|p\wedge lp[g2]|p)\Rightarrow\underset{a,q}{\exists}p=a*q*lc[g1,g2]),$$

$$\underset{g1,g2}{\forall}\bigwedge\left\{\begin{array}{l}lp[g1]|lc[g1,g2]\\ lp[g2]|lc[g1,g2]\end{array}\right.,$$

The reader will recognize in this a definition of the *least common multiple* of $lp[g_1]$ and $lp[g_2]$ (lp is the leading power product of its argument, *is-pp* is true if its argument is a power product).

Lemma (specification 4): 1 generated in the second round is not very useful: the conjecture generator did not eliminate the GB symbol, so it is a conjecture on all unknown symbols. It does not harm though: it states that if the term $df[\ldots,\ldots]$ stays in the final set, the problem can be solved.

In the call to the lazy thinking method above, *ConjectureGenerator* used the semantic matching generalization strategy. In [13] we show that the weaker strategy (term generalization) is not sufficient to solve this problem.

3.4 Ideal Membership

For the ideal membership problem, one of the directions is immediate ($\underset{F}{\forall} ideal[F] \subseteq ideal[GB[F]]$), due to the fact that the initial set is a subset of the final result.

The other direction ($\underset{F}{\forall} ideal[GB[F]] \subseteq ideal[F]$, $Theorem$["Groebner basis specification: 3 $\subseteq$"]) is proved using the *CPCInduction* prover. The property is invariant. The intermediary sets generated by the execution of $Algorithm$["CPC Scheme"] all generate the same ideal. We collect the relevant knowledge in $Theory$["GB knowledge: ideals"], see [13], and then we call lazy thinking (again referring the reader to [13] for the generated proofs):

$Prove[Theorem$["Groebner basis specification: 3 $\subseteq$ "],
  $using \rightarrow Theory$["GB knowledge: ideals"],
  $by \rightarrow CascadeLT[CPCInduction, GenerateConjectures, \ldots]$,
  $ProverOptions \rightarrow \{GRWTarget \rightarrow \{"goal","kb"\}\}]$

LAZY THINKING::::: The proof fails.
After analysing the failing proof, the following conjecture is added to the knowledge base:
•lf[Lemma (specification 2): 1,
$\underset{\boldsymbol{F02},\boldsymbol{G102},\boldsymbol{g302},\boldsymbol{g402}}{\forall}(\boldsymbol{g302} \in \boldsymbol{G102} \wedge \boldsymbol{g402} \in \boldsymbol{G102} \wedge \text{trd}[\text{rd}[\text{lc}[\boldsymbol{g302},\boldsymbol{g402}],\boldsymbol{g302}],\boldsymbol{G102}] \neq$
$\text{trd}[\text{rd}[\text{lc}[\boldsymbol{g302},\boldsymbol{g402}],\boldsymbol{g402}],\boldsymbol{G102}] \wedge \text{ideal}[\boldsymbol{G102}] \subseteq \text{ideal}[\boldsymbol{F02}] \Rightarrow$
$\text{ideal}[\boldsymbol{G102} \frown \text{df}[\text{trd}[\text{rd}[\text{lc}[\boldsymbol{g302},\boldsymbol{g402}],\boldsymbol{g302}],\boldsymbol{G102}],\text{trd}[\text{rd}[\text{lc}[\boldsymbol{g302},\boldsymbol{g402}],\boldsymbol{g402}],\boldsymbol{G102}]]] \subseteq$
$\text{ideal}[\boldsymbol{F02}])$, •finfo[]]
Now attempt the proof with the updated knowledge base.
LAZY THINKING ::::: The proof is completed!!!!

After one lazy thinking exploration round the proof succeeds. Although the generated Lemma (specification 2): 1 (no conflict with the conjecture from the previous Subsection, as these are separate sessions) looks potentially intimidating, what it really says is that for an intermediary set G, and the initial F, $ideal[G \frown df[h_1, h_2]] \subseteq ideal[F]$ (with h_1, h_2 as in $Algorithm$["CPC Scheme"]).

This is as far as lazy thinking will take us. But can we find any function that satisfies this specification? From the definitions of reduction and total reduction, see [7], both h_1 and h_2 can be written as (linear) combinations of elements from G (and by application of induction, essentially F) plus $lc[g_1, g_2]$. We want to show that $df[h_1, h_2]$ is a (linear) combination of elements form F. It is clear that if we reduce the common $lc[g_1, g_2]$ by substraction, we are left with a (linear) combination of elements from F. Therefore, with some simple exploration steps, we identified the substraction operation on polynomials as satisfying df.

Now both *lc* and *df* are known (least common multiple of the leading power products and substraction, respectively). Together, these reconstruct the essential notion of S-polynomials in Gröbner bases theory, see [4]. With these *lc* and *df* we can prove termination and complete the synthesis.

3.5 Termination

Showing termination amounts to showing that we can find a well-founded ordering on the arguments in the recursive call of GB. We look for a lexicographic ordering: we first compare the first argument (polynomial set), w.r.t. a well-founded ordering. If we have equality, we move to comparing the second argument (set of pairs of polynomials):

- In the case $h_1 = h_2$, the lexicographic ordering clearly holds. The first argument does not change in the recursive call. The second argument decreases w.r.t the length of the tuples (which is well-founded).
- In the other case, however, it is not clear that the argument decreases. In fact we add elements to the first argument in the recursive call (and also to the second).

In the theory of Gröbner bases, it can be shown, for the *lc* (least common multiple) and *df* (substraction) discovered in previous lazy thinking exploration rounds, that if $h_1 \neq h_2$, then the leading power product of $h_1 - h_2$ is not a multiple of any of the leading power products already in the polynomial set.

Then, by Dickson's lemma, see [15], there is no infinite such sequence of power products. Since to every polynomial added to the polynomial set there is a corresponding power product that verifies the condition in Dickson's lemma, this means that only finitely many new polynomials can be added to the polynomial set. This ensures termination.

This is the usual way to prove termination in Gröbner bases theory, see, for instance, [2] where Dickson's lemma was reinvented. We included the termination argument for completeness purposes, as this part of the case study is ongoing work in *Theorema*. We could also refer to previous work in Gröbner bases such as [26, 22] where termination is proved, i.e. use those results. Naturally, the formalisms are different and a complete *Theorema* exploration is preferable. However, we want to emphasize that the essence of this case study – algorithm synthesis – is described in the previous Subsections, and is carried out in *Theorema*.

Now the synthesis process is complete: We have synthesized by lazy thinking exploration the unknown subalgorithms in *Algorithm*["CPC scheme"], essentially reconstructing the crucial notion of S-polynomials and through the synthesis process provided the proof of correctness. We also proved termination, which, in turn, ensures that the inference rules we used to prove properties of the algorithm scheme are correct.

4 Related Work

Algorithm (program) synthesis is a well established research field, and an overview of the various approaches can be found in [13], or the survey papers [20, 1]. Lazy thinking is similar to existing approaches, such as deductive approaches [21, 19], in that a proof of the correctness of the specification is set up, but different in the method for obtaining the algorithm (specifications for subalgorithms vs. instantiation of metavariables).

Also, our use of algorithm schemes has the same motivation with that of scheme based synthesis in logic programming such as [16] or functional programming like [25]. However, the focus of lazy thinking is different. Both the scheme based approaches mentioned focus on synthesis by transformation, where a framework and strategies for using algorithm schemes are available, and they were proved offline to ensure correctness of transformations. Lazy thinking is working on proving *online* correctness of programs (at various levels of abstraction). *Failure analysis and conjecture generation* are the identifying features of lazy thinking, when compared to other approaches.

The method of lazy thinking was presented in our earlier work, [9]. In the work presented in this paper we added new failure analysis and conjecture generation strategies and we integrate lazy thinking in the exploration model.

The synthesis of a Gröbner bases algorithm is, to our knowledge, unique. Other approaches to formal Gröbner bases theory, such as [26, 22, 24] are formalizations of parts of the theory, and most focus on the proof of termination (Dickson's lemma). Some algorithms are extracted from constructive proofs, but we feel there is no invention involved: all the relevant knowledge (like the definition of the S-polynomials) is provided.

5 Conclusions

In this paper we presented the lazy thinking algorithm synthesis method in the context of a scheme-based theory exploration model and presented its implementation in the *Theorema* system.

We used our implementation to synthesize an algorithm for Gröbner bases computation, following a "pen and paper" outline given in [7]. The development in this paper is that the case study was explored with the *Theorema* implementation of lazy thinking, and we solved many of the things left implicit in the outline: the preprocessing of CPC, the generalization strategies needed for the synthesis of lc, the synthesis (retrieval) of df.

Lazy thinking (as part of the scheme based exploration model) has a great pedagogical value, giving insight into how new notions can be invented as solutions to problems. The case study we have successfully handled shows that the method is suitable in nontrivial case studies.

Future directions for research include the integration of lazy thinking into a framework to support the scheme-based theory exploration (based on functors, allowing queries on theories and scheme libraries), carrying out other case

studies, the investigation and development of libraries of schemes similar to the hierarchy of schemes described in [25].

For the Gröbner bases case study, the development of the termination proof in the scheme-based exploration model is ongoing. We develop a theory of well-founded relations, and then apply this to establish the well-foundedness of the ordering between the recursive calls of the *GB* algorithm. We also look into the derivation by lazy thinking of improved algorithms (criteria, reduced Gröbner bases).

Acknowledgements. For the first author, this work was partially supported by the Transnational Access Programme at RISC-Linz, in the frame of the European Commission Framework 6 Programme for Integrated Infrastructures Initiatives, project SCIEnce (contract No 026133) and the Austrian Federal Ministry of Science and Research (BMWF) Project 45.551 "Automated Exploration of Mathematical Theories for Industrial Applications" at the e-Austria Research Institute Timişoara, Romania.

References

[1] D. Basin, Y. Deville, P. Flenner, A. Hamfelt, and J.F. Nilsson. *Program Development in Computational Logic*, volume 3049 of *LNCS*, chapter Synthesis of Programs in Computational Logic, pages 30–65. Springer Verlag, 2004.

[2] B. Buchberger. Ein algorithmisches Kriterium fuer die Loesbarkeit eines algebraischen Gleichungssystems (An Algorithmic Criterion for the Solvability of Algebraic Systems of Equations). *Aequationes mathematicae*, 3:374–383, 1970. (english transl.: B. Buchberger, F. Winkler: Groebner Bases and Applications, Proc. of the International Conference "33 Years of Groebner Bases", 1998, RISC, Austria, London Math. Society Lecture Note Series 251, Cambridge Univ. Press, 1998, pp.535 -545).

[3] B. Buchberger. History and Basic Features of the Critical-Pair/Completion Procedure. *Journal of Symbolic Computation*, 3(1/2):3–38, 1987. (Earlier version appeared in: Proceedings of the Conference on Rewrite Technique and Applications, Dijon, May 1985, Lecture Notes in Computer Science, Vol. 202, Springer, 1985, pp. 1-45).

[4] B. Buchberger. Introduction to Gröbner Bases. In B. Buchberger and F. Winkler, editors, *Gröbner Bases and Applications*, volume 251 of *London Mathematical Society Lectures Notes Series*, pages 3–31. Cambridge University Press, Johannes Kepler University of Linz, 1998.

[5] B. Buchberger. Algorithm Invention and Verification by Lazy Thinking. In D. Petcu, V. Negru, D. Zaharie, and T. Jebelean, editors, *Proceedings of SYNASC 2003, 5th International Workshop on Symbolic and Numeric Algorithms for Scientific Computing Timisoara*, pages 2–26, Timisoara, Romania, 1-4 October 2003. Copyright: Mirton Publisher.

[6] B. Buchberger. Algorithm Supported Mathematical Theory Exploration: A Personal View and Stragegy. In B. Buchberger and J. Campbell, editors, *Proceedings of AISC 2004 (7 th International Conference on Artificial Intelligence and*

Symbolic Computation), volume 3249 of *Springer Lecture Notes in Artificial Intelligence*, pages 236–250. RISC, Johannes Kepler University, Austria, Springer, Berlin-Heidelberg, 2004.

[7] B. Buchberger. Towards the Automated Synthesis of a Groebner Bases Algorithm. *RACSAM - Revista de la Real Academia de Ciencias (Review of the Spanish Royal Academy of Science)RACSAM - Revista de la Real Academia de Ciencias (Review of the Spanish Royal Academy of Science), Serie A: Mathematicas*, 98(1):65–75, 2004.

[8] B. Buchberger and O. Caprotti. First International Workshop on Mathematical Knowledge Management (MKM 2001). Technical report, RISC, Johannes Kepler University, Austria, September 2001.

[9] B. Buchberger and A. Crăciun. Algorithm Synthesis by Lazy Thinking: Examples and Implementation in Theorema. In F. Kamareddine, editor, *Proc. of the Mathematical Knowledge Management Workshop, Edinburgh, Nov. 25, 2003.*, volume 93 of *Electronic Notes in Theoretical Computer Science*, pages 24–59, February 2004.

[10] B. Buchberger and A. Crăciun. Algorithm Synthesis by Lazy Thinking: Using Problem Schemes. In D. Petcu, V. Negru, D. Zaharie, and T. Jebelean, editors, *Proceedings of SYNASC 2004, 6th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing Timisoara*, pages 90–106, Timisoara, Romania, 2004.

[11] B. Buchberger, A. Crăciun, T. Jebelean, L. Kovacs, T. Kutsia, K. Nakagawa, F. Piroi, N. Popov, J. Robu, M. Rosenkranz, and W. Windsteiger. Theorema: Towards Computer-Aided Mathematical Theory Exploration. *Journal of Applied Logic*, pages 470–504, 2006.

[12] B. Buchberger, C. Dupre, T. Jebelean, F. Kriftner, K. Nakagawa, D. Vasaru, and W. Windsteiger. The Theorema Project: A Progress Report. In M. Kerber and M. Kohlhase, editors, *Symbolic Computation and Automated Reasoning (Proceedings of CALCULEMUS 2000, Symposium on the Integration of Symbolic Computation and Mechanized Reasoning)*, pages 98–113. St. Andrews, Scotland, Copyright: A.K. Peters, Natick, Massachusetts, 6-7 August 2000.

[13] A. Crăciun. *Lazy Thinking Algorithm Synthesis in Gröbner Bases Theory.* PhD thesis, Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, 2008.

[14] A. Crăciun and M. Hodorog. Decompositions of Natural Numbers: From A Case Study in Mathematical Theory Exploration. In D. Petcu, V. Negru, D. Zaharie, and T. Jebelean, editors, *Proceedings of the 9th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC07)*. IEEE Computer Society, 2008. to appear as IEEE volume.

[15] L. E. Dickson. Finiteness of the Odd Perfect and Primitive Abundant Numbers with n Distinct Prime Factors. *Am. J. Math.*, 35:413–422, 1913.

[16] P. Flener, K.-K. Lau, M. Ornaghi, and J. Richardson. An Abstract Formalization of Correct Schemas for Program Synthesis. *J. Symb. Comput.*, 30(1):93–127, 2000.

[17] M. Hodorog and A. Crăciun. Scheme-Based Systematic Exploration of Natural Numbers. In *Proceedings of the 8th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, Timisoara, Romania, September 26-29, 2006*, pages 23–34. IEEE Computer Society, 2007.

[18] D. E. Knuth and P. B. Bendix. Simple Word Problems in Universal Algebra. In J. Leech, editor, *Proc. of the Conf. of Computational Problems in Abstract Algebra, Oxford, 1967*. Pergamon Press, 1970.

[19] I. Kraan, D. A. Basin, and A. Bundy. Middle-Out Reasoning for Synthesis and Induction. *J. Autom. Reasoning*, 16(1-2):113–145, 1996.

[20] C. Kreitz. *Automated Deduction - A Basis for Applications*, chapter Program Synthesis, pages 105–134. Kluwer Academic Publishers, 1998.

[21] Z. Manna and R. Waldinger. Fundamentals of Deductive Program Synthesis. *IEEE Transactions on Software Engineering*, 18(8):674–704, 1992.

[22] I. Medina-Bulo, F. Palomo-Lozano, J. A. Alonso-Jiménez, and J.-L. Ruiz-Reina. Verified Computer Algebra in ACL2. gröbner Bases Computation. In B. Buchberger and J. A. Campbell, editors, *AISC: Artificial Intelligence and Symbolic Computation, 7th International Conference, AISC 2004, Linz, Austria, September 22-24, 2004, Proceedings*, pages 171–184, 2004.

[23] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1):23–41, 1965.

[24] C. Schwarzweller. Groebner Bases - Theory Refinement in the Mizar System. In *4th International Conference on Mathematical Knowledge Management, Proceedings*, number 3863 in Lecture Notes in Artificial Intelligence, pages 299–314. Springer Verlag, 7 2005.

[25] D. R. Smith. Mechanizing the Development of Software. In M. Broy, editor, *Calculational System Design, Proceedings of the International Summer School Marktoberdorf*, NATO ASI Series, Amsterdam, 1999. IOS Press.

[26] L. Théry. A Machine-Checked Implementation of Buchberger's Algorithm. *J. Autom. Reason.*, 26(2):107–137, 2001.

[27] E. Tomuţa. *An Architecture for Combining Provers and its Applications in the Theorema System.* PhD thesis, RISC, Johannes Kepler University of Linz, 1998.

Practical Program Verification by Forward Symbolic Execution: Correctness and Examples

EXTENDED ABSTRACT

Mădălina Eraşcu and Tudor Jebelean

Research Institute for Symbolic Computation,
Johannes Kepler University, Linz, Austria
{Madalina.Erascu,Tudor.Jebelean}@risc.uni-linz.ac.at

Abstract. We present the theoretical aspects and a prototype implementation in the *Theorema* system of a method for the verification of recursive imperative programs. The method is based on forward symbolic execution and functional semantics and generates first order verification conditions for the total correctness which use only the underlying theory of the program. All verification conditions are generated automatically by our prototype implementation in the frame of the *Theorema* system based on *Mathematica*.
The termination property is expressed as an induction principle depending on the structure of the program with respect to recursion. It turns out that part of the verification conditions (notably the termination condition) are crucial for the existence of the function defined by the program, without which the total correctness formula is trivial due to inconsistency of the assumptions.
The formal description of the method is the basis for the implementation and also for the proof of its correctness.

1 Introduction

We present a formal verification method for imperative programs based on symbolic execution [4, 1], forward reasoning [5, 3] and functional semantics [6]. The distinctive features of our approach are:

- All verification conditions are formulated in the theory of the objects which are manipulated by the program (the object theory – see below).
- The notions of program, semantics, verification condition, and termination are precisely formalized in predicate logic.
- Termination is treated in a purely logical way, namely as existence and uniqueness of the function implemented by the program.

Our approach is purely logical. We assume that the properties of the constants, functions and predicates which are used in the program are specified in an *object theory* Υ. (By a *theory* we understand a set of formulae in the language of predicate logic with equality.) For the purpose of reasoning about imperative

programs we construct a certain *meta–theory* containing the properties of the meta–predicate Π (which checks a program for syntactical correctness) and the meta–functions Σ (which defines the semantics of a program), Γ (which generates the verification conditions) and trm (which generates one termination condition). The programming constructs (statements), the program itself, as well as the terms and the formulae from the object theory are *meta–terms* from the point of view of the meta-theory, and they behave like *quoted* (because the meta–theory does not contain any equalities between programming constructs, and also does not include the object theory).

The programming constructs are: *abrupt statement* (`Return`), *assignments* (allowing *recursive calls*) and *conditionals* (`If` with one and two branches). *Recursive calls* are indicated by the presence of the function symbol f, conventionally corresponding to the function realized by the program. A program is a list of statements, it takes as formal arguments a certain number of variables (denoted conventionally by $\overline{x}$) and it returns a single value (denoted conventionally by y). The meta–predicate Π checks whether a program is syntactically correct, and also that each branch terminates with `Return` and that each variable is initialized before its usage. (We call these variables *active*.)

The semantics of a program is defined via Σ as being an implicit definition *at object level* (that is, using only the signature of the object theory) of the function (conventionally denoted as f) which is implemented by the program. Practically, Σ works by forward symbolic execution on all branches of the program, using as *state* the current substitution for the active variables. Σ produces a conjunction of clauses – conditional definitions for $f[\overline{x}]$. Each clause depends on the accumulated [negated] conditions of the `If` statements leading to a certain `Return` statement, whose argument (symbolically evaluated) represents the corresponding value of $f[\overline{x}]$. Note that Σ effectively translates the original program into a *functional* program. From this point on, one could reason about the program using the Scott fixpoint theory ([5], pag. 86), however we prefer a purely logical approach.

The meta–function Γ produces the verification conditions as a conjunction of formulae at object level. The verification of the program is performed with respect to a given *specification*, composed of two predicates: the input condition $I_f[\overline{x}]$ and the output condition $O_f[\overline{x}, y]$, whose definitions are assumed to be present in the object–theory.

Moreover, some of the functions present in the object theory also have a specification – we call these *additional* functions. These functions will not be present in the verification conditions, but only their specifications. Typical examples of additional functions are those implemented by other programs.

The other functions from the object theory (we call them *basic*) have only input conditions, but no output conditions. These functions will occur in the verification conditions, thus the proof of such conditions will use the properties of the basic functions from the object theory. Typical examples of basic functions are the arithmetic operations in various number domains.

Γ operates similarly to Σ by using symbolic execution on all branches of the program, but in addition generates formulae using the following principles:

- coherence (safety): the arguments of every function (including the currently defined f) call must be satisfy the respective input condition;
- functional correctness: the return value on each branch must satisfy the output condition;

All the verification conditions are first order formulae at object level. This includes the termination condition, because the universally quantified predicate present in the induction principle is represented by a new constant.

The meta-function Θ generates one termination condition: a certain induction principle corresponding to the structure of the recursive calls, which in fact ensures the existence and uniqueness of the function f defined implicitly by Σ.

1.1 Related Work

Our approach follows the principles of the symbolic execution approach introduced in [4], but gives formal definitions in a meta-theory for the for meta-level functions and predicate which characterize the object computation.

These definitions give the possibility of reflective view of the system by describing how the data (the state, the program, the verification conditions) are manipulated and by introducing a causal connection between the data and the status of computation (a certain statement of the program determines a certain computation of $\Sigma[P]$ and Γ to be performed).

We mention that our approach keeps the verification process very simple: the verification conditions generated are first order logic formulae and the proofs of correctness is kept at object-level without introducing a model of computation.

Approaches for solving the correctness of symbolic executed programs exists due to [5, 8, 2]; for the imperative programs containing *assignments*, *conditionals* and `while` *loops bounded* on the number of times they are executed, the proof of correctness is given by analyzing the verification conditions on each branch of the program. For the programs containing loops with unbounded number of iterations, the branches of the program are infinite and have to be traversed inductively in order to prove/disprove their correctness. In the inductive traversal of the tree, additional assertions have to be provided, called *inductive assertions*. But the inductive assertions method applies to partial correctness proofs [5], while our approach concentrates in proving the total correctness of programs.

2 The Formal Meta–Theory

2.1 Program Syntax

The meta–level predicate Π checks whether a program is syntactically correct and additionally that every variable (except the input variables) is used only after being assigned, and that each branch contains `Return`.

As *states* of the execution we use substitutions σ (set of replacements of the form $\{var \rightarrow expr\}$). We sometimes write $\{\overline{var} \rightarrow \overline{expr}\}$ for $\{var_1 \rightarrow expr_1, var_2 \rightarrow expr_2, ...\}$. We denote by $\bar{x}$ the sequence of input variables. The meta–function *Vars* returns the variables which occur in a term.

The formulae composing the meta–definitions below are to be understood as universally quantified over the meta–variables of various types as described in the sequel: We denote by $t \in \mathcal{T}$ a term from the set of object level terms, by $v \in \mathcal{V}$ a variable from the set of variables, by $V \subset \mathcal{V}$ the set of active variables and by φ an object level formula. P_T, P_F, P are tuples of statements representing parts of programs: P_T is the tuple of statements executed if φ is evaluated to $True$, P_F in the case of $False$ evaluation of φ, while P represents the rest of the program to be executed. The tuple P_F can be also empty, case which corresponds to the *one branch* `If` *statement*.

Definition 1

1. $\Pi[P] \Longleftrightarrow \Pi[\{\bar{x}\}, P]$
2. $\Pi[V, \langle \texttt{Return}[t] \rangle \smile P] \Longleftrightarrow Vars[t] \subseteq V$
3. $\Pi[V, \langle v := t \rangle \smile P] \Longleftrightarrow \bigwedge \begin{cases} Vars[t] \subseteq V \\ \Pi[V \cup \{v\}, P] \end{cases}$
4. $\Pi[V, \langle \texttt{If}[\varphi, P_T, P_F] \rangle \smile P] \Longleftrightarrow \bigwedge \begin{cases} Vars[\varphi] \subseteq V \\ \Pi[V, P_T \smile P] \\ \Pi[V, P_F \smile P] \end{cases}$
5. $\Pi[V, P] \Longleftrightarrow \mathbb{F}$, in all other cases

2.2 Program Semantics

The meta–level function Σ creates an object–level formula containing a new function constant f. We consider this formula as being the *semantics* of the program P in the following sense: the function implemented by the program satisfies $\Sigma[P]$.

The formula $\Sigma[P]$ has the shape:

$$\underset{\bar{x}:I_f}{\forall} \bigwedge \left\{ p_i[\bar{x}] \Rightarrow (f[\bar{x}] = g_i[\bar{x}]) \right\}_{i=1}^{n}$$

(We denoted by „$\bar{x} : I_f$" in the condition "$\bar{x}$ satisfies I_f".) Here f is a new (second order) symbol – a name for the function defined by the program. In the case of recursive programs, f may occur in some p_i's and g_i's.

Each of the n paths of the program has associated a object–level formula $p_i[x]$ – the accumulated `If`–conditions on that path, and the object–level term $g_i[x]$ – the symbolic expression of the return value obtained by composing all the assignments (symbolic execution). Note that $p_i[x]$ and $g_i[x]$ do not contain other free variables than x.

Definition 2

1. $\Sigma[P] = \underset{\bar{x}}{\forall}\big(\Sigma[\{\bar{x} \to \bar{x_0}\}, P]_{\{\bar{x_0} \to \bar{x}\}}\big)$
2. $\Sigma[\sigma, \langle \texttt{Return}[t]\rangle \smile P] = (f[\bar{x_0}] = t\sigma)$
3. $\Sigma[\sigma, \langle v := t\rangle \smile P] = \Sigma[\sigma \circ \{v \to t\sigma\}, P]$
4. $\Sigma[\sigma, \langle \texttt{If}[\varphi, P_T, P_F]\rangle \smile P] = \bigwedge \begin{cases} \varphi\sigma \Longrightarrow \Sigma[\sigma, P_T \smile P] \\ \neg\varphi\sigma \Longrightarrow \Sigma[\sigma, P_F \smile P] \end{cases}$

Remark 1. The way Σ handles the `If` statement insures: $\underset{\bar{x}}{\forall}\bigvee\limits_i p_i[\bar{x}] = I_f[\bar{x}]$ (all branches are covered) and $\underset{i \neq j}{\forall} \neg(\underset{\bar{x}}{\forall} p_i[\bar{x}] \wedge p_j[\bar{x}])$ (branches are mutually disjoint).

2.3 Partial Correctness

As we mentioned before, the meta–level function Γ generates two kinds of verification conditions: *coherence (safety)* conditions and *functional* conditions.

In this section we present the first two groups of conditions, which together insure *partial correctness*. The termination condition is subject to next section.

The *coherence* conditions have the shape $\Phi \Rightarrow I_h[t_1, t_2, \ldots]$, where I_h is the input condition of h, and $t_1, t_2, \ldots$ are the symbolic values of the function call. The formula Φ accumulates the conditions on the current branch, namely:

- the conditions from `If` statements;
- the input conditions and the output conditions for the previous function calls.

We consider γ, $\overline{\gamma}$ - a variable or a constant and respectively a sequence of variable and/or constants from the theory Υ, basic functions h, additional functions g, arbitrary function u. The symbol y is a new constant name. An expression like $e_{\tau \leftarrow w}$ denotes that τ is replaced by w in e, where w is a new variable name. The function ,,OccursIn" returns *True* if the first argument appears in the second argument, and *False* otherwise.

Definition 3

1. $\Gamma[P] = \underset{\bar{x}}{\forall}\big(\Gamma[\{\bar{x} \to \bar{x}_0\},\ I_f[\bar{x}_0],\ P]_{\{\bar{x}_0 \to \bar{x}\}}\big)$
2. $\Gamma[\sigma,\ \Phi,\ \langle \texttt{Return}[\gamma]\rangle \smile P] = \Big(\Phi \Rightarrow O_f[\bar{x}_0, \gamma\sigma]\Big)$
3. $\Gamma[\sigma,\ \Phi,\ \langle \texttt{Return}[t_{\tau \leftarrow u[\overline{\gamma}]}]\rangle \smile P] = \Gamma[\sigma,\ \Phi,\ \langle w := u[\overline{\gamma}],\ \texttt{Return}[t_{\tau \leftarrow w}]\rangle \smile P]$
4. $\Gamma[\sigma,\ \Phi,\ \langle v := \gamma\rangle \smile P] = \Gamma[\sigma \circ \{v \to \gamma\sigma\},\ \Phi,\ P]$
5. $\Gamma[\sigma,\ \Phi,\ \langle v := h[\overline{\gamma}]\rangle \smile P] = \bigwedge \begin{cases} \Phi \Rightarrow I_h[\overline{\gamma}\sigma] \\ \Gamma[\sigma \circ \{v \to h[\overline{\gamma}\sigma]\},\ \Phi \wedge I_h[\overline{\gamma}\sigma],\ P] \end{cases}$

6. $\Gamma[\sigma,\, \Phi,\, \langle v := g[\overline{\gamma}]\rangle \smile P] = \bigwedge \left\{ \begin{array}{l} \Phi \Rightarrow I_g[\overline{\gamma}\sigma] \\ \Gamma[\sigma \circ \{v \to c\},\, \Phi \wedge I_g[\overline{\gamma}\sigma] \wedge O_g[\overline{\gamma}\sigma, c],\, P] \end{array} \right.$

7. $\Gamma[\sigma,\, \Phi,\, \langle v := t_{\tau \leftarrow u[\overline{\gamma}]}\rangle \smile P] = \Gamma[\sigma,\, \Phi,\, \langle w := u[\overline{\gamma}],\, v := t_{\tau \leftarrow w}\rangle \smile P]$

8. $\Gamma[\sigma,\, \Phi,\, \langle \mathtt{If}[\varphi_{\tau \leftarrow u[\overline{\gamma}]},\, P_T,\, P_F]\rangle \smile P] =$
$\Gamma[\sigma,\, \Phi,\, \langle w := u[\overline{\gamma}],\, \mathtt{If}[\varphi_{\tau \leftarrow w},\, P_T,\, P_F]\rangle \smile P]$

9. $\Gamma[\sigma,\, \Phi,\, \langle \mathtt{If}[\varphi,\, P_T,\, P_F]\rangle \smile P] = \bigwedge \left\{ \begin{array}{l} \Gamma[\sigma,\, \Phi \wedge \varphi\sigma,\, P_T \smile P] \\ \Gamma[\sigma,\, \Phi \wedge \neg\varphi\sigma,\, P_F \smile P] \end{array} \right.$

The order of the above clauses of Γ has a semantic meaning. Namely, we use this as an abbreviation for additional conditions which should be added to clauses of the definition in order to specify that, for instance, the equality (3.9) is applied only if no subterm of φ is of the form $u[\overline{\gamma}]$ – as specified in the clause (3.8).

2.4 Termination

We want to generate verification conditions which ensure that a program is correct with respect to a specification composed of two object–level formulae: the input condition $I_f[x]$ and the output condition $O_f[x, y]$. Apparently, the correctness could be expressed as: "The formula $\underset{x}{\forall} I_f[x] \Rightarrow O_f[x, P[x]]$ is a logical consequence of the theory Υ augmented with $\Sigma[P]$ and with the verification conditions." However, this always holds in the case that $\Sigma[P]$ is contradictory to Υ, which may happen when the program is recursive. Therefore, *it is crucial that the existence (and possibly the uniqueness) of an f satisfying $\Sigma[P]$ is a logical consequence of the object theory augmented with the verification conditions.* More concretely, before using $\Sigma[P]$ as an assumption, one should prove $\underset{f}{\exists} \Sigma[P]$. The later is ensured by the termination condition which is expressed as an induction scheme developed from the structure of the recursion.

The meta–function Θ generates the termination condition (for simplicity of presentation we assume that `Return`, assignments, and `If` conditions do not contain composite terms – the elimination of these by introducing new assignments can be done as in the definition of Γ).

Definition 4

1. $\Theta[P] = \left(\underset{\bar{x}:I_f}{\forall} \Theta[\{\bar{x} \to \bar{x_0}\},\, \mathbb{T},\, P]_{\{\bar{x}_0 \leftarrow \bar{x}\}} \right) \implies \underset{\bar{x}:I_f}{\forall} \pi[\bar{x}]$

2. $\Theta[\sigma,\, \Phi,\, \langle \mathtt{Return}[\gamma]\rangle \smile P] = \left(\Phi \Rightarrow \pi[\bar{x_0}]\right)$

3. $\Theta[\sigma,\, \Phi,\, \langle v := \gamma\rangle \smile P] = \Theta[\sigma \circ \{v \to \gamma\sigma\},\, \Phi,\, P]$

4. $\Theta[\sigma,\, \Phi,\, \langle v := h[\overline{\gamma}]\rangle \smile P] = \Theta[\sigma \circ \{v \to h[\overline{\gamma}\sigma]\},\, \Phi,\, P]$

5. $\Theta[\sigma,\, \Phi,\, \langle v := f[\overline{\gamma}]\rangle \smile P] = \Theta[\sigma \circ \{v \to y\},\, \Phi \wedge O_f[\overline{\gamma}\sigma, y] \wedge \pi[\overline{\gamma}\sigma],\, P]$

6. $\Theta[\sigma,\, \Phi,\, \langle v := g[\overline{\gamma}]\rangle \smile P] = \Theta[\sigma \circ \{v \to y\},\, \Phi \wedge O_g[\overline{\gamma}\sigma, y],\, P]$

$$7.\ \Theta[\sigma,\ \Phi,\ \langle \texttt{If}[\varphi,\ P_T,\ P_F]\rangle \smile P] = \bigwedge \begin{cases} \Theta[\sigma,\ \Phi \wedge \varphi\sigma,\ P_T \smile P] \\ \Theta[\sigma,\ \Phi \wedge \neg\varphi\sigma,\ P_F \smile P] \end{cases}$$

One single formula is generated, which uses a new constant symbol π standing for an arbitrary predicate. The function Θ operates similarly to Γ by inspecting all possible branches and collecting the respective `If` conditions (in Φ). Moreover it collects the characterizations by output conditions of the values produced by calls to additional functions (4.6), including for the currently defined function f. However, in the case of f (4.5) one also collects the condition $\pi[\overline{\gamma}\sigma]$ – that is the arbitrary predicate applied to the current symbolic values of the arguments of the recursive call to f. On each branch, the collected conditions are used as premise of $\pi[\bar{x}_0]$, and then the conjunction of all these clauses (after reverting to free variables $\bar{x}$) is universally quantified over the input condition and is used as a premise in the final formula.

Example 1 (Primitive recursive functions:). If $\Sigma[P]$ is

$$q[x] \Rightarrow f[x] = s[x], \quad \neg q[x] \Rightarrow f[x] = c[x, f[r[x]]],$$

then the termination condition is:

$$\big(\underset{x:I_f}{\forall}(q[x] \Rightarrow \pi[x]) \wedge ((\neg q[x] \wedge \pi[r[x]]) \Rightarrow \pi[x])\big) \ \Rightarrow \ \underset{x:I_f}{\forall} \pi[x].$$

Note that in this formula π a new symbol standing for an arbitrary predicate. As in illustration, when $I_f[x]$ is "x natural", $q[x]$ is $x = 0$ and $r[x]$ is decrement, then this is the usual induction over natural numbers.

The termination condition is always an implication between two formulae quantified over all x satisfying I_f, with the right-hand side being as above. The left-hand side of the termination condition is a (quantified) conjunction of implications having $f[x]$ on the right-hand side, each implication corresponding to one branch of the program. Namely, if the program branch is represented in $\Sigma[P]$ by $p_i[x] \Rightarrow (f[x] = g_i[x])$, then the corresponding implication in the termination condition is: $(p_i[x] \wedge R_i[x]) \Rightarrow \pi[x]$, where $R_i[x]$ is the conjunction of a number of atoms of the form $\pi[t]$, one for each occurrence of $f[t]$ in $g_i[x]$.

The rule above is applicable only for programs which do not contain nested recursions (that is, terms of the form $f[\ldots, f[\ldots], \ldots]$), and also no occurrences of f in the `If` conditions. Otherwise, f would occur in the termination condition, which we do not want to allow. In this case one can eliminate the un-wanted occurrences of f by using new (universally quantified) variables pre-conditioned by the output condition O_f – as in the function Γ above.

We conjecture that (for terminating programs) the formula $\underset{f}{\exists!}\ \underset{x:I_f}{\forall}\ \Sigma[P]$ is a logical consequence of the the above termination condition. To prove the existence of f appears to be a challenge, however we were able to develop the proof for the general case of primitive recursive functions, as well as for some particular examples with nested occurrences of P (the McCarthy 91 function, matching of expressions).

The uniqueness is a straightforward consequence of the termination condition, by considering $\pi[x]$ as $f_1[x] = f_2[x]$, where f_1 and f_2 are two functions. The property π holds for $q[x]$ and by using the termination condition we prove that holds also in the case $\neg q[x]$. Therefore π holds for all x.

Furthermore, if we assume the existence of f, then, by setting $\pi[x]$ to $I_f[x] \Rightarrow O_f[x, f[x]]$, one obtains in a straightforward way a proof of the total correctness formula $\underset{x}{\forall} I_P[x] \Rightarrow O_P[x, P[x]]$ from the other verification conditions.

3 Implementation and Example

Our prototype implementation (`FwdVCG`) is based on the theoretical aspects presented in the previous sections. It is built on top of the computer algebra system *Mathematica* and uses *Theorema* building blocks and procedural language for writing imperative programs.

Example 2. We illustrate with the following code fragment:

```
Program["GCD", GCD[↓ a, ↓ b]],
 Module[{}],
  If[a = 0,
   Return[b]];
  If[b ! = 0,
   If[a > b,
    a := GCD[a - b, b],
    a := GCD[a, b - a]]];
 Return[a]],
Pre → a ≥ 0 ∧ b ≥ 0,
Post→ ∃k1 ∃k2 ((a = k1 ∗ y) ∧(a = k2 ∗ y))
```

The automatically generated verification conditions for the previous example are the following:

$$(a \geq 0 \wedge b \geq 0) \wedge (a = 0) \Rightarrow \underset{k1}{\exists}\,\underset{k2}{\exists}((a = k1 * b) \wedge (b = k2 * b)) \tag{1}$$

$$(a \geq 0 \wedge b \geq 0) \wedge a \neq 0 \wedge b \neq 0 \wedge a > b \Rightarrow a \geq b \tag{2}$$

$$(a \geq 0 \wedge b \geq 0) \wedge a \neq 0 \wedge b \neq 0 \wedge a > b \wedge a \geq b \Rightarrow a - b \geq 0 \wedge b \geq 0 \tag{3}$$

$$(a \geq 0 \wedge b \geq 0) \wedge a \neq 0 \wedge b \neq 0 \wedge a > b \wedge a \geq b \wedge (a - b \geq 0 \wedge b \geq 0) \wedge \underset{k1}{\exists}\,\underset{k2}{\exists}((a - b = k1 * \mathrm{y1}) \wedge (b = k2 * \mathrm{y1})) \Rightarrow \underset{k1}{\exists}\,\underset{k2}{\exists}((a = k1 * \mathrm{y1}) \wedge (b = k2 * \mathrm{y1})) \tag{4}$$

$$(a \geq 0 \wedge b \geq 0) \wedge a \neq 0 \wedge b \neq 0 \wedge a \not> b \Rightarrow a \geq b \tag{5}$$

$$(a \geq 0 \wedge b \geq 0) \wedge a \neq 0 \wedge b \neq 0 \wedge a \not> b \wedge a \geq b \Rightarrow a \geq 0 \wedge b - a \geq 0 \tag{6}$$

$$(a \geq 0 \wedge b \geq 0) \wedge a \neq 0 \wedge b \neq 0 \wedge a \not> b \wedge a \geq b \wedge (a \geq 0 \wedge b - a \geq 0) \wedge \underset{k1}{\exists}\,\underset{k2}{\exists}((a = k1 * \mathrm{y2}) \wedge (b - a = k2 * \mathrm{y2})) \Rightarrow \underset{k1}{\exists}\,\underset{k2}{\exists}((a = k1 * \mathrm{y2}) \wedge (b = k2 * \mathrm{y2})) \tag{7}$$

$$(a \geq 0 \wedge b \geq 0) \wedge a \neq 0 \wedge \neg(b \neq 0) \Rightarrow \underset{k1}{\exists}\,\underset{k2}{\exists}((a = k1 * a) \wedge (b = k2 * a)) \tag{8}$$

Remark 2. 1. Each verification condition is universally quantified; the bound variables are: $a, b, y1, y2$;
2. The function GCD can not occur in the verification conditions thus their occurrences are replaced by the variables $y1$ and $y2$ (e.g. (4), (7));
3. The input condition of the basic function ,,-"has to be fulfilled thus verification conditions are generated at this aim (e.g. (2), (5)).
4. The program output has the expressions: b (in (1)), $y1$ (in (4)), $y2$ (in (7)), a (in (8)).

The termination condition is:

$$\Big(\underset{\substack{a,b \\ a\geq 0, b\geq 0}}{\forall} \bigwedge \begin{cases} a = 0 \Rightarrow \pi[a,b] \\ (a \neq 0 \wedge b \neq 0 \wedge a > b \wedge \pi[a-b,b]) \Rightarrow \pi[a,b] \\ (a \neq 0 \wedge b \neq 0 \wedge a \neq b \wedge \pi[a,b-a]) \Rightarrow \pi[a,b] \\ (a \neq 0 \wedge b = 0) \Rightarrow \pi[a,b] \end{cases} \Big) \Longrightarrow \Big(\underset{\substack{a,b \\ a\geq 0, b\geq 0}}{\forall} \pi[a,b] \Big)$$

4 Conclusion and Future Work

The method presented in this paper combines forward symbolic execution and functional semantics for generating the verification conditions necessary for checking the imperative program correctness. We implemented it in the *Theorema* system and we tested it on programs written in our mini-programming language.

For checking the validity of the verification conditions we built a prototype of a simplifier that reduces the verification conditions to (systems of) equalities and inequalities. The simplified formulae were obtained using first order logic inference rules, truth constants and simple algebraic simplifications.

As future work we want to apply and automate advanced algebraic and combinatorial algorithms and methods in order to check their validity. A promising approach seems to be the Fourier-Motzkin Elimination method ([7]), especially for systems of inequalities with small number of constraints and variables because of the time complexity reasons.

References

1. D. Coward, *Symbolic execution systems – a review*, Softw. Eng. J. **3** (1988), no. 6, 229–239.
2. L. Deutsch, *An interactive program verifier*, Ph.D. thesis, University of California - Berkley, USA, 1973.
3. G. Dromey, *Program derivation: the development of programs from specifications*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.
4. J. King, *A new approach to program testing*, Proceedings of the international conference on Reliable software (New York, NY, USA), ACM Press, 1975, pp. 228–233.

5. J. Loeckx, K. Sieber, and R. Stansifer, *The foundations of program verification*, John Wiley & Sons, Inc., New York, NY, USA, 1984.
6. J. McCarthy, *A Basis for a Mathematical Theory of Computation*, Computer Programming and Formal Systems (P. Braffort and D. Hirschberg, eds.), North-Holland, Amsterdam, 1963, pp. 33–70.
7. A. Schrijver, *Theory of linear and integer programming*, John Wiley & Sons, Inc., New York, NY, USA, 1986.
8. R. Topor, *Interactive Program Verification using Virtual Programs*, Ph.D. thesis, University of Edinburgh, Scotland, 1975.

Computational Origami of Angle Quintisection

Fadoua Ghourabi, Tetsuo Ida, and Hidekazu Takahashi *

Department of Computer Science, University of Tsukuba,
Tsukuba 305-8573, Japan
{ghourabi, ida, hidekazu}@score.cs.tsukuba.ac.jp

Abstract. Computational Origami is a branch of the science of shapes, where we study computational and mathematical aspects of origami. One of the foundational studies of the computational origami is the axiomatic definition of origami foldability by Huzita in 1989. While Huzita's axioms allow solving equations up to degree 4, it is possible to use other techniques of origami to solve higher order equations. Multifold is a fold operation that involves the creation of more than one crease. The division of an angle into five equal angles requires multifold operations to solve a fifth degree equation known as quintisection. We discuss the construction and the proof of an angle quintisection using a logical and algebraic representation of the fold operation.

1 Introduction

Origami is a traditional Japanese art of paper fold. Starting from a square piece of paper and using only hands, origamists can realize impressive paper works. Origami is not only an art but also a geometric tool for constructing geometric shapes. Now there is computational Origami, the computer assisted study of origami geometry. The field of computational origami come up with further analysis of folding patterns. Computational origami may answer questions such as: "Given an origami paper, how can we fold it to obtain specific creases and points?" or "Can we prove some properties of the origami object that we made?".

One of the fundamental studies of the computational origami is the axiomatic definition of origami foldability by Huzita in 1989 [1]. Huzita found out that there is a finite number of basic folds, where a basic fold consists in making one and only one crease with one fold operation. Given existing lines and points in origami paper, each one of Huzita's six basic folds realizes the creation of a new line which is the fold line. These basic folds are known in literature as Huzita's axioms [2, 3]. Later, a seventh axiom was proposed by Hatori [2]. We call the set of seven axioms Huzita's axioms.

The advantage that origami has over compass and straightedge lies in the set of points that can be constructed [4]. Compass and straightedge can construct only numbers that are solutions of equations of degree of up to 2. On the other

* This research is supported by the JSPS Grants-in-Aid for Exploratory Research No. 19650001 and by Japan-Austria Research Cooperative Program of JSPS and FWF.

hand, Huzita's axioms define what is constructable by making sequential folds along one and only one line at each fold step. It has been mathematically proven that those folds permit the construction of solutions to arbitrary equations of up to degree 4 [4]. This relevant result allows the construction of famous impossibilities, namely doubling a cube, squaring a circle and trisecting an angle. However, by other computational origami techniques such as multifold, it is possible to solve equations of higher degree. Multifold is a fold operation that involves the creation of more than one creases [5].

Symbolic COmputational REsearch Group (SCORE) has developed a computational origami system called *Eos* [6]. In brief, *Eos* has capabilities of offering a fold methodology, constructing and visualizing origami objects, algebraically analyzing origami folds, and proving the correctness of origami constructions. In this paper, we are mostly concerned with the practical aspect of computational origami, in particular the usage of computational origami results towards constructing and proving geometric objects realized with origami as a branch of geometry. We focus on the division of an angle into five equal angles. This problem requires multifold operations to perform the construction.

The rest of the paper is organized as follows. In Sect. 2 we explain elements of the logical specification of the axioms together with a translation into algebraic form. In Sect. 3 we show an example of quintisecting an angle by multifolds using *Eos*. In Sect. 4 we explain how *Eos* can be used to verify the correctness of origami constructs based on Gröbner bases method. In Sect. 5 we summarize our results and point out a direction of further research.

2 Formalization of Computational Origami Method

Huzita's basic folds are considered as an axiomatization effort of origami construction [1, 7]. In this paper we attempt to present the elements of our logical and algebraic formalization of Huzita's axioms.

2.1 Huzita's Axioms Restated

Logical Representation of Huzita's Axioms In an attempt to better understand the elements of Huzita axiom system and their semantics, we introduce a formal language. We define a many sorted first-order language $\mathcal{L}$ over a signature $(\mathcal{P}, \mathcal{F})$ consisting of a set $\mathcal{P}$ of predicate symbols and a set $\mathcal{F}$ of function symbols. We consider the set of sorts $\mathcal{S} = \{\text{Point}, \text{Line}, \mathbb{A}, \text{B}\}$. Point and Line are the sorts of points and lines, respectively. B is the sort of Boolean values and $\mathbb{A}$ is the sort of algebraic numbers over $\mathbb{Q}$. By adding the set membership $P \in \text{Point}$ and $l \in \text{Line}$ to our syntax, we can specify the types of variables.

We also define predicates and functions that constitute $\mathcal{L}$. These symbols have a meaning in algebraic geometry and rational trigonometry. Each axiom is then described as a prenex normal form formula in $\mathcal{L}$ [8]. For instance (O3) the third axiom asserts that given two lines m and n, we can find a line k such that

the fold along k brings m onto n. (O3) can be written as follows

$$\forall m, n \in \text{Line } \exists k \in \text{Line } \forall P \in \text{Point lineReflection}[m,\ k] == n \tag{1}$$

Formula (1) tells that we can find a line k such that the image of m with respect to k is n. The function lineReflection computes the image of a line by line symmetry.

Algebraic Interpretation of Huzita's Axioms The language $\mathcal{L}$ is given an algebraic interpretation. Huzita's axioms are translated into algebraic forms. We work in the Cartesian coordinate system. We study the origami construction in the environment of plane geometry. Thus, we consider an environment ρ that maps points to pairs of their coordinates, and lines to triples of their coefficients. Furthermore, let $\mathcal{A}[\![\cdot]\!]\rho$ be the mapping from symbols of $\mathcal{L}$ to the interpretation of $\mathcal{L}$. $\mathcal{A}[\![\cdot]\!]\rho$ maps the formulas of prenex normal form of Huzita's axioms into a system of polynomial equalities with rational coefficients.

Suppose we apply (O3) to bring m onto n by folding along line k. Let $\rho(m) = \langle a_1, b_1, c_1\rangle$, $\rho(n) = \langle a_2, b_2, c_2\rangle$, and assume that $\rho(k) = \langle a, b, c\rangle$. Let ϕ be the formula in (1). Then the algebraic interpretation of ϕ is

$$\begin{aligned}\mathcal{A}[\![\phi]\!]_{\rho\cup\{k\mapsto(a,b,c)\}} = \{&-a^2\mathrm{a1} + \mathrm{a1}b^2 - 2ab\mathrm{b1} - \mathrm{a2}t = 0,\\ &-2a\mathrm{a1}b + a^2\mathrm{b1} - b^2\mathrm{b1} - \mathrm{b2}t = 0, -2a\mathrm{a1}b + a^2\mathrm{b1} - b^2\mathrm{b1} - \mathrm{c2}t = 0,\\ &\xi t - 1 = 0\}\end{aligned} \tag{2}$$

The set (2) represents algebraic constraints on the coefficients of m, n and k. When we apply $\mathcal{A}[\![.]\!]_\rho$, the universal quantifiers over m and n are eliminated since m and n are well defined on the origami paper. Unknown fold line k is existentially quantified in ϕ. We eliminate the existential quantifier by adding the unknown coefficients a, b and c of k to ρ. Line

$$(-a^2\mathrm{a1} + \mathrm{a1}b^2 - 2ab\mathrm{b1}, -2a\mathrm{a1}b + a^2\mathrm{b1} - b^2\mathrm{b1}, -2a\mathrm{a1}b + a^2\mathrm{b1} - b^2\mathrm{b1}) \tag{3}$$

is the image of m with respect to k [9]. Line n and the line defined in (3) are equivalent which explains the introduction of new non zero variable t. The equation $\xi t - 1 = 0$ in (2) tells that $t \neq 0$ (Rabinovich trick).

The algebraic interpretation of $\mathcal{L}$ is useful for achieving the origami construction by a computer. That is, origami construction consists mainly in folding along a line that satisfies some constraints. Having described the constraints in term of polynomial equalities, our next task is to obtain fold line is constraint solving. Specifically, we solve $\mathcal{A}[\![\phi]\!]_{\rho\cup\{k\mapsto\langle a,b,c\rangle\}}$ which is a set of polynomial equalities where a, b and c are the unknown variables. We need also to eliminate null lines. Therefore, we add to the set of equations constraints on lines coefficients [8]. We designed a user interface `HFold` for origami construction. This interface is an implementation of the language $\mathcal{L}$ and the interpretation mapping $\mathcal{A}[\![.]\!]_\rho$.

2.2 Multifold Origami Method

Multifold operation consists in making folds along more than one line. Alperin and Lang proposed a definition of multifold operations [4]. They noticed that Huzita's axioms can be viewed as combination of three alignments: (i) Point-point alignment specifies that two points are equal, (ii) Line-line alignment tells that two lines are equivalent, and finally, (iii) Point-line alignment tells that a given point is on a given line. The fold alignment is a fold operation which is combination of (i), (ii) and (iii). For example, (O3) can be viewed as line-line alignment. In the case of one fold operation, the fold line that satisfies the alignments is the same. In the case of multifold, we can have more than one fold lines to satisfies the alignments.

We extend the logical and algebraic formalism to describe multifold operations. We are able to call

$$\texttt{HFold}\ [\ \langle\text{handle}\rangle,\ \texttt{Constraint} \rightarrow \langle\text{formula}\rangle,\ \langle\text{options}\rangle\ ]$$

to specify a multifold. Multifold satisfies geometric properties specified by the argument Constraint $\rightarrow \langle$formula$\rangle$ of `HFold`. We explain the usage of `HFold` is Sect. 3.2.

3 Quintisection of an Angle

Dividing an angle into n equal part is a relevant problem when n is an odd natural number. In this paper, we discuss the quintisection construction by computationally supported paper folds. First, we study the problem of quintisection in Euclidean geometry.

3.1 Geometric Problem of Quintisection

Euclidean geometry is based on a set of axioms used to create geometric shapes. Generating shapes involves the use of two tools: a straightedge and a compass. Euclidian geometric construction problem is reduced to the problem of constructing numbers that represent lengths of line segments. A number α is said to be constructible if given a segment of unity length, we can construct a segment of length $|\alpha|$ using straightedge and compass. There is a well known theorem in abstract geometry that asserts that if a number α is constructible by Euclidean tools then α is algebraic over $\mathbb{Q}$ and the degree of the irreducible polynomial of α over $\mathbb{Q}$ is a power of 2 [10].

Quintisection is the problem of dividing an arbitrary angle into five equal parts. The problem of quintisection is not always possible using Euclidean classical tools straightedge and compass. We can find angles that cannot be quintisected. For instance, we take an angle whose tangent is equal to $T = \sqrt{\frac{2}{1+\sqrt{5}}}$. In Fig. 1, T is the tangent of the angle θ. Moreover, we have $T^4+T^2-1 = 0$. Thus, θ can be constructed since $\tan\theta$ is a constructible number (zero of irreducible

polynomial whose degree is power of 2). Now, if $t = \tan\frac{\theta}{5}$ is constructible then the degree of its irreducible polynomial over $\mathbb{Q}$ is a power of 2. Thus, if the irreducible polynomial of t is not of degree 2 than it follows that t is not constructible. In trigonometry we have the following result

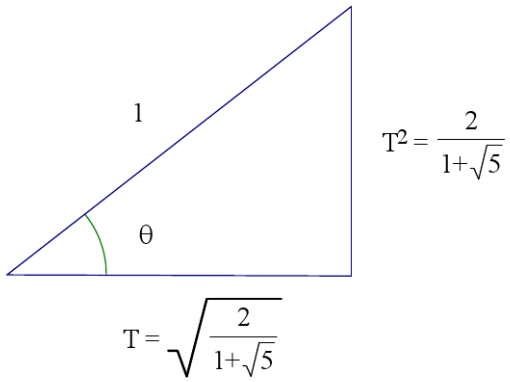


Fig. 1. $\tan\theta = \mathrm{T} = \sqrt{\frac{2}{1+\sqrt{5}}}$

$$T = t\frac{5-10t^2+t^4}{1-10t^2+5t^4} \tag{4}$$

Equation (4) shows that t is a solution of polynomial of degree 5. If we substitute $\sqrt{\frac{2}{1+\sqrt{5}}}$ for T and perform few mathematical manipulations, we obtain

$$(2(1-10t^2+5t^4)\text{^}2 - t^2(5-10t^2+t^4)^2)^2 - 5(t^2(5-10t^2+t^4)^2)^2 = 0$$

After expanding and substituting t^2 by x, we obtain the following polynomial

$$P(x) = -4(-1+65x-595x^2+4460x^3-16370x^4+23126x^5-15070x^6+5260x^7 - 605x^8 - 15x^9 + x^{10})$$

To prove that the monomial polynomial $-\frac{1}{4}P(x)$ is irreducible, we use Eisenstein's irreducibility criterion [10]. First, we substitute $x+2$ for x, we obtain

$$Q(x) = 58805 + 191525x + 259525x^2 + 183500x^3 + 58850x^4 - 9090x^5 - 15910x^6 - 5620x^7 - 695x^8 + 5x^9 + x^{10}$$

Q(x) is irreducible since the prime number 5 is a factor of the coefficients of x^0, x, x^2, x^3, x^5, x^6, x^7, x^8 and x^9 and is not a factor of the coefficient of x^10. Moreover, 5^2 is not a factor of the coefficient of x^0. All the conditions of Eisenstein's irreducibility criterion are verified which make $Q(x)$ an irreducible

polynomial. Then, $Q(t^2)$ is the irreducible polynomial with $\tan \frac{\theta}{5}$ as zero and whose degree is not a power of 2. Therefore, $\tan \frac{\theta}{5}$ is not constructible by compass and straightedge.

On the other hand, quintisection can be realized by computationally supported paper folds. In the following subsection, we show a method of quintisecting an angle using origami techniques.

3.2 Computational Origami of Quintisection

We first assume that the initial origami is a square paper $\square ABCD$. Let E be an arbitrary point on segment $\overline{CD}$[1]. We will consider only the case where $E \in \overline{CD}$ for the simplicity and clarity of explanation.

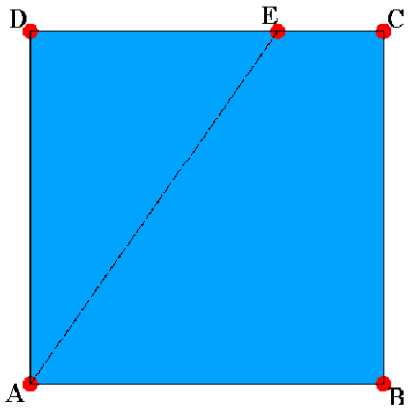


Fig. 2. Computational origami construction of $\angle EAB$

Next, is the crucial step of quintisection. We apply four simultaneous line-line alignments (axioms (O3)). To quintisect $\angle EAB$, we need to find four fold lines x, y, z and t. x is the bisector of the angle between lines AB and y. In other words, x brings AB onto y. Line y is the bisector of the angle between lines x and z. Line z is the bisector of the angle between lines y and t. Finally, t is the bisector of the angle between lines AE and z. The fold operation should satisfy the following formula in $\mathcal{L}$

$$\begin{aligned} \exists x, y, z, t \in \text{Line} \quad & \text{BringLineQ}[z, \text{AE}, t] \wedge \text{BringLineQ}[y, t, z] \wedge \\ & \text{BringLineQ}[x, z, y] \wedge \text{BringLineQ}[\text{AB}, y, x] \end{aligned}$$

BringLineQ$[x, z, y]$ tells that y is the fold line that brings x onto z. Thus, we

[1] $\overline{XY}$ refers to the line segment defined by the points X and Y. XY refers to the line obtained by extending the line segment $\overline{XY}$.

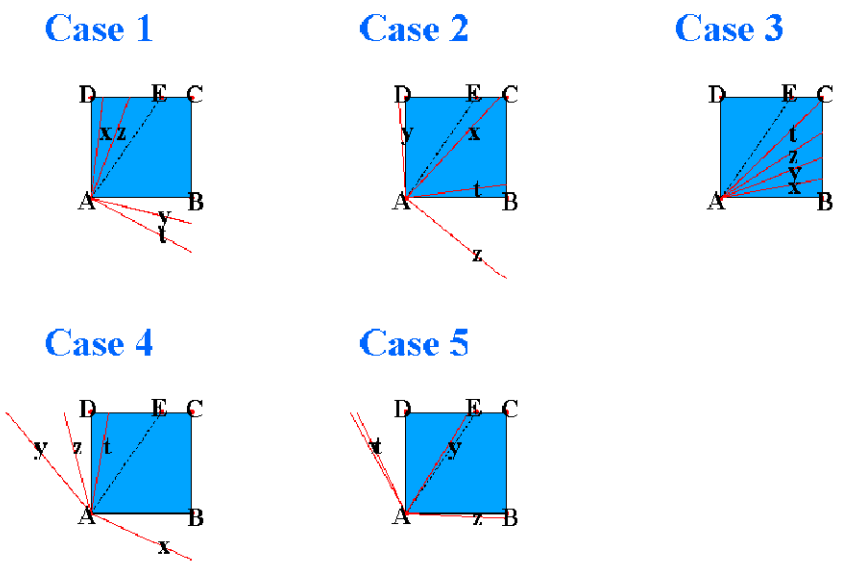


Fig. 3. Five possible quintisections

have BringLineQ$[x, z, y] = (\ \text{lineReflection}[x, z] == y)$. These constraints on the four fold lines are specified in the following call of `HFold`.

$$\begin{aligned}&\texttt{HFold}[\{\texttt{B},\texttt{F},\texttt{B},\texttt{F}\},\texttt{Constraint} \rightarrow \texttt{Exists}[\{\texttt{x},\texttt{y},\texttt{z},\texttt{t}\},\\ &\quad \{\texttt{x} \in \texttt{Line},\texttt{y} \in \texttt{Line},\texttt{z} \in \texttt{Line},\texttt{t} \in \texttt{Line}\},\\ &\quad \texttt{BringLineQ}[\texttt{z},\texttt{AE},\texttt{t}] \wedge \texttt{BringLineQ}[\texttt{y},\texttt{t},\texttt{z}] \wedge \texttt{BringLineQ}[\texttt{x},\texttt{z},\texttt{y}] \wedge\\ &\quad \texttt{BringLineQ}[\text{AB},\texttt{y},\texttt{x}]]]\end{aligned}$$

The algebraic interpretation of the constraints of the fold lines is shown in Fig. 4.

In Fig. 4, ($a2$, $b2$, $c2$) are the coefficients of x, ($a3$, $b3$, $c3$) are the coefficients of y, ($a4$, $b4$, $c4$) are the coefficients of z and ($a5$, $b5$, $c5$) are the coefficients of t. In Sect. 2.1, we have explained the algebraic interpretation of axiom (O3). If we examine the polynomials appearing in Fig. 4, we can identify the constraints on the fold lines x, y, z and t. For instance, the first three equations represent the algebraic interpretation of BringLineQ$[\texttt{y},\texttt{t},\texttt{z}]$ (see the algebraic interpretation of (O3) in Sect. 2.1).

The first parameter of `HFold` is the list of handles. Points B, F, B and F are handles of x, y, z and t, respectively. An origami paper is modeled as stack of faces which are convex polygons created by folding [11]. The origami has a complex structure so that it is crucial to identify the faces that are affected by the fold. In our example, all the faces containing point B are to be moved when we fold along x and z. Besides, all the faces containing point F are to be moved

$$1.42857\mathtt{a5}^2 - 2\mathtt{a5b5} - 1.42857\mathtt{b5}^2 - \mathtt{a4}\kappa 1 == 0 \&\& \mathtt{a5}^2 + 2.85714\mathtt{a5b5} - \mathtt{b5}^2 - \mathtt{b4}\kappa 1 == 0 \&\&$$
$$0.\mathtt{a5}^2 + 0.\mathtt{b5}^2 + 2.85714\mathtt{a5c5} - 2\mathtt{b5c5} - \mathtt{c4}\kappa 1 == 0 \&\& -1 + \kappa 1\xi 1 == 0 \&\&$$
$$-\mathtt{a4}^2\mathtt{a5} + \mathtt{a5b4}^2 - 2\mathtt{a4b4b5} - \mathtt{a3}\kappa 2 == 0 \&\& -2\mathtt{a4a5b4} + \mathtt{a4}^2\mathtt{b5} - \mathtt{b4}^2\mathtt{b5} - \mathtt{b3}\kappa 2 == 0 \&\&$$
$$-2\mathtt{a4a5c4} - 2\mathtt{b4b5c4} + \mathtt{a4}^2\mathtt{c5} + \mathtt{b4}^2\mathtt{c5} - \mathtt{c3}\kappa 2 == 0 \&\& -1 + \kappa 2\xi 2 == 0 \&\&$$
$$-\mathtt{a3}^2\mathtt{a4} + \mathtt{a4b3}^2 - 2\mathtt{a3b3b4} - \mathtt{a2}\kappa 3 == 0 \&\& -2\mathtt{a3a4b3} + \mathtt{a3}^2\mathtt{b4} - \mathtt{b3}^2\mathtt{b4} - \mathtt{b2}\kappa 3 == 0 \&\&$$
$$-2\mathtt{a3a4c3} - 2\mathtt{b3b4c3} + \mathtt{a3}^2\mathtt{c4} + \mathtt{b3}^2\mathtt{c4} - \mathtt{c2}\kappa 3 == 0 \&\& -1 + \kappa 3\xi\ \ 3 == 0 \&\&$$
$$-\mathtt{a2}^2\mathtt{a3} + \mathtt{a3b2}^2 - 2\mathtt{a2b2b3} + 0.\kappa 4 == 0 \&\& -2\mathtt{a2a3b2} + \mathtt{a2}^2\mathtt{b3} - \mathtt{b2}^2\mathtt{b3} - \kappa 4 == 0 \&\&$$
$$-2\mathtt{a2a3c2} - 2\mathtt{b2b3c2} + \mathtt{a2}^2\mathtt{c3} + \mathtt{b2}^2\mathtt{c3} + 0.\kappa 4 == 0 \&\& -1 + \kappa 4\xi 4 == 0 \&\&$$
$$(-1 + \mathtt{b5})\mathtt{b5} == 0 \&\& (-1 + \mathtt{a5})(-1 + \mathtt{b5}) == 0 \&\& (-1 + \mathtt{b4})\mathtt{b4} == 0 \&\&$$
$$(-1 + \mathtt{a4})(-1 + \mathtt{b4}) == 0 \&\& (-1 + \mathtt{b3})\mathtt{b3} == 0 \&\& (-1 + \mathtt{a3})(-1 + \mathtt{b3}) == 0 \&\&$$
$$(-1 + \mathtt{b2})\mathtt{b2} == 0 \&\& (-1 + \mathtt{a2})(-1 + \mathtt{b2}) == 0$$

Fig. 4. Polynomials generated by the call of HFold for performing quintisection

when we fold along y and t. There are five solutions that are depicted in Fig. 3. Cases 1, 2, 3, 4 and 5 are quintisections of the angles $2\pi + \measuredangle EAB$, $\pi + \measuredangle EAB$, $\measuredangle EAB$, $4\pi + \measuredangle EAB$ and $3\pi + \measuredangle EAB$, respectively. Since we are interested in quintisecting the internal angle $\angle EAB$, case 3 has to be selected.

Next, we quintisect the internal angle $\angle EAB$ by selecting case 3 of Fig. 3. We obtain a multifolded origami of Fig. 5. The new points F, G, J and I are intersections between lines BC and x, y, z and t, respectively. The final origami is shown in Fig. 6. Four new lines are created, which are the four quintisectors of $\angle EAB$. After performing the quintisection, we need to prove that lines AF, AG, AI and AJ are quintisectors of $\angle EAB$. This will be the topic of next section.

4 Proof of Correctness of Quintisection

In computational origami, the work of the origamist is not completed just by presenting the construction. The construction has to be accompanied by a correctness proof.

4.1 Property to Prove

We will prove the following theorem.

Theorem 41 *Given the origami in Fig. 6, if*

(a) $\measuredangle GAB = 2\measuredangle FAB \ (mod\ 2\pi)$, *and*
(b) $\measuredangle IAB = 3\measuredangle FAB \ (mod\ 2\pi)$, *and*
(c) $\measuredangle JAB = 4\measuredangle FAB \ (mod\ 2\pi)$, *and*

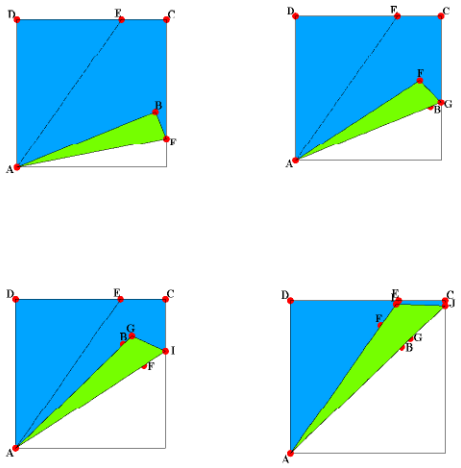


Fig. 5. Performing quintisection at step 4

(d) $\measuredangle EAB = 5\measuredangle FAB \ (mod\ 2\pi)$

then lines AF*,* AG*,* AI *and* AJ *are quintisectors of* $\angle EAB$*.*

We will prove theorem 41 by showing that

$$(\tan 2\measuredangle FAB = \tan \measuredangle GAB) \wedge \tag{5}$$
$$(\tan 3\measuredangle FAB = \tan \measuredangle IAB) \wedge \tag{6}$$
$$(\tan 4\measuredangle FAB = \tan \measuredangle JAB) \wedge \tag{7}$$
$$(\tan 5\measuredangle FAB = \tan \measuredangle EAB) \tag{8}$$

Let T be equal to $\tan \measuredangle FAB$. Furthermore, let $T1$, $T2$, $T3$ and $T4$ be equal to $\tan \measuredangle GAB$, $\tan \measuredangle IAB$, $\tan \measuredangle JAB$ and $\tan \measuredangle EAB$, respectively. Thus, based on well known results in trigonometry, equations (5), (6), (7) and (8) are equivalent to (9), (10), (11) and (12), respectively.

$$(-\frac{2\mathrm{T}}{-1+\mathrm{T}^2} == \mathrm{T1}) \wedge \tag{9}$$
$$(\frac{\mathrm{T}\left(-3+\mathrm{T}^2\right)}{-1+3\mathrm{T}^2} == \mathrm{T2}) \wedge \tag{10}$$
$$(-\frac{4\mathrm{T}\left(-1+\mathrm{T}^2\right)}{1-6\mathrm{T}^2+\mathrm{T}^4} == \mathrm{T3}) \wedge \tag{11}$$
$$(\frac{\mathrm{T}\left(5-10\mathrm{T}^2+\mathrm{T}^4\right)}{1-10\mathrm{T}^2+5\mathrm{T}^4} == \mathrm{T4}) \tag{12}$$

What follows is the illustration of how we can realize the proof of (9), (10), (11) and (12) with *Eos*.

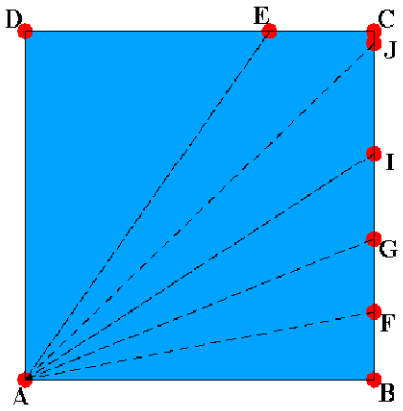


Fig. 6. AF, AG, AI and AJ are quintisectors of $\angle EAB$

4.2 Proof by *Eos*

Besides origami construction, *Eos* also provides means to prove geometric properties of the construction. *Eos* records the geometric properties of all points and fold lines used during the construction, as first-order logic formulas of $\mathcal{L}$. These properties form the premises. Note that we use $\mathcal{P}$ and $\mathcal{C}$ to refer to the premises and the conclusion of the proof, respectively.

The next step is to formulate the property that we want to prove, also as a formula of $\mathcal{L}$. This property forms the conclusion of the proof. The conclusion $\mathcal{C}$ can be written as $\mathcal{C}_1 \wedge \mathcal{C}_2 \wedge \mathcal{C}_3 \wedge \mathcal{C}_4$, where $\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3, \mathcal{C}_4$ are equivalent to (9), (10), (11) and (12), respectively. Thus we need to prove the following implication

$$\mathcal{P} \Rightarrow \mathcal{C}_1 \wedge \mathcal{C}_2 \wedge \mathcal{C}_3 \wedge \mathcal{C}_4 \text{ is valid}$$

$$\Updownarrow$$

$$\mathcal{P} \wedge \neg(\mathcal{C}_1 \wedge \mathcal{C}_2 \wedge \mathcal{C}_3 \wedge \mathcal{C}_4) \text{ is not satisfied}$$

$$\Updownarrow$$

$$(\mathcal{P} \wedge \neg\mathcal{C}_1) \vee (\mathcal{P} \wedge \neg\mathcal{C}_2) \vee (\mathcal{P} \wedge \neg\mathcal{C}_3) \vee (\mathcal{P} \wedge \neg\mathcal{C}_4) \text{ is not satisfied}$$

Eos translates each of the formulas $\phi_i \equiv \mathcal{P} \wedge \neg\mathcal{C}_i$ for $i = 1, \ldots, 4$ into polynomial form with the translation function $\mathcal{A}$ described in Sect. 2.1. In our example, the outcome is a set of polynomial equations $\{f_i ==_i 0 \mid 1 \leq i \leq n\}$ where f_i are polynomials for $i = 1, \ldots, n$. Here, the negation of $\mathcal{C}_i$ is turned into equalities by using Rabinovich trick. Without loss of generality, we consider an initial origami $\square ABCD$ with length equal to 1. Besides, we consider a Cartesian coordinates system with point A being the origin and point B with coordinates $(1, 0)$. E is an arbitrary point on origami paper whose coordinates are (u, v).

Only equalities are generated by the algebraic translation in the proof of quintisection. Therefore, we make use of the following well known results [12, 13]: Let $\phi_i \equiv \mathcal{P} \wedge \neg\mathcal{C}_i$. The formula ϕ_i is not satisfied iff the system of polynomial equations $\mathcal{A}[\![\mathcal{P} \wedge \neg\mathcal{C}_i]\!]_\rho$ is unsolvable iff the reduced Gröbner bases of $\mathcal{A}[\![\mathcal{P} \wedge \neg\mathcal{C}_i]\!]_\rho$ is $\{1\}$.

The computation of the Gröbner bases is performed in the field of multivariate polynomials whose coefficients are in $\mathbb{Q}(u, v)$. With degree reverse lexicographic ordering, the computation of the *Mathematica 6*'s implementation of Gröbner bases of $\mathcal{A}[\![\mathcal{P} \wedge \neg\mathcal{C}_1]\!]_\rho$, $\mathcal{A}[\![\mathcal{P} \wedge \neg\mathcal{C}_2]\!]_\rho$, $\mathcal{A}[\![\mathcal{P} \wedge \neg\mathcal{C}_3]\!]_\rho$ and $\mathcal{A}[\![\mathcal{P} \wedge \neg\mathcal{C}_4]\!]_\rho$ took 14.078 seconds, 16.797 seconds, 22.656 seconds and 21.484 seconds, respectively. We made these computation on Microsoft XP notebook with 1.6 GHz Intel Pentium M processor and 1 GB RAM.

5 Conclusion

Quintisection of an angle is a relevant construction problem that goes beyond Euclidean geometry. We have realized the quintisection of an angle by origami multifold. Besides, we have proven the correctness of the construction based on Gröbner bases. Our approach is to extend the usage of the formalisms ($\mathcal{L}$ and $\mathcal{A}$) of Huzita's axioms to deal with multifold. We have used *Eos* which supports the whole process of construction and proof.

To perform quintisection we based on multifold operation. The construct is clear and simple. However, it is interesting to examine the quintisection problem in Huzita's axiomatization system. A mathematical proof of impossibility or possibility of quintisection by Huzita's axioms has to be studied.

Although the geometric proofs based on the algebraic specification of geometric problems are simple and efficient, they do not necessarily reflect the geometric nature of our construction. It is difficult to give a geometric interpretation of polynomials generated by the construction and proof steps. Logical inferences based on the logical view of origami construction can be augmented to the algebraically based proof method.

References

1. H. Huzita. Axiomatic Development of Origami Geometry. In H. Huzita, editor, *Proceedings of the First International Meeting of Origami Science and Technology*, pages 143–158, 1989.
2. K. Hatori. K's origami - fractional library - origami construction. 2005. http://origami.ousaan.com/library/conste.html.
3. R. J. Lang. Origami and geometric constructions. Available at http://www.langorigami.com/science/hha/origami_constructions.pdf. Copyright: 1996–2003.
4. R. C. Alperin. A Mathematical Theory of Origami Constructions and Numbers. *New York Journal of Mathematics*, 6:119–133, 2000.

5. R. C. Alperin and R. J. Lang. One-, two, and multi-fold origami axioms. In *Proceedings of 4th International Conference on Origami, Science, Mathematics and Education*. 4OSME, 2006. Caltech, Pasadena CA.
6. T. Ida, H. Takahashi, M. Marin, A. Kasem, and F. Ghourabi. Computational Origami System Eos. In *Proceedings of 4th International Conference on Origami, Science, Mathematics and Education*, page 69. 4OSME, 2006. Caltech, Pasadena CA.
7. T. Ida, H. Takahashi, M. Marin, F. Ghourabi, and A. Kasem. Computational Construction of a Maximal Equilateral Triangle Inscribed in an Origami. In *Mathematical Software - ICMS 2006*, volume 4151 of *Lecture Notes in Computer Science*, pages 361–372. Springer, 2006.
8. Fadoua Ghourabi, Tetsuo Ida, Hidekazu Takahashi, Mircea Marin, and Asem Kasem. Logical and algebraic view of huzita's origami axioms with applications to computational origami. In *Proceedings of the 22nd ACM Symposium on Applied Computing*, pages 767–772. ACM Press, 2007. The symposium was held in Seoul, Korea on March 11 - 15, 2007.
9. N. J. Wildberger. *Divine Proportions*. Wild Egg Pty Ltd, 2005.
10. K. R. Pearson A. Jones, S. A. Morris. *Abstract Algebra and Famous Impossiblities*. Spring-Verlag, 1994.
11. T. Ida, H. Takahashi, M. Marin, and F. Ghourabi. Modelling origami for computational construction and beyond. In *The 2007 International Conference on Computational Science and Its Applications (ICCSA 2007)*.
12. B. Buchberger. Ein algorithmisches Kriterium für die Lösbarkeit eines algebraischen Gleichungssystems. *Aequationes mathematicae*, 4(3):374–383, 1970. English translation in: Bruno Buchberger and Franz Winkler (eds.), Gröbner Bases and Applications, Proceedings of the International Conference "33 Years of Gröbner Bases", 1998, London Mathematical Society Lecture Note Series, Vol. 251, Cambridge University Press, 1998, pp. 535 -545.
13. B. Buchberger. Introduction to Gröbner Bases. In B. Buchberger and F. Winkler, editors, *Gröbner Bases and Applications*, pages 3–31. Cambridge University Press, 1998.

Graph Rewriting in Computational Origami

Tetsuo Ida and Hidekazu Takahashi*

Department of Computer Science, University of Tsukuba
Tsukuba 305-8573, Japan
{ida, hidekazu}@score.cs.tsukuba.ac.jp

Abstract. We formalize paper fold (origami) by graph rewriting. Origami construction is abstractly described by a rewrite system $(\mathcal{O}, \nrightarrow)$, where $\mathcal{O}$ is the set of abstract origami's and $\nrightarrow$ is a binary relation on $\mathcal{O}$, called *fold*. An abstract origami is a triplet $(\Pi, \backsim, \succ)$, where Π is a set of faces constituting an origami, and $\backsim$ and $\succ$ are binary relations on Π, each representing adjacency and superposition relations of the faces. Origami construction is modeled as a rewrite sequence of abstract origami's.
We then address the problems of representation and transformation of abstract origami's and of reasoning about the construction for computational purposes. We present a hypergraph of origami and define origami fold as algebraic graph transformation. The algebraic graph theoretic formalism enables us to reason origami in two separate domains of discourse, i.e. pure combinatoric domain and geometric domain $\mathcal{R} \times \mathcal{R}$, and thus helps us to further tackle challenging problems in origami research.

1 Introduction

Origami provides the methodology of constructing a geometrical object with a piece of paper only by means of folding by hands. Computational origami studies the mathematical aspects of origami. By the assistance of a computer we will be able to formalize origami with rigor and capability that are beyond the methods performed by hands.

In this paper we give graph theoretic formalization of origami. Our main motivation of this study is to give more abstract view of origami fold. Although paper fold appears to be a simple operation to humans, an anatomy of origami reveals that it is not an easy operation. There are two distinct operations in paper fold, i.e. division and reflection of origami faces. These operations lend themselves to distinct modes of computations; algebraic and numeric computation on geometric objects, e.g. finding intersection of lines and checking the overlap of two faces, on one hand, and purely combinatoric computation on discrete objects, e.g. computing transitive closure of the adjacency relation on faces, on the other.

These computations tend to be mixed when origami is analyzed mathematically in our earlier work [3]. Indeed our current implementation of Eos [4] relies

* This research is supported by the JSPS Grants-in-Aid for Exploratory Research No. 19650001 and by Japan-Austria Research Cooperative Program of JSPS and FWF.

very much on algorithms which resort to mixtures of algebraic, numeric and symbolic computing. Sometimes algorithms are not easy to describe mathematically because of this intricacy. There should be clearer separation of computations of discrete and continuous objects in origami. By doing so, we not only clarify the algorithms developed for the implementation of Eos, but also to extend the capability of Eos to allow for more complex origami construction such as of 3D and modular origami.

The rest of the paper is organized as follows. In Section 2 we will formalize basic origami operations. In Section 3 we explain the bases for graph theoretic modeling of origami. In Section 4, we show how basic operations of origami are formalized in the algebraic and graph theoretic framework. In Section 5, we summarize the results and point out the direction of our research.

2 Formalizing origami

2.1 First glimpse of origami

We start an origami construction with a single piece of paper, and repeats folding of the paper until it becomes a desired shape. Here, we see that an origami can be modeled as a set of faces. During the construction, some of the faces get divided by a fold line, get rotated along the fold line and become above or below the others. The faces form layers. The layers of faces exhibit a remarkable shape, which may be regarded as a piece of art such as illustrated in Fig. 1.

The left origami in Fig. 1 is the top view of the constructed object. We see the faces in two different colors in the figure. This is because the initial origami has two sides, each colored differently. During the construction, some faces become up and the others become down, resulting in the two colored object. We can imagine that this origami models a cicada. The right is a 3D view of the same origami after stretching it vertically and making overlapping faces slightly far apart. From the shapes in Fig. 1, we will be able to observe that an origami can be formalized as a set of faces together with the relations that express relative positions, horizontally and vertically, among the faces.

2.2 Abstract origami

An origami can be modeled at several abstraction levels. A most abstract view is to take an origami as an algebra $\langle A, R\rangle$, where A is a set and R is a binary relation on A, where we identify A to be a set of faces that constitute an origami, and R to be a geometrical relation on the faces. The origami construction is then a transformation of the algebras viewed as an abstract rewrite system. We begin with this view of abstraction and gradually make our modeling concrete.

We consider a finite set Π of faces to be the object of our study, and introduce two binary relations on Π, expressing horizontal and vertical arrangements of faces rather than a single binary relation R mentioned before. Then, we have the following definition of an origami.

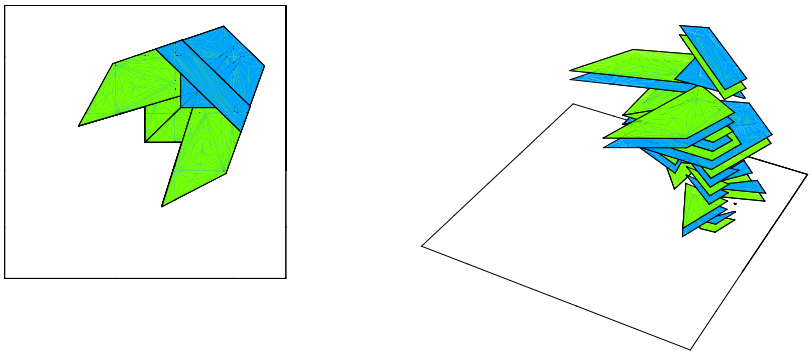


Fig. 1. Origami cicada: art piece (left) and face layers (right)

Definition 1 (Abstract origami). *An abstract origami is a structure*

$$\langle \Pi, \backsim, \succ \rangle$$

where

- Π *is the finite set of faces, and*
- $\backsim$ *and* $\succ$ *are binary relations on* Π*, called adjacency and superposition relations, respectively.*

From our observations so far it is clear that our modeling needs the relations of adjacency and superposition.

Definition 2 (Abstract origami system). *An abstract origami system is an abstract rewrite system* $(\mathcal{O}, \looparrowright)$ *where*

- $\mathcal{O}$ *is the set of abstract origami's.*
- $\looparrowright$ *is the binary relation on* $\mathcal{O}$ *called* abstract fold*, and is denoted by* $O \looparrowright O'$ *for* $O, O' \in \mathcal{O}$*.*

Origami construction proceeds stepwise. Namely, we start with an initial origami ($i = 0$) and perform folds along fold lines repeatedly until we obtain a desired shape. Suppose that we are at the beginning of step i of the construction, having an origami $O_{i-1} = (\Pi_{i-1}, \backsim_{i-1}, \succ_{i-1})$. We make the next fold and obtain the next origami $O_i = (\Pi_i, \backsim_i, \succ_i)$. Thus we have the following:

Definition 3 (Abstract origami construction). *An abstract origami construction is a finite sequence of abstract folds:*

$$O_0 \looparrowright O_1 \looparrowright \cdots \looparrowright O_n, \quad \textit{where } O_0, O_1, \ldots, O_n \in \mathcal{O}$$

Although some properties of origami can be studied with necessary rigor at this level, more geometric information is needed to understand many of the properties of origami. We are lead to the definition of face in Def. 4.

Before we proceed, we note the following definition of an n-gon. An $n(n \geqslant 3)$-gon is a polygon consisting of n edges none of which intersect each other. We also use the notion of overlapping. Let an expression p° denote the interior of an n-gon p. We identify the interior of an n-gon with the set of all the points in the interior. N-gons p and q are called *overlapping* if $p^\circ \cap q^\circ \neq \emptyset$.

Definition 4 (Face). *A face is a convex n-gon.*

Then we can define the adjacency relation as follows:

Definition 5 (Face adjacency). *Two faces are adjacent if they share an edge.*

We can determine whether a face is adjacent to the other face.

Concerning the superposition relation, we assume a decision procedure of determining above or below relation among the faces. Namely, given two faces f and g, we can determine whether:

(1) f is above g or
(2) g is above g or
(3) f and g are not related.

We also say that g is below f if f is above g. Now we have the following definition of the superposition relation.

Definition 6 (Face superposition). *Face f superposes face g if f and g are overlapping, f is above g and no faces that are above g is below f.*

2.3 Formalization of fold

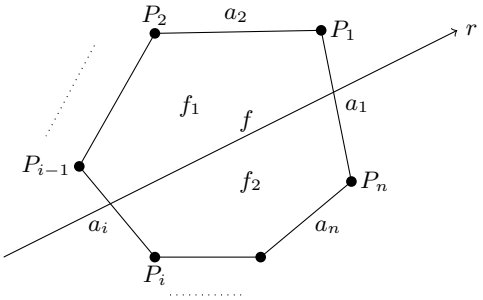
Fold of an origami is a complex operation consisting of the following sub-operations.

(1) Specify the set $\mathcal{F}$ of the faces of concern and decide a basic fold operation. We may use one of Huzita's basic folds [2] or classical fold methods such as mountain and valley folds.
(2) Compute a fold line l and make it a direct line called a ray r.
(3) For each face f in $\mathcal{F}$, do the following until $\mathcal{F} = \emptyset$.
   (a) Divide f by r into two faces f_1 and f_2, where f_2 is to the right of r. See below for more details.
   (b) Update $\mathcal{F}$ by removing f from $\mathcal{F}$ and adding faces that are affected by this division using the superposition and adjacency relations.
(4) Compute new superposition and adjacency relations induced by the division.
(5) Rotate the faces to the right of r along r.
(6) Compute new superposition relation induced by the rotation.

As for the division of a face, we distinguish four cases of geometrical configurations:

1. The line l intersects with the edges of f at two distinct points. The face f is divided into two faces f_1 and f_2, where f_2 is to be rotated.
2. The line l overlaps with some edge of f.
3. The line l passes through f at only one vertex.
4. The line l intersects with none of the edges of f.

Let us now make our modeling more concrete. We represent an n-gon as a sequence of points $\langle P_1, \ldots, P_n \rangle$, where points $P_1, \ldots, P_n$ are vertices of the n-gon. A face is thus represented as a sequence of points. When points $P_1, \ldots, P_n$ are arranged counterclockwise, we say that the face is up, and when clockwise, it is down. The division of an up face is illustrated in Fig. 2. This case is further investigated for the graph transformation in the next section.



The face f is divided by the ray r into $\langle f_1, f_2 \rangle$. The face f_2 is to be rotated.

Fig. 2. Face division

3 Graph formalization

3.1 Hypergraph and graph term

To make origami amenable to computation, we further concretize the abstract origami by graph theoretic formalism. We use a labeled hypergraph for this purpose. Since we do not need algebraic graph theories in full generality such as discussed in [1], we work with hypergraphs defined as follows.

Definition 7 (Hypergraph). *A hypergraph is a quadruple $\langle V, E, s, t \rangle$, where*

- *V is the set of nodes*[1]*,*
- *E is the set of edges, and*
- *$s, t : E \to V^*$ are source and target functions.*

[1] We use of the word *node* here to avoid the clashes with *vertices* of a polygon. We cannot avoid the clashes of the word *edge* of a polygon and of a graph, however.

Definition 8 (Hypergraph labeling). *A hypergraph labeling consists of a pair* $\langle \mathcal{L}_E, \mathcal{L}_V \rangle$ *of label alphabets together with functions* $\tau_s, \tau_t : \mathcal{L}_E \to \mathcal{L}_V^*$.

Definition 9 (Labeled hypergraph). *Given a pair* $\mathcal{L} = \langle \mathcal{L}_V, \mathcal{L}_E \rangle$ *of label alphabets, an* $\mathcal{L}$*-labeled hypergraph is a 6-tuple* $\langle V, E, s, t, l_V, l_E \rangle$*, where*

- $\langle V, E, s, t \rangle$ *is a hypergraph and*
- $l_V : V \to \mathcal{L}_V$ *and* $l_E : E \to \mathcal{L}_E$ *are functions satisfying* $\tau_s \cdot l_E = l_V^* \cdot s$ *and* $\tau_t \cdot l_E = l_V^* \cdot t$.

Hereafter, we only consider hypergraphs. Therefore we call hyprergraph and hyperedges without prefix "hyper" unless we want to emphasize "hyper".

Definition 10 (Graph term). *Given an* $\mathcal{L}$*-labeled graph* $G = \langle V, E, s, t, l_V, l_E \rangle$*, graph term representation* $\mathcal{G}$ *of a graph* G *is defined as*

$$\{l_E(e)[s(e), t(e)] \mid e \in E\} \biguplus \{l_V(v)[v] \mid v \in V\} \tag{3.1}$$

The expression $l_E(e)[s(e), t(e)]$ is of the form $F[v_1, \ldots, v_n]$, which is actually the term representation in our language for graph transformation as we will see shortly. Note that $s(e), t(e)$ is a sequence of nodes.

In Eq.(3.1), $\{l_E(e)[s(e), t(e)] \mid e \in E\}$ is a multi-set and $\biguplus$ is the union operation on multi-sets. The element of $\{l_E(e)[s(e), t(e)] \mid e \in E\}$ is called *edge term* of the graph and the element of $\{l_V(v)[v] \mid v \in V\}$ is called *node term*. Both terms are called *graph terms*, *g-term* in short. This representation allows us to reason with tree structures even when we are dealing with graphs. Note that terms are representation of trees. A hypergraph is now represented as a set of g-terms. In most algebraic graph transformations to follow, our focus is more on the manipulation on edges than of nodes. The rewrite rules for the graph transformation are designed on the basis of the manipulation of edge terms.

Example 1. Let $\mathcal{L}_V$ and $\mathcal{L}_E$ be {F} and {A, R, L}, respectively, and let G be an $\mathcal{L}$-labeled graph $\langle V, E, s, t, l_V, l_E \rangle$, where

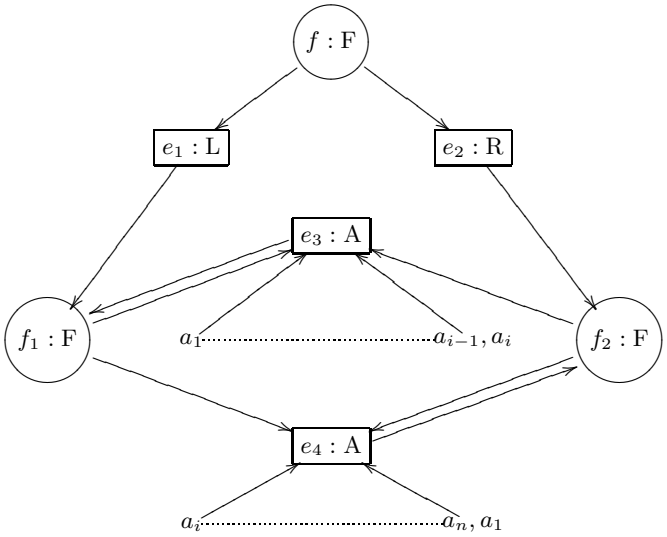
- $V = \{f, f_1, f_2\}$
- $E = \{e_1, e_2, e_3, e_4\}$
- $s = \{e_1 \mapsto f, e_2 \mapsto f, e_3 \mapsto \langle f_1, a_1, \ldots, a_i, f_2 \rangle, e_4 \mapsto \langle f_2, a_i, \ldots, a_n, a_1, f_1 \rangle\}$
- $t = \{e_1 \mapsto f_1, e_2 \mapsto f_2, e_3 \mapsto \langle f_1 \rangle, e_4 \mapsto \langle f_2 \rangle\}$
- $l_E = \{e_1 \mapsto \mathrm{L}, e_2 \mapsto \mathrm{R}, e_3 \mapsto \mathrm{A}, e_4 \mapsto \mathrm{A}\}$
- $l_V = \{f_1 \mapsto \mathrm{F}, f_2 \mapsto \mathrm{F}, f \mapsto \mathrm{F}\}$.

The hyperedge e_3 forms a loop visiting the nodes $f_1, a_1, \ldots, a_i, f_2$ in this order. The hyperedge e_4 also forms a loop. A looped hyperedge visiting nodes v_1, …, v_n in this order may be described as a hyperedge visiting nodes v_i, …, v_n, v_1, …, v_{i-1} for any $i \in \{1, \ldots, n\}$, as well. We, however, use a unique representation of the hyperedge by imposing the condition that the first node is always a newly created node(face), as seen in this example.

The g-term representation $\mathcal{G}$ of G is:

$\mathcal{G}$ = { L[f, f_1], R[f, f_2],
A[$f_1, a_1, \ldots, a_i, f_2.f_1$], A[$f_2, a_i, \ldots, a_n, a_1, f_1, f_2$],
F[f], F[f_1], F[f_2] }

The graph is shown in Fig. 3. In the graph the node v_i with label L is represented as a circled v_i: L. The edge e_i with label L is represented as a boxed e_i: L. This graph represents the face division given in Fig. 2.



Nodes $a_1, a_2, \ldots, a_n$ are not shown.

Fig. 3. Graph of an origami created by face division

3.2 Graph rewriting

In this section we give a language of graph rewriting. G-term t is defined by the following grammar:

$$t := x \mid \underline{x} \mid f[t_1, \ldots, t_n]$$

Here, x denotes a variable, and the underlined one is a sequence variable. The sequence variable is indispensable since a function symbol f in g-term $f[t_1, \ldots, t_n]$ representing a hyperedge has an unfixed arity. A g-pattern is a g-term possibly with a condition c. A g-pattern p is defined by the following grammar:

$$p := t \mid t/; c$$

The syntax of condition c is not given here, as it is defined by an external language. Readers can assume that a condition c has the same syntax as a g-term, or more concretely has the syntax of Mathematica, which we are suing

as our implementation language. The expression of the form $t/;c$ is called a conditional g-term. The intended use of the conditional g-term is as a left-hand side of a graph rewrite rule, which is defined below. It is used for conditional pattern matching. During pattern matching using a substitution θ for selecting a subgraph, if $c\theta$ evaluates (by an external evaluator) to True, $(t/;c)\theta$ represents $t\theta$, and otherwise it represents $\perp$. We use u to denote either a g-term or a g-pattern.

As a meta notation we use $\langle u_1, \ldots, u_n \rangle$ to denote List$[u_1, \ldots, u_n]$.

Definition 11 (Graph rewrite rule). *A graph rewrite rule (rewrite rule for short) is a pair of g-terms*

$$g_l \rightharpoonup g_r$$

where $g_l := \langle u_1, \ldots, u_m \rangle$ *and* $g_r := \langle t_1, \ldots, t_n \rangle$.

Definition 12 (Graph rewriting).

A graph $\mathcal{G}$ *is rewritten to* $\mathcal{G}'$ *by a rewrite rule* $r := g_l \rightharpoonup g_r$*, denoted by*

$$\mathcal{G} \Rightarrow_r \mathcal{G}'$$

if there exist terms $s_1, \ldots, s_m$*, and a substitution* θ *such that* $\{s_1, \ldots, s_m\} \subseteq \mathcal{G}$*,* $g_l\theta$ *after the evaluation of the conditions, if any, is* $\langle s_1, \ldots, s_m \rangle$ *and* $\mathcal{G}' = \mathcal{G} \setminus \{s_1, \ldots, s_m\} \cup \{t_1, \ldots, t_n\}$*, where* $g_r\theta = \langle t_1, \ldots, t_n \rangle$.

4 Fold as a graph rewriting

We are now ready to describe the fold explained in Subsection 2.3 in graph rewriting framework. We recall that the fold consists of the following operations:

(1) division of faces,
(2) update of the adjacency relation,
(3) update of superposition relation induced by face division, and
(4) update of superposition relation induced by face rotation.

These operations are preformed in sequence, and we describe them in graph rewriting.

Face division We consider the division of a face f into f_1 and f_2 by a ray r as shown in Figs. 2 and 3. Figure 3 is the subgraph of the entire graph of the origami that we are working on.

The graph was transformed in the following steps from the graph of the previous step:

(1) Construct nodes f_1 and f_2.
(2) Construct the edge e_1 that links f with f_1 and e_2 that links f with f_2. Depending on whether the divided faces are to the left or right of the ray r, attach the labels R (R for Right) or L (L for Left) to the edges. In this case, the label of e_1 is L since face f_1 is to the left of the ray, and the label of e_2 is R.

(3) Construct the edges e_3 and e_4 issuing from f_1 and from f_2, respectively. We have $s(e_3) = \langle f_1, a_1, \ldots, a_i, f_2 \rangle$, $t(e_3) = f_1$, $s(e_4) = \langle f_2, a_i, \ldots, a_n, a_1, f_1 \rangle$ and $t(e_4) = f_2$. We label those edges by A (A for Adjacency) since the constructed edges represent the adjacency relation.

Face update The graph constructed in the face division step has to be updated since some of other faces are also divided, but the edges still link to those nodes of previous (undivided) faces. This step does this face update. This update is straightforward by the following rewrite rules:

$$\{\mathtt{L}[f, f_1]\,,\ \mathtt{A}[f_1, \underline{x}]\,,\ \mathtt{L}[g, g_1]/; g \neq g_1 \wedge g \in \{x\}\ \} \\ \rightarrow \{\mathtt{L}[f, f_1]\,,\ \mathtt{A}[f_1, \underline{x}\{g \rightarrow g_1\}]\,,\ \mathtt{L}[g, g_1]\}$$

$$\{\mathtt{R}[f, f_1]\,,\ \mathtt{A}[f_1, \underline{x}]\,,\ \mathtt{R}[g, g_1]/; g \neq g_1 \wedge g \in \{x\}\ \} \\ \rightarrow \{\mathtt{R}[f, f_1]\,,\ \mathtt{A}[f_1, \underline{x}\{g \rightarrow g_1\}]\,,\ \mathtt{R}[g, g_1]\}$$

Update of superposition relation induced by division Suppose that faces f and g such that $f \succ g$ are divided into $\langle f_1, f_2 \rangle$ and $\langle g_1, g_2 \rangle$, respectively. In the graph of Fig. 4, the edge e_5 is labeled S (S for Superposition) since $f \succ g$. We have $f_1 \succ g_1$ if $f_1^\circ \cap g_1^\circ \neq \emptyset$, and $f_2 \succ g_2$ if $f_2^\circ \cap g_2^\circ \neq \emptyset$. In Fig. 4, we assume that $f_1^\circ \cap g_1^\circ \neq \emptyset$ and $f_2^\circ \cap g_2^\circ \neq \emptyset$. Therefore, the edges e_6 and e_7 are added in this step. The A labeled edges are omitted for simplicity.

This transformation is realized by the following rewrite rule:

$$\{\mathtt{S}[f, g]\,,\mathtt{L}[f, f_1]\,,\mathtt{R}[f, f_2]\,,\mathtt{L}[g, g_1]/;f_1^\circ \cap g_1^\circ \neq \emptyset\,,\mathtt{R}[g, g_2]/;f_2^\circ \cap g_2^\circ \neq \emptyset\} \\ \rightarrow \{\mathtt{S}[f_1, g_1]\,,\mathtt{S}[f_2, g_2]\,,\mathtt{S}[f, g]\,,\mathtt{L}[f, f_1]\,,\mathtt{R}[f, f_2]\,,\mathtt{L}[g, g_1]\,,\mathtt{R}[g, g_2]\}$$

The g-terms $\mathrm{S}[f_1, g_1]$, $\mathrm{S}[f_2, g_2]$, $\mathrm{S}[f, g]$, $\mathrm{L}[f, f_1]$, $\mathrm{R}[f, f_2]$, $\mathrm{L}[g, g_1]$ and $\mathrm{R}[g, g_2]$ match with edges e_6, e_7, e_5, e_1, e_2, e_3 and e_4, respectively.

Depending on whether the faces are divided and non-divided, and on whether they are to the right or left of the ray in the case of non-divided, we have other four cases. Defining the rewrite rules for these cases is straightforward.

Update of superposition relation induced by rotation For any pair of faces f and g, we have to check if they are related by the superposition. We distinguish the following three cases for each pair.

(1) We rotate the face f that is to the right of r after the division. It may overlap with the face g that is to the left of r. In this case, $f \succ g$ if no other faces above g is below f.
(2) We rotate the faces f and g that are $f \succ g$ and are to the right of r. We delete the relation $f \succ g$ and newly add $g \succ f$.
(3) Otherwise no superposition is added.

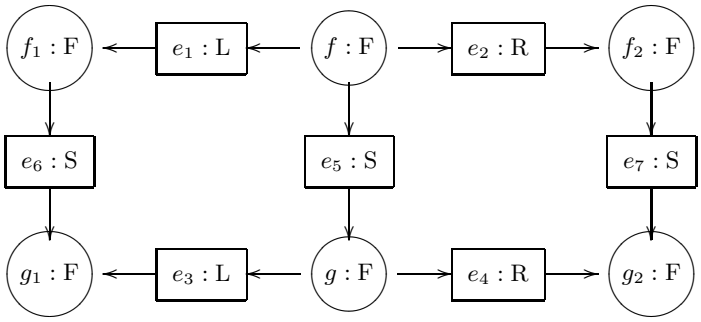


Fig. 4. Addition of superposition relation by division

Example 2. Figure 5 shows the origami at the intermediary step during the fold. We are about to make a fold along the fold line indicated by the dotted line. The ray r corresponding to the fold line runs from lower right to upper left. The faces that form the base layer of the origami has been divided by r into faces a and f. At this step, the origami consists of faces a, b, c, d, f, g and h. We see that the faces a, b, c and d are to the right of r, and that the faces f, g and h are to the left of r.

Figure 6 shows the result of the rotation along r, where the faces to the right are moved.

Figure 7 shows the graph representation of the intermediary origami of Fig. 5. Faces g and h superpose face f. Faces b and c superpose a. Face d superposes b. For simplicity we omit the A, L and R labeled edges.

Figure 8 shows the graph representation of the origami of Fig. 6. The rotation effects the graph transformation from the graph of Fig. 7 to that of Fig. 8. The newly added S labeled edges are e_6, e_7, e_8, e_9 and e_{10}. This is the result of the following computation: Let $RF = \{\mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{d}\}$ and $LF = \{\mathrm{f}, \mathrm{g}, \mathrm{h}\}$.

(1) We take d from RF and g from LF.
(2) Since $\mathrm{d}^\circ \cap \mathrm{g}^\circ = \emptyset$, we have no superposition relation between d and g.
(3) We take h from LF, and check the overlap of d and h.
(4) We have $\mathrm{d}^\circ \cap \mathrm{h}^\circ \neq \emptyset$, we have $\mathrm{d} \succ \mathrm{h}$. This is shown by the edge e_6.
(5) Likewise, we add the edge e_7.
(6) We add the edge e_8 since we had $\mathrm{d} \succ \mathrm{b}$, the edge e_9 since we had $\mathrm{b} \succ \mathrm{a}$ and edge e_{10} since we had $\mathrm{c} \succ \mathrm{a}$.

5 Conclusion

We have presented an abstract model of origami. The abstraction lead to graph theoretic modelling and transformation of origami. More concretely, we formalized an origami as a hypergraph and define the fold as algebraic graph transformations. The algebraic graph theoretic formalism enables us to reason origami in

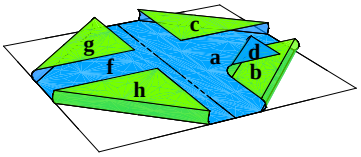


Fig. 5. Origami at the intermediary step during the fold

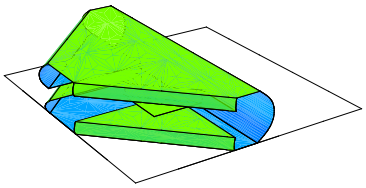


Fig. 6. Origami after the fold

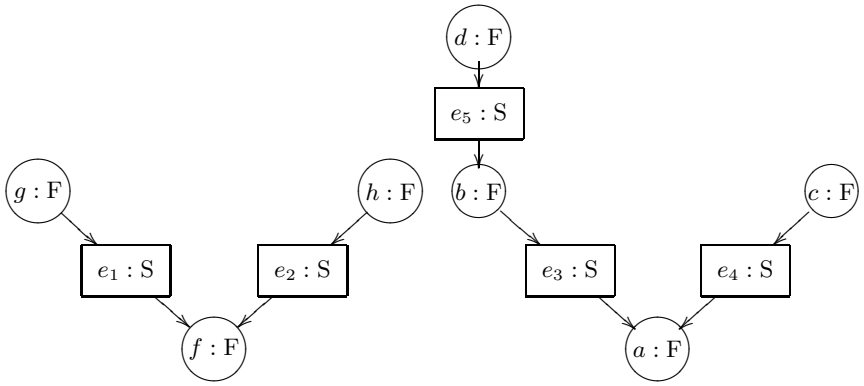


Fig. 7. Graph representation of the origami of Fig. 5

two separate domains of discourse, i.e. pure combinatoric domain, and geometric domain $\mathcal{R} \times \mathcal{R}$, and thus helps us to further tackle challenging problems such as of discovering a new construction given an origami shape, and of discovering a new origami that has certain geometric properties.

Our formalism follows closely that of algebraic and categorical graph theory, and we anticipate the rich theory in this area will be applicable to our computational origami research.

References

1. H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer-Verlag, 2006.

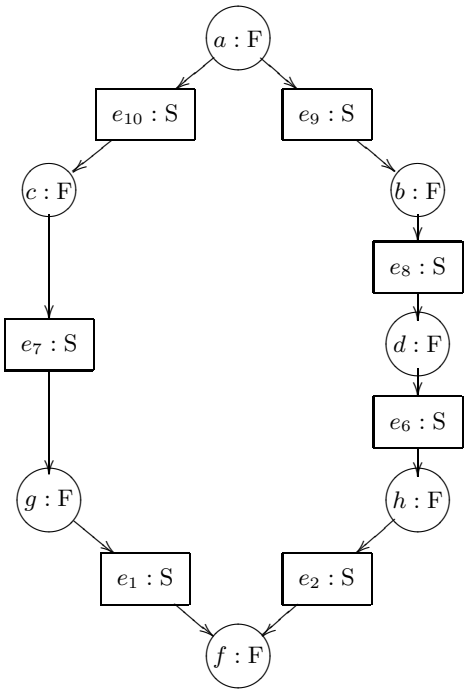


Fig. 8. Graph representation of the origami of Fig. 6

2. H. Huzita. Axiomatic Development of Origami Geometry. In H. Huzita, editor, *Proceedings of the First International Meeting of Origami Science and Technology*, pages 143–158, 1989.
3. T. Ida, H. Takahashi, M. Marin, and F. Ghourabi. Modelling origami for computational construction and beyond. In O. Gervasi and M. Gavrilova, editors, *International Conference on Computational Science and Its Applications 2007 (ICCSA 2007)*, volume 4151 of *Lecture Notes in Computer Sciences*, pages 653 – 665. Springer-Verlag Berlin Heidelberg.
4. T. Ida, H. Takahashi, M. Marin, F. Ghourabi, and A. Kasem. Computational Construction of a Maximal Equilateral Triangle Inscribed in an Origami. In *Mathematical Software - ICMS 2006*, volume 4151 of *Lecture Notes in Computer Science*, pages 361–372. Springer, 2006.

Productivity of Algorithmic Systems

Ariya Isihara

Department of Computer Science, Vrije Universiteit, Amsterdam
ariya@few.vu.nl

Abstract. Algorithmic systems are devised as a framework providing the possibility of defining infinite objects and functions on them, as in a lazy functional programming language. Our concern is the 'productivity' of such definitions, which may be compared to lazy termination in functional programming.
Algorithmic systems are a kind of orthogonal infinitary rewriting systems, that are also constructor TRSs. The constructor function symbols are divided into two classes: inductive and coinductive ones. In algorithmic systems, infinite nesting is only allowed of coinductive constructor symbols.
In the present work, we formulate our notion of algorithmicity, and investigate a criterion for the productivity of an algorithmic system. Productivity is divided into two parts, which we will call Domain Normalisation (DN) and Constructor Normalisation (CN). The DN property gives a certain requirement on the infinitary normal forms, and the CN property is the counterpart of lazy normalisation in functional programming. We investigate criteria to establish these properties.

1 Introduction

Infinitary objects are not only theoretically interesting to consider, but also naturally arise as the semantics of computation, in particular, the semantics of cyclic objects. Moreover, periodic infinitary objects are also something to deal with; consider the TRS

$$\begin{aligned}
\mathsf{filter}(x : y, \mathsf{0}, m) &\to \mathsf{0} : \mathsf{filter}(y, m, m) \\
\mathsf{filter}(x : y, \mathsf{s}(n), m) &\to x : \mathsf{filter}(y, n, m) \\
\mathsf{sieve}(\mathsf{0} : y) &\to \mathsf{sieve}(y) \\
\mathsf{sieve}(\mathsf{s}(n) : y) &\to \mathsf{s}(n) : \mathsf{sieve}(\mathsf{filter}(y, n, n)) \\
\mathsf{nats}(n) &\to n : \mathsf{nats}(\mathsf{s}(n)) \\
\mathsf{primes} &\to \mathsf{sieve}(\mathsf{nats}(\mathsf{s}(\mathsf{s}(\mathsf{0}))))
\end{aligned}$$

which appears in [12, Sect. 12]. This system provides a version of the sieve of Eratosthenes where the filter function replaces struck-out members of the list by zeros (0), which are subsequently deleted by the sieve function. The $\mathsf{nats}(n)$ function generates the list of natural numbers starting with n, with the successor constructor s. Thus, we can compute (or imagine)

$$\mathsf{primes} \twoheadrightarrow \ulcorner 2 \urcorner : \ulcorner 3 \urcorner : \ulcorner 5 \urcorner : \ulcorner 7 \urcorner : \ulcorner 11 \urcorner : \ldots \qquad (\star)$$

where $\twoheadrightarrow$ denotes the infinite reduction, and $\ulcorner n \urcorner$ denotes the representation of a natural number; $\ulcorner n \urcorner = \mathsf{s}^n(\mathsf{0})$. Though we cannot obtain the entire sequence ($\star$) in finite time, each element of the sequence can be obtained in finite time. That is known as lazy evaluation, and thereby we regard primes well specified as a(n infinite) sequence of natural numbers. We refer to this feature as *productivity*. In this example, the primes function is productive, for the system determines the sequence.

Unfortunately, determining productivity is generally very difficult; the productivity problem (whether a definition is productive or not) covers a lot of mathematical open problems. For example, we can code the twin prime problem by

$$\begin{aligned} \mathsf{plusTwo}(x : y) &\to \mathsf{s}(\mathsf{s}(x)) : \mathsf{plusTwo}(y) \\ \mathsf{twins} &\to \mathsf{intersect}(\mathsf{primes}, \mathsf{plusTwo}(\mathsf{primes})) \end{aligned}$$

where the $\mathsf{intersect}$ function computes the intersection of two ordered sequences, which can be implemented as follows:

$$\begin{aligned} \mathsf{intersect}(n : x, m : y) &\to \mathsf{intersectAux}(\mathsf{cmp}(n, m), n : x, m : y) \\ \mathsf{intersectAux}(\mathsf{eq}, n : x, m : y) &\to n : \mathsf{intersect}(x.y) \\ \mathsf{intersectAux}(\mathsf{gt}, n : x, m : y) &\to \mathsf{intersect}(n : x, y) \\ \mathsf{intersectAux}(\mathsf{lt}, n : x, m : y) &\to \mathsf{intersect}(x, m : y) \end{aligned}$$

with the cmp function as above. If the twins function is productive, then there exist infinitely many twin primes, and vice versa.

In fact, productivity is undecidable; one can code the Halting problem of Turing machines, which is known to be undecidable [13]. Actually, the above example of primes is already difficult; the productivity of the sieve function is based on the mathematical fact that there exist infinitely many prime numbers. The difficulty is caused by the rules for the sieve function:

$$\begin{aligned} \mathsf{sieve}(\mathsf{0} : y) &\to \mathsf{sieve}(y) \\ \mathsf{sieve}(\mathsf{s}(n) : y) &\to \mathsf{s}(n) : \mathsf{sieve}(\mathsf{filter}(y, n, n)). \end{aligned}$$

The behaviour varies depending on whether the given sequence begins with $\mathsf{0}$ or a successor $\mathsf{s}(n)$. In the first case it only consumes the leading $\mathsf{0}$ from the sequence to produce nothing, while in the second case it produces the element $\mathsf{s}(n)$ followed by $\mathsf{sieve}(\mathsf{filter}(y, n, n))$. So, the computation of $\mathsf{sieve}(\mathsf{0} : \mathsf{0} : \mathsf{0} : \dots)$ yields nothing:

$$\mathsf{sieve}(\mathsf{0} : \mathsf{0} : \mathsf{0} : \dots) \to \mathsf{sieve}(\mathsf{0} : \mathsf{0} : \mathsf{0} : \dots) \to \mathsf{sieve}(\mathsf{0} : \mathsf{0} : \mathsf{0} : \dots) \dots .$$

Of course, $\mathsf{sieve}(\mathsf{0} : \mathsf{0} : \mathsf{0} : \dots)$ is never called during the computation of primes. But, how could we know that, if we did not know that there are infinitely many primes? In the present work, we wish to forget this annoyance arising from case-distinctive analysis, and are concerned with what we will call 'data-oblivious'

analysis of productivity. We refer to [4] where the concept of data-oblivious is defined and a detailed analysis is provided.

As an easy example of data-oblivious analysis, we recall the plusTwo function which appears in the implementation of the twin prime problem:

$$\mathsf{plusTwo}(x : y) \to \mathsf{s}(\mathsf{s}(x)) : \mathsf{plusTwo}(y).$$

Being oblivious on data, all the data now look like '•' (something):

$$\mathsf{plusTwo}(\bullet : y) \to \bullet : \mathsf{plusTwo}(y).$$

Moreover, any sequence of natural numbers is now no more than

$$\bullet : \bullet : \bullet : \dots$$

for us. Thus, the result of plusTwo is always computed as follows:

$$\begin{aligned} \mathsf{plusTwo}(\bullet : \bullet : \bullet : \dots) &\to \bullet : \mathsf{plusTwo}(\bullet : \bullet : \bullet : \dots) \\ &\to \bullet : \bullet : \mathsf{plusTwo}(\bullet : \bullet : \bullet : \dots) \\ &\dots \\ &\to\!\!\to\!\!\to \bullet : \bullet : \bullet : \dots . \end{aligned}$$

Here we can imagine that the infinite stream of pebbles (•) goes into the gate plusTwo, and that the gate yields again an infinite stream of pebbles. In [5], we call this abstracted view 'pebbleflow'. There we studied data-oblivious analysis of productivity, and presented a definition scheme for which productivity is decidable.

In the present work, we study this data-oblivious analysis with a weaker restriction and a general framework in which we can deal with infinitary objects other than streams.

Running Example

As the running example, we consider the system which computes the Hamming numbers. (In our framework we will be able to deal with much more complicated objects than streams, however, streams would be still suitable as paradigmatic infinitary objects.)

Figure 1 shows our running example (ignore the rightside for this moment). Natural numbers are represented by 0 and s, and streams are represented by the stream constructor ':'. Formally, we regard $n : x$ as an abbreviated form of $\mathsf{cons}(n, x)$, where cons is the formal name of ':'. The functions add and mul compute addition and multiplication, using the rules of Dedekind's TRS [3], and merge together with aux computes the union of two ordered streams with help of the comparison function cmp. The function scalar multiplicates each element of a stream by a fixed number. The function nats is the same as given above. The Ham function computes the Hamming numbers with mutually recursively called ham2, ham3, and ham5 functions.

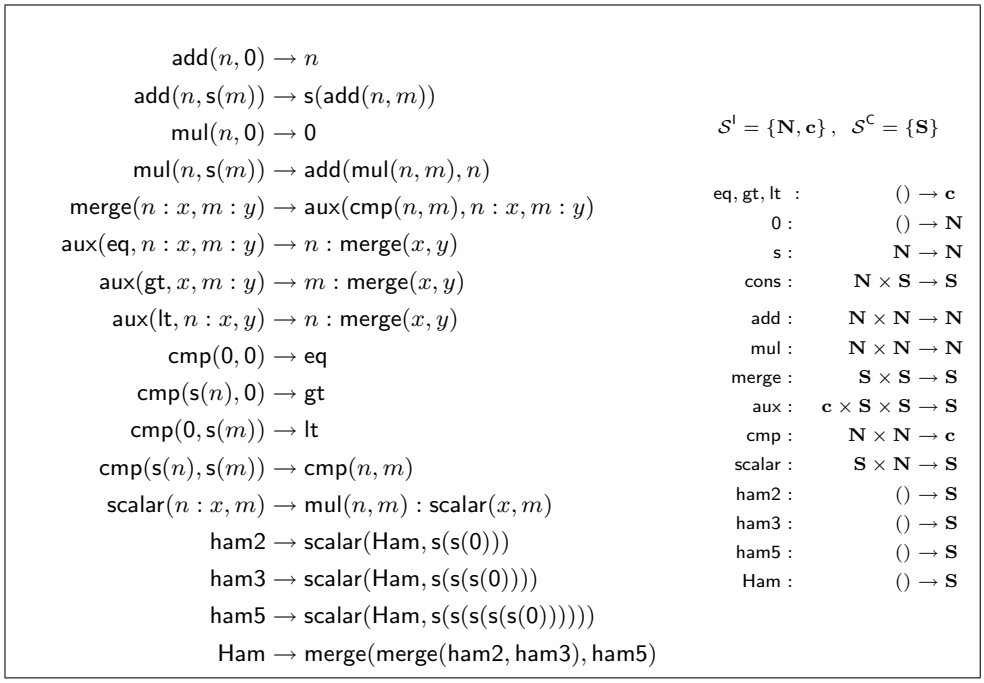


Fig. 1. The running example

Overview

The remainder of the paper is organized as follows: Section 2 presents notations and concisely recalls infinitary term rewriting. In Section 3 we formalise our new notion of algorithmicity, and define productivity in our framework. Section 4 divides the productivity property into two properties, which will be called Domain Normalisation (DN) and Constructor Normalisation (CN); those properties are studied in Section 5 and Section 6, respectively. Section 7 gives a concluding remark.

2 Preliminaries

2.1 Notations

The set of natural numbers including zero is denoted by $\mathbb{N}$. The set $\overline{\mathbb{N}}$ consists of coinductive natural numbers; $\overline{\mathbb{N}} = \mathbb{N} \cup \{\infty\}$. We stimulate $\infty + n = \infty - n = \infty$ for all $n \in \mathbb{N}$. The set of countable ordinals is denoted by $\mathbf{\Omega}$. The set $\mathbb{P}$ of *positions* consists of the finite sequences of positive integers. The empty sequence is denoted by ϵ. The depth of p, written $|p|$, is defined as the length of p. The set $\mathbb{S}$ of *streams* consists of infinite sequences of natural numbers; $\mathbb{S} = \mathbb{N}^{\omega}$.

2.2 Infinitary Rewriting

We assume familiarity with finitary TRSs; we refer to [12] as a standard reference. Here we briefly recall some notions and propositions on infinitary rewriting based on strongly convergent reductions. We refer to [12, Sect. 12], [7], and [8] for further information.

Let Σ, $\mathcal{X}$, $\mathcal{R}$ be fixed sets of symbols, variables, and rewrite rules, respectively. The sets Σ and $\mathcal{R}$ can be finite or infinite, and $\mathcal{X}$ is infinite. Each symbol f has a fixed finite arity, and each variable has the arity 0. Then, a finite term can be regarded as a partial function $t : \mathbb{P} \rightharpoonup (\Sigma \cup \mathcal{X})$ satisfying the following conditions:

1. The domain $\mathrm{dom}(t)$ is finite.
2. The root ϵ is in $\mathrm{dom}(t)$.
3. For every $p \in \mathbb{P}$, we have $p \cdot n \in \mathrm{dom}(t)$ if and only if $p \in \mathrm{dom}(t)$ and $1 \leqslant n \leqslant k$, where k is the arity of the symbol $t(p)$.

Each $p \in \mathrm{dom}(t)$ represents a node in the term tree, labeled by $\Sigma \cup \mathcal{X}$. An infinite term is a partial function satisfying the above conditions 2 and 3; we allow the domain (the nodes) to be infinite. A term is called *ground* if the term has no variable.

A rewrite rule is a pair of terms (l, r). We call l and r respectively lefthand-side (lhs) and righthand-side (rhs). Requirements on rewrite rules are the same as in finitary rewriting; both the terms should be finite, every variable in r should occur also in l, and l should not be a variable. Then, the single-step reduction relation $\to$ on finite terms can be naturally extended to that on infinite terms. We write $\twoheadrightarrow$ to denote multi-step reductions, the reflexive transitive closure of $\to$.

For an infinite sequence $t_0, t_1, \ldots$ of terms, we write $\lim t = s$ if for any $n \in \mathbb{N}$, there exists $m \in \mathbb{N}$ such that $i > m$ and $|p| < n$ implies $t_i(p) = s(p)$. An infinite[1] sequence of single-step reductions

$$t_0 \to t_1 \to t_2 \to \ldots$$

weakly converges to s if $\lim t = s$. Moreover, it *strongly converges* if the depth of redexes that are contracted tends to infinity. We write $t \to^{\omega} s$ if there exists a strongly convergent sequence from t leading to s. We write $\twoheadrightarrow$ to denote $\twoheadrightarrow \cup \to^{\omega}$.

Proposition 1. *In an orthogonal TRS,*

1. *The system is finitary confluent, viz. If* $t \twoheadrightarrow s$ *and* $t \twoheadrightarrow s'$*, then there exists* u *such that* $s \twoheadrightarrow u$ *and* $s \twoheadrightarrow u'$*. This* u *is called a* common reduct *of* s *and* s'*.*
2. *If* $t \twoheadrightarrow s$ *and* $t \twoheadrightarrow s'$ *where* s *and* s' *are normal forms, then* $s \equiv s'$*. This property is called* infinitary unique normal form property (UN^{∞})*.*

[1] We can formalise transfinite reduction of any ordinal length. However, in an orthogonal TRS, any transfinite reduction can be reduced to an infinite reduction of length at most ω (Compression Lemma). Since we consider only orthogonal TRS, infinite reduction suffices for the present paper.

3. *The following conditions are all equivalent.*
   - *For every term t, there exists a normal form s such that $t \twoheadrightarrow s$. This property is called* infinitary weak normalisation (WN^{∞}).
   - *Every infinite reduction sequence strongly converges. This property is called* infinitary strong normalisation (SN^{∞}).
4. *If $t \twoheadrightarrow s$ and $t \twoheadrightarrow u$ where u is a normal form, then $s \twoheadrightarrow u$.*

Proof. See [12] for the first one, and [8] for the others. □

We write t/p to denote the subterm of t at the position p, and write $t\sigma$ to denote the substitution result, where the substitution σ is a mapping from variables to terms.

3 Algorithmic Systems

We deal with our running example in the framework of infinitary term rewriting systems. As seen in Figure 1, it already forms a TRS. However, in order to formalise productivity, we need some further properties. We shall refer to TRSs satisfying these properties as 'algorithmic (term rewriting) systems', for the computation of each function is algorithmically determined in an algorithmic system.

3.1 An Informal Aspect of Productivity

Before we formalise productivity, we roughly present a semantical aspect of productivity.

First, we consider the spaces of $\mathbb{N}$ and $\mathbb{S}$. The space $\mathbb{N}$ is algebraically specified by $\langle 0, S\rangle : 1 + \mathbb{N} \to \mathbb{N}$ and hence the constructors $0 : () \to \mathbb{N}$ and $S : \mathbb{N} \to \mathbb{N}$ arise, constructing the space of natural numbers $\{0, S0, SS0, SSS0 \dots\}$. On the other hand, $\mathbb{S}$ is coalgebraically specified by $\langle hd, tl\rangle : \mathbb{S} \to \mathbb{N} \times \mathbb{S}$ and hence the destructors $hd : \mathbb{S} \to \mathbb{N}$ and $tl : \mathbb{S} \to \mathbb{S}$ arise with the function $cons : \mathbb{N} \times \mathbb{S} \to \mathbb{S}$ to satisfy $cons \circ \langle hd, tl\rangle = \mathrm{id}_{\mathbb{S}}$.

Second, we embed those two spaces into the term space. The algebraic space $\mathbb{N}$ can be directly translated by embedding the constructors as given by $0 \overset{\ulcorner - \urcorner_{\mathbb{N}}}{\longmapsto} \mathsf{0}$ and $S \overset{\ulcorner - \urcorner_{\mathbb{N}}}{\longmapsto} \mathsf{s}$, resulting the same representation as $\ulcorner - \urcorner$ given in Introduction. For example, we have $3 \equiv SSS0 \overset{\ulcorner - \urcorner_{\mathbb{N}}}{\longmapsto} \mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{0})))$. The coalgebraic space $\mathbb{S}$ is not so easy, because the terms, even the infinite terms, are constructed not coalgebraically but algebraically. Hence, we regard *cons* as a constructor and map it to cons as $cons \overset{\ulcorner - \urcorner_{\mathbb{S}}}{\longmapsto} \mathsf{cons}$, and leave the destructors not embedded. Thus, the only difference between the constructions of $\mathbb{N}$ and $\mathbb{S}$ is whether we allow infinite nesting of constructors; $\mu x.\mathsf{s}(x)$ is not accepted but $\mu x.\mathsf{cons}(\mathsf{0}, x)$ is. The first one represents something like ∞, which is not included in $\mathbb{N}$, while the second one represents the stream of zeros.

We call such terms 'values of $\mathbb{N}$ or $\mathbb{S}$' representing an element of $\mathbb{N}$ or $\mathbb{S}$, and write $\mathcal{V}_{\mathbb{N}}$ or $\mathcal{V}_{\mathbb{S}}$, respectively. Note that $\mathcal{V}_{\mathbb{N}}$ and $\mathcal{V}_{\mathbb{S}}$ are the images of $\ulcorner - \urcorner_{\mathbb{N}}$ and $\ulcorner - \urcorner_{\mathbb{S}}$.

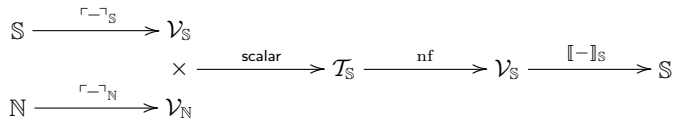


Fig. 2. Productive behaviour of scalar

Third, consider the productivity of the scalar function in the running example. Given $v \in \mathcal{V}_{\mathbb{S}}$ and $w \in \mathcal{V}_{\mathbb{N}}$, we can construct a term $\mathsf{scalar}(v, w)$. Here we define the productivity of scalar, as

> the scalar function is productive if $\mathsf{scalar}(v, w)$ has a unique normal form u in $\mathcal{V}_{\mathbb{S}}$, viz. $\mathsf{scalar}(v, w) \twoheadrightarrow\!\!\!\!\!\rightarrow u \in \mathcal{V}_{\mathbb{S}}$ for every $v \in \mathcal{V}_{\mathbb{S}}$ and $w \in \mathcal{V}_{\mathbb{N}}$.

This is the surface aspect of productivity. We write $\mathrm{nf}(\mathsf{scalar}(v, w))$ to denote that u. Notice the restrictions $v \in \mathcal{V}_{\mathbb{S}}$ and $w \in \mathcal{V}_{\mathbb{N}}$; in this respect, the productivity property is not so strong as SN^{∞}, and those restrictions imply also that our system will be sorted. On the other hand, notice also the restriction $u \in \mathcal{V}_{\mathbb{S}}$; in this respect, the productivity property is not so weak as SN^{∞}. Thus, productivity can be regarded as SN^{∞} from a certain class of terms to a certain class of normal forms.

Finally, we project the above computed u from $\mathcal{V}_{\mathbb{S}}$ back to $\mathbb{S}$ by $\mathsf{cons} \overset{[\![-]\!]_{\mathbb{S}}}{\longmapsto} \mathit{cons}$, i.e. $\mathsf{cons}(n, x) \overset{[\![-]\!]_{\mathbb{S}}}{\longmapsto} \mathit{cons}([\![n]\!]_{\mathbb{N}}, [\![x]\!]_{\mathbb{S}})$ coinductively defined. On the other hand, $[\![-]\!]_{\mathbb{N}}$ is inductively given by $\mathsf{0} \overset{[\![-]\!]_{\mathbb{N}}}{\longmapsto} 0$ and $\mathsf{s}(n) \overset{[\![-]\!]_{\mathbb{N}}}{\longmapsto} S[\![n]\!]_{\mathbb{N}}$.

Figure 2 summarises the whole story of this subsection, with an example case of the scalar function, where $\mathcal{G}_{\mathcal{S}}$ is the set of the proper ground terms, which will be defined in the next subsection (Definition 2).

3.2 Algorithmic Systems

In this subsection, we formalise the notion of algorithmic systems, and productivity of those systems. First of all, we formalise properness of terms, making explicit which kind of terms we wish to evaluate, and which kind of constructor normal forms we expect as results of computation. Productivity will be then defined as infinitary normalisation from proper terms to proper constructor normal forms.

We divide the symbols Σ into two kinds: The set Σ^{F} of *(defined) function symbols* consists of the symbols that occur at the root of the lhs of a rule, i.e. $\Sigma^{\mathsf{F}} = \{l(\epsilon) \mid (l, r) \in \mathcal{R}\}$. The set Σ^{C} of *constructor (function) symbols* consists of the others; $\Sigma^{\mathsf{C}} = \Sigma \setminus \Sigma^{\mathsf{F}}$. A term which contains only constructor symbols is called a *constructor normal form*. Note that every constructor normal form is a normal form.

We fix the set $\mathcal{S}$ of *sorts*, and also divide it into $\mathcal{S}^{\mathsf{I}}$ of *inductive sorts* and $\mathcal{S}^{\mathsf{C}}$ of *coinductive sorts*. According to this division on $\mathcal{S}$, a constructor symbol c sorted $S_1 \times \cdots \times S_n \to T$ is called *inductive constructor symbol* if $T \in \mathcal{S}^{\mathsf{I}}$; *coinductive constructor symbol* if $T \in \mathcal{S}^{\mathsf{C}}$. We write Σ^{CI} and Σ^{CC} to denote

the sets of inductive and coinductive constructor symbols, respectively. For the running example, we set $\mathcal{S}^{\mathsf{I}}$ and $\mathcal{S}^{\mathsf{C}}$ as in the rightside of Figure 1. The following figure summarises the partition of Σ.

$$\text{symbols } (\Sigma) \begin{cases} \text{function symbols } (\Sigma^{\mathsf{F}}) \\ \text{constructor symbols } (\Sigma^{\mathsf{C}}) \begin{cases} \text{inductive constructor symbols } (\Sigma^{\mathsf{CI}}) \\ \text{coinductive constructor symbols } (\Sigma^{\mathsf{CC}}) \end{cases} \end{cases}$$

Definition 2 (Proper Terms).

1. An infinite path in a term t is an infinite sequence of positive integers such that every finite prefix of it is in $\mathrm{dom}(t)$.
2. An infinite path $n_0 n_1 n_2 \dots$ in t is a *forbidden path* if there exist infinitely many i such that $t(n_0 \dots n_i) \notin \Sigma^{\mathsf{CC}}$.
3. A term t is *proper* if there exists no forbidden path in t.

We write $\mathcal{T}$ to denote the set of proper terms; $\mathcal{T}^{\mathsf{fin}}$ for finite terms; $\mathcal{G}$ for proper ground terms; $\mathcal{N}$ for constructor normal forms; $\mathcal{V}$ for proper constructor normal forms ($\mathcal{T} \cap \mathcal{N}$). We say that a substitution σ is proper if σ maps every variable to a proper term of the required sort.

Thus, $\mathcal{T}_{\mathbb{S}}$ in Figure 2 denotes the set of proper terms of the sort $\mathbb{S}$. Note that $\mathcal{V}_{\mathbb{N}}$ and $\mathcal{V}_{\mathbb{S}}$ as seen in the top of this section exactly consist of the proper constructor normal forms of the respective sorts. So, we refer to proper constructor normal forms as *values*. It should be noticed that proper terms are allowed to be infinite only due to the construction of coinductive objects. This stipulation will be technically discussed in the next section.

We are now prepared to formalise our notion of algorithmic term rewriting systems and productivity of those systems.

Definition 3.
- The system is *algorithmic* if the following conditions are satisfied.
  - (Alg–1) The system is orthogonal.
  - (Alg–2) The lhs of every rule contains exactly one function symbol, which occurs only at the root.
  - (Alg–3) The lhs and the rhs of each rule are proper and of the same sort.
  - (Alg–4) Every well-sorted term of the form $f(v_1, \dots, v_n)$ has a redex at the root, where f is a function symbol and $v_1, \dots, v_n$ are values.
- A function symbol f is *productive* if for every well-sorted $t \equiv f(v_1, \dots, v_n)$ where $v_1, \dots, v_n \in \mathcal{V}$, there exists a unique $s \in \mathcal{V}$ such that $t \twoheadrightarrow s$.
- An algorithmic system is *productive* if every function symbol is productive.

One who is familiar with functional programming will recognise the condition (Alg–2) stating that every rule is of the form of a pattern-matching, and the condition (Alg–4) guarantees that those pattern-matching covers every pattern of constructor prefixes.

Remark 4. The primes function as in the head of the Introduction is productive, but the whole system is not productive because of the term $\mathsf{sieve}(0 : 0 : \dots)$. So, productivity may be understood in a local and a global way, referring to individual function symbols or the whole system. In the present work, we focus on the latter.

The next proposition will help us to check orthogonality:

Proposition 5. *Given the condition* (Alg–2)*, the possible overlaps of the rules are all root-overlaps.*

Proof. We recall the definition of overlap: a pair of rules $(l, r), (l', r') \in \mathcal{R}$ overlaps if there exists $p \in \mathrm{dom}(l)$ and substitutions σ, τ such that $(l/p)\sigma \equiv l'\tau$, $l(p) \notin \mathcal{X}$, and $(l \not\equiv l'$ or $p \neq \epsilon)$. If $p = \epsilon$, we call the overlap root-overlap. Thus, we suppose $(l/p)\sigma \equiv l'\tau$ and $l(p) \notin \mathcal{X}$ to show $p = \epsilon$. By the definition of Σ^{F}, we have $l'\tau(\epsilon) \in \Sigma^{\mathsf{F}}$, Since $l(p) \notin \mathcal{X}$, we have $((l/p)\sigma)(\epsilon) = (l/p)(\epsilon) = l(p)$. Therefore, $l(p) = l'\tau(\epsilon) \in \Sigma^{\mathsf{F}}$. From the condition (Alg–2), $l(p) \notin \Sigma^{\mathsf{F}}$ unless $p = \epsilon$. □

Observe that our running example is algorithmic.

4 Analysing Productivity

From now on, we assume that the system is algorithmic, and analyse its productivity.

We divide the productivity into two conditions:

Definition 6.
- The system is *domain normalising* (DN) if $t \in \mathcal{G}$, $t \twoheadrightarrow s$, and $s \in \mathcal{N}$ implies $s \in \mathcal{V}$.
- The system is *constructor normalising* (CN) if for every $t \in \mathcal{G}$, there exists a unique $s \in \mathcal{N}$ such that $t \twoheadrightarrow s$. We write $\mathrm{nf}(t)$ to denote that s.

Clearly, productivity is the conjunction of DN and CN (cf. Proposition 1). The DN property assures that normalisation leads all the proper terms to the intended domain, and CN guarantees that every normal form is fully evaluated, viz. contains no function symbol.

In the following two sections we will study a way to recognise these DN and CN properties by observing the behaviour of each function symbol.

5 Domain Normalisation

This section studies the DN property, which is closely related to the notion of properness.

For a term t, we write $\mathbf{I}^t$ or $\mathbf{F}^t$ to denote the uppermost positions where a inductive constructor symbol or a function symbol occurs, respectively. We slice the values $\mathcal{V}$ by countable ordinals $\mathbf{\Omega}$.

Definition 7. For each $\alpha \in \mathbf{\Omega}$, $\mathcal{V}^\alpha \subseteq \mathcal{V}$ is defined by transfinite induction on $\mathbf{\Omega}$ by $\mathcal{V}^\alpha = \{v \in \mathcal{V} \mid \forall p \in \mathbf{I}^v.\, \exists \beta < \alpha.\, \forall n.\, (p \cdot n \in \mathrm{dom}(v) \Rightarrow v/(p \cdot n) \in \mathcal{V}^\beta)\}$. Note that $\mathcal{V}^\alpha \subseteq \mathcal{V}^\beta$ if $\alpha \leqslant \beta$.

The above definition is a little technical because of the infinitary construction of values. Roughly, if $c(v_1, \ldots, v_n) \in \mathcal{V}^\alpha$, then each argument $v_1, \ldots, v_n$ must be in $\mathcal{V}^\alpha$, and moreover, if c is not a coinductive constructor symbol, then each argument must be in the class indexed by a smaller ordinal.

Lemma 8. *For every value v, there exists $\alpha \in \mathbf{\Omega}$ such that $v \in \mathcal{V}^\alpha$.*

Proof. Let v be a constructor normal form such that $v \notin \mathcal{V}^\alpha$ for any α. To show that v is not proper will suffice. Let $P = \{p \cdot n \in \mathrm{dom}(v) \mid v(p) \in \Sigma^{\mathsf{CC}}\}$. If there exists β_p such that $v/p \in \mathcal{V}^{\beta_p}$ for every $p \in P$, then we have $v \in \mathcal{V}^\beta$ where $\beta = \sup_{p \in P} \beta_p$, which conflicts to the assumption. Hence, we can find $p \in P$ such that v/p does not belong to any $\mathcal{V}^\alpha$. Let $v' = v/p$ and iterate this argument. Thus, we construct a forbidden path in v. □

Now, we formalise the notion of 'DN certificate'.

Definition 9. – Given monotonous functions $[\![f]\!]_{\Sigma^{\mathsf{F}}} : \mathbf{\Omega}^n \to \mathbf{\Omega}$ for each n-ary function symbol f, and ordinals $[\![x]\!]_{\mathcal{X}} \in \mathbf{\Omega}$ for each variable $x \in \mathcal{X}$, we generate the function $[\![-]\!]_{\mathcal{T}^{\mathsf{fin}}} : \mathcal{T}^{\mathsf{fin}} \to \mathbf{\Omega}$ inductively by

$$
\begin{aligned}
[\![c(t_1, \ldots, t_n)]\!]_{\mathcal{T}^{\mathsf{fin}}} &= \sup\{[\![t_1]\!]_{\mathcal{T}^{\mathsf{fin}}}, \ldots, [\![t_n]\!]_{\mathcal{T}^{\mathsf{fin}}}\} + \begin{cases} 0 & (c \in \Sigma^{\mathsf{CC}}) \\ 1 & (c \notin \Sigma^{\mathsf{CC}}) \end{cases} \\
[\![f(t_1, \ldots, f_n)]\!]_{\mathcal{T}^{\mathsf{fin}}} &= [\![f]\!]_{\Sigma^{\mathsf{F}}}([\![t_1]\!]_{\mathcal{T}^{\mathsf{fin}}}, \ldots, [\![t_n]\!]_{\mathcal{T}^{\mathsf{fin}}}) \\
[\![x]\!]_{\mathcal{T}^{\mathsf{fin}}} &= [\![x]\!]_{\mathcal{X}}.
\end{aligned}
$$

– A tuple of monotonous functions $[\![f]\!]_{\Sigma^{\mathsf{F}}} : \mathbf{\Omega}^n \to \mathbf{\Omega}$ forms a *DN certificate* if for every variable assignment $[\![-]\!]_{\mathcal{X}} : \mathcal{X} \to \mathbf{\Omega}$, the above generated $[\![-]\!]_{\mathcal{T}^{\mathsf{fin}}}$ satisfies $[\![l]\!]_{\mathcal{T}^{\mathsf{fin}}} \geqslant [\![r]\!]_{\mathcal{T}^{\mathsf{fin}}}$ for every $(l, r) \in \mathcal{R}$.

Theorem 10. *The system is* DN *if there exists a DN certificate.*

Proof. Let $[\![-]\!]_{\mathcal{G}} : \mathcal{G} \to \mathbf{\Omega}$ be a function recursively given as follows:

1. Let $t \in \mathcal{G}$.
2. For every $p \in \mathbf{I}^t$, let $\kappa_p = \max\{[\![t/(p \cdot i)]\!]_{\mathcal{G}} \mid i = 1, \ldots, n\} + 1$, where n is the arity of $t(p)$.
3. For every $p \in \mathbf{F}^t$, let $\kappa_p = [\![t(p)]\!]_{\Sigma^{\mathsf{F}}}([\![t/(p \cdot 1)]\!]_{\mathcal{G}}, \ldots, [\![t/(p \cdot n)]\!]_{\mathcal{G}})$, where n is the arity of $t(p)$.
4. Let $[\![t]\!]_{\mathcal{G}} = \sup\{\kappa_p \mid p \in \mathbf{I}^t \cup \mathbf{F}^t\}$, where we stipulate $\sup \emptyset = 0$.

Observe that this $[\![-]\!]_{\mathcal{G}}$ is well-defined (cf. the proof of Lemma 8). By the definition of DN certificate, $t \twoheadrightarrow s$ implies $[\![t]\!]_{\mathcal{G}} \geqslant [\![s]\!]_{\mathcal{G}}$.

Suppose $t \twoheadrightarrow\!\!\!\!\!\rightarrow s$ and $s \in \mathcal{N}$, and assume that $s \notin \mathcal{V}$. Then, we have $\mathbf{F}^s = \emptyset$ and can find $p \in \mathbf{I}^s$ such that s/p is not proper. From the definition of strong

convergence, we can find t' such that $t \twoheadrightarrow t'$ and t' agrees on s up to the position p. Note that $[\![t]\!]_{\mathcal{G}} \geqslant [\![t']\!]_{\mathcal{G}} > [\![t'/p]\!]_{\mathcal{G}}$ and that $t'/p \twoheadrightarrow s/p$. Replace t and s respectively by t'/p and s/p, and iterate this argument. That produces an infinite descending chain of ordinals. Hence, s must be proper. □

To show how we can find a DN certificate, we consider the following subsystem of the running example.

$$\begin{aligned}
\mathsf{add}(n, \mathsf{0}) &\to n \\
\mathsf{add}(n, \mathsf{s}(m)) &\to \mathsf{s}(\mathsf{add}(n, m)) \\
\mathsf{mul}(n, \mathsf{0}) &\to \mathsf{0} \\
\mathsf{mul}(n, \mathsf{s}(m)) &\to \mathsf{add}(\mathsf{mul}(n, m), n) \\
\mathsf{scalar}(n : x, m) &\to \mathsf{mul}(n, m) : \mathsf{scalar}(x, m)
\end{aligned}$$

Then, compute $[\![l]\!]_{\mathcal{T}^{\mathsf{fin}}}$ and $[\![r]\!]_{\mathcal{T}^{\mathsf{fin}}}$ for each rule (l, r), to get the inequalities that a DN certificate has to satisfy:

$$\begin{aligned}
[\![\mathsf{add}]\!]_{\Sigma^{\mathsf{F}}}([\![n]\!]_{\mathcal{X}}, 1) &\geqslant [\![n]\!]_{\mathcal{X}} \\
[\![\mathsf{add}]\!]_{\Sigma^{\mathsf{F}}}([\![n]\!]_{\mathcal{X}}, [\![m]\!]_{\mathcal{X}} + 1) &\geqslant [\![\mathsf{add}]\!]_{\Sigma^{\mathsf{F}}}([\![n]\!]_{\mathcal{X}}, [\![m]\!]_{\mathcal{X}}) + 1 \\
[\![\mathsf{mul}]\!]_{\Sigma^{\mathsf{F}}}([\![n]\!]_{\mathcal{X}}, 1) &\geqslant 1 \\
[\![\mathsf{mul}]\!]_{\Sigma^{\mathsf{F}}}([\![n]\!]_{\mathcal{X}}, [\![m]\!]_{\mathcal{X}} + 1) &\geqslant [\![\mathsf{add}]\!]_{\Sigma^{\mathsf{F}}}([\![\mathsf{mul}]\!]_{\Sigma^{\mathsf{F}}}([\![n]\!]_{\mathcal{X}}, [\![m]\!]_{\mathcal{X}}), [\![n]\!]_{\mathcal{X}}) \\
[\![\mathsf{scalar}]\!]_{\Sigma^{\mathsf{F}}}(\max\{[\![n]\!]_{\mathcal{X}}, [\![x]\!]_{\mathcal{X}}\}, [\![m]\!]_{\mathcal{X}}) &\geqslant [\![\mathsf{mul}]\!]_{\Sigma^{\mathsf{F}}}([\![n]\!]_{\mathcal{X}}, [\![m]\!]_{\mathcal{X}}) \\
[\![\mathsf{scalar}]\!]_{\Sigma^{\mathsf{F}}}(\max\{[\![n]\!]_{\mathcal{X}}, [\![x]\!]_{\mathcal{X}}\}, [\![m]\!]_{\mathcal{X}}) &\geqslant [\![\mathsf{scalar}]\!]_{\Sigma^{\mathsf{F}}}([\![x]\!]_{\mathcal{X}}, [\![m]\!]_{\mathcal{X}}).
\end{aligned}$$

It is not so difficult to find a tuple of functions which satisfies the above inequalities. For example, let $[\![\mathsf{add}]\!]_{\Sigma^{\mathsf{F}}}(\alpha, \beta) = \alpha + \beta$, $[\![\mathsf{mul}]\!]_{\Sigma^{\mathsf{F}}}(\alpha, \beta) = \alpha \cdot \beta$, and $[\![\mathsf{scalar}]\!]_{\Sigma^{\mathsf{F}}}(\alpha, \beta) = \sup\{\iota \cdot \beta \mid \iota < \alpha\}$. Similarly, we can find $[\![\mathsf{merge}]\!]_{\Sigma^{\mathsf{F}}}(\alpha, \beta) = \sup\{\alpha, \beta\}$, $[\![\mathsf{cmp}]\!]_{\Sigma^{\mathsf{F}}}(\alpha, \beta) = 0$, $[\![\mathsf{aux}]\!]_{\Sigma^{\mathsf{F}}}(\alpha, \beta, \gamma) = \sup\{\beta, \gamma\}$, and $[\![\mathsf{ham2}]\!]_{\Sigma^{\mathsf{F}}}() = [\![\mathsf{ham3}]\!]_{\Sigma^{\mathsf{F}}}() = [\![\mathsf{ham5}]\!]_{\Sigma^{\mathsf{F}}}() = [\![\mathsf{Ham}]\!]_{\Sigma^{\mathsf{F}}}() = \omega$ for a DN certificate for the whole system. Observe that the above functions certainly form a DN certificate. Thus, the running example is DN.

Remark 11. Admittedly, it requires a little ingenuity to find a DN certificate. However, for many algorithmic systems, we would be able to compute a DN certificate automatically; those functions are constructed by the following operations: taking the successor, taking the maximal of finite ordinals, and accumulating operations. The set of countable ordinals is closed under those operations.

6 Constructor Normalisation

In [5], we have performed a 'data-oblivious' analysis, with a focus on the number of leading data that are already evaluated. Also in the present work, we perform quantitative analysis on the abstract coinductive data-space.

In this section, we assume the existence of a DN certificate $[\![-]\!]_{\Sigma^{\mathsf{F}}}$.

Definition 12. Let $t \in \mathcal{G}$.

- For $p \in \mathrm{dom}(t)$, the *coinductive depth of p in t*, written $\lfloor p \rfloor_t$, is the number of coinductive constructor symbols on the finite path from the root up to p, i.e.

$$\lfloor \epsilon \rfloor_t = 0$$
$$\lfloor p \cdot n \rfloor_t = \lfloor p \rfloor_t + \begin{cases} 1 & (t(p) \in \Sigma^{\mathsf{CC}}) \\ 0 & (\text{otherwise}) \end{cases}$$

- The *rank* of t, written $\mathrm{rank}(t)$, is the minimal depth where a function symbol occurs. Formally, $\mathrm{rank}(t) = \inf\{\lfloor p \rfloor_t \mid t(p) \in \Sigma^{\mathsf{F}}\}$ where we stipulate $\inf \emptyset = \infty$. Note that $\mathrm{rank}(t) = \infty$ iff $t \in \mathcal{V}$.
- The *potential rank* of t, written $\overrightarrow{\mathrm{rank}}^*(t)$, is given by $\overrightarrow{\mathrm{rank}}^*(t) = \sup\{\mathrm{rank}(s) \mid t \twoheadrightarrow s\}$.

Lemma 13. *If $\overrightarrow{\mathrm{rank}}^*(t) = \infty$, then the normal form $\mathrm{nf}(t)$ exists. Moreover, $\mathrm{nf}(t) \in \mathcal{N}$ if $t \in \mathcal{G}$.*

Proof. We can find $t_n \in \mathcal{G}$ such that $t \twoheadrightarrow t_n$ and $\mathrm{rank}(t_n) = n$ for every $n \in \mathbb{N}$. Let $t'_{-1} = t$ and t'_n be a common reduct of t'_{n-1} and t_n. Then, $t \twoheadrightarrow t'_0 \twoheadrightarrow t'_1 \twoheadrightarrow \ldots$ witnesses $t \twoheadrightarrow\!\!\!\!\twoheadrightarrow \lim t'_n$, and $\mathrm{rank}(\lim t'_n) = \infty$. The latter claim is obvious. □

Definition 14. A tuple (l, r', k) where $l, r' \in \mathcal{T}^{\mathsf{fin}}, k \in \mathbb{N}$ is a *dependency* if there exists $(l, r) \in \mathcal{R}$ such that $r' = r/p$ where p is an uppermost occurrence of a function symbol in r, and $k = \lfloor p \rfloor_r$. We write Δ to denote the set of dependencies, and let $\Delta_f = \{(l, r', k) \in \Delta \mid l(\epsilon) = f\}$ for every $f \in \Sigma^{\mathsf{F}}$.

Roughly, dependency indicates dependence of evaluation. For example, from the rule $\mathsf{scalar}(n : x, m) \to \mathsf{mul}(n, m) : \mathsf{scalar}(x, m)$, we have $\Delta_{\mathsf{scalar}} = \{\delta_1, \delta_2\}$ where $\delta_1 = (\mathsf{scalar}(n : x, m), \mathsf{mul}(n, m), 0)$ and $\delta_2 = (\mathsf{scalar}(n : x, m), \mathsf{scalar}(x, m), 1)$. So, for terms $t_n, t_x, t_m \in \mathcal{G}$, to have $\overrightarrow{\mathrm{rank}}^*(\mathsf{scalar}(t_n : t_x, t_m)) \geqslant i$, we need $\overrightarrow{\mathrm{rank}}^*(\mathsf{mul}(t_n, t_m)) \geqslant i$ and $\overrightarrow{\mathrm{rank}}^*(\mathsf{scalar}(t_x, t_m)) + 1 \geqslant i$. Moreover, it should be noticed that the funciton mul does not depend on scalar so that the dependency δ_1 will not occur for infinitely many times in a chain of dependencies. On the other hand, δ_2 can be infinitely nested, because of the recursive call. But, not all recursive calls cause infinitely nested dependencies. For counterexample, consider the rule $\mathsf{add}(n, \mathsf{s}(m)) \to \mathsf{s}(\mathsf{add}(n, m))$, from which a dependency $\delta_3 = (\mathsf{add}(n, \mathsf{s}(m)), \mathsf{add}(n, m), 0)$ arises. Since we do not allow s continually infinitely nested, the other rule $\mathsf{add}(n, \mathsf{0}) \to n$ will be eventually applied. So, we divide Δ into two kinds: those can be continually infinitely nested such as δ_2, and those can be not, such as δ_1 and δ_3.

Definition 15. Let $\langle \Xi, \preccurlyeq \rangle$ be a well-founded partial order. Then, a *complexity index* consists of functions $\xi^f : \mathbf{\Omega}^n \to \Xi$ for each n-ary function symbol f. For a proper term t such that $t(\epsilon) \in \Sigma^{\mathsf{F}}$, we write $\xi(t)$ to denote $\xi^{t(\epsilon)}(\llbracket t/1 \rrbracket_{\mathcal{G}}, \ldots, \llbracket t/n \rrbracket_{\mathcal{G}})$. Given a complexity index, a dependency (l, r', k) is

decreasing if $\xi(l) \succ \xi(r')$ for any $[\![-]\!]_{\mathcal{X}} : \mathcal{X} \to \mathbf{\Omega}$; *infinitely nestable* otherwise. We write $\Delta^{>}$ and Δ^{∞} to denote decreasing and infinitely nestable dependencies, respectively.

For the running example, we set

$$\begin{aligned}\Xi =& \{\mathsf{add}_\iota, \mathsf{mul}_\iota, \mathsf{cmp}_\iota \mid \iota \in \mathbf{\Omega}\} \\ &\cup \{\mathsf{merge}, \mathsf{aux}, \mathsf{nats}, \mathsf{scalar}, \mathsf{ham2}, \mathsf{ham3}, \mathsf{ham5}, \mathsf{Ham}\}\end{aligned}$$

with $\mathsf{nats} \prec \mathsf{cmp}_{\iota_1} \prec \mathsf{add}_{\iota_2} \prec \mathsf{mul}_{\iota_3} \prec \mathsf{scalar} \prec \mathsf{ham2} \prec \mathsf{ham3} \prec \mathsf{ham5} \prec \mathsf{aux} \prec \mathsf{merge} \prec \mathsf{Ham}$ for every $\iota_1, \iota_2, \iota_3 \in \mathbf{\Omega}$, and $\mathsf{cmp}_\iota \prec \mathsf{cmp}_\kappa$, $\mathsf{add}_\iota \prec \mathsf{add}_\kappa$, and $\mathsf{mul}_\iota \prec \mathsf{mul}_\kappa$ if $\iota < \kappa$. The complexity index is given by $\xi^{\mathsf{cmp}}(\iota) = \mathsf{cmp}_\iota$, $\xi^f(\kappa, \iota) = f_\iota$ for $f = \mathsf{add}, \mathsf{mul}$, and $\xi^f(\dots) = f$ for the other function symbols. Then, the infinitely nestable dependencies in the running example are those from aux, δ_2 as mentioned above, and $(\mathsf{nats}(n), \mathsf{nats}(\mathsf{s}(n)), 1)$ from the nats rule.

Definition 16. For an n-ary function symbol f and $i = 1, \dots, n$, the *lookahead of f at i*, written LA_i^f, is defined by $LA_i^f = \sup\{\lfloor i \cdot p \rfloor_l + 1 \mid l(i \cdot p) \notin \mathcal{X}\}$, where we stipulate $\sup \emptyset = 0$.

In data-oblivious analysis, lookahead indicates the required rank of each argument to determine which rule to apply.

Definition 17. 1. A *CN estimation* ρ consists of functions $\rho^f : \overline{\mathbb{N}}^n \to \overline{\mathbb{N}}$ for every n-ary function symbol f.
2. The *minimal CN estimation* o is given by $o^f(\dots) = 0$ for every $f \in \Sigma^{\mathsf{F}}$.
3. Given $E \subseteq \Delta$ and a CN estimation ρ, $E\rho$ is the CN estimation given by

$$(E\rho)^f(k_1, \dots, k_n) = \begin{cases} 0 & (\exists i.\, k_i < LA_i^f) \\ \inf\{[\![r']\!] + k \mid (l, r', k) \in E, l(\epsilon) = f\} & (\text{otherwise}) \end{cases}$$

where $[\![-]\!] : \mathcal{T}^{\mathsf{fin}} \to \overline{\mathbb{N}}$ is inductively defined by

$$\begin{aligned} [\![c(t_1, \dots, t_m)]\!] &= \inf\{[\![t_i]\!] + 1 \mid i = 1, \dots, m\} && (c \in \Sigma^{\mathsf{CC}}) \\ [\![c(t_1, \dots, t_m)]\!] &= \inf\{[\![t_i]\!] \mid i = 1, \dots, m\} && (c \in \Sigma^{\mathsf{CI}}) \\ [\![g(t_1, \dots, t_m)]\!] &= \rho^g([\![t_1]\!], \dots, [\![t_m]\!]) && (g \in \Sigma^{\mathsf{F}}) \\ [\![x]\!] &= k_i - \lfloor i \cdot p \rfloor_l. && (x \in \mathcal{X}, l(i \cdot p) = x) \end{aligned}$$

4. The *approximated CN estimation* π is given by $\pi^f(\dots) = \sup \pi^f_{i0}(\dots)$ where

$$\begin{aligned} \pi_{00} &= o \\ \pi_{i,j+1} &= \Delta^{>}\pi_{i,j} \\ \pi^f_{i+1,j}(\dots) &= \Delta^{\infty}(\inf_j \pi^f_{i,j}(\dots)) \end{aligned}$$

Roughly, the approximated CN estimation approximates the production of each function, viz. a term $f(t_1, \ldots, t_n)$ has the potential rank at least $\pi^f(\overrightarrow{\mathrm{rank}}^*(t_1), \ldots, \overrightarrow{\mathrm{rank}}^*(t_n))$. First, we start from the minimal estimation o, since any term has the potential rank at least 0. The recursive definition of π_{ij} reflects the fact that decreasing dependencies are never be infinitely nestedly applied.

Thus, the following theorem holds:

Theorem 18. *If* $\pi^f(\infty, \ldots, \infty) = \infty$ *for every* $f \in \Sigma^{\mathsf{F}}$*, then the system is* CN.

Proof. We can prove that $\pi^f_{n0}(\overrightarrow{\mathrm{rank}}^*(t_1), \ldots, \overrightarrow{\mathrm{rank}}^*(t_n)) \leqslant \overrightarrow{\mathrm{rank}}^*(f(t_1, \ldots, t_n))$ for every n, by nested induction on $\mathbb{N}$ and Ξ. The claim will follow from the above definition and Lemma 13. □

Observe that the running example satisfies the condition, and therefore CN.

7 Conclusion

We have presented a new framework of algorithmic systems, and given criteria to recognise productivity of an algorithmic system.

The productivity of stream (or list) specifications has been studied from many directions [2, 5, 6, 10, 11, 14]. The mixture of inductive and coinductive specifications has been also studied in [1, 9].

As to future work, we wish to automate or partially automate the procedure to find a DN certificate. CN estimation will be closely related to the production function and I/O sequence in [5].

Acknowledgement

I would thank Roel de Vrijer and Jan Willem Klop for their useful comments.

References

1. A. Abel. Mixed inductive/coinductive types and strong normalization. In *APLAS 2007*, pp. 286–301, 2007.
2. Th. Coquand. Infinite Objects in Type Theory. In H. Barendregt, T. Nipkow, eds., *TYPES*, vol. 806, pp. 62–78. Springer-Verlag, Berlin, 1994.
3. R. Dedekind. *Was sind und was sollen die Zahlen?* Vieweg, 1888.
4. J. Endrullis, C. Grabmayer, D. Hendriks. Data-Oblivious Stream Productivity. Available via: `http://infinity.few.vu.nl/productivity/`, January 2008.
5. J. Endrullis, C.A. Grabmayer, D. Hendriks, A. Isihara, J.W. Klop. Productivity of stream definitions. In *FCT 2007*, LNCS 4639, pp. 274–287, 2007.
6. J. Hughes, L. Pareto, A. Sabry. Proving the correctness of reactive systems using sized types. In *POPL '96*, pp. 410–423, 1996.
7. R. Kennaway, J.W. Klop, M.R. Sleep, F.J. de Vries. Transfinite reductions in orthogonal term rewriting systems. *Information and Computation*, 119(1):18–38, 1995.

8. J.W. Klop, R.C. de Vrijer. Infinitary normalization. In S. Artemov, H. Barringer, A.S. d'Avila Garcez, L.C. Lamb, J. Woods, eds., *We Will Show Them: Essays in Honour of Dov Gabbay*, vol. 2, pp. 169–192. College Publications, 2005.
9. M. Niqui. Productivity of Edalat-Potts exact arithmetic in constructive type theory. *Theory of Computing Systems*, 41(1):127–154, 2007.
10. B.A. Sijtsma. On the productivity of recursive list definitions. *ACM Transactions on Programming Languages and Systems*, 11(4):633–649, 1989.
11. A. Telford, D. Turner. Ensuring the productivity of infinite structures. Technical Report 14-97, The Computing Laboratory, University of Kent at Canterbury, 1997.
12. Terese. *Term Rewriting Systems*. Cambridge University Press, 2003.
13. A. Turing. On computable numbers, with an application to the entscheidungsproblem. In *Proceedings of the London Mathematical Society*, number 2 in 42, pp. 230–265, 1936.
14. W. W. Wadge. An extensional treatment of dataflow deadlock. *TCS*, 13:3–15, 1981.

Strong Normalization of Polymorphic Calculus for Delimited Continuations

Yukiyoshi Kameyama[1] and Kenichi Asai[2]

[1] Department of Computer Science, University of Tsukuba
kameyama@acm.org
[2] Department of Information Science, Ochanomizu University
asai@is.ocha.ac.jp

Abstract. The notion of delimited continuations has been proved useful in various areas of computer programming such as partial evaluation, mobile computing, and web transaction. In our previous work, we proposed polymorphic calculi with control operators for delimited continuations. This paper presents a proof of strong normalization (SN) of these calculi based on a refined (i.e. administrative redex-free) CPS translation.
Keywords: Type System, Delimited Continuation, Control Operator, CPS Translation, Predicative Polymorphism.

1 Introduction

Control operators in functional languages allow the explicit manipulation of control flow of programs, and thus give more flexibility and expressiveness than those programs without them. Scheme has the control operator "call/cc" for continuations, which has been intensively studied from theory to implementation and practical applications. Type-theoretic studies on "call/cc" have revealed that it corresponds to classical logic [10].

Delimited continuation is a similar notion to (unlimited) continuation, but it represents *part* of the rest of the computation rather than whole rest of the computation. Since Danvy and Filinski have proposed the control operators "shift" and "reset" for delimited continuations [6], they have been proved useful in various applications such as backtracking [6], A-normalization in direct style [2], let-insertion in partial evaluation [17], type-safe "printf" [3], and web transaction [15].

This paper investigates the type structure of "shift" and "reset", and in particular, proves strong normalization of a polymorphic calculus for them. In our previous work [4], we have introduced a polymorphic type system for "shift" and "reset", and proved a number of properties such as type soundness (subject reduction and progress). The strong normalization property, however, was only mentioned as a theorem without a proof, which is the subject of the present paper.

Strong normalization (SN) is the property that no reduction sequence can be infinite, and is considered as one of the most fundamental properties of many typed lambda calculi which correspond to logical systems under the Curry-Howard isomorphism. For instance, strong normalization of Girard's System F [9] implies its (logical) consistency.

On the other hand, strong normalization of computational calculi with control operators is a subtle issue as shown by the following list:

$$
\begin{array}{lll}
v ::= c \mid x \mid \lambda x.e & & \text{value} \\
e ::= v \mid e_1 e_2 \mid \mathcal{S}k.e \mid \langle e \rangle \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2 & & \\
\quad \mid \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 & & \text{expression}
\end{array}
$$

Fig. 1. Syntax of $\lambda_{let}^{s/r}$.

- A typed calculus with "call/cc" is strongly normalizing, while that with exception in Standard ML is not strongly normalizing (see, for instance, [11]).
- The calculus with "shift" and "reset" under Filinski's typing[3] is not normalizing, while, under Danvy and Filinski's type system, it is strongly normalizing [1].
- A typed calculus with "control" and "prompt", the other control operators for delimited continuations, is not normalizing [13]. Similarly, a typed calculus with the control operator `cupto` is not normalizing [14].

Hence, we can say SN for the calculi with control operators is a non-trivial issue. This paper solves the problem for the case of "shift" and "reset" under the polymorphic type system.

The rest of this paper is organized as follows: Section 2 gives the syntax and semantics of the polymorphic calculus for shift and reset in [4], and Section 3 reviews the definitional CPS translation for this calculus. In Section 4 we introduce a refined CPS translation and study its properties. In Section 5, we prove strong normalization of our calculus by making use of the refined CPS translation. Section 6 gives conclusion.

2 A Polymorphic Calculus with Shift/Reset

In this section we introduce the polymorphic typed calculi $\lambda_{let}^{s/r}$ for shift and reset in [4]. Following the literature, we distinguish two versions of polymorphism: *predicative* polymorphism (or let-polymorphism) found in ML families and *impredicative* polymorphism which is based on the second order lambda calculus (Girard's System F [9]). In this paper, we concentrate on the predicative version.

2.1 Syntax and Operational Semantics

We assume that the sets of constants (denoted by c), variables (denoted by x, y, k, f), type variables (denoted by t), and basic types (denoted by b) are mutually disjoint, and that each constant is associated with a basic type. We assume `bool` is a basic type which has constants `true` and `false`.

The syntax of $\lambda_{let}^{s/r}$ is given in Figure 1. A value is either a constant, a variable or a lambda abstraction. An expression is either a value, an application, a shift expression (denoted by

[3] Filinski did not give a type system explicitly, but his well-known implementation of "shift" and "reset" [8] specifies a certain type system.

$$
\begin{aligned}
(\lambda x.e)v &\rightsquigarrow e[v/x] \\
\langle v \rangle &\rightsquigarrow v \\
\langle F[\mathcal{S}k.e] \rangle &\rightsquigarrow \langle \texttt{let}\ k = \lambda x.\langle F[x] \rangle\ \texttt{in}\ e \rangle \\
\texttt{let}\ x = v\ \texttt{in}\ e &\rightsquigarrow e[v/x] \\
\texttt{if true then}\ e_1\ \texttt{else}\ e_2 &\rightsquigarrow e_1 \\
\texttt{if false then}\ e_1\ \texttt{else}\ e_2 &\rightsquigarrow e_2
\end{aligned}
$$

Fig. 2. Reduction rules for $\lambda_{let}^{s/r}$

$\mathcal{S}k.e$), a reset expression (denoted by $\langle e \rangle$), a let expression, or a conditional. Note that we omit the fixpoint operator from the calculus in [4].

Variables are bound by lambda or shift (k is bound in the expression $\mathcal{S}k.e$), and are free otherwise. $\texttt{FV}(e)$ denotes the set of free variables in e.

We give call-by-value operational semantics for $\lambda_{let}^{s/r}$. Contexts, pure evaluation contexts (abbreviated as pure e-contexts), and redexes are defined as follows:

$$
\begin{aligned}
C ::= {} & [\,] \mid \lambda x.C \mid eC \mid Ce \mid \mathcal{S}k.C \mid \langle C \rangle \mid \texttt{let}\ x = C\ \texttt{in}\ e \mid \texttt{let}\ x = e\ \texttt{in}\ C \\
& \mid \texttt{if}\ C\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{if}\ e\ \texttt{then}\ C\ \texttt{else}\ e \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ C && \text{context} \\
F ::= {} & [\,] \mid vF \mid Fe \mid \texttt{if}\ F\ \texttt{then}\ e\ \texttt{else}\ e && \text{pure e-context} \\
R ::= {} & (\lambda x.e)v \mid \langle v \rangle \mid \langle F[\mathcal{S}k.e] \rangle \mid \texttt{let}\ x = v\ \texttt{in}\ e \\
& \mid \texttt{if true then}\ e_1\ \texttt{else}\ e_2 \mid \texttt{if false then}\ e_1\ \texttt{else}\ e_2 && \text{redex}
\end{aligned}
$$

A pure e-context F is an evaluation context such that no reset encloses the hole. Therefore, in the redex $\langle F[\mathcal{S}k.e] \rangle$, the outermost reset is guaranteed to be the one corresponding to this shift, i.e., no reset exists inbetween.

The notion of *one-step reduction* $\rightsquigarrow$ is defined by $C[R] \rightsquigarrow C[e]$ where C is an arbitrary context [4] and $R \rightsquigarrow e$ is an instance of reductions in Figure 2. In this figure, $e[v/x]$ denotes the ordinary capture-avoiding substitution. As usual, $\rightsquigarrow^*$ (and $\rightsquigarrow^+$, resp.) denotes the reflexive-transitive (transitive, resp.) closure of $\rightsquigarrow$.

2.2 Type System

The type system of $\lambda_{let}^{s/r}$ is an extension of Danvy and Filinski's monomorphic type system for shift and reset [5].

Types are defined by Figure 3, which are similar to those in core ML except that the function type is annotated with two answer types as $(\alpha/\gamma \to \beta/\delta)$ where γ (and δ, resp.) denotes the answer type before (after, resp.) the execution of the function body. See Asai and Kameyama [4] for details. A type variable is bound by the universal quantifier $\forall$ as usual, and $\texttt{FTV}(\alpha)$ denotes the set of free type variables in α.

[4] Note that we have slightly extended the notion of one-step reduction from our previous paper [4] where the context enclosing a redex must be an evaluation context, not a general context. Hence the SN property in this paper is slightly stronger than the one in [4].

$$\begin{array}{rl} \alpha, \beta, \gamma, \delta ::= t \mid b \mid (\alpha/\gamma \to \beta/\delta) & \text{monomorphic type} \\ A ::= \alpha \mid \forall t.A & \text{polymorphic type} \end{array}$$

Fig. 3. Types of $\lambda_{let}^{s/r}$.

$$\frac{(x : A \in \Gamma \text{ and } \tau \leq A)}{\Gamma \vdash_p x : \tau}\ \text{var} \qquad \frac{(c \text{ is a constant of basic type } b)}{\Gamma \vdash_p c : b}\ \text{const}$$

$$\frac{\Gamma, x : \sigma;\ \alpha \vdash e : \tau;\ \beta}{\Gamma \vdash_p \lambda x.e : (\sigma/\alpha \to \tau/\beta)}\ \text{fun}$$

$$\frac{\Gamma;\ \gamma \vdash e_1 : (\sigma/\alpha \to \tau/\beta);\ \delta \quad \Gamma;\ \beta \vdash e_2 : \sigma;\ \gamma}{\Gamma;\ \alpha \vdash e_1 e_2 : \tau;\ \delta}\ \text{app} \qquad \frac{\Gamma \vdash_p e : \tau}{\Gamma;\ \alpha \vdash e : \tau;\ \alpha}\ \text{exp}$$

$$\frac{\Gamma, k : \forall t.(\tau/t \to \alpha/t);\ \sigma \vdash e : \sigma;\ \beta}{\Gamma;\ \alpha \vdash \mathcal{S}k.e : \tau;\ \beta}\ \text{shift} \qquad \frac{\Gamma; \sigma \vdash e : \sigma;\ \tau}{\Gamma \vdash_p \langle e \rangle : \tau}\ \text{reset}$$

$$\frac{\Gamma \vdash_p e_1 : \sigma \quad \Gamma, x : \mathtt{Gen}(\sigma; \Gamma);\ \alpha \vdash e_2 : \tau;\ \beta}{\Gamma;\ \alpha \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \tau;\ \beta}\ \text{let}$$

$$\frac{\Gamma;\ \sigma \vdash e_1 : \mathtt{bool};\ \beta \quad \Gamma;\ \alpha \vdash e_2 : \tau;\ \sigma \quad \Gamma;\ \alpha \vdash e_3 : \tau;\ \sigma}{\Gamma;\ \alpha \vdash \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 : \tau;\ \beta}\ \text{if}$$

Fig. 4. Type Inference Rules of $\lambda_{let}^{s/r}$.

A *type context* (denoted by Γ) is a finite list of the form $x_1 : A_1, \cdots, x_n : A_n$ where the variables $x_1, \cdots, x_n$ are mutually distinct, and $A_1, \cdots, A_n$ are (polymorphic) types.

Judgments are either one of the following forms:

$$\begin{array}{rl} \Gamma \vdash_p e : \tau & \text{judgment for pure expression} \\ \Gamma;\ \alpha \vdash e : \tau;\ \beta & \text{judgment for general expression} \end{array}$$

The first form of the judgment signifies the expression e is a pure expression (free from control effects), and the second is for an arbitrary expression. We distinguish pure expressions from other expressions in order to present the restriction of let-polymorphism: polymorphism can be introduced only for pure expressions, as we can see it from the type inference rule for let below.

Figure 4 lists the type inference rules of $\lambda_{let}^{s/r}$ where $\tau \leq A$ in the rule (var) means the instantiation of type variables by monomorphic types. Namely, if $A \equiv \forall t_1.\cdots\forall t_n.\rho$ for some monomorphic type ρ, then $\tau \equiv \rho[\sigma_1, \cdots, \sigma_n/t_1, \cdots, t_n]$ for some monomorphic types $\sigma_1, \cdots, \sigma_n$. The type $\mathtt{Gen}(\sigma; \Gamma)$ in the rule (let) is defined by $\forall t_1.\cdots\forall t_n.\sigma$ where $\{t_1, \cdots, t_n\} = \mathtt{FTV}(\sigma) - \mathtt{FTV}(\Gamma)$.

The type inference rules are a natural extension of the monomorphic type system by Danvy and Filinski [5]. Pure expressions are defined by one of the rules (var), (const), (fun) or (reset). They can be freely turned into general expressions by the rule (exp). Pure expressions can be used polymorphically through the rule (let). It generalizes the standard

let-polymorphism found in ML where the so called value restriction is adopted.[5] Finally, the rule (shift) is extended to cope with the answer type polymorphism of captured continuations: k is given a polymorphic type $\forall t.(\tau/t \to \alpha/t)$.

2.3 Properties

In our previous paper [4], we claimed that our calculus provides a good foundation for studying the interaction between polymorphism and delimited continuations. To support this claim, we have presented the proofs of the following properties: Strong Type Soundness, Existence of Principal Types, and Preservation of types and equality through CPS translation. We have also stated Confluence and Strong Normalization for the calculus, but did not present the proofs.

In this subsection, we quickly review the properties which were proved in [4].[6]

Theorem 1 (Subject Reduction). *If $\Gamma;\ \alpha \vdash e_1 : \tau;\ \beta$ is derivable and $e_1 \rightsquigarrow^* e_2$, then $\Gamma;\ \alpha \vdash e_2 : \tau;\ \beta$ is derivable. Similarly, if $\Gamma \vdash_p e_1 : \tau$ is derivable and $e_1 \rightsquigarrow^* e_2$, then $\Gamma \vdash_p e_2 : \tau$ is derivable.*

Theorem 2 (Progress). *If $\vdash_p \langle e \rangle : \tau$ is derivable, then $\langle e \rangle$ can be reduced.*

By Theorems 1 and 2, we can conclude that our type system is sound (strong type soundness in the sense of [18]).

Theorem 3 (Principal Type and Type Inference). *In $\lambda_{let}^{s/r}$, principal type exists, and we can construct a sound and complete type inference algorithm as an extension of Hindley-Milner's algorithm.*

3 Definitional CPS Translation

A CPS translation is a translation from one calculus (typically with control operators) to a simpler calculus (typically without control operators). It allows us to investigate the semantic structure of the source calculus. The merit of shift and reset over other control operators for delimited continuations comes from the fact that there exists a simple, compositional CPS translation. Danvy and Filinski gave the precise semantics of shift and reset in terms of a CPS translation [6, 7], and based on their translation, various theoretical results as well as applications using shift and reset have been proposed (see, for instance, [12]).

In this section, we present an extension of Danvy and Filinski's CPS translation, namely, a CPS translation from $\lambda_{let}^{s/r}$ to a pure polymorphic lambda calculus λ_{let}. We call this CPS translation as "definitional" one, since it defines the semantics of $\lambda_{let}^{s/r}$.

In the following, we first define the target calculus λ_{let}, and then present the definitional CPS translation.

[5] Note that all values are pure, but pure expressions are not necessarily values.

[6] Strictly speaking, these theorems are extended versions of the corresponding theorems in [4], since the notion of reduction in this paper is slightly extended.

$$\frac{(x : A \in \Gamma \text{ and } \tau \leq A)}{\Gamma \vdash x : \tau}\ \text{var} \qquad \frac{(c \text{ is a constant of basic type } b)}{\Gamma \vdash c : b}\ \text{const}$$

$$\frac{\Gamma, x : \alpha \vdash e : \beta}{\Gamma \vdash \lambda x.e : \alpha \to \beta}\ \text{fun} \qquad \frac{\Gamma \vdash e_1 : \alpha \to \beta \quad \Gamma \vdash e_2 : \alpha}{\Gamma \vdash e_1 e_2 : \beta}\ \text{app}$$

$$\frac{\Gamma \vdash e_1 : \sigma \quad \Gamma, x : \mathtt{Gen}(\sigma; \Gamma) \vdash e_2 : \beta}{\Gamma \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \beta}\ \text{let} \qquad \frac{\Gamma \vdash e_1 : \mathtt{bool} \quad \Gamma \vdash e_2 : \beta \quad \Gamma \vdash e_3 : \beta}{\Gamma \vdash \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 : \beta}\ \text{if}$$

Fig. 5. Type Inference Rules of λ_{let}

$$\begin{aligned} b^* &= b \quad \text{for a basic type } b \\ t^* &= t \quad \text{for a type variable } t \\ ((\alpha/\gamma \to \beta/\delta))^* &= \alpha^* \to (\beta^* \to \gamma^*) \to \delta^* \\ (\forall t.A)^* &= \forall t.A^* \\ (\Gamma, x : A)^* &= \Gamma^*, x : A^* \end{aligned}$$

Fig. 6. CPS translation for types and type contexts.

3.1 Target Calculus λ_{let}

The syntax of values and expressions in λ_{let} are the same as those in $\lambda_{let}^{s/r}$ except that λ_{let} does not have control operators shift and reset. Types of λ_{let} are standard and given by:

$$\begin{aligned} \alpha, \beta &::= t \mid b \mid \alpha \to \beta && \text{monomorphic type} \\ A &::= \alpha \mid \forall t.A && \text{polymorphic type} \end{aligned}$$

Figure 5 defines the type inference rules of λ_{let}. Note that, in the type inference rule for (let), there is no side condition on the expression e_1. Hence, for instance, an expression $\mathtt{let}\ x = yz\ \mathtt{in}\ x$ is not typable in $\lambda_{let}^{s/r}$, but is typable in λ_{let}.

The reduction rules for λ_{let} are the same as those for $\lambda_{let}^{s/r}$ restricted to the expressions in λ_{let}, and are omitted.

3.2 Definitional CPS Translation from $\lambda_{let}^{s/r}$ to λ_{let}

Figures 6 and 7 define the definitional CPS translation for $\lambda_{let}^{s/r}$ where the variables κ, κ', m and n are fresh. The type $(\alpha/\gamma \to \beta/\delta)$ is translated to the type of a function which, given a parameter of type α^* and a continuation of type $\beta^* \to \gamma^*$, returns a value of type δ^*.

In [4], we proved that the CPS translation preserves types and equality.

Theorem 4 (Preservation of Types). *If $\Gamma;\ \alpha \vdash e : \tau;\ \beta$ is derivable in $\lambda_{let}^{s/r}$, then $\Gamma^* \vdash [\![e]\!] : (\tau^* \to \alpha^*) \to \beta^*$ is derivable in λ_{let}.*

If $\Gamma \vdash_p e : \tau$ is derivable in $\lambda_{let}^{s/r}$, then $\Gamma^* \vdash [\![e]\!] : (\tau^* \to \gamma) \to \gamma$ is derivable for an arbitrary type γ in λ_{let}.

$$
\begin{aligned}
c^* &= c \\
x^* &= x \\
(\lambda x.e)^* &= \lambda x.[\![e]\!] \\
[\![v]\!] &= \lambda\kappa.\kappa v^* \\
[\![e_1 e_2]\!] &= \lambda\kappa.[\![e_1]\!](\lambda m.[\![e_2]\!](\lambda n.mn\kappa)) \\
[\![\mathcal{S}k.e]\!] &= \lambda\kappa.\texttt{let } k = \lambda n\kappa'.\kappa'(\kappa n) \texttt{ in } [\![e]\!](\lambda m.m) \\
[\![\langle e \rangle]\!] &= \lambda\kappa.\kappa([\![e]\!](\lambda m.m)) \\
[\![\texttt{let } x = e_1 \texttt{ in } e_2]\!] &= \lambda\kappa.\texttt{let } x = [\![e_1]\!](\lambda m.m) \texttt{ in } [\![e_2]\!]\kappa \\
[\![\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]\!] &= \lambda\kappa.[\![e_1]\!](\lambda m.\texttt{if } m \texttt{ then } [\![e_2]\!]\kappa \texttt{ else } [\![e_3]\!]\kappa)
\end{aligned}
$$

Fig. 7. CPS translation for values and expressions.

Theorem 5 (Preservation of Equality). *If* $\Gamma;\ \alpha \vdash e_1 : \tau;\ \beta$ *is derivable and* $e_1 \rightsquigarrow^* e_2$ *in* $\lambda_{let}^{s/r}$*, then* $[\![e_1]\!] = [\![e_2]\!]$ *in* λ_{let} *where* $=$ *is the least congruence relation containing* $\rightsquigarrow$ *in* λ_{let}*.*

Note that Theorem 5 only guarantees that the equality is preserved through the CPS translation. In fact, we cannot show that $e_1 \rightsquigarrow e_2$ implies $[\![e_1]\!] \rightsquigarrow^* [\![e_2]\!]$.

4 Refined CPS Translation

The definitional CPS translation is useful in the semantic study of shift and reset. However, it does not preserve reductions, and hence cannot be used to prove SN. The failure of preservation of reduction is due to the fact that the CPS translation introduces a lot of administrative redexes through the translation.

To overcome this difficulty, we refine the definitional CPS translation so that it may produce fewer administrative redexes. There are several ways to define such optimized CPS translations since Plotkin proposed Colon Translation [16]. Here we use an extended version of two-level lambda calculus [7] as the target calculus of the translation, and define a refined CPS translation from $\lambda_{let}^{s/r}$ to it.

4.1 Two-Level Version of Polymorphic Lambda Calculus

In this subsection we introduce λ_{let}^{2L}, a two-level version of polymorphic typed lambda calculus (without control operators). In this calculus, function spaces are classified into two - static one and dynamic one. Accordingly, each occurrence of λ and application (explicitly denoted by "@") is annotated by overlines (static) as $\overline{\lambda}$ and $\overline{@}$, or underlines (dynamic) as $\underline{\lambda}$ and $\underline{@}$. In their original article, Danvy and Filinski classified every construct into two, but here we only classify lambda's and applications, and we assume that the other constructs are implicitly classified as dynamic ones.

Figure 8 gives the syntax of λ_{let}^{2L}, which is an annotated variant of λ_{let}.

$$
\begin{array}{rl}
e ::= & c \mid x \mid \overline{\lambda} x.e \mid \underline{\lambda} x.e \mid e_1 \overline{@} e_2 \mid e_1 \underline{@} e_2 \\
& \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \mid \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \qquad \text{expression} \\
\alpha, \beta ::= & t \mid b \mid \alpha \overline{\to} \beta \mid \alpha \underline{\to} \beta \qquad \text{monomorphic type} \\
A ::= & \alpha \mid \forall t.A \qquad \text{polymorphic type}
\end{array}
$$

Fig. 8. Syntax of Two-Level Polymorphic Lambda Calculus.

$$
\frac{\Gamma, x : \alpha \vdash e : \beta}{\Gamma \vdash \overline{\lambda} x.e : \alpha \overline{\to} \beta}\ \text{static fun}, x \in \texttt{FV}(e) \qquad \frac{\Gamma, x : \alpha \vdash e : \beta}{\Gamma \vdash \underline{\lambda} x.e : \alpha \underline{\to} \beta}\ \text{dynamic fun}
$$

$$
\frac{\Gamma \vdash e_1 : \alpha \overline{\to} \beta \quad \Gamma \vdash e_2 : \alpha}{\Gamma \vdash e_1 \overline{@} e_2 : \beta}\ \text{static app} \qquad \frac{\Gamma \vdash e_1 : \alpha \underline{\to} \beta \quad \Gamma \vdash e_2 : \alpha}{\Gamma \vdash e_1 \underline{@} e_2 : \beta}\ \text{dynamic app}
$$

Fig. 9. Type Inference Rules of Two Level Polymorphic Calculus.

Figure 9 gives the type system of λ^{2L}_{let}, where the type inference rules for (var), (const), (let), and (if) are the same as those in λ_{let}, and are omitted.

The crucial difference of the type system of λ^{2L}_{let} from that of λ_{let} (besides the annotations) is the side condition $x \in \texttt{FV}(e)$ in the static function:

$$
\frac{\Gamma, x : \alpha \vdash e : \beta}{\Gamma \vdash \overline{\lambda} x.e : \alpha \overline{\to} \beta}\ \text{static fun}, x \in \texttt{FV}(e)
$$

The condition imposes that the abstracted variable x must occur freely in e.

We put this side condition by the following reason: we will use the static lambda abstraction to constitute an administrative redex (a redex which does not exist in the source expression, and is created by the CPS translation). When we prove that the CPS translation preserves reductions, it is important to guarantee that reducing an administrative redex does not discard any subexpressions, hence we put the side condition.

Note that the static lambda expression is not necessarily linear, namely, x may appear more than once in e, since CPS translating conditional expressions (if-then-else) may duplicate the arguments of continuations. Note also that the side condition is not applied to dynamic lambda abstraction which corresponds to lambda abstraction in the source expression. In other words, the actual continuations in $\lambda^{s/r}_{let}$ may discard their arguments.

The operational semantics of λ^{2L}_{let} is given as regarding the only redex as the static β-redex. Namely, the following single rule constitutes the notion of reduction in λ^{2L}_{let}:

$$
(\overline{\lambda} x.e_1) \overline{@} e_2 \rightsquigarrow e_1[e_2/x]
$$

Note that this is full β-reduction, rather than the call-by-value variant.

For this notion of reduction, we have subject reduction, strong normalization and confluence as follows.

Theorem 6 (Subject Reduction). *If $\Gamma \vdash e : \alpha$ is derivable in λ^{2L}_{let}, and $e \rightsquigarrow e'$ by reducing static β-redexes only, then $\Gamma \vdash e' : \alpha$ is derivable in λ^{2L}_{let}.*

$$
\begin{aligned}
b^* &= b \quad \text{for a basic type } b \\
t^* &= t \quad \text{for a type variable } t \\
(\alpha/\gamma \to \beta/\delta)^* &= \alpha^* \underline{\to} (\beta^* \underline{\to} \gamma^*) \underline{\to} \delta^* \\
(\forall t.A)^* &= \forall t.A^* \\
(\Gamma, x : A)^* &= \Gamma^*, x : A^*
\end{aligned}
$$

Fig. 10. Refined CPS translation for types.

Note that this theorem is not trivial, as we have a side condition in the typing rule for static lambda abstractions.

Proof. Since all static lambda abstractions $\overline{\lambda}x.e$ satisfy the side condition $x \in \mathtt{FV}(e)$, all free variables in $(\overline{\lambda}x.e)\overline{@}e'$ occur in $e[e'/x]$ freely, so the result of static β-reduction also satisfies the side condition, too.

Theorem 7. *Static reduction $\rightsquigarrow$ in λ^{2L}_{let} is confluent and strongly normalizing.*

Proof. Since we can embed λ^{2L}_{let} into the second order lambda calculus (where we only consider static reductions in λ^{2L}_{let}), strong normalization is apparent. We can easily prove that static reduction is Church-Rosser, since dynamic constructs are not reduced through the reduction.

By this theorem, for each expression e in λ^{2L}_{let}, its normal form uniquely exists (the normality is defined with respect to the static β-reduction). The normal form of e is denoted by $\mathtt{NF}(e)$.

4.2 Refined CPS Translation

The refined CPS translation is a syntax-directed translation from $\lambda^{s/r}_{let}$ to λ^{2L}_{let} where we use static constructs ($\overline{\lambda}$ and $\overline{@}$) for administrative redexes, and dynamic constructs ($\underline{\lambda}$, $\underline{@}$ and all other constructs) for source redexes.

Given an expression e in $\lambda^{s/r}_{let}$ and an expression K in λ^{2L}_{let}, we define an expression $[\![e, K]\!]$ in λ^{2L}_{let} as the CPS translation for e with respect to the continuation K. We first define the translation for types in Figure 10.

Figure 11 gives the CPS translation for expressions and values where m, m_1, m_2, n, κ, and κ' are fresh variables. In the definition for shift, we have added a redundant redex $\mathtt{let}\ m_1 = \mathtt{true}\ \mathtt{in} \cdots$ for the purpose of SN proof.

The complete CPS transform of an expression e may be defined by $\mathcal{C}[\![e]\!] \equiv \underline{\lambda}\kappa.[\![e, \overline{\lambda}x.\kappa\underline{@}x]\!]$, though we do not need this definition in the proof of strong normalization.

Theorem 8 (Preservation of Types).

1. *Suppose $\Gamma;\ \alpha \vdash e : \tau;\ \beta$ is derivable in $\lambda^{s/r}_{let}$, $\Delta \vdash K : \tau^* \overline{\Rightarrow} \alpha^*$ is derivable in λ^{2L}_{let}, and Γ^*, Δ is a valid type context in λ^{2L}_{let}. Then $\Gamma^*, \Delta \vdash [\![e, K]\!] : \beta^*$ is derivable in λ^{2L}_{let}.*

$$
\begin{aligned}
x^* &= x \\
c^* &= c \\
(\lambda x.e)^* &= \underline{\lambda} x.\underline{\lambda}\kappa.[\![e, \overline{\lambda} m.\ \kappa \underline{@} m]\!] \\
[\![v, K]\!] &= K \overline{@} v^* \\
[\![e_1\ e_2, K]\!] &= [\![e_1, \overline{\lambda} m_1.[\![e_2, \overline{\lambda} m_2.(m_1 \underline{@} m_2)\underline{@}(\underline{\lambda} n.\ K \overline{@} n)]\!]]\!] \\
[\![\langle e \rangle, K]\!] &= K \overline{@} [\![e, \overline{\lambda} m.m]\!] \\
[\![\mathcal{S}k.e, K]\!] &= \texttt{let}\ m_1 = \texttt{true}\ \texttt{in} \\
&\qquad \texttt{let}\ k = \underline{\lambda} n.\underline{\lambda}\kappa'.\kappa' \underline{@}(K \overline{@} n)\ \texttt{in}\ [\![e, \overline{\lambda} m.m]\!] \\
[\![\texttt{let}\ x = e_1\ \texttt{in}\ e_2, K]\!] &= \texttt{let}\ x = [\![e_1, \overline{\lambda} m.m]\!]\ \texttt{in}\ [\![e_2, K]\!] \\
[\![\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3, K]\!] &= [\![e_1, \overline{\lambda} m.\texttt{if}\ m\ \texttt{then}\ [\![e_2, K]\!]\ \texttt{else}\ [\![e_3, K]\!]]\!]
\end{aligned}
$$

Fig. 11. Refined CPS translation for expressions and values.

2. *Suppose* $\Gamma \vdash_p e : \tau$ *is derivable in* $\lambda_{let}^{s/r}$ *and* $\Delta \vdash K : \tau^* \Rrightarrow \gamma$ *is derivable in* λ_{let}^{2L}*, and* Γ^*, Δ *is a valid type context in* λ_{let}^{2L}*. Then* $\Gamma^*, \Delta \vdash [\![e, K]\!] : \gamma$ *is derivable in* λ_{let}^{2L}*.*

Proof. We can prove this theorem by induction on the derivation of $\Gamma;\ \alpha \vdash e : \tau;\ \beta$ and $\Gamma \vdash_p e : \tau$. Here, we give proofs for a few cases.

(Case $e = e_1 e_2$) We assume that $\Gamma;\ \alpha \vdash e_1 e_2 : \tau;\ \beta$ is derivable in $\lambda_{let}^{s/r}$ and $\Delta \vdash K : \tau^* \Rrightarrow \alpha^*$ is derivable in λ_{let}^{2L}.

By inversion, we have

$$
\begin{aligned}
&\Gamma;\ \gamma \vdash e_1 : (\sigma/\alpha \to \tau/\beta);\ \delta && \text{in } \lambda_{let}^{s/r} \\
&\Gamma;\ \beta \vdash e_2 : \sigma;\ \gamma && \text{in } \lambda_{let}^{s/r}
\end{aligned}
$$

Then by induction hypothesis on e_2, we have:

$$
\Gamma^*, \Delta \vdash [\![e_2,\ \overline{\lambda} m_2.(m_1 \underline{@} m_2)\underline{@}(\underline{\lambda} n.\ K \overline{@} n)]\!] : \gamma^* \qquad \text{in } \lambda_{let}^{2L}
$$

and by induction hypothesis on e_1, we have:

$$
\Gamma^*, \Delta \vdash [\![e_1,\ \overline{\lambda} m_1.[\![e_2,\ \overline{\lambda} m_2.(m_1 \underline{@} m_2)\underline{@}(\underline{\lambda} n.\ K \overline{@} n)]\!]]\!] : \delta^* \qquad \text{in } \lambda_{let}^{2L}
$$

Hence we are done.

(Case $\Gamma;\ \alpha \vdash \mathcal{S}k.e : \tau;\ \beta$) By inversion, we have

$$
\Gamma, k : \forall t.(\tau/t \to \alpha/t);\ \sigma \vdash e : \sigma;\ \beta \qquad \text{in } \lambda_{let}^{s/r}
$$

By induction hypothesis on e, we have:

$$
\Gamma^*, k : \forall t.(\tau^* \underline{\to} (\alpha^* \underline{\to} t) \underline{\to} t), \Delta \vdash [\![e, \overline{\lambda} m.m]\!] : \beta^* \qquad \text{in } \lambda_{let}^{2L}
$$

and then it is easy to derive:

$$
\Gamma^*, \Delta \vdash \texttt{let}\ m_1 = \texttt{true}\ \texttt{in}\ \texttt{let}\ k = \underline{\lambda} n.\underline{\lambda}\kappa'.\kappa' \underline{@}(K \overline{@} n)\ \texttt{in}\ [\![e, \overline{\lambda} m.m]\!] : \beta^* \qquad \text{in } \lambda_{let}^{2L}
$$

hence we are done.

4.3 Summary of this Section

We can summarize the results in this section as the properties on the following translations:

$$\lambda_{let}^{s/r} \Longrightarrow \lambda_{let}^{2L} \Longrightarrow \lambda_{let}$$

In the first step, the refined CPS translation maps an expression in $\lambda_{let}^{s/r}$ to an expression in λ_{let}^{2L}. Theorem 8 guarantees that this step preserves the type.

In the second step, an expression in λ_{let}^{2L} is normalized to its unique normal form, which can be viewed as an expression in λ_{let} by removing all the overlines and underlines.[7] Theorem 6 guarantees that this step preserves the type.

We know that the calculus λ_{let} is strongly normalizing, since it can be embedded in, for instance, the second order lambda calculus [9]. Hence, in order to prove the strong normalizability of $\lambda_{let}^{s/r}$, it only remains to show that the composed translation from $\lambda_{let}^{s/r}$ to λ_{let} preserves reductions, which will be proved in the next section.

5 Strong Normalization

In this section, we prove that reductions in $\lambda_{let}^{s/r}$ are preserved by the composed translation of the refined CPS translation and the static reduction in λ_{let}^{2L}.

Theorem 9 (Preservation of Reduction). *Suppose $\Gamma;\ \alpha \vdash e_1 : \tau;\ \beta$ is derivable in $\lambda_{let}^{s/r}$, and $\Delta \vdash K : \tau^* \Rightarrow \alpha^*$ is derivable in λ_{let}^{2L}. Then we have:*

1. *If $e_1 \rightsquigarrow e_2$ by a reduction rule other than the reset-value reduction ($\langle v \rangle \rightsquigarrow v$), then $\mathit{NF}([\![e_1, K]\!]) \rightsquigarrow^+ \mathit{NF}([\![e_2, K]\!])$ in λ_{let}.*
2. *If $e_1 \rightsquigarrow e_2$ by the reset-value reduction ($\langle v \rangle \rightsquigarrow v$), then $\mathit{NF}([\![e_1, K]\!]) \equiv \mathit{NF}([\![e_2, K]\!])$ in λ_{let}.*

In the theorem above, we regard expressions in λ_{let}^{2L} as those in λ_{let} by erasing all overlines and underlines.

Proof. The first part of this theorem is proved by the case analysis of reduction rules used in $e_1 \rightsquigarrow e_2$.

- If the reduction is the call-by-value β reduction (the first reduction in Figure 2), or reductions for let, or conditional, then the theorem can be proved easily.
- For the reduction $\langle F[\mathcal{S}k.e] \rangle \rightsquigarrow \langle \mathtt{let}\ k = \lambda x.\langle F[x] \rangle\ \mathtt{in}\ e \rangle$, we first prove that $\mathtt{NF}([\![F[e], K]\!]) \equiv \mathtt{NF}([\![e, \overline{\lambda} m.[\![F[m], K]\!]]\!])$. This property can be easily proved by induction on F. Note that this property holds for typable expressions only.
  Then we can prove:

$$\begin{aligned}
&\mathtt{NF}([\![\langle F[\mathcal{S}k.e] \rangle, K]\!]) \\
&\equiv \mathtt{NF}(K\overline{@}[\![F[\mathcal{S}k.e], \overline{\lambda} m.m]\!]) \\
&\equiv \mathtt{NF}(K\overline{@}[\![\mathcal{S}k.e, \overline{\lambda} m'.[\![F[m'], \overline{\lambda} m.m]\!]]\!]) \\
&\equiv \mathtt{NF}(K\overline{@}(\mathtt{let}\ m_1 = \mathtt{true}\ \mathtt{in}\ \mathtt{let}\ k = \underline{\lambda} n.\underline{\lambda}\kappa'.\kappa'\underline{@}((\overline{\lambda} m'.[\![F[m'], \overline{\lambda} m.m]\!])\overline{@} n)\ \mathtt{in}\ [\![e, \overline{\lambda} m.m]\!])) \\
&\equiv \mathtt{NF}(K\overline{@}(\mathtt{let}\ m_1 = \mathtt{true}\ \mathtt{in}\ \mathtt{let}\ k = \underline{\lambda} n.\underline{\lambda}\kappa'.\kappa'\underline{@}[\![F[n], \overline{\lambda} m.m]\!]\ \mathtt{in}\ [\![e, \overline{\lambda} m.m]\!])) \\
&\rightsquigarrow^+ \mathtt{NF}(K\overline{@}(\mathtt{let}\ k = \underline{\lambda} n.\underline{\lambda}\kappa'.\kappa'\underline{@}[\![F[n], \overline{\lambda} m.m]\!]\ \mathtt{in}\ [\![e, \overline{\lambda} m.m]\!]))
\end{aligned}$$

[7] Note that static constructs may remain in the normal forms.

In the last step above, since K does not discard its argument by the side condition of the static lambda expression, the reduction $\texttt{let}\ m_1 = \texttt{true}\ \texttt{in}\ e_1 \rightsquigarrow e_1$ is preserved, and at least one step reduction occurs during this sequence. (Recall that we have added a dummy redex in the refined CPS translation of the shift expression.) We also have:

$$\begin{aligned}&\mathtt{NF}([\![\langle \texttt{let}\ k = \lambda x.\langle F[x]\rangle\ \texttt{in}\ e\rangle, K]\!])\\ &\equiv \mathtt{NF}(K\overline{@}\texttt{let}\ k = \underline{\lambda} x.\underline{\lambda}\kappa.\kappa\underline{@}[\![F[x], \overline{\lambda} m.m]\!]\ \texttt{in}\ [\![e, \overline{\lambda} m.m]\!])\end{aligned}$$

and therefore the resulting expressions are the same up to α-equivalence, hence we have:

$$\mathtt{NF}([\![\langle F[\mathcal{S}k.e]\rangle, K]\!]) \rightsquigarrow^{+} \mathtt{NF}([\![\langle \texttt{let}\ k = \lambda x.\langle F[x]\rangle\ \texttt{in}\ e\rangle, K]\!])$$

The second part of this theorem is proved by a simple calculation, hence we are done.

We now give the strong normalization property for $\lambda_{let}^{s/r}$ as a theorem.

Theorem 10 (Strong Normalization). *If $\Gamma;\ \alpha \vdash e : \tau;\ \beta$ is derivable in $\lambda_{let}^{s/r}$, then there is no infinite reduction sequence starting from e.*

Proof. Suppose there is an infinite reduction sequence $e_1 \rightsquigarrow e_2 \rightsquigarrow \cdots$ in $\lambda_{let}^{s/r}$. Since the reset-value reduction ($\langle v\rangle \rightsquigarrow v$) cannot be applied to an expression infinitely many times, the reduction sequence must contain infinitely many reductions which are not the reset-value reduction. Then by Theorem 9, we have an infinite sequence $\mathtt{NF}(\mathcal{C}[\![e_1]\!]) \rightsquigarrow^{+} \mathtt{NF}(\mathcal{C}[\![e_2]\!]) \rightsquigarrow^{+} \cdots$. But, since the target calculus λ_{let} is a strongly normalizing calculus, we get contradiction.

Hence, $\lambda_{let}^{s/r}$ does not have an infinite reduction sequence.

As a corollary of strong normalization, we obtain confluence of $\lambda_{let}^{s/r}$, though it can be proved directly.

Theorem 11 (Confluence). *The notion of reduction in $\lambda_{let}^{s/r}$ is confluent.*

Proof. Since the reductions in $\lambda_{let}^{s/r}$ are not overlapping, they are weakly Church-Rosser (WCR). Church-Rosser property is subsumed by WCR and SN.

6 Conclusion

In this paper, we have presented a proof of strong normalization of the polymorphic calculus for shift and reset introduced by our previous work. The calculus allows let-polymorphism with a less restricted condition than the value restriction in ML families.

Let us emphasize that our proof is simple and easy to understand compared with the SN proofs for the calculi with call/cc and $\lambda\mu$, for which one needs more involved proof. The simplicity of our proof partly comes from the modularity of the proof, but mainly from the design of the control operators shift and reset and the naturality of the type system [4].

For future work, we plan to extend this result to the calculi with impredicative polymorphism given in [4]. Finding a better perspective of strong normalizability of calculi with various control operators is also left for future work.

Acknowledgments. We would like to thank Koji Nakazawa, Oleg Kiselyov and Chung-chieh Shan for helpful comments. The anonymous referees provided insightful comments.

The first author was partly supported by JSPS and FWF under the Japan-Austria Research Cooperative Program and by JSPS Grant-in-Aid for Scientific Research 20650003. The second author was partly supported by JSPS Grant-in-Aid for Scientific Research (C) 18500005.

References

1. Z. M. Ariola, H. Herbelin, and A. Sabry. A type-theoretic foundation of continuations and prompts. In *ICFP*, pages 40–53, 2004.
2. K. Asai. Logical Relations for Call-by-value Delimited Continuations. In *Trends in Functional Programming*, volume 6, pages 63–78, 2007.
3. K. Asai. On Typing Delimited Continuations: Three New Solutions to the Printf Problem. Technical Report OCHA-IS 07-1, Dept. of Information Science, Ochanomizu University, September 2007. 9 pages.
4. K. Asai and Y. Kameyama. Polymorphic Delimited Continuations. In *APLAS*, pages 239–254, 2007.
5. O. Danvy and A. Filinski. A Functional Abstraction of Typed Contexts. Technical Report 89/12, DIKU, University of Copenhagen, July 1989.
6. O. Danvy and A. Filinski. Abstracting Control. In *Proc. 1990 ACM Conference on Lisp and Functional Programming*, pages 151–160, 1990.
7. O. Danvy and A. Filinski. Representing Control: a Study of the CPS Transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.
8. A. Filinski. Representing Monads. In *POPL*, pages 446–457, 1994.
9. J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.
10. T. Griffin. A Formulae-as-Types Notion of Control. In *POPL*, pages 47–58, 1990.
11. R. Harper and M. Lillibridge. Explicit Polymorphism and CPS Conversion. In *POPL*, pages 206–219, 1993.
12. Y. Kameyama and M. Hasegawa. A Sound and Complete Axiomatization for Delimited Continuations. In *ICFP*, pages 177–188, 2003.
13. Y. Kameyama and T. Yonezawa. Typed Dynamic Control Operators for Delimited Continuations. In *FLOPS, LNCS 4989*, pages 239–254, 2008.
14. O. Kiselyov. Simply Typed Lambda-Calculus with a Typed-Prompt Delimited Control is not Strongly Normalizing. 2006. `http://okmij.org/ftp/Computation/Continuations.html`.
15. O. Kiselyov. Demo of Persistent Delimited Continuations in OCaml for Nested Web Transactions. In *Talk presented at Continuation Fest2008, Tokyo, Japan*, 2008.
16. G. D. Plotkin. Call-by-Name, Call-by-Value, and the λ-Calculus. *Theoretical Computer Science*, 1:125–159, 1975.
17. P. Thiemann. Combinators for Program Generation. *J. Funct. Program.*, 9(5):483–525, 1999.
18. A. K. Wright and M. Felleisen. A Syntactic Approach to Type Soundness. *Inf. Comput.*, 115(1):38–94, 1994.

Experiences with Web Environment Origamium: Examples and Applications

Asem Kasem and Tetsuo Ida

Department of Computer Science, University of Succuba,
Succuba 305-8573, Japan
{kasem, ida}@score.cs.tsukuba.ac.jp

Abstract. We present a web environment for origami theorem proving and show the experimental results for using its functionalities. The environment consists of a mathematical engine for computational origami, a graphical interface for origami construction and visualization through the web, and a system of web services for symbolic computation, which allows web users to access computer algebra software systems. We demonstrate our experiences with this environment through a set of examples, where we use computational origami to construct objects representing geometrical theorems, and then access via web to computer algebra systems to prove those theorems automatically.

Keywords. Computational origami, symbolic web services, geometrical theorem proving, symbolic computation, Gröbner bases.

1 Introduction

Origami, the traditional Japanese art of paper folding, is a powerful tool for constructive geometry. Despite its simplicity, it can be used to solve several challenging problems from classical geometry, such as trisecting an arbitrary angle or doubling the volume of a given cube [18].

The growing interest in manipulating origami with computers led to the discipline termed as computational origami. It is concerned with simulating origami constructions on computers, and moreover, with the geometrical and mathematical aspects of origami constructions [6].

As a part of our research in computational origami at SCORE[1] laboratory, we have developed a software called Eos (E-origami system) [11, 14]. It has capabilities of symbolic and numeric constraint solving, visualization of origami constructions, and assists the user in proving geometric theorems about origami by decision methods from computer algebra.

We built WebEos system to provide interested origamists with a web interface to the features of Eos. Through graphical interactive interfaces, web users can easily construct origami pieces using their web browsers. To enable those

[1] The Symbolic Computation Research Group at University of Tsukuba

users to reason about their constructed origami, we also built a system for symbolic computation web services, called SCORUM. It allows access to mathematical software for performing symbolic computations related to origami theorem proving. These web systems, along with EOS, are integrated to form a web environment for computational origami. Details about this environment have already been published [16].

In this paper, we focus on showing some experimental results through a set of examples. We present our trials to prove geometrical theorems, and the results of some intensive symbolic computations. We also compare the computation time among symbolic computation requests run on different mathematical systems, with different parameters and algorithms. The paper serves to show the merits of using the environment for origami theorem proving, and how it helps to organize and properly utilize our available resources.

2 Overview of the Environment

In this section, we will give a brief overview of the environment which we think is important to help the reader to understand the context of the research.

The web environment for computational origami is called ORIGAMIUM [16]. The diagram in Fig. 1 shows its architecture and the interaction between its subsystems. The environment integrates three main systems.

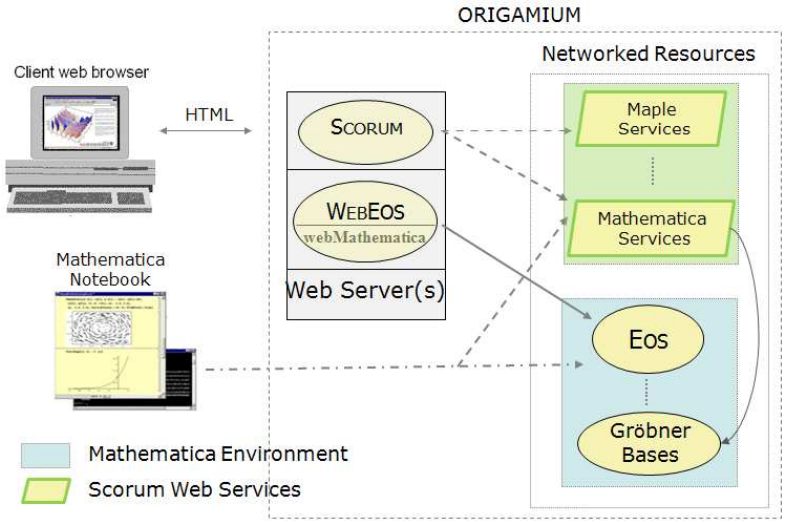


Fig. 1. Overview of ORIGAMIUM Architecture

First is the mathematical engine for origami manipulation, EOS. It is developed in *Mathematica* [21] and provides the implementation of a set of functions

to create origami and manipulate it by folding, unfolding, creating points, cutting, etc. Accessing Eos is done by executing text-based commands through *Mathematica*'s frontend.

Second is WebEos, the web interface of Eos for interactive origami construction. WebEos is based on Ajax technology for web applications development. It allows asynchronous access to various functions of Eos, and facilitates dynamic origami constructing using a graphical user interface running on a web browser. It aims to reach a wide number of users interested in origami, and helps to demonstrate interesting examples and constructions, and share them publicly on the web. Further details about WebEos system can be found in [17, 16].

A snapshot for one of the web pages of WebEos is shown in Fig. 2. It shows an intermediate step of constructing a regular triangle inscribed in origami.

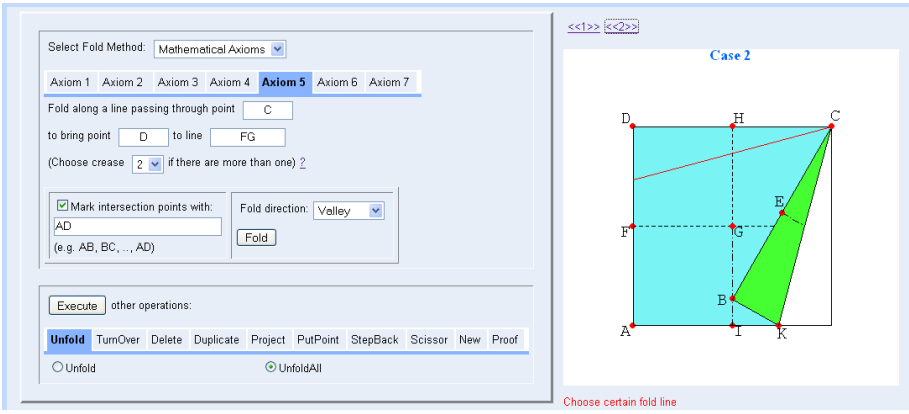


Fig. 2. Execution of origami fold operation using WebEos. Two fold lines satisfying the fold conditions are possible, and they can be viewed by clicking the hyperlinks above the origami image

Third is the Symbolic Computation Research Forum, Scorum, a system that exposes web services for symbolic computation. It is designed to facilitate standard web access to the functionalities of various mathematical systems, such as *Mathematica*, *Maple* [19], *CoCoa* [4], etc. Scorum also provides user-friendly interfaces to access these web services, and to input and visualize mathematical expressions using web pages.

In Scorum, we distinguish two kinds of symbolic computations, i.e. interactive and time-consuming (non-interactive) computations. Interactive computations are expected to finish within allowed time slice. Thus, their result is delivered immediately to the caller. Time-consuming computations are those which are expected to take long time before they finish, and thus they are not suited for web services interaction due to the timeout problems.

For non-interactive services, the user can check the status and the result of the submitted requests. Besides, (s)he can view the list of completed requests and sort it according to the computation time, or other criteria. This helps to compare and check the efficiency of computations related to each other, such as submitting the same computation request for *Mathematica* and *Maple* systems at the same time. Further details about SCORUM can be found in [20, 16][2].

2.1 Origami Theorem Proving

Each origami construction has geometric properties that can represent a geometrical problem or theorem, such as: the theorem of Morley's triangle, the maximum equilateral triangle inscribed in a square, creating a regular heptagon, etc.

These theorems or problems can be illustrated by computational origami. To prove their correctness, we can use methods from computer algebra to reason about their geometric properties. We usually follow the following approach:

1. Construct the geometrical object by origami folding.
2. Transform the geometrical properties of the construction into polynomials, which we call the premises P.
3. A conclusion C is then formulated as polynomial equations.
4. The final formula that we try to prove becomes $P \Rightarrow C$.
5. To prove this implication, we use techniques of automated theorem proving over polynomials, namely Gröbner bases method. Checking whether the implication holds or not leads to proving the correctness of the construction\theorem.

There are various implementations for these methods of computer algebra. The design of our environment enables users to access this variety of symbolic computation implementations through SCORUM.

3 Experimental Results

We will present several examples to show how the web environment can organize our research activities on origami, and facilitate the interaction with symbolic computation software. We use WEBEOS to make the origami constructions of our examples. Then, we generate the polynomial equations that represent these constructions, and formulate the desired conclusions.

Automated proofs for the theorems are performed by computing Gröbner bases for the polynomials, as explained in the previous section. This requires accessing computer algebra systems to perform this time-and-memory consuming computation. Almost always, we need to make several trial runs of such computations to investigate a given problem. The investigation involves experimenting with several trials where we may change one or all of the following:

[2] Please note that services of WEBEOS and SCORUM systems are not yet made public on the web. The reasons are discussed in the referred papers

- the construction method to build the geometric object, which will affect the generated polynomials.
- the conclusion formulation, which may reduce computation time.
- the assigned coordinates mappings of the geometric object. Careful selection can simplify the generated polynomials and lead to faster computation.
- the parameters for Gröbner bases computation, such as monomial orders, computing method, and the order of variables, which are known to substantially affect computing time.
- the used mathematical system, which can expose different potential parameters and methods.

It is clear that trying multiple combinations of these possible variations consumes a lot of time and effort. The usage of Scorum enables us to reduce this effort and focus on the creative part of solving the problem, while using the system to skip most of the routine and manual interventions.

Scorum can facilitate and organize submitting requests for computations of Gröbner bases to different mathematical systems. The computations are executed on dedicated servers, which have powerful hardware specifications. This allows a better utilization of our networked resources.

We use MathML as the standard representation of mathematical data in Scorum. This helps the user to make uniform access to multiple mathematical systems which may use different data representations.

In Fig. 3, we view the web page that allows users to submit requests for Gröbner bases computation. The page contains information about Gröbner bases method and allows the user to input the necessary parameters. Inputting mathematical expressions is made possible using MathML text, or using graphical pallets. The pallets are provided using the Java applets of *WebEQ* [15] product.

Figure 4 shows a listing of all requests for Gröbner bases computations submitted by an authenticated user. Through this page, the user can access the requests to check their results, compare or delete them. As indicated by the status icons in the figure, some of the requests have been computed and their results are shown as MathML text. We can also see that one computation has started and wasn't finished at the time of accesssing the page. The rest of computations are pending requests waiting for the execution to start.

By clicking on the "Select" icon, all the data of the selected request will be shown, with visual representation of its mathematical expressions. By clicking on the headers of the list, the user can sort the requests in many ways, such as according to their consumed computation time. This offers a possibility to easily compare certain computations with each other.

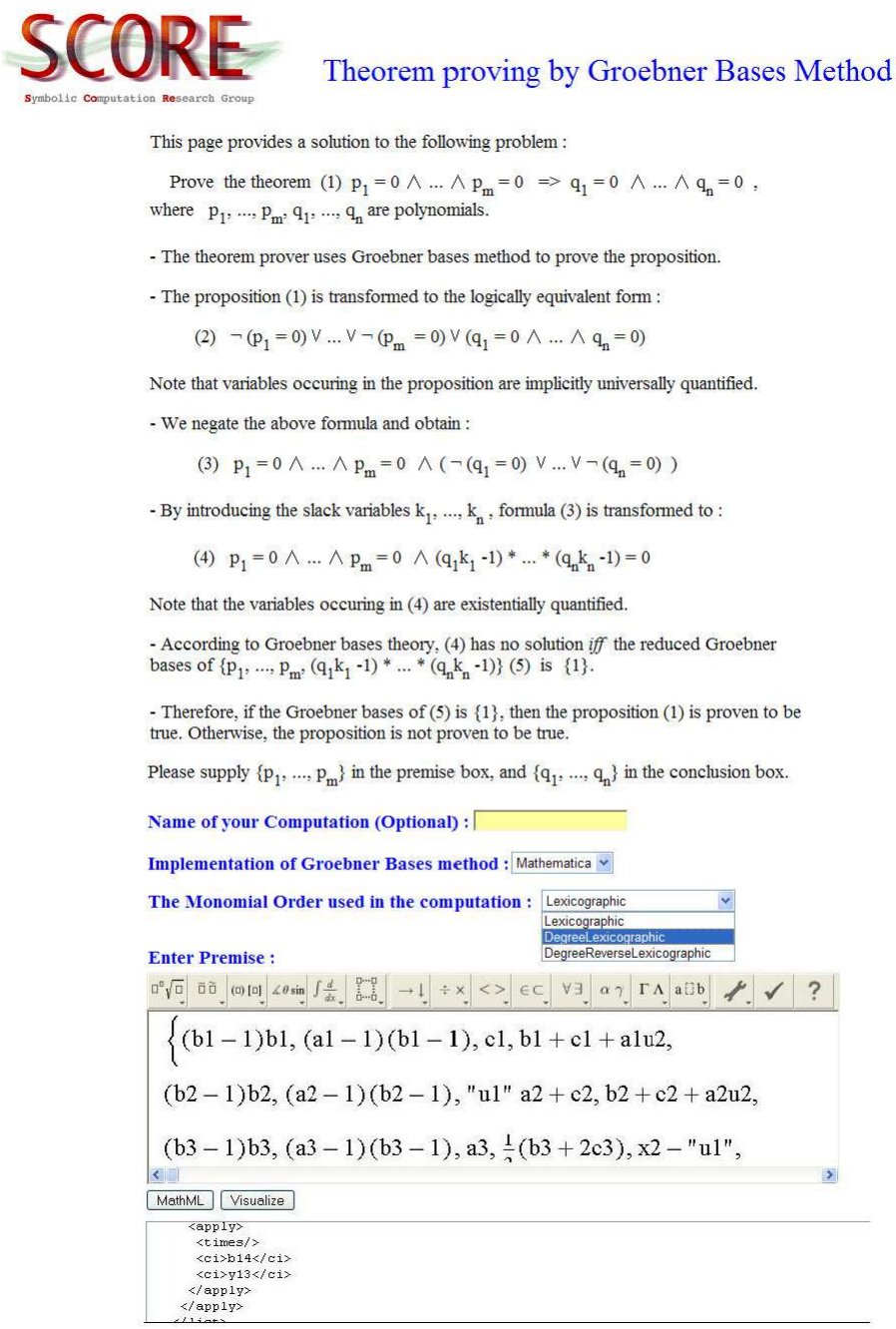


Fig. 3. SCORUM web page for submitting a Gröbner bases computation

Fig. 4. Web page of SCORUM showing the submitted requests for Gröbner bases computation

In the following parts of the paper, we show the results of our investigations and experiments to use ORIGAMIUM to prove Morley's theorem, and to make angle quintisecting (division of an angle into 5 equal angles).

3.1 Morley's Triangle Theorem

The theorem is also known as Morley's trisector theorem. It states that in any triangle in plane geometry, the three points of intersection of the adjacent angle trisectors form an equilateral triangle [2]. In previous publications [13, 12], we have shown an automated proof of this theorem by computational origami using EOS system. However, in this paper, we focus on the usage of the web environment, and comparing the results obtained by several trials to prove it. Here, we use two methods to construct Morley's triangle.

The first is based on Abe's method for trisecting an angle [9, 1]. It depends on the basic folding axioms, known as Huzita's origami axioms [10]. In Fig. 5, we show a snapshot of the triangle being constructed on the web using WEBEOS.

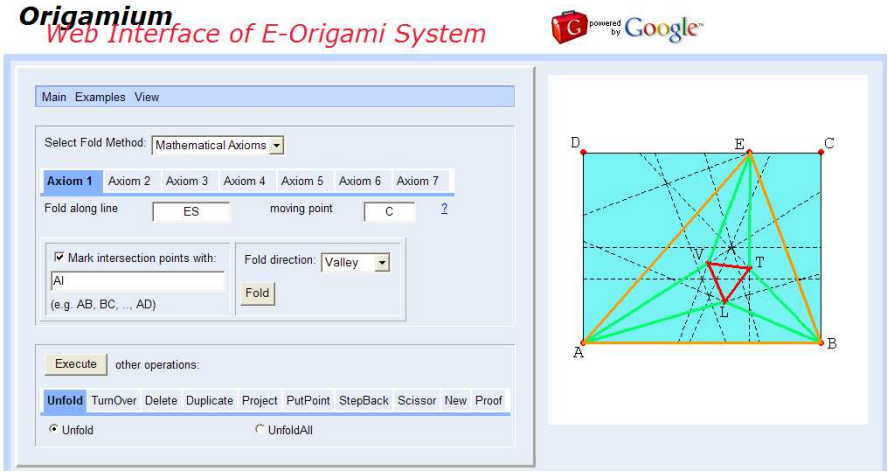


Fig. 5. Construction of Morley's triangle using WEBEOS

The second method is based on general folding method that finds multiple fold lines satisfying a given constraint. EOS system provides implementation of this fold operation, and we use it in order to simplify construction steps for trisecting the angles of the triangle. This also leads to reduced polynomial form, compared with the previous method.

It is important to mention that the usage of general folding method is limited to the implementation of EOS system in *Mathematica 6*. Interfacing this method to WEBEOS is not feasible currently, because *webMathematica 3*, which allows access to *Mathematica 6* from the web, is still a Beta-version release. In

WEBEOS, we use *webMathematica 2* [22] which is compatible with *Mathematica 5.2* only.

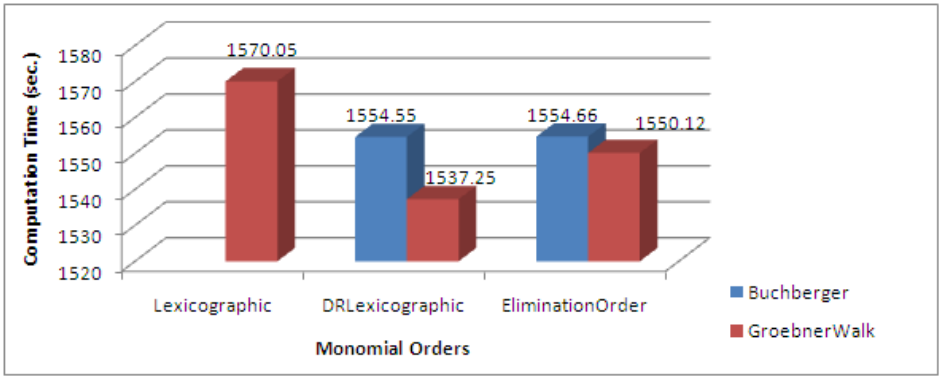
Using the web page of Gröbner bases in SCORUM, we submit several computation requests with different parameters, and then wait for their computation results.

In the case of Abe's method, the number of generated polynomials is 65, with 51 variables. Computation timings (always in seconds) are shown in Table 1[3].

Table 1. Proof of Morley's triangle theorem, using Abe's method

Available algorithms: Buchberger [3] and GroebnerWalk [5] methods
Monomial orders: Lexicographic, DRLexicographic, and Elimination Order
Hardware spec.: *Mathematica 6* on Intel Core 2 Duo 2.67 GHz, 2.00 GB of RAM

	Lexicographic	**Degree Reverse Lexicographic**	**Elimination Order**
Buchberger		1554.55	1554.66
GroebnerWalk	1570.05	1537.25	1550.12



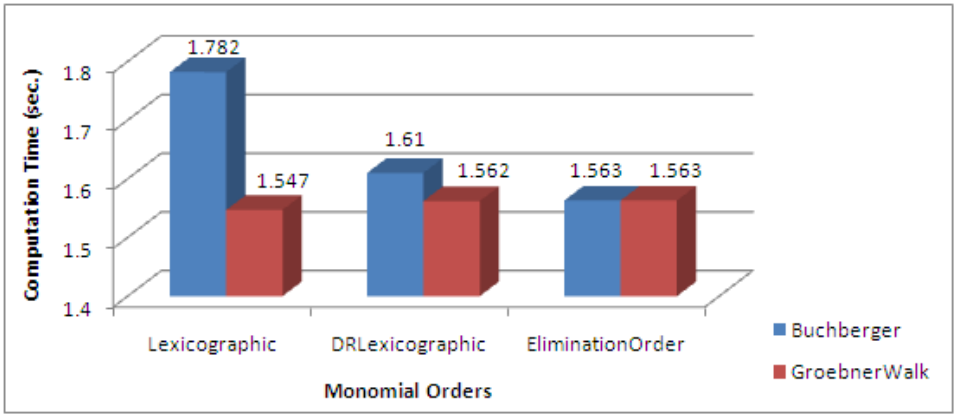
While in the case of using the general fold method, the number of generated polynomials is 53, with 41 variables. Computations times are shown in Table 2, where we use the same hardware specifications as before. The results show a significant improvement to prove the theorem using the general fold method.

With the same hardware specifications, we have also performed trials to compute Gröbner bases using *Mathematica 5.2*. The results are shown in Table 3.

[3] Empty cells in the table refer to the failure of the computation. This is due to the limitation of memory space, or to interruption of computation after taking long time without any result

Table 2. Proof of Morley's triangle theorem, using general fold method

	Lexicographic	**Degree Reverse Lexicographic**	**Elimination Order**
Buchberger	1.782	1.61	1.563
GroebnerWalk	1.547	1.562	1.563



We also used *Maple 11* to compute Gröbner bases with the same hardware specifications. *Maple 11* provides a wider set of algorithms and monomial orders, compared to *Mathematica*, which allows us to try new possibilities of computations as shown in Table 4.

3.2 Angle Quintisection

The other example in this paper is angle quintisection. Using the general folding method to find multiple fold lines, we are able to divide an arbitrary angle into 5 equal angles, as shown in Fig. 6.

The automated proof of the construction requires 4 sub-computations of Gröbner bases. The numbers of polynomials in each is 42,46,42, and 42. The corresponding numbers of variables are 36,38,36, and 36. The times for the 4 computations are summed up and presented in Table 5.

4 Discussion and Conclusion

We have presented our web environment for origami theorem proving through several examples. We conducted experiments that illustrate its usage, and allow us to make interesting comparisons.

Table 3. Morley's triangle with general fold method, using *Mathematica 5.2*

Available algorithms: Buchberger method
Monomial orders: Lexicographic, DRLexicographic, and DLexicographic

	Lexicographic	**Degree Reverse Lexicographic**	**Degree Lexicographic**
Buchberger	8910.03	3.25	19.219

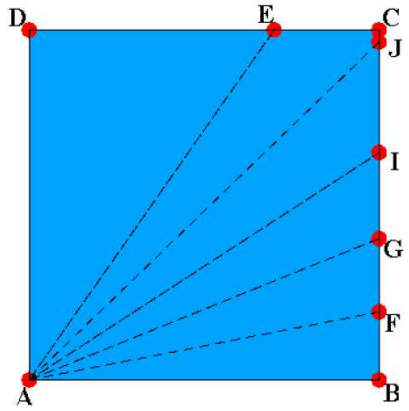


Fig. 6. Angle quintisection using general fold method of Eos

During the different trials to compute Gröbner bases, we encountered some problems that terminated these computations, and caused the absence of results in the tables of Section 3.

In some cases, the reason was insufficient memory. The usage of more powerful hardware could have solved the problem. However, we preferred to use the same hardware specification for all computations, in order to make fair and reasonable comparisons. In other cases, the computations continued for several hours (6, 12, or 18 hours) without obtaining any result, so we manually interrupted them. In the future, we may allow longer time trials to check these computations.

Through out our experiments, we noticed good improvements in the efficiency of computing Gröbner bases in *Mathematica 6* compared to *Mathematica 5.2*. This can be shown from Table 2 and Table 3.

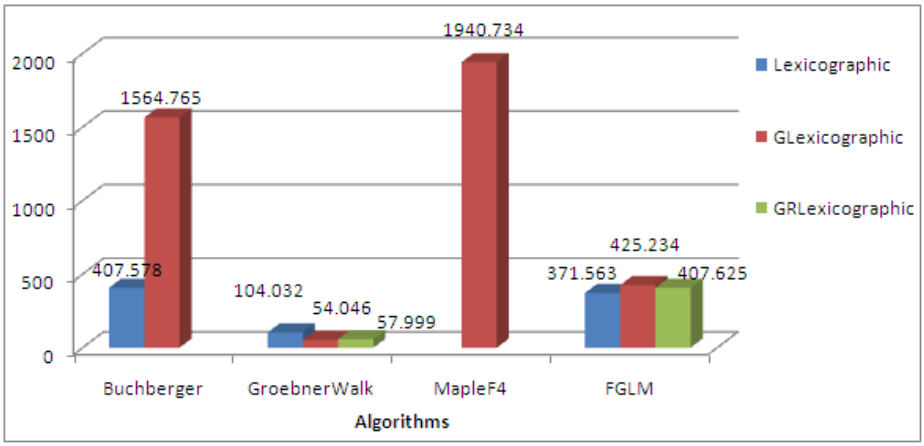
By comparing the values in Table 2 with Table 4, we can see that *Mathematica 6* was faster than *Maple 11* in computing Gröbner bases for the same input.

It is important to add that, in the paper, we showed only few of the variations of computations that we tried in order to prove the theorems. However, we

Table 4. Morley's triangle with general fold method, using *Maple 11*

Available algorithms: Buchberger, GroebnerWalk, MapleF4 [7], and FGLM [8]
Monomial orders: Lexicographic, GradedLexicographic, and GradedReverseLexicographic

	Lexicographic	**GLexicographic**	**GRLexicographic**
Buchberger	407.578	1564.765	
GroebnerWalk	104.032	54.046	57.999
MapleF4		1940.734	
FGLM	371.563	425.234	407.625



actually made many other trials which included changes to the order of variables in Gröbner bases, usage of different coordinates mappings to generate simplified polynomials, and different formalizations of the desired conclusion. We think that these changes are not of great importance to be presented in the paper, however, awareness should be given to this fact to show how the environment assists us through all of these trials.

Acknowledgements

This research is supported by the JSPS Grants-in-Aid for Scientific Research (B) No. 20300001 and by Japan-Austria Research Cooperative Program of JSPS and FWF.

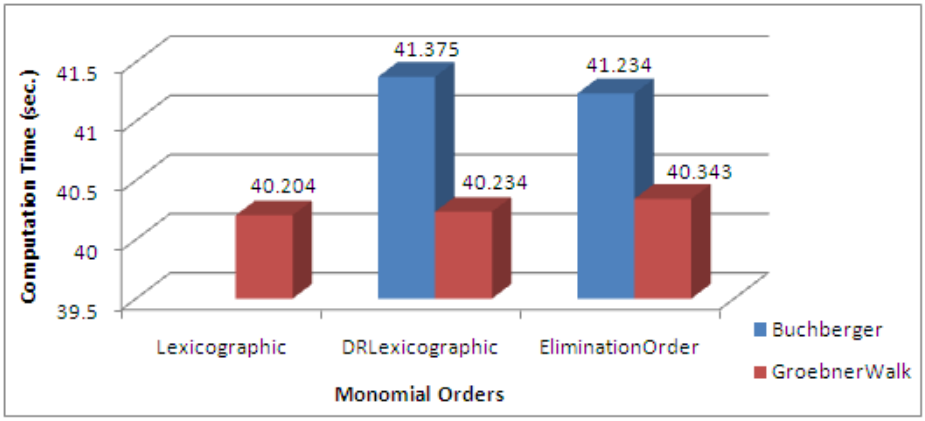
References

1. Tsune Abe. described in British Origami, no. 108, p. 9, 1984.

Table 5. Proof of angle quintisection using *Mathematica 6*

Hardware spec.: *Mathematica 6* on Intel Core 2 Duo 2.67 GHz, 2.00 GB of RAM

	Lexicographic	**Degree Reverse Lexicographic**	**Elimination Order**
Buchberger		41.375	41.234
GroebnerWalk	40.204	40.234	40.343



2. A. Bogomolny. Morley's Miracle. http://www.cut-the-knot.org/triangle/Morley, 1996. ©1996-2005.
3. B. Buchberger. Groebner-Bases: An Algorithmic Method in Polynomial Ideal Theory. In N.K. Bose, editor, *Multidimensional Systems Theory - Progress, Directions and Open Problems in Multidimensional Systems*, chapter 6, pages 184–232. Copyright: Reidel Publishing Company, Dordrecht - Boston - Lancaster, The Netherlands, 1985. (Second editioin: N.K.Bose (ed.): Multidimensional Systems Theory and Application, Kluwer Academic Publisher, 2003, pp.89-128.).
4. CoCoATeam. CoCoA: a system for doing Computations in Commutative Algebra. Available at `http://cocoa.dima.unige.it`.
5. S. Collart, M. Kalkbrener, and D. Mall. Converting bases with the gröbner walk. *J. Symb. Comput.*, 24(3-4):465–469, 1997.
6. E. D. Demaine and M. L. Demaine. Recent results in computational origami. In Thomas Hull, editor, *Origami^3: Third International Meeting of Origami Science, Mathematics and Education*, pages 3–16, Natick, Massachusetts, 2002. A K Peters, Ltd.
7. J. Faugère. A new efficient algorithm for computing gröbner basis (f 4). *Journal of Pure and Applied Algebra*, 139(1-3):61–88, 6 1999.
8. J. C. Faugère, P. Gianni, D. Lazard, and T. Mora. Efficient computation of zero-dimensional gröbner bases by change of ordering. *J. Symb. Comput.*, 16(4):329–344, 1993.
9. Koji Fusimi. Trisection of angle by abe. *Saiensu supplement*, page 8, October,1980.

10. H. Huzita. Axiomatic Development of Origami Geometry. In H. Huzita ed., editor, *Proceedings of the First International Meeting of Origami Science and Technology*, pages 143–158, 1989.
11. T. Ida, D. Ţepeneu, B. Buchberger, and J. Robu. Proving and Constraint Solving in Computational Origami. In *Proceedings of the 7th International Symposium on Artificial Intelligence and Symbolic Computation (AISC 2004)*, volume 3249 of *LNAI*, pages 132–142, 2004.
12. T. Ida, M. Marin, and H. Takahashi. Computational origami of a morley's triangle. In Michael Kohlhase, editor, *Proceedings of 4th International Conference on Mathematical Knowledge Management (MKM 2005), LNCS 3863*, pages 267–282. Springer, July 15-16 2005. Bremen, Germany.
13. T. Ida, H. Takahashi, D. Ţepeneu, and M. Marin. Morley's Theorem Revisited through Computational Origami. In *Proceedings of the 7th International Mathematica Symposium (IMS 2005)*, 2005.
14. T. Ida, H. Takahashi, M. Marin, A. Kasem, and F. Ghourabi. Computational Origami System Eos. In *Proceedings of 4th International Conference on Origami, Science, Mathematics and Education*, page 69. 4OSME, 2006. Caltech, Pasadena CA.
15. Design Science Inc. WebEQ developers suite. http://www.dessci.com/en/products/webeq.
16. A. Kasem and T. Ida. Computational origami environment on the web. *Frontiers of Computer Science in China*, 2(1), 2008. To be published.
17. A. Kasem, H. Takahashi, M. Marin, and T. Ida. weborigami2 : A system for origami construction and proving using web 2.0 technologies. In *Proceedings of the Annual Symposium of Japan Society for Software Science and Technology*, Nara, Japan, 2007. JSSST.
18. R. J. Lang. Origami and geometric constructions. http://www.langorigami.com/science/hha/origami_constructions.pdf. Copyright: 1996-2003.
19. Maplesoft. Maple. http://www.maplesoft.com.
20. M. Naifer, A. Kasem, and T. Ida. A system of web services for symbolic computation. In Tetsuo Ida, Qingshan Jiangand, and Dongming Wang, editors, *Proceedings of the 5th Asian Workshop on Foundations of Software*, pages 145–152. Beihang University, 2007. Xiamen, China.
21. Wolfram Research. Mathematica. http://www.wolfram.com/products/mathematica.
22. Wolfram Research. webMathematica 2. http://www.wolfram.com/products/webmathematica.

Invariant Generation with `Aligator`

Laura Kovács *

EPFL, Switzerland
laura.kovacs@epfl.ch

Abstract. We present an automatic method for generating polynomial invariants of a subfamily of imperative loops operating on numbers, called the P-solvable loops. The approach combines algorithmic combinatorics and polynomial algebra with computational logic, and it derives a set of polynomial invariants from which, under additional assumptions, any polynomial invariant can be derived. The technique is implemented in a new software package `Aligator` written in *Mathematica*, and successfully tried on many programs implementing interesting algorithms working on numbers.

1 Introduction

To verify and/or analyze programs containing loops one needs to discover some properties of loops automatically. Such properties are known as loop invariants. Powerful techniques for finding loop invariants are thus crucial for further progress of software verification and program analysis.

By combining symbolic computation with computer aided verification, in this paper we address the question of automatic generation of invariants for loops of a special form. These loops are characterized by the following conditions: (i) they contain only assignments to variables and conditional loops; (ii) tests conditions are omitted; (iii) the variables in assignments range over numeric types, such as integers or rationals; (iv) the variables can be expressed as a polynomial of the initial values of variables (those when the loop is entered), the loop counter, and some new variables, where there are algebraic dependencies among the new variables. We call such loops *P-solvable*. There are many natural examples of P-solvable loops in real-life programs, e.g. *affine* loops are P-solvable.

Finding invariants for P-solvable loops may be a very hard and creative work since non-trivial mathematical knowledge and intuition may be required. In [14], we derived a systematic method for generating *polynomial loop invariants* of P-solvable loops. These invariants are of the form $p_1 = 0 \wedge \cdots \wedge p_r = 0$, where $p_1, \ldots, p_r$ are polynomials over the program variables. In the sequel we will call a *polynomial equality* any equality of the form $p = 0$, where p is a polynomial, thus an invariant is polynomial if it is a conjunction of polynomial equalities. Finding valid polynomial identities (i. e. invariants) has applications in many classical data flow analysis problems [20], e. g., constant propagation, discovery of symbolic constants, discovery of loop induction variables, etc.

* Work was done while the author was at RISC-Linz, Austria.

Our method for invariant generation first translates conditional statements within the loop into a sequence of loops, called inner loops. Then all loop conditions are ignored, thus turning the loop into a non-deterministic program. Next symbolic summation methods are applied to the inner loops to determine if the output values of their variables can be expressed using symbolic expressions in their input values and inner loop counters. If yes, a collection of potential polynomial invariants is generated using a Gröbner basis algorithm to eliminate loop counters. The invariant property of these polynomials are then checked using the weakest precondition strategy, and valid polynomial invariants of P-solvable loops are thus derived. Moreover, we proved that under some conditions the method computes *all* polynomial invariants, i.e. it computes a Gröbner basis of the ideal of polynomial invariants. However, we could not find any example of a P-solvable loop for which our approach fails to be complete. We thus conjecture that the imposed completeness conditions cover a large class of imperative programs, and the completeness proof of our approach without the additional assumptions is a challenging task for further research.

Exploiting the symbolic manipulation capabilities of the computer algebra system *Mathematica*, our approach is implemented in a new software package called `Aligator` [14]. `Aligator` includes algorithms for solving special classes of recurrence relations (those that are either Gosper-summable or C-finite) and generating polynomial dependencies among algebraic exponential sequences. Using `Aligator`, a complete set of polynomial invariants is successfully generated for numerous imperative programs working on numbers [14]; some of these examples are presented in this paper.

The automatically obtained invariant assertions, together with the user-asserted non-polynomial invariant properties, can be subsequently used for proving the partial correctness of programs by generating appropriate verification conditions as first-order logical formulas. This verification process is supported in an imperative verification environment implemented in the *Theorema* system [3].

The work presented in this paper is an overview of already published theoretical results and practical experiments regarding verification of imperative programs [18, 14, 17, 16, 15]. Our goal in this paper is to present and exemplify the main ideas and requirements that are involved in the invariant generation process; for technical details and correctness of the presented algorithms and properties we refer to the papers mentioned before.

2 Related Work

In [10], M. Karr proposed a general technique for finding affine relationships among program variables. However, Karr's work used quite complicated operations (transformations on invertible/non-invertible assignments, affine union of spaces) and had a limitation on arithmetical operations among the program variables. For these reasons, extension of his work has recently become a challenging research topic.

One line of work uses a *generic polynomial relation* of an a priori *fixed degree* [21, 22, 28, 9, 26]. Coefficients of the polynomial are replaced by variables, and constraints over the values of the coefficients are derived. The solution space of this constraint system characterizes the coefficients of all polynomial invariants up to the fixed degree.

However, in case when the program has polynomial invariants of different degrees, these approaches have to be applied separately for the different degrees. This is not the case of our algorithm. Our restriction is not on the degree of sought polynomial relations, but on the type of assignments (recurrence equations) present in the loop body. The shape of assignments restricts our approach to the class of P-solvable loops, and thus we cannot handle loops with arbitrary polynomial assignments, nor tests in loop condition.

A different line of research imposes structural constraints on the assignment statements of the loop. Based on the theory of Gröbner basis, in [27] a fixpoint procedure for invariant generation is presented for so–called *simple* loops having *solvable mappings with positive rational eigenvalues*. This fixpoint is the ideal of polynomial invariants. The restriction of assignment mappings being solvable with positive rational eigenvalues ensures that the program variables can be polynomially expressed in terms of the loop counter and some auxiliary *rational* variables. Hence, the concept of solvable mapping is similar to the definition of P-solvable loop. However, contrarily to [27], in our approach we compute closed form solutions of program variables for a wider class of recurrence equations (assignment statements). The restriction on the closed form solution for P-solvable loops brings our approach also to the case of having closed forms as polynomials in the loop counters and additional new variables, but, unlike [27], the new variables can be arbitrary *algebraic* numbers, and not just rationals. Contrarily to [27] where completeness is always guaranteed, the completeness of our method for loops with conditionals is proved only under additional assumptions over ideals of polynomial invariants. It is worth to be mentioned though that these additional constraints cover a wide class of loops, and we could not find any example for which the completeness of our approach is violated.

3 Preliminaries

This section starts with a brief overview of P-solvable loops. Next, we introduce some notion about recurrence solving and polynomial equalities, and recall some fundamental facts about their algorithmic treatment.

We assume that $\mathbb{K}$ is a field of characteristic zero (e.g. $\mathbb{Q}$, $\mathbb{R}$, etc.), and by $\bar{\mathbb{K}}$ we denote its algebraic closure. Throughout this paper, $X = \{x_1, \ldots, x_m\}$ $(m > 1)$ denotes the set of loop variables with initial values X_0, and $\mathbb{K}[X]$ is the ring of polynomials in the variables X with coefficients from $\mathbb{K}$.

P-solvable Loops. In our approach for generating polynomial invariants, test conditions in the loops are ignored. When we ignore conditions of loops, we will deal with non-deterministic programs. Using regular-expression like notation, in [14] we introduced the syntax and semantics of the class of non-deterministic programs that we consider. We called this class *basic non-deterministic* programs. Essentially, when we omit the condition b from a conditional statement `If`$[b$ `Then` S_1 `Else` $S_2]$, where S_1 and S_2 are sequences of assignments, we will write it as `If`$[\ldots$ `Then` S_1 `Else` $S_2]$ and mean the basic non-deterministic program $S_1|S_2$. Similarly, we omit the condition b from a loop `While`$[b, S]$, where S is a sequence of assignments, and write it in the form `While`$[\ldots, S]$ to mean the basic non-deterministic program S^*. Ignoring the tests

in the conditional branches means that either branch is executed in every possible way, whereas ignoring the test condition of the loop means the loop is executed arbitrarily many nonzero times. We will refer to the loop obtained in this way by dropping the loop condition and all test conditions also as a *P-solvable loop.*

In the rest of this paper we will focus on *non-deterministic P-solvable loops with assignments and conditional branches with ignored conditions*, written as below.

$$\texttt{While}[\ldots, \texttt{If}[\ldots \texttt{ Then } S_1];\ \ldots\ ; \texttt{If}[\ldots \texttt{ Then } S_k]]. \tag{1}$$

In our work, we developed a systematic method for generating (all) polynomial invariants for loops of such a non-deterministic syntax and semantics. Namely, we identified a class of loops with assignments, sequencing and conditionals, called the P-solvable loops, for which tests are ignored, and the value of each loop variable is expressed as a polynomial in the initial values of variables (those when the loop is entered), the loop counter, and some new variables, where there are polynomial relations among the new variables. The class of P-solvable loops includes the simple situations when the expressions in the assignment statements are affine mappings, and thus affine loops are P-solvable [14].

A precise definition of P-solvable loops can be found in [14].

Polynomial Ideals and Invariants. A non-empty subset $I \subseteq \mathbb{K}[X]$ is an *ideal* of $\mathbb{K}[X]$ if $p_1 + p_2 \in I$ for all $p_1,\ p_2 \in I$ and $pq \in \mathbb{K}[X]$ for all $p \in I$ and $q \in \mathbb{K}[X]$. As observed in [25], the set of polynomials p such that $p = 0$ is a polynomial invariant forms a polynomial ideal, called *polynomial invariant ideal*.

By Hilbert's basis theorem [1], any ideal, and in particular thus the polynomial invariant ideal, has a finite basis. Using the Buchberger Algorithm [2], a special ideal basis called Gröbner basis $\{p_1, \ldots, p_r\}$ ($p_i \in \mathbb{K}[X]$) of the polynomial invariant ideal can be effectively computed. Hence, the conjunction of the polynomial equations corresponding to the polynomials from the computed basis (i.e. $p_i(X) = 0$) characterizes completely the polynomial invariants of the loop. Namely, any other polynomial invariant can be derived as a logical consequence of $p_1 = 0 \wedge \cdots \wedge p_r = 0$.

In the process of deriving a basis for the polynomial invariant ideal, we rely on efficient methods from algorithmic combinatorics, as presented below.

Sequences and Recurrences. From the assignments statements of a P-solvable loop, recurrence equations of the variables are built and solved, using the loop counter n as the recurrence index.

In what follows, $f : \mathbb{N} \to \mathbb{K}$ defines a *(univariate) sequence* of values $f(n)$ from $\mathbb{K}$ ($n \in \mathbb{N}$). A *recurrence equation* for the sequence f is a rational function defining the values of $f(n + r)$ in terms of the previous values $f(n), f(n + 1), \ldots, f(n + r - 1)$, where $r \in \mathbb{N}$ is called the *order* of the recurrence. A solution of the recurrence equation $f(n)$, that is a *closed-form solution*, expresses the value of $f(n)$ as a function of the summation variable n and some given initial values, e.g. $f(0), \ldots, f(r-1)$. A detailed presentation of sequences and recurrences can be found in [5, 7]. In our research, we only consider special classes of recurrence equations, as follows.

A *C-finite recurrence* $f(n)$ is of the form $f(n + r) = a_0(n)f(n) + a_1(n)f(n + 1) + \ldots + a_{r-1}(n)f(n + r - 1)$, where the constants $a_0, \ldots, a_{r-1} \in \mathbb{K}$ do not depend on n. The closed form of a C-finite recurrence can always be computed [31, 5], and it is

a linear combination of polynomials in n and algebraic exponential sequences $\theta^n \in \bar{\mathbb{K}}$, with $\theta \in \bar{\mathbb{K}}$, where there exist polynomial relations among the exponential sequences. By adding any linear combination of polynomials and exponential sequences in n to the rhs of a C-finite recurrence, we obtain an *inhomogeneous linear recurrence* $f(n+r) = a_0(n)f(n) + a_1(n)f(n+1) + \ldots + a_{r-1}(n)f(n+r-1) + \sum_i q_i(n)\theta_i^n$ *with constant coefficients*, where $q_i \in \bar{\mathbb{K}}[n]$ and $\theta_i \in \bar{\mathbb{K}}$. Such recurrences can be transformed into C-finite ones, and thus their closed forms can be always computed. For solving such recurrences, we rely on our *Mathematica* implementation integrated into `Aligator`. For example, the closed form of $f(n+1) = 4f(n) + 2$ is $f(n) = 4^n f(0) - \frac{2}{3}(4^n - 1)$, where $f(0)$ is the initial value of $f(n)$.

A *Gosper-summable* recurrence $f(n)$ is of the form $f(n+1) = f(n) + h(n)$, where the sequence $h(n)$ can be a product of rational function-terms, exponentials, factorials and binomials in the summation variable n (all these factors can be raised to an integer power). The closed form solution of a Gosper-summable recurrence can be exactly computed using the decision algorithm given by [6]. In our research, we use a *Mathematica* implementation of the Gosper-algorithm given by the RISC Combinatorics group [23]. For example, the closed form of $f(n+1) = f(n) + n^5$ is $f(n) = f(0) - \frac{1}{12}n^2 + \frac{5}{12}n^4 - \frac{1}{2}n^5 + \frac{1}{6}n^6$, where $f(0)$ is the initial value of $f(n)$.

In our work, we only consider P-solvable loops whose assignment statements describe Gosper-summable or C-finite recurrences. Thus, the closed forms of loop variables can be computed as presented above.

Algebraic Dependencies. As mentioned already, the closed form solutions of the variables of a P-solvable loop are polynomial expressions in the summation variable n and algebraic exponential sequences in n. We thus need to relate this sequences in a polynomial manner, such that the exponential sequences can be eliminated from the closed forms of the loop variables, and polynomial invariants can be subsequently derived. In other words, we need to compute the *algebraic dependencies* among the exponential sequences $\theta_1^n, \ldots, \theta_s^n \in \bar{\mathbb{K}}$ of the algebraic numbers $\theta_1, \ldots, \theta_s \in \bar{\mathbb{K}}$ present in the closed forms.

An *algebraic dependency* (or *algebraic relation*) of these sequences over $\bar{\mathbb{K}}$ is a polynomial $p \in \bar{\mathbb{K}}[y_1, \ldots, y_s]$ in s distinct variables $y_1, \ldots, y_s$, such that $p(\theta_1^n, \ldots, \theta_s^n) = 0, \forall n \in \mathbb{N}$. Computing algebraic dependencies reduces thus to compute a Gröbner basis of the ideal of all algebraic dependencies. We integrated in our framework a *Mathematica* implementation for deriving such a basis [11]. For example, $\theta_1^n\theta_2^n - 1 = 0$ generates the ideal of algebraic dependencies among the exponential sequences of $\theta_1 = 4$ and $\theta_2 = \frac{1}{4}$, whereas there is no algebraic dependency among the exponential sequences of $\theta_1 = 4$ and $\theta_2 = 3$.

4 P-solvable Loops with Conditional Branches

We have now all necessary ingredients to synthesize our invariant generation algorithm for P-solvable loops with assignments and (nested) conditionals. This is achieved in Algorithm 4.3, as described below.

(1) P-solvable loop with conditional branches (1), i.e. outer loop, is transformed into nested P-solvable loops with assignments only, i.e. inner loops.

(2) Further, using fresh variables denoting inner loop counters, the body of each P-solvable inner loop is written as a system of recurrence equations of the loop variables X. Recurrences are then exactly solved, and *polynomial* closed forms of inner loops are derived. Next, the ideal of algebraic dependencies among the exponential sequences from the derived closed forms is computed.

Further, the polynomial invariants of (1) are inferred using the closed forms of its inner loops, as presented in the next steps.

(3) Inner loops are executed in all possible orders. Hence, closed forms are "merged" to express the behavior of a sequence of inner loops as a polynomial system in the inner loop counters, initial values of variables (those before the inner loop sequence), and some new variables standing for the exponential sequences in the inner loop counters, where there are algebraic dependencies among the new variables. Loop counters are next eliminated by Gröbner basis computation, and the ideal of valid polynomial identities after arbitrary inner loop sequences are thus derived.

(4) Further, the intersection of the polynomial ideals of all inner loop sequences is computed, and the ideal of polynomial relations for an *arbitrary iteration* of the outer loop (3) is obtained. In particular, the ideal of polynomial relations after the *first* iteration of the outer loop are inferred, as candidate polynomial invariants of (1) (i.e. after arbitrary many iterations).

(5) Using the weakest precondition strategy, the inductiveness property of candidate polynomial invariants is checked, and finally polynomial invariants for (1) are obtained. Moreover, under the additional assumptions introduced in Theorem 4.9, we prove that our approach is complete. Namely, it returns a basis for the polynomial invariant ideal for some special cases of P-solvable loops with conditional branches and assignments.

In what follows, we discuss the above steps in more detail.

1. Loop Transformation. The transformation rule is given below.

THEOREM 4.1 Let us consider the following two loops:

$$\texttt{While}[b, s_0; \texttt{If}[b_1\ \texttt{Then}\ s_1\ \texttt{Else}\ \ldots\ \texttt{If}[b_{k-1}\ \texttt{Then}\ s_{k-1}\ \texttt{Else}\ s_k]\ldots]; s_{k+1}] \quad (2)$$

and

$$\begin{array}{l} \texttt{While}[b, \\ \texttt{While}[b \wedge b'_1,\ s_0; s_1; s_{k+1}]; \\ \ldots \\ \texttt{While}[b \wedge \neg b'_1 \wedge \cdots \wedge \neg b'_{k-1},\ s_0; s_k; s_{k+1}]], \end{array} \quad (3)$$

where $s_0,\ s_1,\ \ldots,\ s_k,\ s_{k+1}$ are sequences of assignments, and $b'_i = \mathrm{wp}(s_0, b_i)$ is the weakest precondition of s_0 with postcondition b_i, $i = 1, \ldots, k-1$.
Then any formula I is an invariant of the first loop if and only if it is an invariant of the second loop and all of its inner loops.

Since in our approach for invariant generation tests are ignored in the loop and conditional branches, the loop (2) can be equivalently written as (1), by denoting $S_i =$

$s_0; s_i; s_{k+1}$. Further, using our notation for basic non-deterministic programs mentioned on page 125, the outer loop (3) can be written as $(S_1|S_2|\ldots|S_k)^*$. Based on Theorem 4.1, an imperative loop having $k \geq 1$ *conditional branches and assignment statements only* is called *P-solvable* if the inner loops obtained after performing the transformation rule from Theorem 4.1 are P-solvable.

EXAMPLE 4.2 Consider the loop implementing Wensley's algorithm for real division [29].

$$\begin{array}{l} \texttt{While}[(d \geq Tol), \\ \quad \texttt{If}[(P < a + b) \\ \quad\quad \texttt{Then}\ b := b/2; d := d/2 \\ \quad\quad \texttt{Else}\ a := a + b; y := y + d/2; b := b/2; d := d/2]]. \end{array} \tag{4}$$

After applying Theorem 4.1 and omitting all test conditions, the obtained nested loop system is as follows.

$$\begin{array}{ll} & \texttt{While}[\ldots, \\ S_1: & \texttt{While}[\ldots, b := b/2; d := d/2]; \\ S_2: & \texttt{While}[\ldots, a := a + b; y := y + d/2; b := b/2; d := d/2]]. \end{array}$$

2. Closed Forms of Inner Loops. For each P-solvable inner loop S_i, where $i = 1, \ldots, k$, with loop counter $j_i \in \mathbb{N}$, the recurrence equations of the loop variables X are first extracted. Next, the type of recurrences are identified and solved by the methods from page 126. Using the P-solvable loop property, the closed form system of an inner loop S_i with j_i iterations can be thus written as:

$$\begin{cases} x_1[j_i] = q_{i,1}(j_i, \theta_{i1}^{j_i}, \ldots, \theta_{is}^{j_i}) \\ \vdots \\ x_m[j_i] = q_{i,m}(j_i, \theta_{i1}^{j_i}, \ldots, \theta_{is}^{j_i}) \end{cases}, \text{ where } \begin{array}{l} \theta_{i,r} \in \bar{\mathbb{K}},\ q_{i,l} \in \bar{\mathbb{K}}[j_i, \theta_{i1}^{j_i}, \ldots, \theta_{is}^{j_i}], \\ q_{i,l} \text{ are parameterized by } X_0, \\ r = 1, \ldots, s,\ \ l = 1, \ldots, m \end{array} \tag{5}$$

Furthermore, the ideal A_i of algebraic dependencies among $j_i, \theta_{i1}^{j_i}, \ldots, \theta_{is}^{j_i}$ is computed. Conform page 127, we thus have $A_i = I(j_i, \theta_{i1}^{j_i}, \ldots, \theta_{is}^{j_i})$.

The structural restrictions on recurrences from page 126 are crucial. If the recurrences cannot be solved exactly, or their closed forms do not fulfill the P-solvable form, our algorithm fails in generating valid polynomial relations among the loop variables.

EXAMPLE 4.3 For Example 4.2, the closed form systems of its inner loops S_1 and S_2 are given below.

Inner loop S_1:
$j_1 \in \mathbb{N}$
$z_{11} = 2^{-j_1}, z_{12} = 2^{-j_1}$

$$\begin{cases} a[j_1] = a[0_1] \\ b[j_1] \underset{C-finite}{=} b[0_1] * z_{11} \\ d[j_1] \underset{C-finite}{=} d[0_1] * z_{12} \\ y[j_1] = y[0_1] \end{cases}$$

Inner loop S_2:
$j_2 \in \mathbb{N}$
$z_{21} = 2^{-j_2}, z_{22} = 2^{-j_2}, z_{23} = 2^{-j_2}, z_{24} = 2^{-j_2}$

$$\begin{cases} a[j_2] \underset{Gosper}{=} a[0_2] + 2 * b[0_2] - 2 * b[0_2] * z_{21} \\ b[j_2] \underset{C-finite}{=} b[0_2] * z_{22} \\ d[j_2] \underset{C-finite}{=} d[0_2] * z_{23} \\ y[j_2] \underset{Gosper}{=} y[0_2] + d[0_2] - d[0_2] * z_{24}, \end{cases}$$

with the computed algebraic dependencies

$$\{\, z_{11} - z_{12} = 0 \qquad \text{and} \qquad \begin{cases} z_{21} - z_{24} = 0 \\ z_{22} - z_{24} = 0 \\ z_{23} - z_{24} = 0, \end{cases}$$

where $X_{01} = \{a[0_1], b[0_1], d[0_1], y[0_1]\}$ and $X_{02} = \{a[0_2], b[0_2], d[0_2], y[0_2]\}$ are respectively the initial values of a, b, d, y before entering the inner loops S_1 and S_2.

3. Polynomial Relations of Inner Loop Sequences. In the general case of a P-solvable loop (2) with a nested conditional statement having $k \geq 1$ conditional branches, by applying Theorem 4.1, we obtain an outer loop (3) with k P-solvable inner loops $S_1, \ldots, S_k$. Thus an *arbitrary iteration* of the outer loop is described by an arbitrary sequence of the k P-solvable loops. Since the tests are ignored, for any iteration of the outer loop we have $k!$ possible sequences of inner P-solvable loops.
Let us denote the set of permutations of length k over $\{1, \ldots, k\}$ by $\mathfrak{S}_k$. Consider a permutation $W = (w_1, \ldots, w_k) \in \mathfrak{S}_k$ and a sequence of numbers $J = \{j_1, \ldots, j_k\} \in \mathbb{N}^k$. Then we write $S_W^J = S_{w_1}^{j_{w_1}}; S_{w_2}^{j_{w_2}}; \ldots; S_{w_k}^{j_{w_k}}$ to denote an *arbitrary iteration of the outer loop*, i.e. an arbitrary sequence of the k inner loops. By S_i^j we mean the sequence of assignments $\underbrace{S_i; \ldots; S_i}_{j \text{ times}}$.

As previously discussed, for each P-solvable inner loop $S_{w_i}^{j_{w_i}}$ from S_W^J we obtain its system of closed forms together with its ideal of algebraic dependencies among the exponential sequences (steps 1-4 of Algorithm 4.1). Further, the system of closed forms of loop variables after S_W^J is obtained by *merging* the closed forms of its inner loops. Merging is based on the fact that the initial values of the loop variables corresponding to the inner loop $S_{w_{i+1}}^{j_{w_{i+1}}}$ are given by the final values of the loop variables after $S_{w_i}^{j_{w_i}}$ (step 5 of Algorithm 4.1). In [14] we showed that merging of closed forms of P-solvable inner loops yields a polynomial closed form system as well.
We can now compute the ideal of valid polynomial relations among the loop variables X with initial values X_0 corresponding to the sequence of assignments $\underbrace{S_{w_1}; \ldots; S_{w_1}}_{j_{w_1} \text{ times}}$; $\underbrace{S_{w_2}; \ldots; S_{w_2}}_{j_{w_2} \text{ times}}; \ldots \ \ldots; \underbrace{S_{w_k}; \ldots; S_{w_k}}_{j_{w_k} \text{ times}}$. Using notation introduced on page 125, we thus compute the ideal of valid polynomial relations after $S_{w_1}^*; \ldots; S_{w_k}^*$. This is presented in Algorithm 4.1.

Algorithm 4.1 Polynomial Relations of a P-solvable Loop Sequence
Input: k P-solvable inner loops $S_{w_1}, \ldots, S_{w_k}$
Output: The ideal $G \trianglelefteq \mathbb{K}[X]$ of polynomial relations among X with initial values X_0 after $S_{w_1}^*; \ldots; S_{w_k}^*$
Assumption: S_{w_i} are sequences of assignments, $w_i \in \{1, \ldots, k\}, j_i \in \mathbb{N}, k \geq 1$

1 **for** each $S_{w_i}^{j_{w_i}}$, $i = 1, \ldots, k$ **do**
2 Compute the closed form of $S_{w_i}^{j_{w_i}}$
3 Compute the ideal A_{w_i} of algebraic dependencies for $S_{w_i}^{j_{w_i}}$

4 **endfor**

5 Compute the merged closed form of $S_{w_1}^{j_{w_1}}; \ldots; S_{w_k}^{j_{w_k}}$:

$$\begin{cases} x_1[j_1,\ldots,j_k] = f_1(j_{w_1}, \theta_{w_1 1}^{j_{w_1}}, \ldots, \theta_{w_1 s}^{j_{w_1}}, \ldots \ldots, j_{w_k}, \theta_{w_k 1}^{j_{w_k}}, \ldots, \theta_{w_k s}^{j_{w_k}}) \\ \vdots \\ x_m[j_1,\ldots,j_k] = f_m(j_{w_1}, \theta_{w_1 1}^{j_{w_1}}, \ldots, \theta_{w_1 s}^{j_{w_1}}, \ldots \ldots, j_{w_k}, \theta_{w_k 1}^{j_{w_k}}, \ldots, \theta_{w_k s}^{j_{w_k}}) \end{cases}, \text{where}$$

$f_l \in \bar{\mathbb{K}}[z_{10},\ldots,z_{1s},\ldots \ldots,z_{k0},\ldots,z_{ks}]$,
the variables $z_{i0},\ldots,z_{is}$ are standing for the C-finite sequences $j_{w_i}, \theta_{w_i 1}^{j_{w_i}}, \ldots, \theta_{w_i s}^{j_{w_i}}$,
the coefficients of f_l are given by the initial values before $S_{w_1}^{j_{w_1}}; \ldots; S_{w_k}^{j_{w_k}}$

6 $A_* = \sum_{i=1}^{k} A_{w_i}$

7 $I = \langle x_1 - f_1, \ldots, x_m - f_m \rangle + A_* \subset \bar{\mathbb{K}}[z_{10},\ldots \ldots,z_{ks}, x_1,\ldots,x_m]$

8 **return** $G = I \cap \mathbb{K}[x_1,\ldots,x_m]$.

Elimination of $z_{10},\ldots,z_{ks}$ at step 8 is performed by Gröbner basis computation of I w.r.t. an elimination order $\succ$ such that $z_{10} \succ \cdots \cdots \succ z_{ks} \succ x_1 \cdots \succ x_m$.

EXAMPLE 4.4 For Example 4.2, the steps of Algorithm 4.1 are presented below.
Steps 1-4. The closed form systems of the inner loops S_1 and S_2 are as in Example 4.3.
Steps 5-6. For the inner loop sequence $S_1^{j_1}; S_2^{j}$ the initial values X_{02} are given by the values $a[j_1], b[j_1], d[j_1], y[j_1]$ after $S_1^{j_1}$. Hence, the merged closed form of $S_1^{j_1}; S_2^{j_2}$ is given below. For simplicity, let us rename the initial values X_{01} to respectively $X_0 = \{a[0], b[0], d[0], y[0]\}$.

$$\begin{cases} a[j_1,j_2] = a[0] + 2 * b[0] * z_{11} - 2b[0] * z_{21} * z_{11} \\ b[j_1,j_2] = b[0] * z_{22} * z_{11} \\ d[j_1,j_2] = d[0] * z_{12} * z_{23} \\ y[j_1,j_2] = y[0] + d[0] * z_{12} - d[0] * z_{24} * z_{12}, \end{cases} \tag{6}$$

with the already computed algebraic dependencies

$$A_* = \langle z_{11} - z_{12}, z_{21} - z_{24}, z_{22} - z_{24}, z_{23} - z_{24} \rangle. \tag{7}$$

Steps 7, 8. From (6) and (7), by eliminating $z_{11}, z_{12}, z_{21}, z_{22}, z_{23}, z_{24}$, we obtain the ideal of polynomial relations for $S_1^{j_1}; S_2^{j_2}$, as below.

$$G = \langle\, -b[0] * d + b * d[0], a * d[0] - a[0] * d[0] - 2 * b[0] * y + 2 * b[0] * y[0], \\ a * d - a[0] * d - 2 * b * y + 2 * b * y[0] \rangle.$$

4. Polynomial Relations of an Iteration of (3)**.** In order to get all polynomial relations among the loop variables X with initial values X_0 corresponding to an arbitrary iteration of the outer loop (3), we need to apply Algorithm 4.1 on each possible sequence of k inner loops that are in a number of $k!$. This way, for each sequence of k inner loops we get the ideal of their polynomial relations among the loop variables X with initial values X_0 (step 3 of Algorithm 4.2). Using ideal theoretic results, by taking the *intersection* of

all these ideals, we derive *the ideal of polynomial relations among the loop variables X with initial values X_0 that are valid after any sequence of k P-solvable inner loops* (step 4 of Algorithm 4.2). The intersection ideal thus obtained is the ideal of polynomial relations among the loop variables X with initial values X_0 *after an arbitrary iteration of the outer loop* (3). This can be algorithmically computed as follows.

Algorithm 4.2 Polynomial Relations for an Iteration of (3)
Input: P-solvable loop (3) with P-solvable inner loops $S_1, \ldots, S_k$
Output: The ideal $PI \subset \mathbb{K}[X]$ of the polynomial relations among X with initial values X_0 corresponding to an arbitrary iteration of (3)
Assumption: X_0 are the initial values of X before the arbitrary iteration of (3)

1 $PI =$ Algorithm 4.1$\big(S_1,\ \ldots,\ S_k\big)$
2 **for** each $W \in \mathfrak{S}_k \setminus \{(1, \ldots, k)\}$ **do**
3 $\quad G =$ Algorithm 4.1$\big(S_{w_1},\ \ldots,\ S_{w_k}\big)$
4 $\quad PI = PI \cap G$
5 **endfor**
6 **return** PI.

THEOREM 4.5 Algorithm 4.2 is correct. It returns the generators for the ideal PI of polynomial relations among the loop variables X with initial values X_0 after a possible iteration of the outer loop (3).

EXAMPLE 4.6 Similarly to Example 4.4, we compute the ideal of polynomial relations for $S_2^{j_2}; S_1^{j_1}$ for Example 4.2. Further, we take the intersection of the ideals of polynomial relations for $S_1^{j_1}; S_2^{j_2}$ and $S_2^{j_2}; S_1^{j_1}$. We thus obtain

$$PI = \langle\ -b[0] * d + b * d[0], a * d[0] - a[0] * d[0] - 2 * b[0] * y + 2 * b[0] * y[0],$$
$$a * d - a[0] * d - 2 * b * y + 2 * b * y[0]\rangle.$$

5. Polynomial Invariants of Outer Loop (1)**.** What remains is to identify the relationship between the polynomial invariants among the loop variables X of the *outer loop* (1) and the computed polynomial relations using Algorithm 4.2 for *an arbitrary iteration of the outer loop* (3). For doing so, we proceed as follows.

1. Note that the initial values X_0 of the loop variables X at the entry point of the outer loop are also the initial values of the loop variables X before the *first* iteration of the outer loop (3). We thus firstly compute by Algorithm 4.2 the ideal of all polynomial relations among the loop variables X with initial values X_0 corresponding to the *first* iteration of the outer loop (3). We denote this ideal by PI_1.
2. Next, from (the generators of) PI_1 we keep only the set GI of polynomial relations that are invariants among the loop variables X with initial values X_0: they are preserved by any iteration of the outer loop (3) starting in a state in which the initial values of the loop variables X are X_0. By correctness of Theorem 4.1, the polynomials from GI thus obtained are invariants among the loop variables X with initial values X_0 of the P-solvable loop (2), and thus of (1) (see Theorem 4.7).

Finally, we can now formulate our algorithm for polynomial invariant generation for P-solvable loops with conditional branches and assignments.

Algorithm 4.3 P-solvable Loops with Non-deterministic Conditionals
Input: P-solvable loop (2) with $k \geq 1$ conditional branches and assignments
Output: Polynomial invariants of (2) among X with initial values X_0

1 Apply Theorem 4.1, yielding a nested loop (3) with k P-solvable inner loops $S_1, \ldots, S_k$
2 Apply Algorithm 4.2 for computing the ideal PI_1 of polynomial relations among X after the first iteration of the outer loop (3)
3 From PI_1 keep the set GI of those polynomials whose conjunction is preserved by each $S_1, \ldots, S_k$:

$GI = \{p \in PI_1 \mid wp(S_i, p(X) = 0) \in \langle GI \rangle, i = 1, \ldots, k\} \subset PI_1$, where $wp(S_i, p(X) = 0)$ is the weakest precondition of S_i with postcondition $p(X) = 0$

4 $\underline{return}\ GI$.

THEOREM 4.7 Algorithm 4.3 is correct. It returns polynomial invariants among the loop variables X with initial values X_0 of the P-solvable loop (2) (and thus of (1)).

EXAMPLE 4.8 From Example 4.6 we already have the set PI_1 for Example 4.2. By applying step 3 of Algorithm 4.3, the set of polynomial invariants for Example 4.2 is

$$GI = \{b[0] * d + b * d[0], a * d[0] - a[0] * d[0] - 2 * b[0] * y + 2 * b[0] * y[0], \\ a * d - a[0] * d - 2 * b * y + 2 * b * y[0]\}.$$

In what follows, we state under which additional assumptions Algorithm 4.3 returns a basis of the polynomial invariant ideal. We fix some further notation.
– J_* the polynomial invariant ideal among X with initial values X_0 of the P-solvable loop (2).
– J_W denotes the ideal of polynomial relations among X with initial values X_0 after S_W^J.
– For all $i = 1, \ldots, k$ and $j \in \mathbb{N}$, we denote by $J_{W,i}$ the ideal of polynomial relations among X with initial values X_0 after $S_W^J; S_i^j$.
For proving completeness of our method, we impose structural conditions on the ideal of polynomial relations among X with initial values X_0 corresponding to sequences of k and $k+1$ inner loops, as presented below.

THEOREM 4.9 Let $\mathfrak{a}_k = \bigcap_{W \in \mathfrak{S}_k} J_W$ and $\mathfrak{a}_{k+1} = \bigcap_{\substack{W \in \mathfrak{S}_k \\ i=1,\ldots,k}} J_{W,i}$. Let GI be as in Algorithm 4.3.

1. If $\mathfrak{a}_k = \mathfrak{a}_{k+1}$ then $J_* = \mathfrak{a}_k$.
2. If $\langle GI \rangle = \mathfrak{a}_k \cap \mathfrak{a}_{k+1}$ then $J_* = \mathfrak{a}_k \cap \mathfrak{a}_{k+1}$.
3. If $\langle GI \rangle = \mathfrak{a}_k$ then $J_* = \mathfrak{a}_k$.

EXAMPLE 4.10 From Examples 4.6 and 4.8 we obtain $GI = PI_1$. By Theorem 4.9 we thus derive $GI = J_*$, yielding the completeness of Algorithm 4.3 for Example 4.2.

Note, that P-solvable loops with assignments only are a special case of loop (1). Algorithm 4.3 is thus applicable to this class of P-solvable loops as well (i.e. for $k = 1$). Moreover, for P-solvable loops with assignments only filtering and completeness check of the result is not needed; for these loops a basis of the polynomial invariant ideal is always derived by variable elimination from the closed form system of the (single inner) loop [14, 17, 16].

5 Further Examples

Our approach to invariant generation is implemented in a new software package, called `Aligator`. The package combines algorithms from symbolic summation and polynomial algebra with computational logic, and is applicable to the rich class of P-solvable loops. `Aligator` contains routines for checking the P-solvability of loops, transforming them into a system of recurrence equations, solving recurrences and deriving closed forms of loop variables, computing the ideal of polynomial invariants by variable elimination, invariant filtering and completeness check of the resulting set of invariants

`Aligator` was implemented in *Mathematica* 5.2 [30], and is available from:

`http://mtc.epfl.ch/software-tools/Aligator/`

Using `Aligator`, we have successfully tested our method on a number of interesting number theoretic examples [14], some of them being listed in Table 1. The first column of the table contains the name of the example, the second and third columns specify the applied combinatorial methods and the number of generated polynomial invariants for the corresponding example, whereas the fourth column shows the timing (in seconds) needed on a machine with 2.0GHz CPU and 2GB of memory. The fifth columns shows whether our method was complete.

6 Conclusion

We described a framework for generating loop invariants for a family of imperative programs operating on numbers. The approach is implemented as a *Mathematica* package, called `Aligator`. `Aligator` offers software support for **a**utomated **i**nvariant **g**eneration by **a**lgebraic **t**echniques **o**ver the **r**ationals. The successful application of the package on a number of examples demonstrates the value of using symbolic summation and polynomial algebra together with computational logic for program verification.

So far, the focus has been on generating polynomial equations as loop invariants. We believe that it should be possible to identify and generate polynomial inequalities in addition to polynomial equations, as invariants as well. Quantifier elimination methods on theories, including the theory of real closed fields, should be helpful. We are also interested in generalizing the framework to programs on nonnumeric data structures.

Acknowledgements. The author wishes to thank Tudor Jebelean, Andrei Voronkov, Deepak Kapur and Manuel Kauers for their help and comments.

Example	Comb. Methods	Nr.Poly.	(sec)	Compl.
P-solvable loops with assignments only				
Division [4]	Gosper	1	0.031	yes
Integer square root [12]	Gosper	2	0.063	yes
Integer square root [13]	Gosper	1	0.046	yes
Integer cubic root [13]	Gosper	2	0.094	yes
Fibonacci [14]	C-finite, Alg.Dependencies	1	0.219	yes
Sum of powers n^5 [24]	Gosper	1	0.125	yes
P-solvable loops with conditional branches and assignments				
Wensley's Algorithm [29]	Gosper, C-finite, Alg.Dependencies	3	0.25	yes
LCM-GCD computation [4]	Gosper	1	0.437	yes
Extended GCD [13]	Gosper	5	3.094	yes
Fermat's factorization [13]	Gosper	1	0.109	yes
Square root [32]	Gosper, C-finite, Alg.Dependencies	1	0.406	yes
Binary Product [13]	Gosper, C-finite, Alg.Dependencies	1	0.219	yes
Binary Product [27]	Gosper, C-finite, Alg.Dependencies	1	0.297	yes
Binary Division (2nd Loop) [8]	C-finite, Alg. Dependencies	1	0.219	yes
Hardware Integer Division [19]	Gosper, C-finite, Alg.Dependencies	3	0.25	yes
Hardware Integer Division [28]	Gosper, C-finite, Alg.Dependencies	3	0.25	yes
Factoring Large Numbers [13]	C-finite, Gosper	1	0.906	yes

Table 1. Experimental Results

References

1. W. W. Adams and P. Loustaunau. *An Introduction to Gröbner Bases*. Graduate Studies in Math. 3. AMS, 1994.
2. B. Buchberger. An Algorithm for Finding the Basis Elements of the Residue Class Ring of a Zero Dimensional Polynomial Ideal. *Journal of Symbolic Computation*, 41(3-4):475–511, 2006. Phd thesis 1965, University of Innsbruck, Austria.
3. B. Buchberger, A. Craciun, T. Jebelean, L. Kovacs, T. Kutsia, K. Nakagawa, F. Piroi, N. Popov, J. Robu, M. Rosenkranz, and W. Windsteiger. Theorema: Towards Computer-Aided Mathematical Theory Exploration. *Journal of Applied Logic*, 4(4):470–504, 2006.
4. E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
5. G. Everest, A. van der Poorten, I. Shparlinski, and T. Ward. *Recurrence Sequences*, volume 104 of *Mathematical Surveys and Monographs*. American Mathematical Society, 2003.
6. R. W. Gosper. Decision Procedures for Indefinite Hypergeometric Summation. *Journal of Symbolic Computation*, 75:40–42, 1978.
7. R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics*. Addison-Wesley Publishing Company, 2nd edition, 1989.
8. A. Kaldewaij. *Programming. The Derivation of Algorithms*. Prentince-Hall, 1990.
9. D. Kapur. A Quantifier Elimination based Heuristic for Automatically Generating Inductive Assertions for Programs. *Journal of Systems Science and Complexity*, 19(3):307–330, 2006.
10. M. Karr. Affine Relationships Amomg Variables of Programs. *Acta Informatica*, 6:133–151, 1976.

11. M. Kauers and B. Zimmermann. Computing the Algebraic Relations of C-finite Sequences and Multisequences. Technical Report 2006-24, SFB F013, 2006.
12. M. Kirchner. Program Verification with the Mathematical Software System Theorema. Technical Report 99-16, RISC-Linz, Austria, 1999. Diplomaarbeit.
13. D. E. Knuth. *The Art of Computer Programming*, volume 2. Seminumerical Algorithms. Addison-Wesley, 3rd edition, 1998.
14. L. Kovács. *Automated Invariant Generation by Algebraic Techniques for Imperative Program Verification in Theorema*. PhD thesis, RISC, Johannes Kepler University Linz, 2007.
15. L. Kovács. Aligator: A Mathematica Package for Invariant Generation. In *Proc. of IJCAR*, LNCS, Sydney, Australia, 2008. To appear.
16. L. Kovács. Invariant Generation for P-solvable Loops with Assignments. In *Proc. of CSR*, volume 5010 of *LNCS*, pages 349–359, Moscow, Russia, 2008. To appear.
17. L. Kovács. Reasoning Algebraically About P-Solvable Loops. In *Proc. of TACAS*, volume 4963 of *LNCS*, pages 249–264, Budapest, Hungary, 2008.
18. L. Kovács, N. Popov, and T. Jebelean. Combining Logic and Algebraic Techniques for Program Verification in Theorema. In *Proc. of ISOLA 2006*, 2006.
19. Z. Manna. *Mathematical Theory of Computation*. McGraw-Hill Inc., 1974.
20. M. Müller-Olm, M. Petter, and H. Seidl. Interprocedurally Analyzing Polynomial Identities. In *Proc. of STACS 2006*, 2006.
21. M. Müller-Olm and H. Seidl. Computing Polynomial Program Invariants. *Indormation Processing Letters*, 91(5):233–244, 2004.
22. M. Müller-Olm and H. Seidl. Precise Interprocedural Analysis through Linear Algebra. In *Proc. of the 31st POPL*, pages 330–341, 2004.
23. P. Paule and M. Schorn. A Mathematica Version of Zeilberger's Algorithm for Proving Binomial Coefficient Identities. *Journal of Symbolic Computation*, 20(5-6):673–698, 1995.
24. M. Petter. Berechnung von polynomiellen Invarianten. Master's thesis, Technical University Munich, Germany, 2004.
25. E. Rodriguez-Carbonell and D. Kapur. Automatic Generation of Polynomial Loop Invariants: Algebraic Foundations. In *Proc. of ISSAC 04*, 2004.
26. E. Rodriguez-Carbonell and D. Kapur. Automatic Generation of Polynomial Invariants of Bounded Degree using Abstract Interpretation. *Science of Computer Programming*, 64(1), 2007.
27. E. Rodriguez-Carbonell and D. Kapur. Generating All Polynomial Invariants in Simple Loops. *J. of Symbolic Computation*, 42(4):443–476, 2007.
28. S. Sankaranaryanan, H. B. Sipma, and Z. Manna. Non-Linear Loop Invariant Generation using Gröbner Bases. In *Proc. of POPL 2004*, 2004.
29. B. Wegbreit. The Synthesis of Loop Predicates. *Communication of the ACM*, 2(17):102–112, 1974.
30. S. Wolfram. *The Mathematica Book. Version 5.0*. Wolfram Media, 2003.
31. D. Zeilberger. A Holonomic System Approach to Special Functions. *Journal of Computational and Applied Mathematics*, 32:321–368, 1990.
32. K. Zuse. *The Computer - My Life*. Springer, 1993.

Approximation of String Operations in the PHP String Analyzer *

Yasuhiko Minamide

Department of Computer Science
University of Tsukuba
minamide@cs.tsukuba.ac.jp

Abstract. The PHP string analyzer approximates the possible string output of a PHP program with a context-free grammar. The approximation is obtained by repeatedly transforming grammars with finite state transducers approximating string operations. For the approximation of the regular expression functions such as those performing regular expression matching and replace, we approximate the functions as combinations of transducers. We first review how we approximate the functions in the current version of the PHP string analyzer. The approximation does not take the disambiguation strategy of the regular expression functions and thus produces a rough approximation for ambiguous regular expressions. To improve the precision of approximation, we adopt transducers with regular look-ahead and show that the disambiguation strategies can be modeled precisely.

1 Introduction

PHP is one of the most popular server-side scripting languages used to generate Web pages dynamically [AB+05]. The PHP string analyzer developed by the author is a static analysis tool that approximates the string output of a program as a context-free grammar [Min05]. The analysis is an extension of the string expression analysis of Christensen *et al.* approximating a string value of an expression with a regular language [CMS03]. The PHP string analyzer was applied to check validity of dynamically generated HTML documents [MT06,NM08] and to detect command injection vulnerabilities in server-side programs [WS07].

The analyzer obtains the context-free grammar approximating the possible string output of a program by repeatedly transforming grammars with finite state transducers approximating string operations. In order to obtain a precise approximation, it is crucial to precisely model string operations as transducers. Scripting languages such as Perl and PHP offer advantages in string manipulation by providing powerful string manipulation functions based on regular

* This work has been partially supported by CREST of JST (Japan Science and Technology Agency), and JSPS and FWF under the Japan-Austria Research Cooperative Program.

expressions. Critical operations on strings such as sanitization are often conducted by regular expression functions. Thus, we study how to approximate the regular expression functions precisely in this paper.

We first review how the current version of the analyzer approximates the functions as transducers. The approximation does not take the disambiguation strategy of the regular expression functions, and thus produces a rough approximation for ambiguous regular expressions. To improve the precision of approximation, we adopt transducers with regular look-ahead and show that the disambiguation strategies can be modeled precisely with transducers with regular look-ahead.

This paper is organized as follows. In Sections 2 and 3, we review the PHP string analyzer and the regular expression functions in PHP. In Section 4, we present how we modeled the regular expression functions as transducers in the current version of the analyzer. In Section 5, we introduce transducers with regular look-ahead and precisely model disambiguation strategies of the regular expression functions. Section 6 presents the decomposition of transducers with regular look-ahead that makes it possible to incorporate them in the PHP string analyzer. Finally, we present some conclusions.

2 PHP String Analyzer

In this section, we review the PHP string analyzer and present how transducers are utilized to model string operations. The analyzer takes two inputs: a PHP program and an input specification. The input specification is given as a regular expression and describes the set of possible inputs to the PHP program.

To illustrate the string analysis, let us consider the following program.

```
for ($i = 0; $i < $n; $i++)
  $x = "0".$x."1";
echo $x;
```

In PHP, the infix operator dot represents string concatenation. This program concatenates the same number of `"0"`s to the left and `"1"`s to the right of `$x`: the number depends on the value of `$n`.

The input specification is given by specifying the initial values of global variables in our analyzer. The initial values of `$x` and `$n` are described in the following specification.

```
$x : /abc|xyz/
$n : int
```

The specification `/abc|xyz/` is a regular expression representing the set of strings $\{abc, xyz\}$. Only the type is specified for the variable `$n`.

The idea of string analysis is to consider assignments as production rules of a context-free grammar. By considering assignments as production rules and translating the input specification into production rules, we can obtain the following

grammar approximating the output of the program.

$$X \to abc \mid xyz \mid 0X1$$

This grammar represents the set of strings, $\{0^n abc1^n \mid n \geq 0\} \cup \{0^n xyz1^n \mid n \geq 0\}$, as we expect.

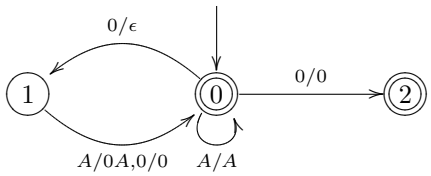
This analysis works even if a program contains string operations other than concatenation. Let us consider the following revised program:

```
for ($i = 0; $i < $n; $i++)
  $x = "0".$x."1";
echo str_replace("00","0",$x);
```

where `str_replace("00","0",$x)` replaces the string `00` in `$x` with `0`. The set of strings the variable `$x` may contain after the `for` loop is represented by the context-free grammar above. Therefore, we can obtain the context-free grammar for the output of the revised program if we know how a grammar is transformed by `str_replace("00","0",$x)`.

A finite automaton with output called a transducer plays a crucial rule here. A transducer has the key property that the image of a context-free language under a transducer is context-free [Ber79]. Furthermore, many string operations can be realized by transducers.

Let us consider `str_replace("00","0",$x)` in the example: this operation can be realized by the following transducer:



where A is any character except 0. There are three states 0, 1 and 2 in this transducer: the state 0 is the start state, and 0 and 2 are the final states. The transitions labeled with $0/\epsilon$ and $A/0A$ mean that the transducer produces ϵ and $0A$ for the inputs 0 and A, respectively. For example, this transducer outputs $0abc11$ for the input $00abc11$.

We developed an algorithm to compute the image of a context-free language under a transducer based on the the context-free graph reachability algorithm [Rep00,MR00]. By the algorithm, we obtain the following context-free grammar with the start symbol S by computing the image of the previous context-free grammar.

$$S \to abc \mid xyz \mid X1$$
$$X \to 0abc \mid 0xyz \mid 0S1$$

3 Regular Expression Operations in PHP

Scripting languages such as Perl and PHP offer advantages in string manipulation by providing powerful string manipulation functions based on regular expressions. Let us consider the following regular expression replace function.

```
preg_replace("/a([0-9]*)b/", "x\\1y", $x)
```

This function replaces substrings matching `a([0-9]*)b` in `$x` with `x\\1y` where `\\1` is replaced with the string matching the first grouped subexpression `([0-9]*)`. The following example clarifies the operation.

```
preg_replace("/a([0-9]*)b/", "x\\1y", "a01ba234b") => "x01yx234y"
```

In the PHP string analyzer, we need to approximate this kind of powerful string operations as precisely as possible.

The following two families of regular expression functions are provided in PHP.

- POSIX-compatible
  - `ereg`: perform a regular expression match.
  - `ereg_replace`: perform a regular expression search and replace.
- Perl-compatible
  - `preg_match`: perform a regular expression match.
  - `preg_replace`: perform a regular expression search and replace.

The two families adopt different disambiguation strategies and behave differently for an ambiguous regular expression.

- a POSIX-compatible regular expression matches the longest prefix of a string matching the expression.
- Perl-compatible regular expressions adopt the first and greedy strategy we will explain later.

The differences of various implementations of regular expression functions are discussed into details in [Fri06]. Vansummeren formalized several disambiguation strategies including POSIX, and showed that it is possible to precisely infer the type of regular expression matching [Van06]. The type inference for greedy matching were also discussed in [TSY02,HVP05].

Let us consider the following regular expression replace operation. It replaces the substrings matching `a+` with `x` in `aaabaaa`.

```
ereg_replace("a+", "x", "aaabaa") => "xbx"
```

Although `a+` matches any number of `a`'s except zero, by the longest matching strategy the regular expression first matches `aaa` and then matches `aa`. Thus, we obtain `xbx`.

The choice operator behaves in the similar manner for a POSIX-compatible regular expression. The regular expression `a|aa` matches `aa`, `a`, and then `aa` in the example below.

```
ereg_replace("a|aa", "x", "aaabaa") => "xxbx"
```

A Perl-compatible regular expression behaves differently. The first choice of `|` has higher priority. Thus, in the following expression `a|aa` matches five single `a`'s and produces `"xxxbxx"`.

```
preg_replace("/a|aa/", "x", "aaabaa") => "xxxbxx"
```

Kleene closure in a Perl-compatible regular expression is greedy.

```
pereg_replace("/a+/", "x", "aaabaa") => "xbx"
```

The behavior of $r*$ in Perl-compatible regular expressions can be interpreted by $rr^* + \epsilon$ where the first choice has a higher priority. That makes the Kleene closure in a Perl-compatible regular expression greedy.

4 Approximation of Regular Expression Functions

In this paper, we concentrate the approximation of the regular expression replace function. Furthermore, we do not consider the case where matched substrings are used in the replacement like `x\\1y`. The approximation for that case is briefly discussed in [Min05] and the discussion in this paper can also be extended to treat that case.

In this section, we review the definition of regular transducers and then show how the regular expression replace function can be approximated by a combination of regular transducers. The current version of the PHP string analyzer adopts the method presented in this section.

A regular transducer is an automaton with output and formalized as follows.

Definition 1. *A regular transducer T is a 6-tuple $(Q, \Sigma_i, \Sigma_o, \Delta, q_0, F)$ where Q is the finite set of states, Σ_i is the input alphabet, Σ_o is the output alphabet, Δ is the finite transition-and-output relation, $\Delta \subseteq (Q \times \Sigma_i \times Q \times \Sigma_o^*) \cup (Q \times Q)$, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of final states.*

If $(q_1, a, q_2, w) \in \Delta$, then at the state q_1 and for the input a, the transducer outputs w and changes its state to q_2. If $(q_1, q_2) \in \Delta$, the transducer can perform ϵ-transition from q_1 to q_2. In this definition, we exclude transducers that output a string for ϵ-transition.

Let us consider the regular expression replace functions. To avoid using the concrete syntax of PHP, let us write $\text{replace}^{\text{perl}}(r, w)$ and $\text{replace}^{\text{posix}}(r, w)$ where r is a regular expression defined below and w is a replacement string:

$$r ::= \emptyset \mid \epsilon \mid a \mid r + r \mid r^*$$

where a is a symbol in the alphabet. The examples in the previous section are rewritten as follows.

$$\text{replace}^{\text{posix}}(a + aa, x)(aaabaa) = xxbx$$

$$\text{replace}^{\text{perl}}(a + aa, x)(aaabaa) = xxxbxx$$

We approximate these regular expression replace functions by a combination of two transducers used by Mohri and Sproat [MS96] to compile context-sensitive rewrite rules on strings.

The first transducer is called markers of type 1. For a regular expression r, it inserts marker # every position in w where w is divided to w_1w_2 and at that position w_2 is divided to w_3w_4 for $w_3 \in L(r)$. The marker for r can be constructed as follows [1].

1. Construct a deterministic automaton $A_1 = (Q, \Sigma, \delta, q_0, F)$ accepting $\Sigma^* r^{\text{rev}}$.
2. T_1' is $(Q, \Sigma, \Sigma \cup \{\#\}, \Delta, q_0, F)$ where Δ has the following transition.

$$\begin{array}{c} (q, a, q', a) \in \Delta \ \text{ if } \delta(q, a) = q' \text{ and } q' \notin F \\ (q, a, q', a\#) \in \Delta \text{ if } \delta(q, a) = q' \text{ and } q' \in F \end{array}$$

3. The marker T_1 is the reverse of T_1'.

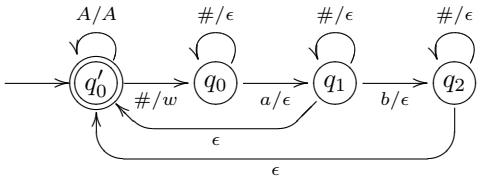
The second transducer replaces substrings matching r with a string w. Since the left ends of the substrings matching r are marked with #, it starts replacing a matching substring with w when it encounters #.

1. Construct a deterministic automaton $A_2 = (Q, \Sigma, \delta, q_0, F)$ accepting r.
2. T_2 is $(Q \cup \{q_0'\}, \Sigma \cup \{\#\}, \Sigma, \Delta, q_0', \{q_0'\})$ where Δ has the following transition.

$$\begin{array}{rl} (q_0', \#, q_0, w) \in \Delta & \\ (q_0', a, q_0', a) \in \Delta & \text{if } a \neq \# \\ (q, a, q', \epsilon) \in \Delta & \text{if } \delta(q, a) = q' \\ (q, \epsilon, q_0') \in \Delta & \text{if } q \in F \\ (q, \#, q, \epsilon) \in \Delta & \text{if } q \neq q_0' \end{array}$$

At a final state of A_2, it nondeterministacally changes its state to q_0' with ϵ-transition. This corresponds to stopping matching at the point. Because # may appear inside a substring matching r after applying T_1, we need the transitions that ignore #: $(q, \#, q, \epsilon) \in \Delta$.

Example 1. We obtain the following transducer for $a + ab$.



where A is any symbol in Σ.

[1] If $\epsilon \in L(r)$, a slight adjustment is necessary.

By applying these two transducers, we can approximate the regular expression replace function. Let us consider the following example.

$$\text{replace}^{\text{posix}}((a + ab)(b + bb), x)(abbb) = x$$

By applying T_1 and T_2 to $abbb$, we obtain the rough approximation $\{x, xb, xbb\}$. This is because we have not taken into account the longest matching strategy of the POSIX regular expression functions.

5 Modeling Regular Expression Functions with Transducers with Regular Look-Ahead

5.1 Transducers with Regular Look-Ahead

It is rather difficult to precisely model regular expression functions with the disambiguation strategies of POSIX and Perl by transducers directly. Thus, we employ regular transducers with regular look-ahead used by Engelfrite in the study of top-down tree transducers [Eng77]. Transducers with regular look-ahead make it much easier to model disambiguation strategies of POSIX and Perl.

It was shown that regular transducers with regular look-ahead can be decomposed into two transducers [Eng77].

- The first transducer preprocesses an input from the end and annotates the input.
- The second transducer simulates the transducer with look-ahead by taking advantages of the annotation added by the first transducer.

This decomposition can be considered a generalization of the construction in the previous section. Regular look-ahead was also implicitly used by the algorithms of regular expression matching with contexts of Kearns [Kea91] and greedy regular expression matching of Frisch and Cardelli [FC04].

We formalize transducers with regular look-ahead that decides whether a transition is permitted or not with a regular language associated to ϵ-transition. If a regular language r is associated with an ϵ-transition, the transition can be taken only if the rest of the input string is not in $r\Sigma^*$.

Definition 2. *A transducer with regular look-ahead T is a 6-tuple $(Q, \Sigma_i, \Sigma_o, \Delta, q_0, F)$. The only difference from the standard transducer is the transition relation: $\Delta \subseteq (Q \times \Sigma_i \times Q \times \Sigma_o^*) \cup (Q \times Q \times 2^{\Sigma_i^*})$. For $(q, q', r) \in \Delta$, r is the look-ahead regular language for the ϵ-transition from q to q' and must be regular.*

We write $(q, \epsilon, q', r) \in \Delta$ when $(q, q', r) \in \Delta$.

The configuration of a transducer is a tuple of a state, the rest of input, and the output string. The transition between configurations $(q, w, v) \vdash (q', w', v')$ is defined as follows.

$$\begin{array}{lll} (q, aw, v) \vdash (q', w, vv') & \text{if} & (q, a, q', v') \in \Delta \\ (q, w, v) \vdash (q', w, v) & \text{if} & (q, \epsilon, q', r) \in \Delta \wedge w \notin r\Sigma^* \end{array}$$

5.2 POSIX-Compatible Regular Expressions

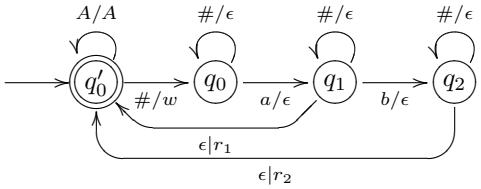
We model the POSIX-compatible regular expression replace function as a transducer with regular look-ahead. We construct a transducer T_2 replacing the longest substrings matching a regular expression r below. We assume that the mark # of the substrings matching r are inserted by T_1 in Section 4.

1. Construct a deterministic automaton $A_2 = (Q, \Sigma, \delta, q_0, F)$ accepting r. We write $L(q)$ for $\{w \mid \delta(q, w) \in F\}$. It is clear that $L(q)$ is regular for any $q \in Q$.
2. T_2 is $(Q \cup \{q_0'\}, \Sigma, \Sigma \cup \{\#\}, \Delta, q_0', \{q_0'\})$ where Δ has the following transition.

$$\begin{array}{rl} (q_0', \#, q_0, w) \in \Delta & \\ (q_0', a, q_0', a) \in \Delta & \text{if } a \neq \# \\ (q, a, q', \epsilon) \in \Delta & \text{if } \delta(q, a) = q' \\ (q, \epsilon, q_0', L(q) - \{\epsilon\}) \in \Delta & \text{if } q \in F \\ (q, \#, q, \epsilon) \in \Delta & \text{if } q \neq q_0' \end{array}$$

The construction is almost the same as T_2 in the previous section. The only difference is the look-ahead in the ϵ-transition from a final state of A_2 to q_0'. It allows the transducer to start searching a next matching substring only when no nonempty $w' \in L(q)$ is a prefix of the rest of input. That ensures that at that point the transducer is at the end of a longest matching substring.

Example 2. We obtain the following transducer for $a + ab$.



where A is any symbol and $r_1 = L(q_1) - \{\epsilon\} = \{b\}$ and $r_2 = L(q_2) - \{\epsilon\} = \emptyset$.

By applying the marker in the previous section and this transducer, we can precisely model the POSIX compatible regular expression replace function.

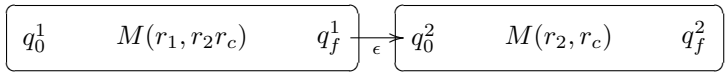
5.3 Perl-Compatible Regular Expressions

Perl-compatible regular expressions require us to insert look-ahead at not only final states but all the states that have choices. In order to achieve this, we extend the standard construction of an ϵ-NFA from a regular expression.

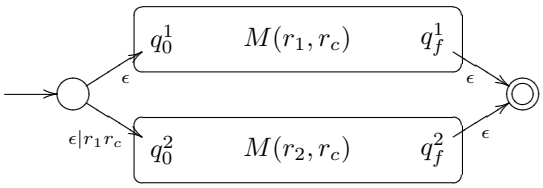
We define ϵ-NFA with regular look-ahead $M(r, r_c)$ modeling the disambiguation strategy of Perl for a regular expression r and a continuation regular expression r_c. The continuation regular expression r_c is required to annotate ϵ-transition with regular look-ahead.

The cases for symbols, the empty string, and the empty set are standard. We show the construction for $r_1 r_2$, $r_1 + r_2$, and r^* below.

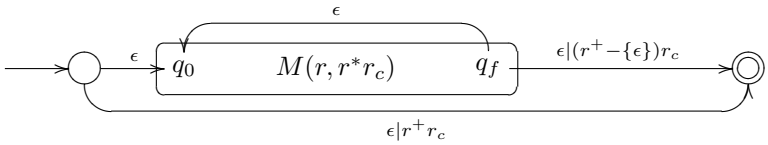
$M(r_1 r_2, r_c) =$



$M(r_1 + r_2, r_c) =$



$M(r^*, r_c) =$



In the regular expression $r_1 + r_2$, r_2 is taken only when the rest of the input is not in $r_1 r_c \Sigma^*$. Thus, the ϵ-transition for r_2 is annotated with $r_1 r_c$.

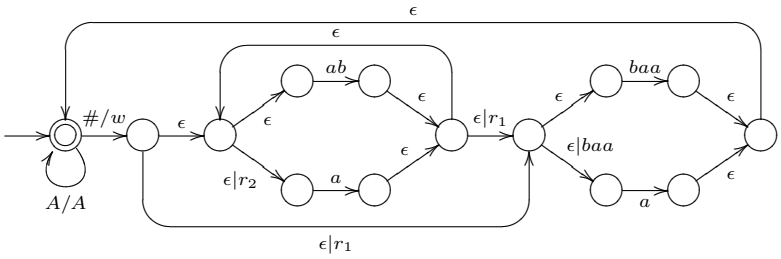
For r^*, the greedy strategy is enforced by the two ϵ-transition with look-ahead $r^+ r_c$ and $(r^+ - \{\epsilon\})r_c$. In order to treat problematic regular expressions such at $(a^*)^*$, one of the ϵ-transition must be annotated with $(r^+ - \{\epsilon\})r_c$ instead of $r^+ r_c$. The construction above models Perl-compatible regular expressions. For the greedy matching strategy of Frisch and Cardelli [FC04], the other ϵ-transition of the two must be annotated with $(r^+ - \{\epsilon\})r_c$ instead.

The ϵ-NFA with look-ahead $M(r, \epsilon)$ constructed in this way can be easily converted into a transducer with look-ahead to perform the regular expression replace function for r.

Example 3. Let us consider the following regular expression replace.

$$\text{replace}^{\text{perl}}((ab + a)^*(baa + a), x)(abaa) = xa$$

The transducer for the function is constructed as follows.



where $r_1 = (ab + a)^+(baa + a)$ and $r_2 = ab(ab + a)^*(baa + a)$. In this diagram, we omit the transitions $(q, \#, q, \epsilon) \in \Delta$ for all q that is not the initial state. It is easy to check that by apply T_1 and this transducer we do not obtain x, but xb.

6 Decomposing Transducers with Regular Look-Ahead

In order to incorporate the transducers with regular look-ahead in the PHP string analyzer, we need to transform them into combinations of standard transducers. We adopt the transformation of Engelfrite [Eng77]. The transformation decomposes a transducer with look-ahead into two transducers: one preprocesses the input from the end and one simulates the original transducer by utilizing the information added by the other. Although the composition of top-down and bottom-up tree transducers cannot be represented by one transducer, the composition of the two transducers above results in one nondeterministic transducer for strings. Thus, it is also possible to implement a transducer with look-ahead with a standard nondeterministic transducer without look-ahead.

Let T be a transducer with look-ahead and $\mathcal{L} = \{L_1, \ldots, L_k\}$ be the set of all look-ahead sets in T. T is decomposed into a preprocessing transducer T_1 that reads the input from the end and T_2 that simulates T with the annotation added by T_1.

The output alphabet of T_1 is $\Sigma'_i = \Sigma \cup \{0,1\}^k$ and produces $I_0 a_1 I_1 a_2 I_2 \ldots I_{n-1} a_n I_n$ for $a_1 a_2 \ldots a_n$ such that the following holds.

$$\begin{array}{ll} \pi_i(I_j) = 1 & \text{if} \quad a_{j+1} \ldots a_n \in L_i \Sigma^* \\ \pi_i(I_j) = 0 & \text{if} \quad a_{j+1} \ldots a_n \notin L_i \Sigma^* \end{array}$$

T_1 can be constructed in the same manner as the marker in Section 4.

With the annotation added by the T_1, T can be easily simulated. Let $T = (Q, \Sigma_i, \Sigma_o, \Delta, q_0, F)$. Then, $T_2 = (Q \cup (Q \times \{0,1\}^k), \Sigma_i \cup \{0,1\}^k, \Sigma_o, \Delta', q_0, F \times \{0,1\}^k)$ where Δ' is defined as follows.

$$\begin{array}{rl} ((q, I), a, q', w) \in \Delta' & \text{if} \quad (q, a, q', w) \in \Delta \\ (q, I, (q, I), \epsilon) \in \Delta' & \\ ((q, I), \epsilon, (q', I)) \in \Delta' & \text{if} \quad (q, \epsilon, q', L_j) \in \Delta \wedge \pi_j(I) = 0 \end{array}$$

We have implemented the transducers for both POSIX and Perl-compatible regular expression replace functions in this manner in the PHP string analyzer.

7 Conclusion

We have reviewed how we approximate the regular expression functions in the current version of the PHP string analyzer. The approximation is not precise since it ignores the disambiguation strategies of the regular expression functions. To make it more precise, we have adopted transducers with regular look-ahead and precisely modeled the disambiguation strategy of POSIX and Perl. We have implemented the improved transducers in the PHP string analyzer, but have not conducted detailed experiments yet.

The regular expression replace functions have a feature that matched substrings can be referred in a replacement string. Although we have not discussed this feature in this paper, we believe that we can also model this feature precisely if you restrict the use of matched substrings so that they are used in the order that they appear.

References

[AB+05] Mehdi Achour, Friedhelm Betz, et al. *PHP Manual*, 2005. http://www.php.net/docs.php.

[Ber79] Jean Berstel. *Transductions and Context-Free Languages*. Teubner Studienbucher, 1979.

[CMS03] Aske Simon Christensen, Andres Møller, and Michael I. Schwartzbach. Precise analysis of string expressions. In *Proceedings of the Static Analysis Symposium (SAS)*, volume 2694 of *LNCS*, pages 1–18, 2003.

[Eng77] Joost Engelfrite. Top-down tree transducers with regular look-ahead. *Mathematical Systems Theory*, 10(1):289–303, 1977.

[FC04] Alain Frisch and Luca Cardelli. Greedy regular expression matching. In *Proceedings of the 31st International Colloquium on Automata, Languages and Programming*, volume 3142 of *LNCS*, pages 618–629, 2004.

[Fri06] Jeffrey E. F. Friedl. *Mastering Regular Exressions (3rd Edition)*. O'Reilly, 2006.

[HVP05] Haruo Hosoya, Jérôme Vouillon, and Benjamin Pierce. Regular expression types for XML. *ACM Transactions on Programming Languages and Systems*, 27(1):46–90, 2005.

[Kea91] Steven M. Kearns. Extending regular expression with context operators and parse extraction. *Software – Practice and Experience*, 21(8):787–804, 1991.

[Min05] Yasuhiko Minamide. Static approximation of dynamically generated Web pages. In *Proceedings of the 14th International World Wide Web Conference*, pages 432–441. ACM Press, 2005.

[MR00] David Melski and Thomas Reps. Interconvertibility of a class of set constraints and context-free language reachability. *Theoretical Computer Science*, 248(1–2):29–98, 2000.

[MS96] Mehryar Mohri and Richard Sproat. An efficient compiler for weighted rewrite rules. In *34th Meeting of the Association for Computational Linguistics (ACL '96), Proceedings of the Conference*, 1996.

[MT06] Yasuhiko Minamide and Akihiko Tozawa. XML validation for context-free grammars. In *Proc. of The Fourth ASIAN Symposium on Programming Languages and Systems*, volume 4279 of *LNCS*, pages 357–373, 2006.

[NM08] Takuya Nishiyama and Yasuhiko Minamide. A translation from the HTML DTD into a regular hedge grammar. In *Proceedings of the 13th International Conference on Implementation and Application of Automata (to appear)*, volume 5148 of *LNCS*, 2008.

[Rep00] Thomas Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11–12):701–726, 2000.

[TSY02] Naoshi Tabuchi, Eijiro Sumii, and Akinori Yonezawa. Regular expression types for strings in a text processing language. In *Proceedings of International Workshop on Types in Programming*, ENTCS 75, 2002.

[Van06] Stijn Vansummeren. Type inference for unique pattern matching. *ACM Transactions on Programming Languages and Systems*, 28(3):389–428, 2006.

[WS07] Gary Wassermann and Zhendong Su. Sound and precise analysis of web applications for injection vulnerabilities. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 32 – 41, 2007.

A Computer Verified Theory of Compact Sets*

Russell O'Connor**

Institute for Computing and Information Science
Faculty of Science
Radboud University Nijmegen

Abstract. Compact sets in constructive mathematics capture our intuition of what computable subsets of the plane (or any other complete metric space) ought to be. A good representation of compact sets provides an efficient means of creating and displaying images with a computer. In this paper, I build upon existing work about complete metric spaces to define compact sets as the completion of the space of finite sets under the Hausdorff metric. This definition allowed me to quickly develop a computer verified theory of compact sets. I applied this theory to compute provably correct plots of uniformly continuous functions.

1 Introduction

How should we define what computable subsets of the plane are? Sir Roger Penrose ponders this question at one point in his book "The Emperor's New Mind" [9]. Requiring that subsets be decidable is too strict; determining if a point lies on the boundary of a set is undecidable in general. Penrose gives the unit disc, $\{(x, y)|x^2 + y^2 \leq 1\}$, and the epigraph of the exponential function, $\{(x, y)|\exp(x) \leq y\}$, as examples of sets that intuitively ought to be considered computable [2]. Restricting one's attention to pairs of rational or algebraic numbers may work well for the unit disc, but the boundary of the epigraph of the exponential function contains only one algebraic point. A better definition is needed.

To characterize computable sets, we draw an analogy with real numbers. The computable real numbers are real numbers that can be effectively approximated to arbitrary precision. The approximations are usually rational numbers or dyadic rational numbers. We can define computable sets in a similar way.

We need a dense subset of sets that have finitary representations. In the case of the plane, the simplest candidate is the finite subsets of $\mathbb{Q}^2$. Again, $\mathbb{Q}$ could be replaced with the dyadic rationals. How do we measure the accuracy of an approximation? Distances between subsets can be defined by the Hausdorff metric (section 3).

To construct the real numbers, we complete the rational numbers. By reasoning constructively (section 2), the real numbers generated are always computable.

* This document has been produced using TEXMACS(see `http://www.texmacs.org`)

** `r.oconnor@cs.ru.nl`

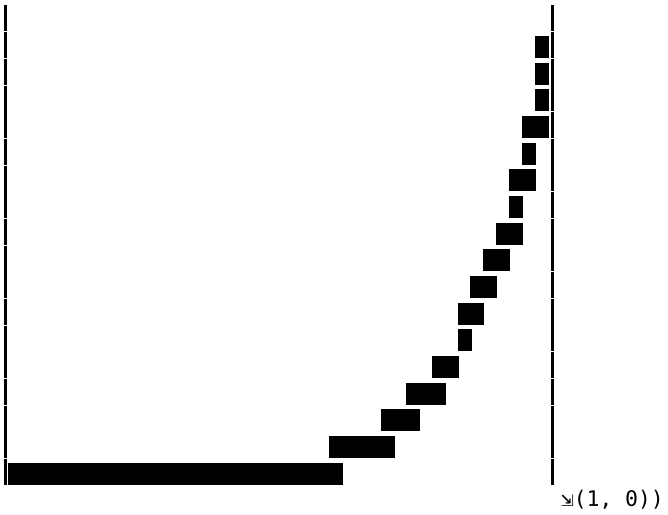
Completing the finite subsets of $\mathbb{Q}^2$ with the Hausdorff metric yields the compact sets (section 5). By reasoning constructively, the generated compact sets are always computable!

The unit disc is constructively compact; it can be effectively approximated with finite sets. When a computer attempts to display the unit disc, only a finite set of the pixels can be shown. So instead of displaying an ideal disc, the computer displays a finite set that approximates the disc. This is the key criterion that Penrose's examples enjoy. They can be approximated to arbitrary precision and displayed on a raster.

Technically the epigraph of the exponential function is not compact; however, it is locally compact. One may wish to consider constructive locally compact sets to be computable. This would mean that any finite region of a computable set has effective approximations of arbitrary precision.

This definition of constructively compact sets has been formalized in the Coq proof assistant [11]. Approximations of compact sets can be rasterized and displayed inside Coq (section 6). For example, figure 1 shows a theorem in Coq certifying that a plot is close the exponential function. The plot itself is computed from the definition of the graph of the exponential function.

```
ball (m:=Compact stableQ2) (324 # 2592)
  graphCR (boundBelow 0 ∘ boundAbove 3 ∘ exp_bound_uc 1) [- (6) .. 1]
  (Cunit
     (- (6), 3)↖
```



```
                                                    ↘(1, 0))
```

Fig. 1. A theorem in Coq stating that a plot on a 42 by 18 raster is close to the graph of the exponential function on $[-6, 1]$.

The standard definition of computable sets used in computable analysis says that a set is computable if the distance to the set is a computable real-valued

function. This definition is equivalent to our definition using computable approximations (although, this has not been verified in Coq). However, I believe defining computable sets by effective approximations of finite sets more accurately matches our intuition about sets that can be drawn by a computer.

2 Constructive Mathematics

Usually constructive logic is presented as a restriction of classical logic where proof by contradiction and the law of the excluded middle are not allowed. While this is a valid point of view, constructive logic can instead be presented as an extension of classical logic.

Consider formulas constructed from universal quantification ($\forall$), implication ($\Rightarrow$), conjunction ($\wedge$), true ($\top$), false ($\bot$), and equality for natural numbers ($=_{\mathbb{N}}$). Define negation by $\neg\varphi := \varphi \Rightarrow \bot$. One can (constructively) prove $\neg\neg\varphi \Rightarrow \varphi$ holds for any formula φ generated from this set of connectives by induction on the structure of φ because the atomic formulas—which in this case are equalities on $\mathbb{N}$—are decidable. Thus, one can deduce classical results with constructive proofs for formulas generated from this restricted set of connectives.

This set of connectives is not really restrictive because it can be used to define the other connectives. One can define the classical disjunction ($\tilde{\vee}$) by $\varphi\tilde{\vee}\psi := \neg(\neg\varphi \wedge \neg\psi)$. Similarly, one can define the classical existential quantifier ($\tilde{\exists}$) by $\tilde{\exists}x.\varphi(x) := \neg\forall x.\neg\varphi(x)$. With this full set of connectives, one can produce classical mathematics. The law of the excluded middle ($\varphi\tilde{\vee}\neg\varphi$) has a constructive proof when the classical disjunction is used.

Given this presentation of classical logic, we can extend the logic by adding two new connectives, the constructive disjunction ($\vee$) and the constructive existential ($\exists$). These new connectives come equipped with their constructive rules of inference (given by natural deduction) [12]. These constructive connectives are slightly stronger than their classical counterparts. Constructive excluded middle ($\varphi \vee \neg\varphi$) cannot be deduced in general, and our inductive argument that $\neg\neg\varphi \Rightarrow \varphi$ holds no longer goes through if φ uses these constructive connectives.

We wish to use constructive reasoning because constructive proofs have a computational interpretation. A constructive proof of $\varphi \vee \psi$ tells which of the two disjuncts hold. A proof of $\exists n : \mathbb{N}.\varphi(n)$ gives an explicit value for n that makes $\varphi(n)$ hold. Most importantly, we have a functional interpretation of $\Rightarrow$ and $\forall$. A proof of $\forall n : \mathbb{N}.\exists m : \mathbb{N}.\varphi(n, m)$ is interpreted as a function with an argument n that returns an m paired with a proof of $\varphi(n, m)$.

The classical fragment also admits this functional interpretation, but formulas in the classical fragment typically end in $\ldots \Rightarrow \bot$. These functions take their arguments and return a proof of false. Of course, there is no proof of false, so it must be the case that the arguments cannot simultaneously be satisfied. Therefore, these functions can never be executed. In this sense, only trivial functions are created by proofs of classical formulas. This is why constructive mathematics aims to strengthen classical results. We wish to create proofs with non-trivial functional interpretations.

From now on, I will leave out the word "constructive" from phrases like "constructive disjunction" and "constructive existential" and simply write "disjunction" and "existential". This follows the standard practice in constructive mathematics of using names from classical mathematics to refer to some stronger constructive notion. I will explicitly use the word "classical" when I wish to refer to classical concepts.

2.1 Dependently Typed Functional Programming

This functional interpretation of constructive deductions is given by the Curry-Howard isomorphism [12]. This isomorphism associates formulas with dependent types, and proofs of formulas with functional programs of the associated dependent types. For example, the identity function $\lambda x : A.x$ of type $A \Rightarrow A$ represents a proof of the tautology $A \Rightarrow A$. Table 1 lists the association between logical connectives and type constructors.

Logical Connective	**Type Constructor**
implication: $\Rightarrow$	function type: $\Rightarrow$
conjunction: $\wedge$	product type: $\times$
disjunction: $\vee$	disjoint union type: $+$
true: $\top$	unit type: $()$
false: $\bot$	void type: $\emptyset$
for all: $\forall x.\varphi(x)$	dependent function type: $\Pi x.\varphi(x)$
exists: $\exists x.\varphi(x)$	dependent pair type: $\Sigma x.\varphi(x)$

Table 1. The association between formulas and types given by the Curry-Howard isomorphism

In dependent type theory, functions from values to types are allowed. Using types parametrized by values, one can create dependent pair types, $\Sigma x : A.\varphi(x)$, and dependent function types, $\Pi x : A.\varphi(x)$. A dependent pair consists of a value x of type A and an value of type $\varphi(x)$. The type of the second value depends on the first value, x. A dependent function is a function from the type A to the type $\varphi(x)$. The type of the result depends on the value of the input.

The association between logical connectives and types can be carried over to constructive mathematics. We associate mathematical structures, such as the natural numbers, with inductive types in functional programming languages. We associate atomic formulas with functions returning types. For example, we can define equality on the natural numbers, $x =_{\mathbb{N}} y$, as a recursive function:

$$\begin{aligned} 0 =_{\mathbb{N}} 0 &:= \top \\ Sx =_{\mathbb{N}} 0 &:= \bot \\ 0 =_{\mathbb{N}} Sy &:= \bot \\ Sx =_{\mathbb{N}} Sy &:= x =_{\mathbb{N}} y \end{aligned}$$

One catch is that general recursion is not allowed when creating functions. The problem is that general recursion allows one to create a fixpoint operator $\text{fix} : (\varphi \Rightarrow \varphi) \Rightarrow \varphi$ that corresponds to a proof of a logical inconsistency. To prevent this, we allow only well-founded recursion over an argument with an inductive type. Because well-founded recursion ensures that functions always terminate, the language is not Turing complete. However, one can still express fast growing functions like the Ackermann function without difficulty [12].

Because proofs and programs are written in the same language, we can freely mix the two. For example, in my previous work [7], I represent the real numbers by the type

$$\exists f : \mathbb{Q}^+ \Rightarrow \mathbb{Q}. \forall \varepsilon_1 \varepsilon_2. |f(\varepsilon_1) - f(\varepsilon_2)| \leq \varepsilon_1 + \varepsilon_2. \tag{1}$$

Values of this type are pairs of a function $f : \mathbb{Q}^+ \Rightarrow \mathbb{Q}$ and a proof of $\forall \varepsilon_1 \varepsilon_2. |f(\varepsilon_1) - f(\varepsilon_2)| \leq \varepsilon_1 + \varepsilon_2$. The idea is that a real number is represented by a function f that maps any requested precision $\varepsilon : \mathbb{Q}^+$ to a rational approximation of the real number. Not every function of type $\mathbb{Q}^+ \Rightarrow \mathbb{Q}$ represents a real number. Only those functions that have coherent approximations should be allowed. The proof object paired with f witnesses the fact that f has coherent approximations. This is one example of how mixing functions and formulas allows one to create precise datatypes.

2.2 Notation

I will use the functional style of defining multivariate functions with Curried types. A binary function will have type $X \Rightarrow Y \Rightarrow Z$ instead of $X \wedge Y \Rightarrow Z$ ($\Rightarrow$ is taken to be right associative). To ease readability, I will still write binary function application as $f(x, y)$, even though it should really be $f(x)(y)$.

Anonymous functions are written using lambda expressions. A function on natural numbers that doubles its input is written $\lambda x : \mathbb{N}.2x$. The type of the parameter will be omitted when it is clear from context what it should be.

The type of propositions is $\star$. Predicates are represented by functions to $\star$. These predicates are often used where power sets are used in classical mathematics. The type $X \Rightarrow \star$ can be seen as the power set of X. I will often write $x \in \mathcal{A}$ in place of $\mathcal{A}(x)$ when $\mathcal{A} : X \Rightarrow \star$ and $x : X$.

The notation $x \in l$ is also used when l is a finite enumeration (section 4). Also $x \in \mathcal{S}$ will be used when $\mathcal{S}$ is a compact set (section 5). The types will make it clear what the interpretation of $\in$ should be.

I will use shorthand to combine membership with quantifiers. I will write $\forall x \in \mathcal{A}. \varphi(x)$ for $\forall x. x \in \mathcal{A} \Rightarrow \varphi(x)$, and $\exists x \in \mathcal{A}. \varphi(x)$ will mean $\exists x. x \in \mathcal{A} \wedge \varphi(x)$.

Quotient types are not used in this theory. In place of quotients, setoids are used. A **setoid** is a dependent record containing a type X (its carrier), a relation $\asymp : X \Rightarrow X \Rightarrow \star$, and a proof that $\asymp$ is an equivalence relation. When we define a function on setoid, we usually prove it is **respectful**, meaning it respects the setoid equivalence relations on its domain and codomain. Respectful functions will also be called **morphisms**.

I will often write $f(x)$ when f is a record (or existential) with a function as its carrier (or witness) and leave implicit the projection of f into a function.

3 Metric Spaces

Traditionally, a metric space is defined as a set X with a metric function $d : X \times X \Rightarrow \mathbb{R}^{0+}$ satisfying certain axioms. The usual constructive formulation requires d be a computable function. In my previous work [7], I have found it useful to take a more relaxed definition for a metric space that does not require the metric be a function. Instead, I represent the metric via a respectful ball relation $B : \mathbb{Q}^+ \Rightarrow X \Rightarrow X \Rightarrow \star$ satisfying five axioms:

1. $\forall x\varepsilon.B_\varepsilon(x,x)$
2. $\forall xy\varepsilon.B_\varepsilon(x,y) \Rightarrow B_\varepsilon(y,x)$
3. $\forall xyz\varepsilon_1\varepsilon_2.B_{\varepsilon_1}(x,y) \Rightarrow B_{\varepsilon_2}(y,z) \Rightarrow B_{\varepsilon_1+\varepsilon_2}(x,z)$
4. $\forall xy\varepsilon.(\forall\delta.\varepsilon < \delta \Rightarrow B_\delta(x,y)) \Rightarrow B_\varepsilon(x,y)$
5. $\forall xy.(\forall\varepsilon.B_\varepsilon(x,y)) \Rightarrow x \asymp y$

The ball relation $B_\varepsilon(x,y)$ expresses that the points x and y are within ε of each other. I call this a ball relationship because the partially applied relation $B_\varepsilon(x) : X \Rightarrow \star$ is a predicate that represents the ball of radius ε around the point x. The first two axioms are reflexivity and symmetry of the ball relationship. The third axiom is a version of the triangle inequality.

The fourth axiom states that the balls are closed balls. Closed balls are used because being closed is usually a classical formula. This means they can be ignored during computation because they have no computational content [4]. We want to minimize the amount of computation needed to get our constructive results.

The fifth axiom states the identity of indiscernibles. This means that if two points are arbitrarily close together then they are equivalent. The reverse implication follows from the reflexivity axiom and the fact that B is respectful. In some instances, axiom 5 can be considered as the definition of $\asymp$ on X.

For example, $\mathbb{Q}$ can be equipped with the usual metric by defining the ball relation as

$$B^{\mathbb{Q}}_\varepsilon(x,y) := |x - y| \leq \varepsilon.$$

This definition satisfies all the required axioms.

3.1 Uniform Continuity

We are interested in the category of metric spaces with uniformly continuous functions between them. A function $f : X \Rightarrow Y$ between two metric spaces is **uniformly continuous with modulus** $\mu_f : \mathbb{Q}^+ \Rightarrow \mathbb{Q}^+$ if

$$\forall x_1x_2\varepsilon.B^X_{\mu_f(\varepsilon)}(x_1,x_2) \Rightarrow B^Y_\varepsilon(f(x_1),f(x_2)).$$

We call a function **uniformly continuous** if it is uniformly continuous with some modulus. We use notation $X \to Y$ with a single bar arrow to denote the type of uniformly continuous functions from X to Y. This record type consists of three parts, a function f of type $X \Rightarrow Y$, a modulus of continuity, and a proof that f is uniformly continuous with the given modulus. Again, we will leave the projection to the function type implicit and allow us to write $f(x)$ when $f : X \to Y$ and $x : X$.

3.2 Classification of Metric Spaces

There is a hierarchy of classes that metrics can belong to. The strongest class of metrics are the **decidable metrics** where

$$\forall xy\varepsilon.B^X_\varepsilon(x,y) \vee \neg B^X_\varepsilon(x,y).$$

The constructive disjunction here implies there is an algorithm for computing whether two points are within ε of each other or not. The metric on $\mathbb{Q}$ has this property; however, the metric on $\mathbb{R}$ does not because of the lack of a decidable equality.

The next strongest class of metrics is what I call **located metrics**. These metrics have the property

$$\forall xy\varepsilon\delta.\varepsilon < \delta \Rightarrow B^X_\delta(x,y) \vee \neg B^X_\varepsilon(x,y).$$

This is similar to being decidable, but there is a little extra wiggle room. If x and y are between ε and δ far apart, then the algorithm has the option of either return a proof of $B^X_\delta(x,y)$ or $\neg B^X_\varepsilon(x,y)$. This extra flexibility allows $\mathbb{R}$ to be a located metric. Every decidable metric is also a located metric. Some metrics are not located. The standard sup-metric on functions between metric spaces may not be located.

The weakest class of metrics we will discuss are the **stable metrics**. A metric is stable when

$$\forall xy\varepsilon.\neg\neg B^X_\varepsilon(x,y) \Rightarrow B^X_\varepsilon(x,y).$$

Every located metric is stable. Although we will discuss the possibility of non-stable metrics in section 7, it appears that metric spaces used in practice are stable. This work relies crucially on stability at one point, so we will be assuming that metric spaces are stable throughout this paper.

3.3 Complete Metrics

Given a metric space X, one can create a new metric space called the **completion of** X, or simply $\mathfrak{C}(X)$. The type $\mathfrak{C}(X)$ is defined to be

$$\exists f : \mathbb{Q}^+ \Rightarrow X.\forall\varepsilon_1\varepsilon_2.B^X_{\varepsilon_1+\varepsilon_2}(f(\varepsilon_1), f(\varepsilon_2))$$

with the ball relation defined to be

$$B^{\mathfrak{C}(X)}_\varepsilon(x,y) := \forall\delta_1\delta_2.B^X_{\delta_1+\varepsilon+\delta_2}(x(\delta_1), y(\delta_2)).$$

The definition of $\mathfrak{C}(X)$ may look familiar. It is a generalization of the type that I gave for real numbers in equation 1. In fact, in my actual implementation the real numbers are defined to be $\mathfrak{C}(\mathbb{Q})$.

A complete metric comes equipped with an injection from the original space $\mathrm{unit} : X \to \mathfrak{C}(X)$ and a function $\mathrm{bind} : (X \to \mathfrak{C}(Y)) \Rightarrow (\mathfrak{C}(X) \to \mathfrak{C}(Y))$ that lifts uniformly continuous functions with domain X to uniformly continuous

function with domain $\mathfrak{C}(X)$. One of the most common way of creating functions that operate on complete metric spaces is by using bind. One first defines a function on X, which is easy to work with when X is a discrete space. Then one proves the function is uniformly continuous. After that, bind does the rest of the work.

A second, similar way of creating functions with complete domains is by using $\mathrm{map} : (X \to Y) \Rightarrow (\mathfrak{C}(X) \to \mathfrak{C}(Y))$. The function map can be defined by $\mathrm{map}(f) := \mathrm{bind}(\mathrm{unit} \circ f)$, but in my implementation, map is more fundamental than bind [7].

I will use the following notation:

$$\hat{x} := \mathrm{unit}(x)$$
$$\check{f} := \mathrm{bind}(f)$$
$$\bar{f} := \mathrm{map}(f)$$

The completion operation, $\mathfrak{C}$, and the functions unit and bind together form a standard construction called a **monad** [6]. Monads have been used in functional programs to capture many different computational notions including exceptions, mutable state, and input/output [13]. We will see another example of a monad in section 4.

3.4 Product Metrics

Given two metric spaces X and Y, their Cartesian product $X \times Y$ forms a metric space with the standard sup-metric:

$$B_\varepsilon^{X \times Y}((x_1, y_1), (x_2, y_2)) := B_\varepsilon^X(x_1, x_2) \wedge B_\varepsilon^Y(y_1, y_2)$$

The product metric interacts nicely with the completion operation. There is an isomorphism between $\mathfrak{C}(X \times Y)$ and $\mathfrak{C}(X) \times \mathfrak{C}(Y)$. One direction I call **couple**. The other direction is defined by lifting the projection functions:

$$\mathrm{couple} : \mathfrak{C}(X) \times \mathfrak{C}(Y) \to \mathfrak{C}(X \times Y)$$
$$\overline{\pi_1} : \mathfrak{C}(X \times Y) \to \mathfrak{C}(X)$$
$$\overline{\pi_2} : \mathfrak{C}(X \times Y) \to \mathfrak{C}(Y)$$

We denote $\mathrm{couple}(x, y)$ by $\langle x, y \rangle$. The following lemmas prove that these functions form an isomorphism.

$$\langle \overline{\pi_1}(z), \overline{\pi_2}(z) \rangle \asymp z$$
$$(\overline{\pi_1}\langle x, y \rangle, \overline{\pi_2}\langle x, y \rangle) \asymp (x, y)$$

3.5 Hausdorff Metrics

Given a metric space X, we can try to put a metric on predicates (subsets) of X. We start by defining the Hausdorff hemimetric. A hemimetric is a metric

without the symmetry and identity of indiscernibles requirement. We define the hemimetric relation over $X \Rightarrow \star$ as

$$H_\varepsilon^{X \Rightarrow \star}(\mathcal{A}, \mathcal{B}) := \forall x \in \mathcal{A}.\tilde{\exists} y \in \mathcal{B}.B_\varepsilon(x, y).$$

Notice the use of the classical existential in this definition. In general, we do not need to know which point in $\mathcal{B}$ is close to a given point in $\mathcal{A}$; it is sufficient to know one exists without knowing which one. Furthermore, there are cases when we cannot know which point in $\mathcal{B}$ is close to a given point in $\mathcal{A}$.

This relation is reflexive and satisfies the triangle inequality. It is not symmetric. We define a symmetric relation by

$$B_\varepsilon^{X \Rightarrow \star}(\mathcal{A}, \mathcal{B}) := H_\varepsilon^{X \Rightarrow \star}(\mathcal{A}, \mathcal{B}) \wedge H_\varepsilon^{X \Rightarrow \star}(\mathcal{B}, \mathcal{A}).$$

This relationship is reflexive, symmetric, and satisfies the triangle inequality. Notice that if $\mathcal{B} \subseteq \mathcal{A}$ then $H_\varepsilon(\mathcal{A}, \mathcal{B})$ holds for all ε. The hemimetric captures the subset relationship. If $\mathcal{B} \subseteq \mathcal{A}$ and $\mathcal{A} \subseteq \mathcal{B}$ (i.e. $\mathcal{A} \asymp \mathcal{B}$), then $B_\varepsilon(\mathcal{A}, \mathcal{B})$ holds for all ε. However, axiom 5 for metric spaces requires the reverse implication; if $B_\varepsilon(\mathcal{A}, \mathcal{B})$ holds for all ε, then we want $\mathcal{A} \asymp \mathcal{B}$. Unfortunately, this does not hold in general. Neither does the closedness property required by axiom 4 hold. To make a true metric space, we need to focus on a subclass of predicates that have more structure.

4 Finite Enumerations

A finite enumeration of points from X is represented by a list. A point x is in a finite enumeration if there classically exists a point in the list that is equivalent to x. We are not required to know which point in the list is equivalent to x; we only need to know that there is one. An equivalent definition can be given by well-founded recursion on lists:

$$x \in \mathrm{nil} := \bot$$
$$x \in \mathrm{cons}\, y l := x \asymp y \tilde{\vee} x \in l$$

Two finite enumerations are considered equivalent if they have exactly the same members:

$$l_1 \asymp l_2 := \forall x.x \in l_1 \Leftrightarrow x \in l_2$$

If X is a metric space, then the space of finite enumerations over X, $\mathfrak{F}(X)$, is also a metric space. The Hausdorff metric with the membership predicate defines the ball relation:

$$B_\varepsilon^{\mathfrak{F}(X)}(l_1, l_2) := B_\varepsilon^{X \Rightarrow \star}(\lambda x.x \in l_1, \lambda y.y \in l_2)$$

This ball relation is both closed (axiom 4) and is compatible with our equivalence relation for finite enumerations (axiom 5), so this truly is a metric space.

Finite enumerations also form a monad (I have yet to verify this in Coq). The unit : $X \to \mathfrak{F}(X)$ function creates an enumeration with a single member. The bind : $(X \to \mathfrak{F}(Y)) \Rightarrow (\mathfrak{F}(X) \to \mathfrak{F}(Y))$ function takes an $f : X \to \mathfrak{F}(Y)$ and applies it to every element of an enumeration $l : \mathfrak{F}(X)$ and returns the union of the results.

4.1 Mixing Classical and Constructive Reasoning

Proving the ball relation for finite enumerations is closed makes essential use of classical reasoning. Given ε, suppose $B_\delta^{\mathfrak{F}(X)}(l_1, l_2)$ holds whenever $\varepsilon < \delta$. We need to show that $B_\varepsilon^{\mathfrak{F}(X)}(l_1, l_2)$ holds. By the definition of the metric, this requires proving (in part) $\forall x \in l_1.\tilde{\exists} y \in l_2.B_\varepsilon^X(x, y)$. From our assumptions, we know that $\forall x \in l_1.\tilde{\exists} y \in l_2.B_\delta^X(x, y)$ holds for every δ greater than ε. If we had used a constructive existential in the definition of the Hausdorff hemimetric, we would have a problem. Each different value δ could produce a *different* y witnessing $B_\delta^X(x, y)$. In order to use the closedness property from X to conclude $B_\varepsilon^X(x, y)$, we need a *single* y such that $B_\delta^X(x, y)$ holds for all δ greater than ε. Classically we would use the infinite pigeon hole principle to find a single y that occurs infinitely often in the stream of ys produced from $\delta \in \{\varepsilon + \frac{1}{n} | n : \mathbb{N}^+\}$. Such reasoning does not work constructively. Given an infinite stream of elements drawn from a finite enumeration, there is no algorithm that will determine which one occurs infinitely often.

Fortunately, because we used classical quantifiers in the definition of the Hausdorff metric, we can apply the the infinite pigeon hole principle to this problem. We classically know there is some y that occurs infinitely often when $\delta \in \{\varepsilon + \frac{1}{n} | n : \mathbb{N}^+\}$, even if we do not know which one. For such y, $B_\delta^X(x, y)$ holds for δ arbitrarily close to ε, and therefore $B_\delta^X(x, y)$ must hold for all δ greater than ε. By the closedness property for X, $B_\varepsilon^X(x, y)$ holds as required. The other half of the definition of $B_\varepsilon^{\mathfrak{F}(X)}(l_1, l_2)$ is handled similarly.

Recall that the classical fragment of constructive logic requires that proof by contradiction hold for atomic formulas in order to deduce the rule $\neg\neg\varphi \Rightarrow \varphi$. Because $B_\varepsilon^X(x, y)$ is a parameter, we do not know if it is constructed out of classical connectives. To use the classical reasoning needed to apply the pigeon hole principle, we assume that $\neg\neg B_\varepsilon^X(x, y) \Rightarrow B_\varepsilon^X(x, y)$ holds. This is the crucial point where stability of the metric for X is used.

5 Compact Sets

Completing the metric space of finite enumerations yields a metric space of compact sets:

$$\mathfrak{K}(X) := \mathfrak{C}(\mathfrak{F}(X))$$

The idea is that every compact set can be represented as a limit of finite enumerations that approximate it. In order for a compact set to be considered a set, we need to define a membership relation. The membership is not over X because compact sets are supposed to be complete and X may not be a complete space itself. Instead, membership is over $\mathfrak{C}(X)$, and it is defined for $x : \mathfrak{C}(X)$ and $\mathcal{S} : \mathfrak{K}(X)$ as

$$x \in \mathcal{S} := \forall \varepsilon_1 \varepsilon_2.\tilde{\exists} y \in \mathcal{S}(\varepsilon_2).B_{\varepsilon_1+\varepsilon_2}^X(x(\varepsilon_1), y).$$

A point is considered to be a member of a compact set $\mathcal{S}$ if it is arbitrarily close to being a member of all approximations of $\mathcal{S}$. Thus $\mathfrak{K}(X)$ represents the space of compact subsets of $\mathfrak{C}(X)$.

5.1 Correctness of Compact Sets

Bishop and Bridges define a compact set in a metric space X as a set that is complete and totally bounded [1]. In our framework, we say a predicate $\mathcal{A} : X \Rightarrow \star$ is complete if for every $x : \mathfrak{C}(X)$ made from approximations in $\mathcal{A}$, then x is in $\mathcal{A}$:

$$\forall x : \mathfrak{C}(X).(\forall \varepsilon.x(\varepsilon) \in \mathcal{A}) \Rightarrow \exists z \in \mathcal{A}.\hat{z} \asymp x$$

A set $\mathcal{B} : X \Rightarrow \star$ is totally bounded if there is an ε-net for every $\varepsilon : \mathbb{Q}^+$. An ε-net is a list of points l from $\mathcal{B}$ such that for every $x \in \mathcal{B}$ there (constructively) exists a point z that is constructively in l and $B_\varepsilon(x, z)$. Bishop and Bridges use the strong constructive definition of list membership that tells which member of the list the value is.

$$\forall \varepsilon : \mathbb{Q}^+.\exists l : \operatorname{list} X.(\forall x \in l.x \in \mathcal{B}) \wedge \forall x \in \mathcal{B}.\exists z \in l.B^Y_\varepsilon(x, z)$$

Does our definition of compact sets correspond with Bishop and Bridges's definition? The short answer is yes, but there is a small caveat. Our definition of metric space is more general than the one that Bishop and Bridges use. Bishop and Bridges require a distance function $d : X \Rightarrow X \Rightarrow \mathbb{R}$. Our more liberal definition of metric space does not have this requirement. I have verified that our definition of compact is the same as Bishop and Bridges's assuming that X is a located metric. If a metric space has a distance function, then it is a located metric. Thus our definition of compact corresponds to Bishop and Bridges's definition of compact for those metric spaces that correspond to Bishop and Bridges's definition of metric space.

It may seem impossible that our definition can be equivalent to Bishop and Bridges's definition when we sometimes use a classical existential quantifier while Bishop and Bridges use constructive quantifiers everywhere. How would one prove Bishop and Bridges version of $x \in \mathcal{S}$ from our version of $x \in \mathcal{S}$? The trick is to use the constructive disjunction from the definition of located metric. Roughly speaking, at some point we need to prove $\exists z \in l.B^Y_\varepsilon(x, z)$ from $\tilde{\exists} z \in l.B^Y_\varepsilon(x, z)$. This can be done by doing a search though the list l using the located metric property to decide for each element $z_0 \in l$ whether $B_{\varepsilon+\delta}(x, z_0)$ or $\neg B_\varepsilon(x, z_0)$ holds. The classical existence is sufficient to prove that this finite search will successfully find some z such that $B_{\varepsilon+\delta}(x, z)$ holds. The extra δ can be absorbed by other parts of the proof. The full proof of the isomorphism is too technical to be presented here. A detailed description can be found in my forthcoming PhD thesis or by examining the formal Coq proofs.

5.2 Distribution of $\mathfrak{F}$ over $\mathfrak{C}$

The composition of two monads $\mathfrak{A} \circ \mathfrak{B}$ forms a monad when there is a distribution function $\operatorname{dist} : \mathfrak{B}(\mathfrak{A}(X)) \to \mathfrak{A}(\mathfrak{B}(X))$ satisfying certain laws [5]. For compact sets, $\mathfrak{K}(X) := (\mathfrak{C} \circ \mathfrak{F})(X)$, the distribution function $\operatorname{dist} : \mathfrak{F}(\mathfrak{C}(X)) \to \mathfrak{C}(\mathfrak{F}(X))$ is defined by

$$\operatorname{dist}(l)(\varepsilon) := \operatorname{map}(\lambda x.x(\varepsilon))l.$$

This function interprets a finite enumeration of points from $\mathfrak{C}(X)$ as a compact set. Thus $\mathfrak{K}$ is also a monad (I have yet to verified this in Coq).

5.3 Compact Image

We define the compact image of a compact set $\mathcal{S} : \mathfrak{K}(X)$ under a uniformly continuous function $\check{f} : \mathfrak{C}(X) \to \mathfrak{C}(Y)$ by first noting that applying f to every point in a finite enumeration is a uniformly continuous function, $\mathrm{map}(f) : \mathfrak{F}(X) \to \mathfrak{F}(\mathfrak{C}(Y))$. Composing this with dist yields a uniformly continuous function from finite enumeration $\mathfrak{F}(X)$ to compact sets $\mathfrak{K}(Y)$. Using bind, this function can be lifted to operate on $\mathfrak{K}(X)$. The result is the **compact image** function:

$$f \upharpoonright \mathcal{S} := \mathrm{bind}(\mathrm{dist} \circ \mathrm{map}(f))(\mathcal{S})$$

Although Bishop and Bridges would agree that the result of this function is compact, they would not say that it is the image of $\mathcal{S}$ because one cannot constructively prove

$$y \in f \upharpoonright \mathcal{S} \Rightarrow \exists x \in \mathcal{S}.\check{f}(x) \asymp y.$$

However, I believe one can prove (but I have not verified this yet) the classical statement

$$y \in f \upharpoonright \mathcal{S} \Rightarrow \tilde{\exists} x \in \mathcal{S}.\check{f}(x) \asymp y.$$

When $\check{f}$ is injective, as it will be for our graphing example in section 6.1, the constructive existential statement holds.

6 Plotting Functions

There are many examples of constructively compact sets. This section illustrates one application of compacts sets, plotting functions.

6.1 Graphing Functions

Given a uniformly continuous function $\check{f} : \mathfrak{C}(X) \to \mathfrak{C}(Y)$ and a compact set $\mathcal{D} : \mathfrak{K}(X)$, the graph of the function over $\mathcal{D}$ is the set of points $\{(x, \check{f}(x)) | x \in \mathcal{D}\}$. This graph can be constructed as a compact set $\mathcal{G} : \mathfrak{K}(X \times Y)$. A single point is graphed by the function $g(x) := \langle \hat{x}, f(x) \rangle$. This function is uniformly continuous, $g : X \to \mathfrak{C}(X \times Y)$. The graph $\mathcal{G}$ is defined as the compact image of $\mathcal{D}$ under g.

$$\mathcal{G} := g \upharpoonright \mathcal{D}$$

6.2 Rasterizing Compact Sets

Given a compact set in the plane $\mathcal{S} : \mathfrak{K}(\mathbb{Q} \times \mathbb{Q})$, we can draw an image of it, or rather we can draw an approximation of it. This process consists of two steps. The first step is to compute an ε-approximation $l := \mathcal{S}(\varepsilon)$. The finite enumeration l is a list of rational coordinates. The next step is to move these points around so that all the points lie on a raster. A raster is simply a two dimensional matrix of Booleans. Given coordinates for the top-left and bottom-right corners, a raster can be interpreted as a finite enumeration. Using advanced notation features in Coq, a raster can be displayed inside the proof assistant. Most importantly, when the constructed raster is interpreted, it is provably close to the original compact set.

6.3 Plotting the Exponential Function.

Given a uniformly continuous function $\check{f} : \mathbb{R} \to \mathbb{R}$ and an interval $[a, b]$, the graph of $\check{f}$ over this compact interval is a compact set. The graph is an ideal mathematical curve. This graph can then be plotted yielding a raster that when interpreted as a finite enumeration is provably close to the ideal mathematical curve.

Recall figure 1 from section 1. It is a theorem in Coq that states the (ideal mathematical) graph of the exponential function (which is uniformly continuous on $(-\infty, 1]$) restricted to the range $[0, 3]$ on the interval $[-6, 1]$ is within $\frac{324}{2592}$ (which is equivalent to $\frac{1}{8}$) of the finite set represented by raster shown with the top-left corner mapped to $(-6, 3)$ and the bottom-right corner mapped to $(1, 0)$. The raster is 42 by 18, so, by considering the domain and range of the graph, each pixel represents a $\frac{1}{6}$ by $\frac{1}{6}$ square. The error between the plot and the graph must always be greater than half a pixel. I chose an ε that produces a graph with an error of $\frac{3}{4}$ of a pixel. In this case $\frac{3}{4} \cdot \frac{1}{6} \asymp \frac{1}{8}$, which is the error given in the theorem.

There is one small objection to this image. Each block in the picture represents an infinitesimal mathematical point lying at the center of the block, but the block appears as a square the size of the pixel. This can be fixed by interpreting each block as a filled square instead of as a single point. This change would simply add an additional $\frac{1}{2}$ pixel to the error term. This has not been done yet in this early implementation.

7 Alternative Hausdorff Metric Definitions

There is another possible definition for the Hausdorff metric. One could define the Hausdorff hemimetric as

$$H'_\varepsilon(\mathcal{A}, \mathcal{B}) := \forall x \in \mathcal{A}. \forall \delta. \exists y \in \mathcal{B}. B^X_{\varepsilon+\delta}(x, y).$$

The extra flexibility given by the δ term also allows one to conclude that there is some $y \in \mathcal{B}$ that is within ε of x without telling us which one (again, it may

be the case that we cannot know which y is the one). Our original definition $H_\varepsilon(\mathcal{A}, \mathcal{B})$ is implied by $H'_\varepsilon(\mathcal{A}, \mathcal{B})$; however, the alternative definition yields more constructive information.

The two definitions are equivalent under mild assumptions. When X is a located metric, then $H'_\varepsilon(\mathcal{A}, \mathcal{B}) \Leftrightarrow H_\varepsilon(\mathcal{A}, \mathcal{B})$. This is very common case and allows us to recover the constructive information in the H' version from the H version.

The constructive existential in the definition of H' would make the resulting metric not provably stable. It is somewhat unclear which version is the right definition for the constructive Hausdorff metric. The key deciding factor for me was that I had declared the ball relation to be in the `Prop` universe. Coq has a `Prop`/`Set` distinction where values in the `Prop` universe are removed during program extraction [11]. To make program extraction sound, values outside the `Prop` universe cannot depend on information inside the `Prop` universe. This means that even if I used the H' definition in the Hausdorff metric, its information would not be allowed by Coq to construct values in `Set`. For this reason, I chose the H version with the classical quantifiers for the definition of the Hausdorff metric. Values with classical existential quantifier type have no information in them and naturally fit into the `Prop` universe.

8 Conclusion

This work shows that one can compute with and display constructively compact sets inside a proof assistant. We showed how to graph uniformly continuous functions and render the results. We have turned a proof assistant into a graphing calculator. Moreover, our plots come with proofs of (approximate) correctness.

Even though a classical quantifier in the Hausdorff metric is used, it does not interfere with the computation of raster images. This development shows that one can combine classical reasoning with constructive reasoning. The classical existential quantifier was key in allowing us to use the pigeon hole principle to prove the closedness property of the Hausdorff metric.

All of the theorems in this paper have been verified by Coq except where indicated otherwise. Those few theorems that have not been verified in Coq are not essential and have not been assumed in the rest of the work (the statements simply do not appear in the Coq formalization). This formalization will be part of the next version of the CoRN library [3], which will be released when Coq 8.2 is released.

Given my previous work about metric spaces and uniformly continuous functions [8], the work of defining compact sets and plotting functions took only one and a half months of additional work.

This work provides a foundation for future work. One can construct more compact sets such as fractals and geometric shapes. Proof assistants could be modified so that the high resolution display of a monitor could be used instead of the "ASCII art" notation that is used in this work.

9 Acknowledgments

I would like to thank my advisor, Bas Spitters, for suggesting the idea that compact sets can be defined as the completion of finite sets.

I would also like to thank Jasper Stein whose implementation of Sokoban in Coq [10] using ASCII art inspired the idea of using Coq's notation mechanism for displaying graphs inside Coq.

References

[1] Errett Bishop and Douglas Bridges. *Constructive Analysis*. Number 279 in Grundlehren der mathematischen Wissenschaften. Springer-Verlag, 1985.

[2] Vasco Brattka. The emperor's new recursiveness: The epigraph of the exponential function in two models of computability. In Masami Ito and Teruo Imaoka, editors, *Words, Languages & Combinatorics III*, pages 63–72, Singapore, 2003. World Scientific Publishing. ICWLC 2000, Kyoto, Japan, March 14–18, 2000.

[3] L. Cruz-Filipe, H. Geuvers, and F. Wiedijk. C-CoRN: the constructive Coq repository at Nijmegen. In A. Asperti, G. Bancerek, and A. Trybulec, editors, *Mathematical Knowledge Management, Third International Conference, MKM 2004*, volume 3119 of *LNCS*, pages 88–103. Springer–Verlag, 2004.

[4] L. Cruz-Filipe and B. Spitters. Program extraction from large proof developments. In D. Basin and B. Wolff, editors, *Theorem Proving in Higher Order Logics, 16th International Conference, TPHOLs 2003*, volume 2758 of *LNCS*, pages 205–220. Springer–Verlag, 2003.

[5] Mark P. Jones and Luck Duponcheel. Composing monads. Technical Report YALEU/DCS/RR-1004, Yale University, 1993.

[6] E. Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in computer science*, pages 14–23, Piscataway, NJ, USA, 1989. IEEE Press.

[7] Russell O'Connor. A monadic, functional implementation of real numbers. *Mathematical. Structures in Comp. Sci.*, 17(1):129–159, 2007.

[8] Russell O'Connor. Certified exact transcendental real number computation in Coq. In Otmane Ait-Mohamed, editor, *TPHOLs*, volume 5170 of *Lecture Notes in Computer Science*, pages 246–261. Springer, 2008.

[9] Roger Penrose. *The Emperor's New Mind: Concerning Computers, Minds, and The Laws of Physics*. Oxford, NY, 1989.

[10] Jasper Stein. Coqoban, September 2003. `http://coq.inria.fr/contribs/Coqoban.html`.

[11] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.0*, April 2004. `http://coq.inria.fr`.

[12] S. Thompson. *Type Theory and Functional Programming*. Addison Wesley, 1991.

[13] P. Wadler. Monads for functional programming. In *Proceedings of the Marktoberdorf Summer School on Program Design Calculi*, August 1992.

Verification of Functional Programs Containing Nested Recursion

Nikolaj Popov, Tudor Jebelean*

Research Institute for Symbolic Computation,
Johannes Kepler University, Linz, Austria
{popov,jebelean}@risc.uni-linz.ac.at

Abstract. We present an environment for proving partial correctness of recursive functional programs which contain nested recursive calls. As usual, correctness is transformed into a set of first-order predicate logic formulae—verification conditions. As a distinctive feature of our method, these formulae are not only sufficient, but also necessary for the correctness. We demonstrate our method on the McCarthy 91 function, which is considered a "challenge problem" for automated program verification.

1 Introduction

We develop a method for the generation of verification conditions for proving partial correctness of recursive functional programs which contain nested recursive calls. Our focus is on the generation of the conditions, and we do not treat here the problem of proving them. We assume that the specification and the program are provided and our task is to generate sound and complete verification conditions. (In fact, we believe that specifications should be developed before the program is written.)

Recursive programs are called nested when an argument to a recursive call contains another invocation of the main recursive program. For example:

$$f[x] = \textbf{ If } x = 0 \textbf{ then } 0 \textbf{ else } f[f[x-1]].$$

We approach the problem of program verification by studying the most frequent program schemata. We have studied so far Simple Recursive [6] and Fibonacci-like [15] schemata.

When deriving necessary (and also sufficient) conditions for program correctness, we actually prove at the meta-level that for any program of a certain class (defined by a certain schema) it suffices to check only the respective verification conditions. This is very important for the automation of the whole process, because the production of the verification conditions is not expensive from the computational point of view.

* The Theorema project is supported by FWF (Austrian National Science Foundation) – SFB project F1302. The program verification project in the frame of e-Austria Timişoara is supported by BMBWK (Austrian Ministry of Education, Science and Culture). Additional support comes from INTAS project Ref. Nr 05-1000008-8144.

In this paper we study, in particular, a class of recursive functional programs which contain nested recursive definitions and we extract the purely logical conditions which are sufficient for the program correctness.

The logical conditions are inferred using Scott induction [1, 11] in the fixpoint theory of programs and constitute two meta-theorems which are proven once for the whole class. The concrete verification conditions for each program are then provable without having to use the fixpoint theory.

In order to illustrate the method and the class of recursive functions which may contain nested recursive definitions, we presented here the McCarthy 91 function [13, 12], which is considered a "challenge problem" for automated program verification.

We consider the partial correctness problem expressed as follows: *given* the program which computes the function F in a domain D and given its specification by a precondition on the input $I_F[x]$ and a postcondition on the input and the output $O_F[x, y]$, *generate* the verification conditions VC_1, ... , VC_n which are sufficient for the program to satisfy the specification. The function F satisfies the specification, if: for any input x satisfying I_F, if F terminates on x, (we write $F[x] \downarrow$) then the condition $O_F[x, F[x]]$ holds. This is also called "partial correctness" of the program F:

$$(\forall x : I_F[x])\ (F[x] \downarrow \implies O_F[x, F[x]]). \tag{1}$$

A Verification Condition Generator (VCG) is a device—normally implemented by a program—which takes a program, actually its source code, and the specification, and produces verification conditions. These verification conditions do not contain any part of the program text, and are expressed in a different language, namely they are logical formulae.

Any VCG should come together with its *Soundness* statement, that is: for a given program F, defined on a domain D, with a specification I_F and O_F if the verification conditions $VC_1, \ldots, VC_n$ hold in the theory $Th[D]$ of the domain D, then the program F satisfies its specification I_F and O_F.

Moreover, we are also interested in the following question: What if some of the verification conditions do not hold? May we conclude that the program is not correct? In fact, the program may still be correct. However, if the VCG is complete, then one can be sure that the program is not correct. A VCG is complete, if whenever the program satisfies its specification, the produced verification conditions hold.

The notion of *Completeness* of a VCG is important for the following two reasons: theoretically, it is the dual of *Soundness* and practically, it helps debugging. Any counterexample for the failing verification condition would carry over to a counterexample for the program and the specification, and thus give a hint on "what is wrong". Indeed, most books about program verification present methods for verifying correct programs. However, in practical situations, it is the failure which occurs more often until the program and the specification are completely debugged.

This work is performed in the frame of the *Theorema* system [3], a mathematical computer assistant which aims at supporting all phases of mathematical

activity: construction and exploration of mathematical theories, definition of algorithms for problem solving, as well as experimentation and rigorous verification of them. *Theorema* provides both functional and imperative programming constructs. Moreover, the logical verification conditions which are produced by the methods presented here can be passed to the automatic provers of the system in order to be checked. The system includes a collection of general as well as specific provers for various interesting domains (e.g., integers, sets, reals, tuples, etc.). More details about *Theorema* are available at `www.theorema.org`.

2 Coherent Programs

In this section we state the general principles we use for writing coherent programs with the aim of building up a non-contradictory system of verified programs. Although, these principles are not our invention (similar ideas appear in [8]), we state them here because we want to emphasize and later formalize them. Similar ideas appear also in software engineering—they are called there *Design by Contract* or *Programming by Contract* [14].

We build our system such that it preserves the modularity principle, that is, each concrete implementation of a program may be replaced by another one at any time.

Building up correct programs: Firstly, we want to ensure that our system of coherent programs would contain only correct (verified) programs. This we achieve, by:

- start from basic (trustful) functions e.g. addition, multiplication, etc.;
- define each new function in terms of already known (defined previously) functions by giving its source text, the specification (input and output predicates) and prove their correctness with respect to the specification.

This simple inductively defined principle would guarantee that no wrong program may enter our system. The next property we want to ensure is the easy exchange (mobility) of our program implementations. This principle is usually referred as:

Modularity: Once we define the new function and prove its correctness, we "forbid" using any knowledge concerning the concrete function definition. The only knowledge we may use is the specification. This gives us the possibility of easy replacement of existing functions.

In order to achieve the modularity, we need to ensure that when defining a new program, all the calls made to the existing (already defined) programs obey the input restrictions of that programs—we call this: *Appropriate values for the function calls.*

We now define the class of coherent programs as those which obey the *appropriate values to the function calls* principle. The general definition comes in two parts: for functions defined by composition and for functions defined by *if-then-else*.

Definition 1. *Let F be obtained from H, G_1, ..., G_n by composition:*

$$F[x] = H[G_1[x], \ldots, G_n[x]]. \tag{2}$$

The program F with the specification (I_F and O_F) is coherent *with respect to its auxiliary functions H, G_i and their specifications (I_H and O_H, I_{G_i} and O_{G_i}), if and only if*

$$(\forall x : I_F[x]) \Longrightarrow I_{G_1}[x] \wedge \cdots \wedge I_{G_n}[x] \tag{3}$$

and

$$(\forall x : I_F[x])\ (\forall y_1 \ldots y_n)\ (O_{G_1}[x, y_1] \wedge \cdots \wedge O_{G_n}[x, y_n] \Longrightarrow I_H[y_1, \ldots, y_n]). \tag{4}$$

Definition 2. *Let F be obtained from H, G by if-then-else:*

$$F[x] = \ \textbf{If}\ Q[x]\ \textbf{then}\ H[x]\ \textbf{else}\ G[x]. \tag{5}$$

The program F with the specification (I_F and O_F) is coherent *with respect to its auxiliary functions H, G and their specifications (I_H and O_H, I_G and O_G) if and only if*

$$(\forall x : I_F[x])\ (Q[x]\ \Longrightarrow I_H[x]) \tag{6}$$

and

$$(\forall x : I_F[x])\ (\neg Q[x]\ \Longrightarrow I_G[x]). \tag{7}$$

Note that H and G may contain invocations of the main program F in their definitions, however, this would be treated as a combination of *if-then-else* and a *composition*. The predicate Q does not contain any invocation of F.

As a first step of the verification process, before going to the real verification, we check if the program is coherent. It is not that programs which are not coherent are necessarily not correct. However, if we want to achieve the modularity of our system, we need to restrict to dealing only with coherent programs.

3 Generation of Verification Conditions

In order to prove partial correctness, we extract the purely logical conditions which are sufficient and also necessary for the program to be partially correct.

By the following schema we define the class of programs which may contain nested recursion, namely we look at programs of the form:

$$F[x] = \ \textbf{If}\ Q[x]\ \textbf{then}\ S[x]\ \textbf{else}\ C_1[x, F[C_2[x, F[\ldots C_k[x, F[R[x]]]]]]], \tag{8}$$

where Q is a predicate and S, C_i, R are auxiliary functions ($S[x]$ is a "simple" function (the bottom of the recursion), $C_1[x, y], \ldots, C_k[x, y]$ are "combinator" functions, and $R[x]$ is a "reduction" function).

We assume that the functions $S, C_1, \ldots, C_k$ and R satisfy their specifications given by $I_S[x]$, $O_S[x, y]$, $I_{C_i}[x, y]$, $O_{C_i}[x, y, z]$, $I_R[x]$, $O_R[x, y]$.

3.1 Coherent Nested Recursive Programs

We start up with instantiating the definitions for coherent programs (1) and (2), namely:

Definition 3. *Let for all i, the functions S, C_i, and R satisfy their specifications (I_S, O_S), (I_{C_i}, O_{C_i}), and (I_R, O_R). Then the program F as defined in* (8) *with its specification (I_F, O_F) is coherent with respect to S, C_i, R, and their specifications, if and only if the following conditions hold:*

$$(\forall x : I_F[x])\ (Q[x] \Longrightarrow I_S[x]) \tag{9}$$

$$(\forall x : I_F[x])\ (\neg Q[x] \Longrightarrow I_F[R[x]]) \tag{10}$$

$$(\forall x : I_F[x])\ (\neg Q[x] \Longrightarrow I_R[x]) \tag{11}$$

$$\begin{gathered}(\forall x, y_1, \ldots, y_{2k} : I_F[x]) \\ (\neg Q[x] \wedge O_F[R[x], y_1] \wedge O_F[y_2, y_3] \wedge \cdots \wedge O_F[y_{2k-2}, y_{2k-1}] \\ \wedge O_{C_k}[x, y_1, y_2] \wedge O_{C_{k-1}}[x, y_3, y_4] \ \wedge \cdots \wedge O_{C_1}[x, y_{2k-1}, y_{2k}] \\ \Longrightarrow \\ I_{C_1}[x, y_1] \wedge I_{C_2}[x, y_3] \wedge \cdots \wedge I_{C_k}[x, y_{2k-1}] \\ \wedge I_F[y_2] \wedge I_F[y_4] \wedge \cdots \wedge I_F[y_{2k-2}])\end{gathered} \tag{12}$$

Now the conditions for coherence look a bit complicated, however, in the example we will see that this is not a case. Moreover, our experience shows that proving the conditions for coherence of concrete examples is relatively easy, compared, for example, to proving partial correctness conditions.

Looking closer at the conditions, we see that our intuition about coherent programs is met, namely:

- (9) treats the special case, that is, $Q[x]$ holds and no recursion is applied, thus the input x must fulfill the precondition of S.
- (10) treats the general case, that is, $\neg Q[x]$ holds and recursion is applied, thus the first new input $R[x]$ must fulfill the precondition of the main function F.
- (11) treats the general case, that is, $\neg Q[x]$ holds and recursion is applied, thus the input x must fulfill the precondition of the reduction function R.
- (12) treats the general case, and expresses in a cascade manner, that all the inputs to the combinator functions $C_1, \ldots, C_k$ must be appropriate and also all the intermediate inputs to the main function F must be appropriate as well.

After having defined the coherence verification conditions, we go towards defining the verification conditions for ensuring partial correctness.

3.2 Partial Correctness Conditions and their Soundness

We introduce the verification conditions for the class of programs with nested recursion, by providing the relevant *Soundness* theorem.

Theorem 1. *Let for all i, the functions S, C_i, and R satisfy their specifications (I_S, O_S), (I_{C_i}, O_{C_i}), and (I_R, O_R). Let also the program F as defined in* (8) *with its specification (I_F, O_F) be coherent with respect to S, C_i, R, and their specifications. Then, F is partially correct with respect to (I_F, O_F) if the following verification conditions hold:*

$$(\forall x : I_F[x])\ (Q[x] \Longrightarrow O_F[x, S[x]]) \tag{13}$$

$$\begin{gathered}(\forall x, y_1, \ldots, y_{2k} : I_F[x]) \\ (\neg Q[x] \wedge O_F[R[x], y_1] \wedge O_F[y_2, y_3] \wedge \cdots \wedge O_F[y_{2k-2}, y_{2k-1}] \\ \wedge O_{C_k}[x, y_1, y_2] \wedge O_{C_{k-1}}[x, y_3, y_4] \wedge \cdots \wedge O_{C_1}[x, y_{2k-1}, y_{2k}] \\ \Longrightarrow \\ O_F[x, y_{2k}])\end{gathered} \tag{14}$$

The above conditions constitute the following principle:

- (13) prove that the base case is correct.
- (14) prove that the recursive expression is correct under the assumption that all the reduced calls are correct.

Proof. Using Scott induction, we will show that F is partially correct with respect to its specification, namely:

$$(\forall x : I_F[x])\ (F[x] \downarrow \Longrightarrow O_F[x, F[x]]).$$

We now consider the following partial correctness property ϕ of functions:

$$(\forall f)\ (\phi[f] \iff (\forall a)\ (f[a] \downarrow\ \wedge\ I_F[a] \implies O_F[a, f[a]])).$$

The first step in Scott induction is to show that ϕ holds for the nowhere defined function Ω (that is, there is no x, such that $\Omega[x] \downarrow$). By the definition of ϕ we obtain:

$$\phi[\Omega] \iff (\forall a)\ (\Omega[a] \downarrow\ \wedge\ I_F[a] \implies O_F[a, \Omega[a]])),$$

and so, $\phi[\Omega]$ holds, since $\Omega[a] \downarrow$ never holds.

In the second step of Scott induction, we assume $\phi[f]$ holds for some function f:

$$(\forall a)\ (f[a] \downarrow\ \wedge\ I_F[a] \implies O_F[a, f[a]]), \tag{15}$$

and show $\phi[f_{new}]$, where f_{new} is obtained from f by the main program (8) as follows:

$$f_{new} = \textbf{If } Q[x] \textbf{ then } S[x] \textbf{ else } C_1[x, f[C_2[x, f[\ldots C_k[x, f[R[x]]]]]]].$$

Now, we need to show that for an arbitrary a,

$$f_{new}[a]\downarrow \ \wedge I_F[a] \implies O_F[a, f_{new}[a]].$$

Assume $f_{new}[a]\downarrow$ and $I_F[a]$. We have now the following two cases:

Case 1: $Q[a]$.
By the definition of f_{new} we obtain $f_{new}[a] = S[a]$ and since $f_{new}[a]\downarrow$, we obtain that $S[a]$ must terminate as well, that is $S[a]\downarrow$. Now using verification condition (13) we may conclude $O_F[a, S[a]]$ and hence $O_F[a, f_{new}[a]]$.

Case 2: $\neg Q[a]$.
By the definition of f_{new} we obtain:

$$f_{new}[a] = C_1[a, f[C_2[a, f[\ldots C_k[a, f[R[a]]]]]]]$$

and since $f_{new}[a]\downarrow$, we obtain that all the programs involved in this computation also terminate, that is:

$$C_1[a, f[C_2[a, f[\ldots C_k[a, f[R[a]]]]]]]\downarrow$$

and say: $C_1[a, f[C_2[a, f[\ldots C_k[a, f[R[a]]]]]]] = y_{2k}$,

$$f[C_2[a, f[\ldots C_k[a, f[R[a]]]]]]\downarrow$$

and say: $f[C_2[a, f[\ldots C_k[a, f[R[a]]]]]] = y_{2k-1}$,

$$C_2[a, f[\ldots C_k[a, f[R[a]]]]]\downarrow$$

and say: $C_2[a, f[\ldots C_k[a, f[R[a]]]]] = y_{2k-2}$,

$$f[C_3[a, f[\ldots C_k[a, f[R[a]]]]]]\downarrow$$

and say: $f[C_3[a, f[\ldots C_k[a, f[R[a]]]]]] = y_{2k-3}$,

$$\ldots$$

$$f[C_k[a, f[R[a]]]]\downarrow$$

and say: $f[C_k[a, f[R[a]]]] = y_3$,

$$C_k[a, f[R[a]]]\downarrow$$

and say: $C_k[a, f[R[a]]] = y_2$,

$$f[R[a]]\downarrow$$

and say: $f[R[a]] = y_1$, and

$$R[a]\downarrow .$$

From here, by the induction hypothesis, we obtain that

$$O_F[R[a], y_1] \wedge O_F[y_2, y_3] \wedge \cdots \wedge O_F[y_{2k-2}, y_{2k-1}].$$

On the other hand, by knowing that all the programs C_1, $C_2, \ldots, C_k$ are partially correct with respect to their specifications, we obtain that

$$O_{C_1}[a, y_{2k-1}, y_{2k}] \wedge O_{C_2}[a, y_{2k-3}, y_{2k-2}] \wedge \cdots$$
$$\wedge O_{C_{k-1}}[a, y_3, y_4] \wedge O_{C_k}[a, y_1, y_2].$$

Concerning the verification condition (14), note that all the assumptions from the left part of the implication are at hand and thus we can conclude:

$$O_F[a, y_{2k}],$$

and thus $O_F[a, f_{new}[a]]$.

We conclude that the property ϕ holds for the least fixpoint of (8) and hence, ϕ holds for the function computed by (8), which completes the proof of the soundness theorem (1).

Now we proceed towards the complement of the soundness theorem, namely, the *Completeness* theorem.

3.3 Completeness of the Verification Conditions

Now, we formulate the *Completeness* theorem for the class of programs with nested recursion.

Theorem 2. *Let for all i, the functions S, C_i, and R satisfy their specifications (I_S, O_S), (I_{C_i}, O_{C_i}), and (I_R, O_R). Let also the program F as defined in* (8) *with its specification (I_F, O_F) be coherent with respect to S, C_i, R, and their specifications, and the output specifications (O_F) and (O_{C_I}) of F and C_i, respectively, be functional ones.*

Then, if F is partially correct with respect to (I_F, O_F) then the verification conditions (13) *and* (14) *hold.*

Proof. We assume now that:

- The functions S, C_i, and R are partially correct with respect to their specifications (I_S, O_S), (I_{C_i}, O_{C_i}), and (I_R, O_R).
- The program F as defined in (8) with its specification (I_F, O_F) is coherent with respect to S, C_i, R, and their specifications.
- The output specifications O_F and O_{C_i} of F and C_i, respectively, are functional ones, that is:

$$(\forall x : I_F[x])(\exists! y)\ (O_F[x, y]),$$
$$(\forall x, y : I_{C_i}[x, y])(\exists! z)\ (O_{C_i}[x, y, z]).$$

- The program F as defined in (8) is partially correct with respect to its specification, that is, the partial correctness formula holds:

$$(\forall x : I_F[x])\ (F[x] \downarrow \implies O_F[x, F[x]]). \tag{16}$$

We show that (13) and (14) are valid as logical formulae by proving them simultaneously.

Take arbitrary but fixed x and assume $I_F[x]$ and $F[x]\downarrow$. We consider the following two cases:

Case 1: $Q[x]$

By the definition of F, we have $F[x] = S[x]$, and by using the partial correctness formula (16) of F, we conclude (13) holds. Proving (14) is trivial, because we have $Q[x]$.

Case 2: $\neg Q[x]$

Now, proving (13) is trivial. Assume $y_1, \ldots, y_{2k}$ are such that:

$$O_F[R[x], y_1] \wedge O_F[y_2, y_3] \wedge \cdots \wedge O_F[y_{2k-2}, y_{2k-1}]$$
$$\wedge O_{C_k}[x, y_1, y_2] \wedge O_{C_{k-1}}[x, y_3, y_4] \wedge \cdots \wedge O_{C_1}[x, y_{2k-1}, y_{2k}].$$

Since F is partially correct and $F[x]\downarrow$, we obtain that

$$C_1[x, F[C_2[x, F[\ldots C_k[x, F[R[x]]]]]]]\downarrow$$
$$F[C_2[x, F[\ldots C_k[x, F[R[x]]]]]]\downarrow$$
$$C_2[x, F[\ldots C_k[x, F[R[x]]]]]\downarrow$$
$$F[C_3[x, F[\ldots C_k[x, F[R[x]]]]]]\downarrow$$
$$\ldots$$
$$F[C_k[x, F[R[x]]]]\downarrow$$
$$C_k[x, F[R[x]]]\downarrow$$
$$F[R[x]]\downarrow$$
$$R[x]\downarrow .$$

From the fact that F is correct follows that it obeys its specification. We assumed $O_f[R[x], y_1]$ holds, and since the output specification is functional, we conclude that $F[R[x]] = y_1$.

Furthermore, since the output specifications of C_i: O_{C_i} are functional predicates, we obtain that:

$$C_1[x, f[C_2[x, f[\ldots C_k[x, f[R[x]]]]]]] = y_{2k},$$
$$f[C_2[x, f[\ldots C_k[x, f[R[x]]]]]] = y_{2k-1},$$
$$C_2[x, f[\ldots C_k[x, f[R[x]]]]] = y_{2k-2},$$
$$f[C_3[x, f[\ldots C_k[x, f[R[x]]]]]] = y_{2k-3},$$
$$\ldots$$
$$f[C_k[x, f[R[x]]]] = y_3,$$
$$C_k[x, f[R[x]]] = y_2,$$

$$f[R[x]] = y_1.$$

On the other hand, by the definition of F, we have:

$$F[x] = C_1[x, f[C_2[x, f[\ldots C_k[x, f[R[x]]]]]]]$$

and hence $F[x] = y_{2k}$. Again, from the correctness of F, we obtain:

$$O_F[x, y_{2k}],$$

which had to be proven.

By this we completed our proof of the *Completeness* theorem.

4 Example and Discussion

In order to illustrate the *Soundness* and the *Completeness* theorems, and the class of recursive functions which may contain nested recursive definitions, we consider the McCarthy 91 function, which is considered a "challenge problem" for automated program verification.

The program itself is defined as follows:

$$M[x] = \textbf{ If } x \geq 101 \textbf{ then } x - 10 \textbf{ else } M[M[x + 11]], \tag{17}$$

with the specification:

$$(\forall x)\ (I_M[x] \Longleftrightarrow x \in \mathbb{N}) \tag{18}$$

and

$$(\forall x, y)\ (O_M[x, y] \Longleftrightarrow (x < 101 \wedge y = 91) \vee (x \geq 101 \wedge y = x - 10)). \tag{19}$$

The (automatically generated) conditions for *coherence* are:

$$(\forall x : x \in \mathbb{N})\ (x \geq 101 \Longrightarrow \mathbb{T}) \tag{20}$$

$$(\forall x : x \in \mathbb{N})\ (x \ngeq 101 \Longrightarrow x + 11 \in \mathbb{N}) \tag{21}$$

$$(\forall x : x \in \mathbb{N})\ (x \ngeq 101 \Longrightarrow \mathbb{T}) \tag{22}$$

$$\begin{gathered}
(\forall x, y_1, y_2, y_3, y_4 : x \in \mathbb{N}) \\
(x \ngeq 101 \wedge \\
((x + 11 < 101 \ \wedge \ y_1 = 91) \vee (x + 11 \geq 101 \ \wedge \ y_1 = x + 11 - 10)) \\
\wedge\ ((y_2 < 101 \ \wedge \ y_3 = 91) \vee (y_2 \geq 101 \ \wedge \ y_3 = y_2 - 10)) \\
\wedge\, y_1 = y_2 \ \wedge \ y_3 = y_4 \\
\Longrightarrow \\
\mathbb{T} \ \wedge \ \mathbb{T} \ \wedge \ y_2 \in \mathbb{N} \ \wedge \ y_4 \in \mathbb{N})
\end{gathered} \tag{23}$$

One sees that the formulae (20) and (22) are trivially valid, because we have the logical constant $\mathbb{T}$ at the right side of an implication. The origin of these $\mathbb{T}$ come from the preconditions of the $x - 10$ ($S[x] = x - 10$) and the projection functions ($C_1[x, y] = y$ and $C_2[x, y] = y$).

The formulae (21) and (23) are easy consequences of the elementary theory of naturals.

For the further check of *correctness* the generated conditions are:

$$\begin{gathered}(\forall x : x \in \mathbb{N})(x \geq 101 \\ \Longrightarrow\ (x < 101 \wedge x - 10 = 91) \vee (x \geq 101 \wedge x - 10 = x - 10)).\end{gathered} \tag{24}$$

$$\begin{gathered}(\forall x, y_1, y_2, y_3, y_4 : x \in \mathbb{N}) \\ (n \not\geq 101 \\ \wedge((x + 11 < 101 \wedge y_1 = 91) \vee (x + 11 \geq 101 \wedge y_1 = x + 11 - 10)) \\ \wedge((y_2 < 101 \wedge y_3 = 91) \vee (y_2 \geq 101 \wedge y_3 = y_2 - 10)) \\ \wedge y_1 = y_2 \wedge y_3 = y_4 \\ \Longrightarrow \\ (x < 101 \wedge y_4 = 91) \vee (x \geq 101 \wedge y_4 = x - 10)).\end{gathered} \tag{25}$$

The proofs of these verification conditions are straightforward, and thus the program (17) is partially correct with respect to the specification (18), (19).

Now comes the question: What if the program is not correctly written? Thus, we introduce now a bug. The program M is now almost the same as the previous one, but in the base case (when $x \geq 101$) the return value is $x - 11$. The new (wrong) definition of M is:

$$M[x] = \ \textbf{If}\ x \geq 101\ \textbf{then}\ x - 11\ \textbf{else}\ M[M[x + 11]], \tag{26}$$

After generating the verification conditions, we see that all but one are valid, namely:

$$\begin{gathered}(\forall x : x \in \mathbb{N})(x \geq 101 \\ \Longrightarrow (x < 101\ \wedge\ x - 11 = 91) \vee (x \geq 101\ \wedge\ x - 11 = x - 10)).\end{gathered} \tag{27}$$

which reduces to proving:

$$x - 11 = x - 10.$$

Therefore, according to the completeness of the method, we conclude that the program M does not satisfy its specification. Moreover, the failed proof gives a hint for "debugging": we need to change the return value in the case $x \geq 101$ to $x - 10$.

A similar experiment shows, that in fact, the input condition (19), that is $x \in \mathbb{N}$ is too strong, and could be successfully replaced by $x \in \mathbb{Z}$.

5 Related Research

Proving correctness of recursive programs in an automatic manner is considered as being challenge, even without nested recursive definitions. Studying the behavior of such programs begins with classical papers (e.g., [18]) and it is still under consideration.

Proofs exposed in classical books (e.g., [10, 11]) are very comprehensive, however, their orientation is theoretical rather than practical and mechanized. On the other hand, most of the tools for proving program correctness automatically or semiautomatically, do not treat that special case of recursion if they do recursion at all.

In the PVS system [16] the approach is type theoretical and relies on exploration of certain subtyping properties.

The approach presented in [2] puts additional restrictions, namely, recursive programs are examined first to satisfy non-nested recursive definitions and then they may be considered as nested recursion.

In the RRL system [7], a specialized "cover set induction method" is introduced and the nested recursion is treated by it.

In the Lambda system [5], the recursive programs are treated as fixpoint operators, however, it does not extract automatically the inductive obligations that would correspond to the general case in our settings.

The paper [17], presents two methods for dealing with nested recursion, including termination. However, termination conditions must be provided manually.

The main (and also very essential) difference of our approach is that we are able to formulate conditions which are not only sufficient but also necessary in order for the program to be correct.

6 Conclusions

In this paper, we defined necessary and sufficient conditions for programs which may contain nested recursion to be partially correct. These are expressed by two theorems, whose proofs are carried over in the fixpoint theory of programs. However, the concrete proof obligations (verification conditions) are first order predicate logic formulae, which are provable in the theory of the domain on which the program is executed.

Furthermore, the concrete proof problems are used as test cases for the provers of *Theorema* and for experimenting with the organization and management of mathematical knowledge.

References

1. J. W. de Bakker and D. Scott. A Theory of Programs. In *IBM Seminar*, Vienna, Austria, 1969.

2. R. S. Boyer and J S. Moore. *A Computational Logic Handbook*. Academic Press Professional, Inc., San Diego, CA, USA, 1988.
3. B. Buchberger, A. Craciun, T. Jebelean, L. Kovacs, T. Kutsia, K. Nakagawa, F. Piroi, N. Popov, J. Robu, M. Rosenkranz, W. Windsteiger. Theorema: Towards Computer-Aided Mathematical Theory Exploration. In *Journal of Applied Logic, vol. 4, issue 4*, pp. 470–504, 2006.
4. B. Buchberger, D. Vasaru. Theorema: The Induction Prover over Lists. In *First International Theorema Workshop*, RISC, Hagenberg, Austria, June 1997.
5. S. Finn, M. P. Fourman, and J. Longley. Partial Functions in a Total Setting. *Journal of Automated Reasoning*, 18(1):85–104, 1997.
6. T. Jebelean, L. Kovács, and N. Popov. Experimental Program Verification in the Theorema System. *Int. Journal on Software Tools for Technology Transfer (STTT)*, 2008. To appear.
7. D. Kapur and M. Subramaniam. Automating Induction over Mutually Recursive Functions. In *AMAST '96: Proceedings of the 5th International Conference on Algebraic Methodology and Software Technology*, pages 117–131, London, UK, 1996. Springer-Verlag.
8. M. Kaufmann and J S. Moore. An Industrial Strength Theorem Prover for a Logic Based on Common Lisp. *Software Engineering*, 23(4):203–213, 1997.
9. L. Kovacs, N. Popov, T. Jebelean. Combining Logic and Algebraic Techniques for Program Verification in Theorema. In *Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISOLA 2006)*, Paphos, Cyprus, 2006.
10. J. Loeckx, K. Sieber. The Foundations of Program Verification. Teubner, second edition, 1987.
11. Z. Manna. *Mathematical Theory of Computation*. McGraw-Hill Inc., 1974.
12. Z. Manna and J. McCarthy. Properties of Programs and Partial Function Logic. *Machine Intelligence*, 5:27–37.
13. Z. Manna and A. Pnueli. Formalization of Properties of Functional Programs. *J. ACM*, 17(3):555–569, 1970.
14. B. Meyer. Applying "Design by Contract". *Computer*, 25(10):40–51, 1992.
15. N. Popov and T. Jebelean. A Prototype Environment for Verification of Recursive Programs. In Z. Istenes, editor, *Proceedings of FORMED'08*, pages 121–130, March 2008. to appear as ENTCS volume, Elsevier.
16. PVS: Specification and Verification System. *http://pvs.csl.sri.com*
17. K. Slind. Another Look at Nested Recursion. In *TPHOLs '00: Proceedings of the 13th International Conference on Theorem Proving in Higher Order Logics*, pages 498–518, London, UK, 2000. Springer-Verlag.
18. W. W. Tait. Nested Recursion. *Mathematische Annalen*, 143(3):236–250, 1961.

External and Internal Syntax of the λ-calculus

Masahiko Sato[1]

Graduate School of Informatics, Kyoto University

Abstract. It is well known that defining the substitution operation on λ-terms appropriately and establish basic properties like the substitution lemma is a subtle task if we wish to do it formally. The main obstacle here comes from the fact that unsolicited capture of free variables may occur during the substitution if one defines the operation naively.
We argue that although there are several approaches to cope with this problem, they are all unsatisfactory since each of them defines the λ-terms in terms of a single fixed syntax. We propose a new way of defining λ-terms which uses an external syntax to be used mainly by humans and an internal syntax which is used to implement λ-terms on computers.
In this setting, we will show that we can define λ-terms and the substitution operation naturally and can establish basic properties of terms easily.

1 Introduction

There is a growing interest in the study of syntactic structure of expressions equipped with the variable binding mechanism. The importance of this study can be justified for various reasons, including those of educational, scientific and engineering reasons. This study is educationally important since in logic and computer science, we cannot avoid teaching the technique of substitution of higher order linguistic objects correctly and rigorously. Scientific importance is obvious as can be seen from the historical facts that correctly defining the substitution operation was difficult and sometimes resulted in erroneous definitions. Engineering importance comes from recent developments of proof assistance and symbolic computation systems which are increasingly used to assist and verify metamathematical results rather than ordinary mathematical results. We cite here only Aydemir et al. [1] which contains an extensive list of literature on this topic.

We share all those reasons above with other researchers in this field as our motivation to study this subject, but we are especially interested in this subject because of the following ontological question.

> What are *syntactic objects* as objects of mathematical structures with variable binding mechanism?

This is a *semantical* question and cannot be answered by simply manipulating symbols syntactically. To answer this question, we have to study syntax semantically. Our contribution in this paper is precisely the result of such a study.

We have already contributed in this study in [23–28] by investigating the mathematical structure of symbolic expressions. We think that Frege [8], McCarthy [16, 17], Martin-Löf [19, Chapter 3] and Gabbay-Pitts [9, 10] contributed very much in the semantical study of syntax. Our work which we report here is influenced by these works and in particular by the works of Frege and Gabbay-Pitts.

Frege not only formulated syntax of a logical language with binders for the first time, he also formulated it by using two disjoint sets of variables, one for *global variables* using Latin letters and the other for *local variables* using German letters [15, page 25]. Later, Gentzen [11], for instance, followed this approach, but traditionally both logic and the λ-calculus have been formulated using only one sort of variables including Gödel [12] and Church [5] perhaps because of the influence of Whitehead and Russell [32]. McCarthy contributed to semantical understanding of syntactic objects by introducing Lisp symbolic expressions [16] and by introducing the concept *abstract syntax* in [17]. He introduced the term 'abstract syntax' by providing functions to analyze and synthesize *syntactic objects* hiding details of concrete representations of these syntactic objects. This approach works well for languages without variable binding mechanism, but it was difficult to provide abstract syntax (in McCarthy's sense[1]) for languages with binders until Gabbay and Pitts [9, 10] invented nominal technique which implemented abstraction using Fraenkel-Mostowski set theory. They utilized the *equivariance property* which holds in FM-set theory over an abstract set of atoms to deal with α-equivalence and abstraction mechanism on languages with binders having explicit variable names (rather than languages with nameless variables based, for instance, on de Bruijn indices).

Our approach is similar to Gabbay-Pitts' in the sense that the equivariance property holds for our languages, but, unlike their case, we work in standard mathematics and develop our theory by introducing a new notion of *B-algebra* ('B' is for 'binding') which is an algebra equipped with the mechanism of variable binding. For a set $\mathbb{X}$ of atoms, we can introduce the set $\mathbb{S}[\mathbb{X}]$ of *symbolic expressions* over $\mathbb{X}$ as a free B-algebra freely generated from $\mathbb{X}$.

A standard method of defining λ-terms (with explicit names for bound variables) goes as follows. First the set Λ of λ-terms is inductively defined as the smallest set satisfying the set equation $\Lambda = \mathbb{X} + \Lambda \times \Lambda + \mathbb{X} \times \Lambda$ where $\mathbb{X}$ is a given set of variables. Unfortunately it is not possible to define substitution operation on this data structure in a meaningful way due to the possibility of variable capture. To get out of this situation, the α-equivalence relation $=_\alpha$ is defined, and various notions and properties of λ-terms are established by *identifying* α-equivalent terms. However, as pointed out by McKinna-Pollack [18], Pitts [20], Urban [30], Vestargaard [31] etc., that we have to work modulo α-equivalence creates many technical difficulties when we reason about properties of λ-terms by structural induction on λ-terms.

[1] The term 'abstract syntax' used in 'HOAS (Higher Order Abstract Syntax)' has different sense. For this reason, structural induction/recursion works for syntactic objects described by abstract syntax in McCarthy's sense but not in HOAS.

We wish to solve this problem by proposing a new way of defining λ-terms which uses an external syntax to be used mainly by humans and an internal syntax which is used to implement λ-terms on computers. Our motivation for introducing two kinds of syntax is as follows.

Firstly, we wish to have a syntax which inductively creates the set $\mathbb{L}$ of λ-terms isomorphic to $\Lambda/{=_\alpha}$, since by doing so we can constructively grasp each λ-term through the process of creating the term inductively. Note that in case of λ-terms as elements of $\Lambda/{=_\alpha}$, we cannot grasp each term as above, since although each element of Λ is inductively created, each element of $\Lambda/{=_\alpha}$ is obtained abstractly by *identifying* α-equivalent elements of Λ. We will call the syntax which defines $\mathbb{L}$ *the internal syntax* since it can be easily implemented on a computer.

Secondly, in addition to the internal syntax, we will also introduce *the external syntax* which is intended to be used by humans. The external syntax is the same as the standard syntax of λ-calculus given for example in Barendregt [2] and we use Λ as the set of λ-terms but work modulo $=_\alpha$. We can never avoid having an external syntax, since we need it to read and write λ-terms. So, the question is the choice of an external syntax which is comfortable for humans to use as a medium to talk about abstract but real λ-terms as syntactic objects. We think that for this purpose we are right in choosing the standard syntax as *the* external syntax *provided that* we can work in it comfortably and smoothly. Our approach achieves this by defining a natural *semantic function* $[\![-]\!]$ which maps each λ-term M in the external syntax to a λ-term $[\![M]\!]$ in the internal syntax in such a way that $[\![M]\!] = [\![N]\!]$ iff $M =_\alpha N$.

This paper is organized as follows. In Section 2 we introduce the system $\mathbb{S}$ of symbolic expressions with binding structure. We also introduce a new notion of B-algebra and characterize the set of symbolic expressions as a free B-algebra. We also define substitutions as endomorphisms on $\mathbb{S}$ and point out that permutations (i.e., bijective substitutions) are automorphisms and that the group of permutations naturally acts on $\mathbb{S}$ and endow the equivariance property on $\mathbb{S}$.

In Section 3, we introduce the internal syntax for λ-calculus, and define the set $\mathbb{L}$ of λ-terms as a subset of the free B-algebra $\mathbb{S}$ generated by the set $\mathbb{X}$ of global variables. The internal syntax has two sorts of variables, global and local variables. These two sorts of variables have explicit *names* and hence, in the case of local variables, these names can be used to directly refer to the corresponding binders. In contrast with this, if we use de Bruijn indices [6], local variables become *nameless* and we need the complex mechanism of lifting so that these nameless variables can correctly refer to the corresponding binders. Substitution on $\mathbb{L}$ is defined as B-algebra endomorphism. So, there is no need of renaming of variables while computing substitutions. In this paper, we take up the untyped λ-calculus as a canonical example of linguistic structure with the mechanism of variable binding. The system is canonical as it is well-known since Church [4] that λ-calculus can be used as an implementation language of other languages with binders.

In Section 4, we introduce the external syntax by the standard method using only one sort of variables which are used both as global variables (aka free variables) and local variables (aka bound variables). The set Λ of λ-terms in this syntax is also a subset of the same base set $\mathbb{S}$ we used to define the internal syntax. The main difference of the external syntax from the internal syntax is that in the former syntax only one sort of variables is used while two sorts of variables are used in the latter syntax. This difference comes from our construction of $\Lambda \subset \mathbb{S}$ without using the binding mechanism of the B-algebra $\mathbb{S}$. The external syntax and the internal syntax are naturally related by the semantic surjective function $[\![-]\!] : \Lambda \to \mathbb{L}$ which is homomorphic with respect to the application constructor and collapses α-equality to the equality on $\mathbb{L}$.

Section 5 concludes the paper by comparing our results with Gabbay-Pitts' approach and with that of Aydemir et al. [1], and finally by remarking that the data structure of our internal syntax is isomorphic to those of the representations proposed by Quine [22], Bourbaki [3], Sato and Hagiya [23] and Sato [24, 26].

2 Symbolic expressions

In this section we define the set of *symbolic expressions* as a free algebra equipped with a binary operation and a binding operation, and generated by a denumerably infinite set $\mathbb{X}$ of *atoms*. In the construction, we will also use the set $\mathbb{N}$ of natural numbers (which includes 0) as binders.

We will write '$M : \mathbb{S}$', '$X : \mathbb{X}$' and '$x : \mathbb{N}$' for the judgments 'M is a symbolic expression', 'X is an atom' and 'x is a natural number' respectively, and define the set $\mathbb{S}$ of symbolic expressions over $\mathbb{X}$ by the following rules. Since $\mathbb{S}$ is defined depending on $\mathbb{X}$, we will write '$\mathbb{S}[\mathbb{X}]$' for $\mathbb{S}$ when we wish to emphasize the dependency. Atoms will also be called *global variables* and natural numbers will also be called *local variables*. We will use letters 'X', 'Y', 'Z' for global variables, 'x', 'y', 'z' for local variables, and 'M', 'N', 'P', 'Q' for symbolic expressions and later elements of B-algebras.

$$\frac{X : \mathbb{X}}{X : \mathbb{S}} \qquad \frac{x : \mathbb{N}}{x : \mathbb{S}} \qquad \frac{M : \mathbb{S} \quad N : \mathbb{S}}{(M\ N) : \mathbb{S}} \qquad \frac{x : \mathbb{N} \quad M : \mathbb{S}}{[x]\,M : \mathbb{S}}$$

The expression '$(M\ N)$' is said to be *the pair of M and N*. The expression '$[x]\,M$' is said to be *the abstraction by x of M*, 'x' is said to be *the binder of this expression* and 'M' is said to be *the scope of the binder 'x'*. The above definition of symbolic expressions reflects our idea that local variables may get bound by a binder but global variables should never get bound.

With each symbolic expression M we assign a set $\mathrm{LV}(M)$ called *the set of free local variables in M* and a set $\mathrm{GV}(M)$ called *the set of global variables in M* as follows.

1. $\mathrm{LV}(X) \triangleq \{\}$.
2. $\mathrm{LV}(x) \triangleq \{x\}$.
3. $\mathrm{LV}((M\ N)) \triangleq \mathrm{LV}(M) \cup \mathrm{LV}(N)$.

4. $\mathrm{LV}(\texttt{[}x\texttt{]}\,M) \triangleq \mathrm{LV}(M) - \{x\}$.

Note: In general, the binder x of an expression $\texttt{[}x\texttt{]}\,M$ may contain x in M again as a binder. In fact, $\texttt{[}x\texttt{]}\,\texttt{[}x\texttt{]}\,x$ is an example of such a case, and in this case we consider that the right-most occurrence of 'x' is bound by the inner binder and not by the left-most binder. We will say that x *occurs free* in M if $x \in \mathrm{LV}(M)$.

1. $\mathrm{GV}(X) \triangleq \{X\}$.
2. $\mathrm{GV}(x) \triangleq \{\}$.
3. $\mathrm{GV}(\texttt{(}M\ N\texttt{)}) \triangleq \mathrm{GV}(M) \cup \mathrm{GV}(N)$.
4. $\mathrm{GV}(\texttt{[}x\texttt{]}\,M) \triangleq \mathrm{GV}(M)$.

Note: The above definition reflects our idea that global variables are never to be bound. We will say that X *occurs* in M if $X \in \mathrm{GV}(M)$.

It is possible to characterize the set $\mathbb{S}$ algebraically by introducing the notion of B-algebra ('B' is for 'binding'). A *B-algebra* is a triple

$$\langle A, \texttt{()} : A \times A \to A, \texttt{[]} : \mathbb{N} \times A \to A \rangle$$

where A is a set which contains $\mathbb{N}$ as its subset. A *magma* (also called a groupoid) is an algebraic structure equipped with a single binary operation, and the notion of B-algebra introduced here is derived from this notion of magma. A B-algebra is a magma equipped with an additional binding operation.

Note: The notion of B-algebra is different from the notion of binding algebra introduced in Firore et al. [7, Section 2]. While our B-algebra has an explicit binding operation $\texttt{[}x\texttt{]}\,M$ which can bind any $x \in \mathbb{N}$ in any $M \in A$, a binding algebra does not have such an explicit algebraic operation of abstraction. Instead, a binding algebra presupposes the existence of the objects obtained by variable binding and operate on these objects.

A *B-algebra homomorphism* is a function h from a B-algebra A to a B-algebra B such that $h(x) = x$, $h(\texttt{(}M\ N\texttt{)}) = \texttt{(}h(M)\ h(N)\texttt{)}$ and $h(\texttt{[}x\texttt{]}\,M) = \texttt{[}x\texttt{]}\,h(M)$ hold for all $M, N \in A$ and $x \in \mathbb{N}$. It is then easy to see that

$$\langle \mathbb{S}[\mathbb{X}], \texttt{()} : \mathbb{S} \times \mathbb{S} \to \mathbb{S}, \texttt{[]} : \mathbb{N} \times \mathbb{S} \to \mathbb{S} \rangle$$

is a *free* B-algebra with the free generating set $\mathbb{X}$. In fact, let B be an B-algebra and consider any $\rho : \mathbb{X} \to B$. Then this ρ can be uniquely extended to a B-algebra homomorphism $[\rho] : \mathbb{S}[\mathbb{X}] \to B$ as follows:

1. $[\rho]X \triangleq \rho(X)$.
2. $[\rho]x \triangleq x$.
3. $[\rho]\texttt{(}M\ N\texttt{)} \triangleq \texttt{(}[\rho]M\ [\rho]N\texttt{)}$.
4. $[\rho]\texttt{[}x\texttt{]}\,M \triangleq \texttt{[}x\texttt{]}\,[\rho]M$.

Here, we are interested in the case where B is $\mathbb{S}$ and $\rho : \mathbb{X} \to \mathbb{S}$ is a finite map. We will call such a map *a finite simultaneous substitution*, or simply *a substitution*. If ρ sends X_i to P_i ($1 \leq i \leq n$, and X_i are distinct) and fixes the rest, $[\rho] : \mathbb{S} \to \mathbb{S}$ is an endomorphism and we will write '$[P_i/X_i]M$' for $[\rho]M$ and call it '*the result of (simultaneously) substituting P_i for X_i in M*'. The substitution operation satisfies the following equations.

1. $[P_i/X_i]X = \begin{cases} P_i & \text{if } X = X_i \text{ for some } i, \\ X & \text{if } X \neq X_i \text{ for all } i. \end{cases}$
2. $[P_i/X_i]x = x$.
3. $[P_i/X_i](M\ N) = ([P_i/X_i]M\ [P_i/X_i]N)$.
4. $[P_i/X_i][x]\,M = [x][P_i/X_i]M$.

It should be noted that, since substitution is an endomorphism, the substitution operation commutes with the operations of B-algebra smoothly. We note that if ρ and σ are substitutions, then their composition $\rho \circ \sigma$ is also a substitution satisfying the identity $[\rho \circ \sigma]M = [\rho][\sigma]M$. This is a useful property of substitutions as first-class objects.

An endoporhism $[\rho]$ becomes an automorphism if and only if ρ is a permutation, that is, the image of ρ is $\mathbb{X}$ and $\rho : \mathbb{X} \to \mathbb{X}$ is a bijecton. We write '$G_{\mathbb{X}}$' for the group of finite permutations on $\mathbb{X}$. The group $G_{\mathbb{X}}$ naturally acts on the B-algebra $\mathbb{S}[\mathbb{X}]$ by defining the group action of $\pi \in G_{\mathbb{X}}$ on M as $[\pi]M$. In particular, we have $[/]M = M$ and $[\pi \circ \sigma]M = [\pi][\sigma]M$. When $\pi = X, Y/Y, X$ is a transposition which transposes X and Y, we will write '$X/\!\!/Y$' for π. A transposition is its own inverse since we have $[X/\!\!/Y] \circ [X/\!\!/Y] = [X, Y/Y, X] \circ [X, Y/Y, X] = [X, Y/X, Y] = [/]$. For each $\pi \in G_{\mathbb{X}}$ the group action $[\pi](-)$ determines a B-algebra automorphism on $\mathbb{S}[\mathbb{X}]$.

We can apply the general notion of *equivariance* to the group $G_{\mathbb{X}}$. Suppose that $G_{\mathbb{X}}$ acts on two sets U, V and consider a map $f : U \to V$. The map f is said to be an *equivariant map* if f commutes with all $\pi \in G$ and $u \in U$, namely, $f([\pi]u) = [\pi]f(u)$. An equivariant map for an n-ary function can be defined similarly. For example, let $P : U \times V \to \mathbb{B}$ be a binary relation whose values are taken in the set $\mathbb{B} = \{\mathtt{t}, \mathtt{f}\}$ of truth values and define the action of $G_{\mathbb{X}}$ on $\mathbb{B}$ to be a trivial one which fixes the two truth values. Then, that P is an equivariant map means that $P([\pi]u, [\pi]v) = P(u, v)$ holds for all $u \in U, v \in V$ and $\pi \in G_{\mathbb{X}}$. This means that an equivariant relation preserves the validity of the relation under permutations, and for this reason, we may call an equivariant relation *an equivariance*. Thus, the action of $G_{\mathbb{X}}$ provides a useful tool for establishing properties about symbolic expressions since all the statements we make about symbolic expressions enjoy *the equivariance property*. Importance of the notion of equivariance in the abstract treatment of syntax seems to be first emphasized by Gabbay and Pitts [9, 20]. We will apply the notion of equivariance in Section 3 and in Section 4, Theorem 3.

We need to define another form of substitution operation on $\mathbb{S}$ which substitutes a symbolic expression P for *free* occurrences of a local variable y in M. We will write '$[P/y]M$' for the result of the operation and define it as follows.

1. $[P/y]X \triangleq X$.
2. $[P/y]x \triangleq \begin{cases} P & \text{if } x = y, \\ x & \text{if } x \neq y. \end{cases}$
3. $[P/y](M\ N) \triangleq ([P/y]M\ [P/y]N)$.
4. $[P/y]\llbracket x \rrbracket M \triangleq \begin{cases} \llbracket x \rrbracket M & \text{if } x = y, \\ \llbracket x \rrbracket [P/y]M & \text{if } x \neq y. \end{cases}$

Note that $[P/y]$ is a function from $\mathbb{S}$ to $\mathbb{S}$ but, unless P is y, it is not a B-algebra homomorphism since it neither preserves y nor commutes with the abstraction operation $\llbracket y \rrbracket(-)$.

The intended meaning of the fourth clause of the above definition is as follows. According to our definition of the substitution $[P/y]$ we have $[P/y]\llbracket x \rrbracket M = \llbracket x \rrbracket M$ if $x = y$. This is natural since $\mathrm{LV}(\llbracket x \rrbracket M)$ does not contain y in this case. If $x \neq y$, then the definition is again natural since it is defined so that the substitution will commute with the abstraction operation. However, it should be noted that, in this case, if P contains free occurrences of x then these occurrences of x will be bound after the substitution. This is an unsolicited situation and known ways to avoid this is either to rename x in $\llbracket x \rrbracket M$ or to rename x in P. The first way is so called α-renaming and the second is called lifting. In Section 3, we will introduce a third way in which we only consider a subset of $\mathbb{S}$ which is rich enough to define λ-terms and at the same time does not create this unsolicited situation. The third way solves the problem by not creating the problem. The height function we define below plays an important role in achieving this.

We can readily show, by induction on the construction of M, the following useful lemmas. We note in passing that, although we can prove it inductively, the Permutation Lemma below follows as an instance of equivariance which says that the substitution function commutes with the action of permutations.

Lemma 1 (GV Lemma). $\mathrm{GV}([P/X]M) \subseteq (\mathrm{GV}(M) - \{X\}) \cup \mathrm{GV}(P)$.

Lemma 2 (Permutation Lemma). *If π is a finite permutation on $\mathbb{X}$, then we have $[\pi][P/Y]M = [[\pi]P/[\pi]Y][\pi]M$.*

Lemma 3 (Substitution Lemma). *If $X \neq Y$ and $X \notin \mathrm{GV}(Q)$, then we have $[Q/Y][P/X]M = [[Q/Y]P/X][Q/Y]M$.*

We conclude this section by defining two functions, namely, the *height function* $\mathrm{H} : \mathbb{X} \times \mathbb{S} \to \mathbb{N}$ and the *birthday function* $|-| : \mathbb{S} \to \mathbb{N}$ as follows.

1. $\mathrm{H}_X(Y) \triangleq \begin{cases} 1 & \text{if } X = Y, \\ 0 & \text{if } X \neq Y. \end{cases}$
2. $\mathrm{H}_X(x) \triangleq 0$.
3. $\mathrm{H}_X((M\ N)) \triangleq \max(\mathrm{H}_X(M), \mathrm{H}_X(N))$.
4. $\mathrm{H}_X(\llbracket x \rrbracket M) \triangleq \begin{cases} 0 & \text{if } \mathrm{H}_X(M) = 0, \\ \mathrm{H}_X(M) & \text{if } x = 0 \text{ or } \mathrm{H}_X(M) > x, \\ x + 1 & \text{otherwise.} \end{cases}$

We will call $\mathrm{H}_X(M)$ *the height of* X *in* M. We note that $\mathrm{H}_X(M)$ is 0 if and only if X is not used in the construction of M, that is, $X \notin \mathrm{GV}(M)$. If $\mathrm{H}_X(M) = n+1$, then it means that (if we write M in tree form) either $n = 0$ and X occurs as a leaf at least once in the tree and all the paths from the root to X do not go through a binder, or n is the largest binder among all the binders we encounter if we go down the tree from the root to all the occurrences of X.

We can easily prove the following lemmas.

Lemma 4. *If* $X \neq Y$ *and* $X \notin \mathrm{GV}(Q)$, *then* $\mathrm{H}_X([Q/Y]M) = \mathrm{H}_X(M)$.

Lemma 5 (Height Lemma). *If* $x = \mathrm{H}_X(M)$, *then* $[X/x][x/X]M = M$.

The *birthday function* $|-|$ is defined as follows.

1. $|X| \stackrel{\triangle}{=} 1$.
2. $|x| \stackrel{\triangle}{=} 1$.
3. $|(M\ N)| \stackrel{\triangle}{=} \max(|M|, |N|) + 1$.
4. $|[x]\,M| \stackrel{\triangle}{=} |M| + 1$.

The birthday function is defined by reflecting our ontological view of mathematical objects according to which each mathematical object must be constructed by applying a *constructor function* to already constructed objects. By assigning the birthday of a symbolic expression as above, we can see that all the four rules we used in our formation rules of symbolic expressions do enjoy this property. The construction, therefore, proceeds as follows. We observe that among the four rules of symbolic expressions, the first two are unary constructors and the last two are binary constructors. We assume that we have no symbolic expressions on day 0 but global variables and local variables are already constructed so that we have them all on day 0. So, on day 1, only the first two constructors are applicable. Hence, on day 1, all the global and local variables are recognized as symbolic expressions. On day 2, all the four rules are applicable, but only the last two rules produce new symbolic expressions, and they are: $(M\ N)$ where M, N are both variables, or $[x]\,M$ where x is a local variable and M is a variable, be it global or local. The construction of symbolic expressions continues in this way day by day, and every symbolic expression shall be born on its birthday.

This construction suggests the following induction principle which can be used to establish general properties about symbolic expressions:

$$\frac{(\forall N : \mathbb{S}.\, |N| < |M| \Rightarrow \Phi(N)) \Rightarrow \Phi(M)}{\Phi(M)}\ .$$

By using this rule, we can see the validity of the following structural induction rule.

If we can derive the following four judgments

1. $\forall X : \mathbb{X}.\, \Phi(X)$,
2. $\forall x : \mathbb{N}.\, \Phi(x)$,
3. $\forall M, N : \mathbb{S}.\, \Phi(M) \wedge \Phi(N) \Rightarrow \Phi((M\ M))$,
4. $\forall x : \mathbb{N}.\, \forall M : \mathbb{S}.\, \Phi(M) \Rightarrow \Phi([x]\,M)$,

then we may conclude the judgment: $\forall M : \mathbb{S}.\, \Phi(M)$.

3 The internal syntax

In this section, we define the internal syntax for the λ-calculus. The internal syntax is more basic than the external syntax we introduce in Section 4. It is so for the following two reasons. Firstly, each λ-term defined by the internal syntax directly corresponds to a λ-term as an abstract mathematical object. Namely, the equality relation on the λ-terms defined by the internal syntax is the syntactical identity relation, while the equality on the external λ-terms must be defined modulo α-equivalence. Secondly, we can later define the equality relation on external λ-terms by giving an interpretation of them in terms of internal terms. For these reasons, we will find internal λ-terms easier to implement on a computer than external terms.

As the domain for representing the λ-terms of the internal syntax, we use the free B-algebra $\mathbb{S}[\mathbb{X} \cup \{\mathtt{app}, \mathtt{lam}\}]$, where $\mathbb{X}$ is a denumerably infinite set containing neither `app` nor `lam` and disjoint from the set $\mathbb{N}$. We will write '$\mathbb{L}$' for the set of λ-terms in this syntax. Although $\mathbb{L}$ is not a subalgebra of $\mathbb{S}$, it enjoys the nice property of being closed under the substitution operation. Namely, for any $X \in \mathbb{X}$ and $M, N \in \mathbb{L}$, we will have $[N/X]M \in \mathbb{L}$ (Theorem 1).

We define the set $\mathbb{L}$ inductively by the following rules. The judgment '$M : \mathbb{L}$' means that M is a λ-expression. We will write '`(app` M N`)`' as an abbreviation of '`(app (`M N`))`'.

$$\frac{X : \mathbb{X}}{X : \mathbb{L}} \qquad \frac{M : \mathbb{L} \quad N : \mathbb{L}}{\mathtt{(app}\ M\ N\mathtt{)} : \mathbb{L}} \qquad \frac{X : \mathbb{X} \quad M : \mathbb{L}}{\mathtt{(lam\ [}x\mathtt{]}[x/X]M\mathtt{)} : \mathbb{L}}\ (*)$$

Note: In the third rule $(*)$, the height of X in M must be x. We see that in case $x = 0$, then the conclusion of the rule becomes `(lam [0]`M`)`.

A λ-term is called an *application* if it is defined by the second rule above, and an *abstract* if defined by the third rule. Each abstract $M =$ `(lam [`x`]`P`)` defines a function $f_M : \mathbb{S} \to \mathbb{S}$ by putting $f_M(N) \stackrel{\triangle}{=} [N/x]P$ for all $N \in \mathbb{S}$. We will write '$M(N)$' for $f_M(N)$ and call it the *instantiation of the abstract* M *by* N.

We explain the notion of equivariance for the set $\mathbb{S} = \mathbb{S}[\mathbb{X} \cup \{\mathtt{app}, \mathtt{lam}\}]$. Here, the equivariance property is the property which reflects the intrinsic internal symmetry of the set $\mathbb{S}$ with respect to the group action $[\sigma](-) : G_{\mathbb{X}} \times \mathbb{S} \to \mathbb{S}$ which sends any $M \in \mathbb{S}$ to $[\sigma]M \in \mathbb{S}$ where σ is any finite permutation on $\mathbb{S}$. Let $\Phi(M)$ be a statement about $M \in \mathbb{S}$. Then the statement has the *equivariance property* if, for any $M \in \mathbb{S}$ and $\sigma \in G_{\mathbb{X}}$, $\Phi(M)$ holds if and only $\Phi([\sigma]M)$ holds. (See also [10].)

We can see, albeit informally, that all the statements we prove in this paper have the equivariance property as follows. Suppose that we have a derivation D of $\Phi(M)$. We can formalize this derivation in a formal language whose syntax is based on $\mathbb{S}' = \mathbb{S}[\mathbb{X} \cup \{\mathtt{app}, \mathtt{lam}\} \cup \mathbb{C}]$ where $\mathbb{C}$ is a set of constants, such as logical symbols, necessary to formalize our derivation. Then we have $D \in \mathbb{S}'$ and $\Phi(M) \in \mathbb{S}'$. Here, the functionality of the group action is $[\sigma](-) : G_{\mathbb{X}} \times \mathbb{S}' \to \mathbb{S}'$ and we have $[\sigma]\Phi(M) = \Phi([\sigma]M)$. Now, since D proves $\Phi(M)$, we have $[\sigma]D$ proves $[\sigma]\Phi(M) = \Phi([\sigma]M)$ since all the axioms and inference rules of our formalized

system are closed under the group action on $\mathbb{S}'$. For example, the result of group action by $\sigma \in G_{\mathbb{X}}$ on the three rules defining the set $\mathbb{L}$ is:

$$\frac{[\sigma]X : \mathbb{X}}{[\sigma]X : \mathbb{L}} \qquad \frac{[\sigma]M : \mathbb{L} \quad [\sigma]N : \mathbb{L}}{\texttt{(app } [\sigma]M \ [\sigma]N\texttt{)} : \mathbb{L}} \qquad \frac{[\sigma]X : \mathbb{X} \quad [\sigma]M : \mathbb{L}}{\texttt{(lam [}x\texttt{]}[x/[\sigma]X][\sigma]M\texttt{)} : \mathbb{L}} \ (*)$$

They are all instances of the same rules including the side condition $(*)$ since we have $\mathrm{H}_{[\sigma]X}([\sigma]M) = \mathrm{H}_X(M)$.

The essential reason for the validity of the equivariance property is the *indistinguishability* of elements in $\mathbb{X}$. Namely, all we know about $\mathbb{X}$ is that it is disjoint from $\mathbb{N}$ and does not contain `app` or `lam`, and hence we are not able to state in our language a property which holds for a particular element of $\mathbb{X}$ but does not hold for some other elements in $\mathbb{X}$. In contrast with this, consider the transposition τ which transposes `app` and `lam`. Then τ induces an automorphism $[\tau]$ on $\mathbb{S}'$, but this automorphism sends a true statement '$\texttt{(app } X \ X\texttt{)} : \mathbb{L}$' to a false statement '$\texttt{(lam } X \ X\texttt{)} : \mathbb{L}$' for any $X \in \mathbb{X}$.

Given any $M \in \mathbb{S}$, we can decide whether $M \in \mathbb{L}$ or not by induction on $|M|$. For example, if M is of the form $\texttt{(lam [}x\texttt{]}N\texttt{)}$, then,

$$\begin{aligned} M \in \mathbb{L} &\iff N = [x/X]P \text{ for } \textit{some } X \text{ and } P \in \mathbb{L} \\ &\iff [X/x]N \in \mathbb{L} \text{ for } \textit{some } X \notin \mathrm{GV}(N) \\ &\iff [X/x]N \in \mathbb{L} \text{ for } \textit{any } X \notin \mathrm{GV}(N). \end{aligned}$$

The last equivalence is an instance of some/any property (Pitts [20]) whose proof we omit here. So, to decide if $M \in \mathbb{L}$, we have only to take an $X \notin \mathrm{GV}(N)$ and decide if $[X/x]N \in \mathbb{L}$. We can decide this, since $|[X/x]N| = |N| < |M|$. The decision for other cases can be made similarly.

We have the following theorems which guarantee that λ-terms are closed under substitution and instantiation.

Theorem 1. *If P, Q are λ-terms and Y is a global variable, then $[Q/Y]P$ is a λ-term.*

Proof. We argue by induction on the birthday of P and prove by the case of the last rule applied to obtain the derivation of $P : \mathbb{L}$. The only nontrivial case is when P is of the form $\texttt{(lam [}x\texttt{]}[x/X]M\texttt{)}$ and it is derived by the rule:

$$\frac{X : \mathbb{X} \quad M : \mathbb{L}}{\texttt{(lam [}x\texttt{]}[x/X]M\texttt{)} : \mathbb{L}} .$$

where $x = \mathrm{H}_X(M)$.

In this case, by induction hypothesis, we have $[Q/Y]M : \mathbb{L}$ and since

$$[Q/Y]\texttt{(lam [}x\texttt{]}[x/X]M\texttt{)} = \texttt{(lam [}x\texttt{]}[Q/Y][x/X]M\texttt{)},$$

our goal is to prove:

$$\texttt{(lam [}x\texttt{]}[Q/Y][x/X]M\texttt{)} : \mathbb{L}.$$

Here, we have the following three possible cases.

Case 1: $X = Y$. In this case, we have

$$\texttt{(lam [}x\texttt{]}[Q/Y][x/X]M\texttt{)} = \texttt{(lam [}x\texttt{]}[Q/X][x/X]M\texttt{)} = \texttt{(lam [}x\texttt{]}[x/X]M\texttt{)}.$$

So, our goal becomes $\texttt{(lam [}x\texttt{]}[x/X]M\texttt{)} : \mathbb{L}$, which we already know to hold.

Case 2: $X \neq Y$ and $X \notin \mathrm{GV}(Q)$. In this case, by the Substitution Lemma 3, we have

$$\texttt{(lam [}x\texttt{]}[Q/Y][x/X]M\texttt{)} = \texttt{(lam [}x\texttt{]}[x/X][Q/Y]M\texttt{)}$$

since $[Q/Y]x = x$. Moreover, since $X \neq Y$ and $X \notin \mathrm{GV}(Q)$, we have

$$\mathrm{H}_X([Q/Y]M) = \mathrm{H}_X(M) = x$$

by Lemma 5. Hence we can apply the rule:

$$\frac{X : \mathbb{X} \quad [Q/Y]M : \mathbb{L}}{\texttt{(lam [}x\texttt{]}[x/X][Q/Y]M\texttt{)} : \mathbb{L}}$$

and obtain the desired result: $[Q/Y]\texttt{(lam [}x\texttt{]}[x/X]M\texttt{)} : \mathbb{L}$.

Case 3: $X \neq Y$ and $X \in \mathrm{GV}(Q)$. In this case, we choose a fresh global variable Z such that $Z \neq X$, $Z \neq Y$, $Z \notin \mathrm{GV}(M)$ and $Z \notin \mathrm{GV}(Q)$. Then, we can easily see that $[x/X]M = [x/Z][Z/X]M$ by the freshness of Z. Hence, by the Substitution Lemma, we have

$$\begin{aligned}[Q/Y][x/X]M &= [Q/Y][x/Z][Z/X]M \\ &= [[Q/Y]x/Z][Q/Y][Z/X]M \\ &= [x/Z][Q/Y][Z/X]M.\end{aligned}$$

So, our goal now becomes:

$$\texttt{(lam [}x\texttt{]}[x/Z][Q/Y][Z/X]M\texttt{)} : \mathbb{L}.$$

Since $|[Z/X]M| = |M|$, we have $[Q/Y][Z/X]M : \mathbb{L}$ by induction hypothesis. Moreover we have

$$\mathrm{H}_Z([Q/Y][Z/X]M) = \mathrm{H}_X([Q/Y]M) = \mathrm{H}_X(M) = x.$$

Hence, we can now apply the following rule to obtain the desired goal:

$$\frac{Z : \mathbb{X} \quad [Q/Y][Z/X]M : \mathbb{L}}{\texttt{(lam [}x\texttt{]}[x/Z][Q/Y][Z/X]M\texttt{)} : \mathbb{L}}\,.$$

□

Theorem 2. *If* $\texttt{(lam }M\texttt{)}$ *and* N *are* λ*-terms, then so is* $\texttt{(lam }M\texttt{)}(N)$.

We are now ready to define the $\lambda\beta$-calculus on the set $\mathbb{L}$ of λ-terms. First we have the following β-reduction rule.

$$\frac{\texttt{(lam}\ M\texttt{)} : \mathbb{L} \quad N : \mathbb{L}}{\texttt{(app (lam}\ M\texttt{)}\ N\texttt{)} \to_\beta \texttt{(lam}\ M\texttt{)}(N)}$$

We recall that $\texttt{(lam}\ M\texttt{)}(N)$ is the instantiation of $\texttt{(lam}\ M\texttt{)}$ by N. Since $\texttt{(lam}\ M\texttt{)} : \mathbb{L}$ implies that M is of the form $\texttt{[}x\texttt{]}P$, we have $\texttt{(lam}\ M\texttt{)}(N) = [N/x]P$.

The reduction relation $M \to N$ of the $\lambda\beta$-calculus is defined here as the binary relation on $\mathbb{S}$ inductively generated by the following rules.

$$\frac{M \to_\beta N}{M \to N} \qquad \frac{X : \mathbb{X}}{X \to X} \qquad \frac{M \to P \quad N \to Q}{\texttt{(app}\ M\ N\texttt{)} \to \texttt{(app}\ P\ Q\texttt{)}}$$

$$\frac{X : \mathbb{X} \quad M \to N}{\texttt{(lam [}x\texttt{]}[x/X]M\texttt{)} \to \texttt{(lam [}y\texttt{]}[y/X]N\texttt{)}}\ (*) \qquad \frac{M \to N \quad N \to P}{M \to P}$$

Note: The fourth rule $(*)$ may be applied only when the following side condition is met:

The height of X is x in M and y in N.

We need this condition to ensure that the conclusion of the rule indeed becomes a relation on λ-terms. Since the height of X may be different in M and N we have to use maybe different binders x and y as the binders of M and N.

We have the following lemma which is useful when we compute a β-redex inside the scope of a binder.

Lemma 6. *If* $\texttt{(lam [}x\texttt{]}M\texttt{)}$ *is a* λ*-term,* X, Y *are global variables such that* $X \notin \mathrm{GV}(M)$ *and* $Y \notin \mathrm{GV}(M)$*, and* Q *is a* λ*-term, then* $[Q/X][X/x]M = [Q/Y][Y/x]M$*.*

Example 1. We give an example of reduction by considering the reduction of a λ-term which corresponds to the λ-term:

$$(\lambda z.\ (\lambda x.\ (\lambda y.\ zy)(xz)))y$$

in traditional notation. In the traditional language, this term is reduced as follows.

$$(\lambda z.\ (\lambda x.\ (\lambda y.\ zy)(xz)))y \to \lambda x.\ (\lambda w.\ yw)(xy) \to \lambda x.\ y(xy)$$

Note that we renamed the bound variable y to w to avoid capturing of the free variable y.

In order to translate this into our λ-term smoothly, we use the following two functions, $app : \mathbb{L} \times \mathbb{L} \to \mathbb{L}$ and $lam : \mathbb{X} \times \mathbb{L} \to \mathbb{L}$ defined by:

- $app(M, N) \triangleq \texttt{(app}\ M\ N\texttt{)}$.
- $lam(X, M) \triangleq \texttt{(lam [}x\texttt{]}[x/X]M\texttt{)}$ where $x = \mathrm{H}_X(M)$ and

Now, the above term corresponds to the following λ-term.

$$\begin{aligned}
&app(lam(Z, lam(X, app(lam(Y, app(X, Y)), app(X, Z)))), Y)\\
&= app(lam(Z, lam(X, app(lam(Y, \texttt{(app } X\ Y\texttt{)}), \texttt{(app } X\ Z\texttt{)}))), Y)\\
&= app(lam(Z, lam(X, app(\texttt{(lam [1](app } X\ 1\texttt{))}, \texttt{(app } X\ Z\texttt{)}))), Y)\\
&= app(lam(Z, \texttt{(lam [2](app (lam [1](app 2 1)) (app 2 } Z\texttt{)))}), Y)\\
&= app(\texttt{(lam [1](lam [2](app (lam [1](app 2 1)) (app 2 1))))}, Y)\\
&= \texttt{(app (lam [1](lam [2](app (lam [1](app 2 1)) (app 2 1)))) } Y\texttt{)}
\end{aligned}$$

We can compute this term as follows.

$$\begin{aligned}
&\texttt{(app (lam [1](lam [2](app (lam [1](app 2 1)) (app 2 1)))) } Y\texttt{)}\\
&\to \texttt{(lam [2](app (lam [1](app 2 1)) (app 2 } Y\texttt{)))}\\
&= lam(X, \texttt{(app (lam [1](app } X\ 1\texttt{)) (app } X\ Y\texttt{)))}\\
&\to lam(X, \texttt{(app } X\ \texttt{(app } X\ Y\texttt{)))}\\
&= \texttt{(lam [1](app 1 (app 1 } Y\texttt{)))}
\end{aligned}$$

We will use the functions *app* and *lam* in the next section to interpret λ-terms in the external syntax by the internal language. □

4 The external syntax

The data structure of the external syntax we introduce in this section is essentially the same as that of the traditional syntax of λ-terms with named variables. In our formulation of the external syntax we will use only global variables and will not use local variables. Also we do not use the binding structure of B-algebra. The mathematical structure of the external syntax is a simple binary tree structure, and as a price for the simplicity of the structure, the definition of substitution involving α-renaming is much more complex than that for the internal syntax. So, in this section, we will not directly work in the language of the external syntax, but instead we will introduce various notions indirectly by translating the syntactic objects of the external language into the objects of the internal language.

We use the same set $\mathbb{S} = \mathbb{S}[\mathbb{X} \cup \{\texttt{app}, \texttt{lam}\}]$ of symbolic expressions as the base set for defining the set Λ of λ-terms in the external syntax. The set Λ is defined inductively as follows.

$$\frac{X : \mathbb{X}}{X : \Lambda} \qquad \frac{M : \Lambda \quad N : \Lambda}{\texttt{(app } M\ N\texttt{)} : \Lambda} \qquad \frac{X : \mathbb{X} \quad M : \Lambda}{\texttt{(lam } X\ M\texttt{)} : \Lambda}$$

In this section, to distinguish λ-terms in the external syntax from λ-terms in the internal syntax, we will call $M \in \Lambda$ a Λ-*term* and $M \in \mathbb{L}$ an $\mathbb{L}$-*term*.

We define an onto function $[\![-]\!] : \Lambda \to \mathbb{L}$ which, for each $M \in \Lambda$, defines its *denotation* $[\![M]\!] \in \mathbb{L}$ as follows.

1. $[\![X]\!] \stackrel{\triangle}{=} X$.
2. $[\![\texttt{(app } M\ N\texttt{)}]\!] \stackrel{\triangle}{=} app([\![M]\!], [\![N]\!])$.
3. $[\![\texttt{(lam } X\ M\texttt{)}]\!] \stackrel{\triangle}{=} lam(X, [\![M]\!])$.

Note: The surjectivity of $[\![-]\!]$ can be verified by induction on the construction of $M \in \mathbb{L}$.

Our view is that each $M \in \Lambda$ is simply a name of the λ-term $[\![M]\!] \in \mathbb{L}$. It is therefore natural to define notions about M in terms of notions about $[\![M]\!]$. As an example, for any $M \in \Lambda$, we can define $\mathrm{FV}(M)$, *the set of free variables in* M, simply by putting: $\mathrm{FV}(M) \stackrel{\triangle}{=} \mathrm{GV}([\![M]\!])$. After *defining* $\mathrm{FV}(M)$ this way, we can *prove* the following equations which characterize the set $\mathrm{FV}(M)$ in terms of the language of the external syntax.

1. $\mathrm{FV}(X) = \{X\}$.
2. $\mathrm{FV}(\texttt{(app } M\ N\texttt{)}) = \mathrm{FV}(M) \cup \mathrm{FV}(N)$.
3. $\mathrm{FV}(\texttt{(lam } X\ M\texttt{)}) = \mathrm{FV}(M) - \{X\}$.

A Λ-term M is *closed* if $\mathrm{FV}(M) = \{\}$.

Defining the α-equivalence relation on Λ is also straightforward. Given $M, N \in \Lambda$, we define M and N to be *α-equivalent*, written '$M =_\alpha N$', if $[\![M]\!] = [\![N]\!]$. For example, we have

$$\texttt{(lam } X\ \texttt{(lam } Y\ \texttt{(app } X\ Y\texttt{)))} =_\alpha \texttt{(lam } Y\ \texttt{(lam } X\ \texttt{(app } Y\ X\texttt{)))},$$

since

$$\begin{aligned}[\![\texttt{(lam } X\ \texttt{(lam } Y\ \texttt{(app } X\ Y\texttt{)))}]\!] &= lam(X, lam(Y, app(X, Y))) \\ &= lam(X, lam(Y, \texttt{(app } X\ Y\texttt{)})) \\ &= lam(X, \texttt{(lam [1](app } X\ 1\texttt{)))} \\ &= \texttt{(lam [2](lam [1](app 2 1)))}\end{aligned}$$

and we have the same result for $[\![\texttt{(lam } X\ \texttt{(lam } Y\ \texttt{(app } X\ Y\texttt{)))}]\!]$.

We now verify the adequacy (see Harper et al. [14]) of our definition of the α-equivalence against the definition of the α-equivalence due to Gabbay and Pitts [10, 20]. Their definition, in our notation, is as follows.

$$\frac{M : \Lambda}{M =_\alpha M} \qquad \frac{M =_\alpha P \quad N =_\alpha Q}{\texttt{(app } M\ N\texttt{)} =_\alpha \texttt{(app } P\ Q\texttt{)}} \qquad \frac{[X/\!\!/Z]M =_\alpha [Y/\!\!/Z]N}{\texttt{(lam } X\ M\texttt{)} =_\alpha \texttt{(lam } Y\ N\texttt{)}}\ (*)$$

The rule $(*)$ may be applied only when $Z \notin \mathrm{GV}(M) \cup \mathrm{GV}(N)$. The adequacy is established by interpreting these rules in our internal syntax and showing the Soundness and Completeness Theorem 3 below, which is preceded by the following lemma.

Lemma 7. *If* $M =_\alpha N$, *then* $\mathrm{H}_X(M) = \mathrm{H}_X([\![M]\!]) = \mathrm{H}_X([\![N]\!]) = \mathrm{H}_X(N)$ *for all* $X \in \mathbb{X}$.

Proof. By induction on the derivation of $M =_\alpha N$. □

Theorem 3. *The judgment $M =_\alpha N$ is derivable by using the above rules if and only if $[\![M]\!] = [\![N]\!]$.*

Proof. We show the soundness part by induction on $|M|$. We only consider the third rule. Suppose that $[X/\!\!/Z]M =_\alpha [Y/\!\!/Z]N$ and $Z \notin \mathrm{GV}(M) \cup \mathrm{GV}(N)$. By induction hypothesis, we have $[\![[X/\!\!/Z]M]\!] = [\![[Y/\!\!/Z]N]\!]$. Our goal is to show that $[\![\texttt{(lam } X\ M\texttt{)}]\!] = [\![\texttt{(lam } Y\ N\texttt{)}]\!]$. We have

$$[\![\texttt{(lam } X\ M\texttt{)}]\!] = \mathit{lam}(X, [\![M]\!]) = \texttt{(lam [}x\texttt{]}[x/X][\![M]\!]\texttt{)},$$

and

$$[\![\texttt{(lam } Y\ N\texttt{)}]\!] = \mathit{lam}(Y, [\![N]\!]) = \texttt{(lam [}y\texttt{]}[y/Y][\![N]\!]\texttt{)},$$

where $x = \mathrm{H}_X([\![M]\!])$ and $y = \mathrm{H}_Y([\![N]\!])$. Now, by the freshness of Z and by Lemma 7, we have $x = \mathrm{H}_X([\![M]\!]) = \mathrm{H}_Z([X/\!\!/Z][\![M]\!]) = \mathrm{H}_Z([Y/\!\!/Z][\![N]\!]) = \mathrm{H}_Y([\![N]\!]) = y$. So, letting $z = x = y$, we will be done if we can show that $[x/X][\![M]\!] = [y/Y][\![N]\!]$. This is indeed the case since:

$$\begin{aligned}
&[\![[X/\!\!/Z]M]\!] = [\![[Y/\!\!/Z]N]\!] \\
\Longrightarrow\ &[X/\!\!/Z][\![M]\!] = [Y/\!\!/Z][\![N]\!] \quad \text{(by equivariance)} \\
\Longrightarrow\ &[z/Z][X/\!\!/Z][\![M]\!] = [z/Z][Y/\!\!/Z][\![N]\!] \quad \text{(by freshness of } Z\text{)} \\
\Longrightarrow\ &[X/\!\!/Z][x/X][\![M]\!] = [Y/\!\!/Z][y/Y][\![N]\!] \quad \text{(by Permutation Lemma)} \\
\Longrightarrow\ &[x/X][\![M]\!] = [y/Y][\![N]\!] \quad \text{(by GV Lemma, freshness of } Z\text{)}.
\end{aligned}$$

The completeness part is also proved by induction on $|M|$. We consider only the case where $[\![M]\!] = [\![N]\!]$ is of the form $\texttt{(lam [}z\texttt{]}[z/Z]P\texttt{)}$ with $P \in \mathbb{L}$ and $z = \mathrm{H}_Z(P)$.

In this case, $M = \texttt{(lam } X\ M'\texttt{)}$ for some X, M' and $N = \texttt{(lam } Y\ N'\texttt{)}$ for some Y, N'. Hence, $[\![M]\!] = \texttt{(lam [}z\texttt{]}[z/X][\![M']\!]\texttt{)}$ and $[\![N]\!] = \texttt{(lam [}z\texttt{]}[z/Y][\![N']\!]\texttt{)}$, so that we have $[z/X][\![M']\!] = [z/Y][\![N']\!]$. Hence, we have

$$\begin{aligned}
&[z/X][\![M']\!] = [z/Y][\![N']\!] \\
\Longrightarrow\ &[X/\!\!/Z][z/X][\![M']\!] = [Y/\!\!/Z][z/Y][\![N']\!] \\
\Longrightarrow\ &[z/Z][\![[X/\!\!/Z]M']\!] = [z/Z][\![[Y/\!\!/Z]N']\!] \\
\Longrightarrow\ &[Z/z][z/Z][\![[X/\!\!/Z]M']\!] = [Z/z][z/Z][\![[Y/\!\!/Z]N']\!] \\
\Longrightarrow\ &[\![[X/\!\!/Z]M']\!] = [\![[Y/\!\!/Z]N']\!] \quad \text{(by Height Lemma)} \\
\Longrightarrow\ &[X/\!\!/Z]M' =_\alpha [Y/\!\!/Z]N' \quad \text{(by induction hypothesis)} \\
\Longrightarrow\ &M =_\alpha N.
\end{aligned}$$

□

We can at once obtain the transitivity of the α-equivalence relation by this theorem. This gives a semantical proof of a syntactical property of Λ-terms.

We now turn to the definition of substitution on Λ-terms. Since we can define substitution only modulo $=_\alpha$, we define substitution not as a function but as a relation

$$[N/X]M \Downarrow P$$

on $\Lambda \times \mathbb{X} \times \Lambda \times \Lambda$ which we read '*the result (modulo* $=_\alpha$*) of substituting* N *for* X *in* M *is* P'. The substitution relation is defined by the following rules. The fifth rule $(*)$ below may be applied when $Y \notin \mathrm{FV}(P)$.

$$\frac{P : \Lambda}{[P/X]X \Downarrow P} \qquad \frac{P : \Lambda \quad X \neq Y}{[P/X]Y \Downarrow Y} \qquad \frac{[P/X]M \Downarrow M' \quad [P/X]N \Downarrow N'}{[P/X]\texttt{(app } M\ N\texttt{)} \Downarrow \texttt{(app } M'\ N'\texttt{)}}$$

$$\frac{}{[P/X]\texttt{(lam } X\ M\texttt{)} \Downarrow \texttt{(lam } X\ M\texttt{)}} \qquad \frac{[P/X]M \Downarrow N \quad X \neq Y}{[P/X]\texttt{(lam } Y\ M\texttt{)} \Downarrow \texttt{(lam } Y\ N\texttt{)}}\ (*)$$

$$\frac{\texttt{(lam } Y\ M\texttt{)} =_\alpha \texttt{(lam } Z\ N\texttt{)} \quad [P/Z]\texttt{(lam } Z\ N\texttt{)} \Downarrow Q}{[P/X]\texttt{(lam } Y\ M\texttt{)} \Downarrow Q}$$

The substitution relation we just defined enjoys the following soundness and completeness theorems.

Theorem 4 (Soundness of Substitution).
If $[N/X]M \Downarrow P$*, then* $[[\![N]\!]/X][\![M]\!] = [\![P]\!]$.

Theorem 5 (Completeness of Substitution). *If* $[N'/X]M' = P'$ *in* $\mathbb{L}$*, then* $[N/X]M \Downarrow P$*,* $[\![M]\!] = M'$*,* $[\![N]\!] = N'$ *and* $[\![P]\!] = P'$ *for some* $N, M, P \in \Lambda$.

By these theorems, we can see that for any N, X, M we can always find a P such that $[N/X]M \Downarrow P$ and all such Ps are α-equivalent with each other.

We omit the development of $=_{\alpha\beta}$ relation on Λ which is a routine work by now.

5 Conclusion

We have introduced the notion of a B-algebra as a magma with an additional operation of local variable binding, and defined the set $\mathbb{S} = \mathbb{S}[\mathbb{X}]$ of symbolic expressions over a set $\mathbb{X}$ of global variables as the free B-algebra with the free generating set $\mathbb{X}$. This setting allowed us to define (simultaneous) substitutions as endomorphisms on $\mathbb{S}$ and permutations as automorphisms on $\mathbb{S}$. As far as we know, this is the first algebraic formulation of *substitution as homomorphism* applicable to symbolic expressions with a variable binding mechanism.

We conclude the paper by comparing our formulation with that by Gabbay-Pitts [10], that by Aydemir et al. [1] and finally those by Quine [22], Bourbaki [3], Sato-Hagiya [23] and Sato [24].

The formulation by Gabbay-Pitts uses FM-set theory over a set of atoms and atoms play the role of variables when they *implement* λ-terms in FM-set theory. Since FM-set theory is close to standard ZFC-set theory except for the indistinguishability of atoms and failure of the axiom of choice, their construction of

λ-terms is set-theoretic and non-constructive, although induction principle for so constructed λ-terms can be introduced and proven to be correct. The λ-terms defined in this way is shown to be isomorphic to the standard λ-terms in Λ modulo α-equivalence. A good point of this formulation is that capture avoiding substitution can be manipulated *rigorously* using arguments similar to standard informal arguments on λ-terms modulo α-equivalence. As pointed out in Section 1, standard informal arguments are often very difficult to formalize rigorously. They use the equivariance property under finite permutations of atoms extensively. Pitts later introduced the notion of *nominal sets* [20, 21] and showed that essentially the same results can be obtained within the framework of standard mathematics.

In contrast with this, our formulation of λ-terms in the internal syntax use two sorts of variables, and define λ-terms constructively by inductive rules of construction. We also use the equivariance property of permutations extensively, but, for us, a permutation is just a special instance of more general notion of the simultaneous substitution. In our setting, substitutions and permutations are endomorphisms and automorphisms on $\mathbb{S}$, respectively, and all the substitutions on λ-terms are always capture avoiding with no need of renaming local variables.

The formulation by Aydemir et al. uses two sorts of variables, one for global variables and the other for local variables just like our internal syntax. However, they use de Bruijn indices for local variables, so that their local variables are *nameless* while ours have explicit names (natural numbers are names!). Their binders do not have names but ours have names. In spite of this difference, substitution of a term for a global variable goes as smoothly as our case since both formulations use two sorts of variables. However, their subsitution operations are not characterized as homomorphisms due to the lack of algebraic structure on their terms.

Another difference concerns the formation rules of abstraction. To explain the difference, we note that our introduction rule of abstracts could equivalently formulated, in a backward way so to speak, as follows.

$$\frac{X : \mathbb{X} \quad [X/x]M : \mathbb{L}}{\texttt{(lam [}x\texttt{]}M\texttt{)} : \mathbb{L}} \ (*)$$

Note: The rule $(*)$ may be applied only if $X \notin \mathrm{GV}(M)$ and $x = \mathrm{H}_X([X/x]M)$.

Although this is a technically correct rule, we must say that this rule is unnatural from our *ontological point of view*. This is because in order to apply this rule and obtain a new λ-term as the result of the application, we must somehow know the very λ-term we wish to construct. As we already stressed in [27], we believe that every *mathematical object*, including of course every λ-term, must be constructed by applying a *constructor* function to *already created objects*. But, this rule does not follow this *ontological condition*, and this is why we did not adopt the above rule but instead adopted the abstraction formation rule in Section 3. Now, if formulated in the style of Aydemir et al. [1], the rule for constructing abstracts

in $\mathbb{L}$ would become like this (cf. the TYPING-ABS rule in [1, Figure 1]):

$$\frac{X : \mathbb{X} \quad M^X : \mathbb{L}}{\texttt{(lam } M\texttt{)} : \mathbb{L}} \ (**)$$

Just like our rule $(*)$, the rule $(**)$ may be applied only when $X \notin \mathrm{GV}(M)$. In this rule, local variables are represented by de Bruijn indices, and $\lambda x.\, x\lambda y.\, yx$, for instance, becomes

```
(lam (app 0 (lam (app 0 1))))
```

while it becomes

```
(lam [2](app 2 (lam [1](app 1 2))))
```

in our formulation. The term M^X in the second premise of the rule $(**)$ is the *opening up* of M by X which corresponds to our instantiation of `(lam [`x`]`M`)` by X, namely, $[X/x]M$. So continuing our example, opening up by X and instantiation by X, respectively, becomes

`(app` X `(lam (app 0` X`)))` and `(app` X `(lam [1](app 1` X`)))`.

Note that in opening up `(app 0 (lam (app 0 1)))` by X we had to replace 0 by X in one place and 1 by X in another place while `[2](app 2 (lam [1](app 1 2)))` could be instantiated by X just by substituting X for two occurrences of 2.

We may thus say that the representation of λ-terms by the method of [1] is more complex than our method and that what we presume to be their rule for *introducing* a new λ-term is ontologically unnatural as it requires us to *mentally* construct the term beforehand. We note, however, that it is possible to replace the rule $(**)$ with an ontologically natural rule which parallels our rule we gave in Section 3. See 4.5 of Aydemir et al. [1] for such a rule where they examine a rule given by Gordon [13]. They did not adopt this rule for technical reasons. We also remark that our internal syntax is more human friendly than that of [1] and hence, if we so wish, can be used as an external syntax replacing the external syntax we gave in Section 4.

Finally, we remark that, as a data structure, our representation of expressions with binders is, in a sense, isomorphic to those by Quine [22, page 70], Bourbaki [3], Sato-Hagiya [23] and Sato [24, 26]. Their representations are nameless since abstraction is realized by providing links between the binding node and the nodes which refer back to the binding node. These representations are usually implemented on a computer by realizing links in terms of pointers. However, except for Sato and Hagiya [23] and Sato [24, 26], these data structures do not admit well-founded induction principle, since these data structures contain *cycles*. Unlike these, our representation admits reasoning by induction on the birthday of each expression, and has a nice algebraic structure.

Acknowledgments

I wish to thank René Vestergaard and Murdoch Gabbay for fruitful discussions on the mechanism of variable binding.

References

1. Aydemir, B., Charguéraud, A., Pierce, B.C., Pollack R. and Weirich, S., Engineering Formal Metatheory, In *POPL '08: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles on Programming Languages*, ACM Press, 2008.
2. Barendregt, H., *The Lambda Calculus*, North-Holland, 1984.
3. Bourbaki, N., *Theory of Sets*, Hermann, 1968.
4. Church, A., A Formulation of the Simple Theory of Types, *Juornal of Symbolic Logic*, **5**, pp. 56–68, 1940.
5. Church, A., *The Calculi of Lambda Conversions*, Princeton University Press, 1941.
6. de Bruijn, N.G. Lambda Calculus Notation with Nameless Dummies, A Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem, *Indag. Math.*, **34**, pp. 381–392, 1972.
7. Fiore, M., Plotkin, G. and Turi, D., Abstract Syntax and Variable Binding, Proc. 14th Annual IEEE Symposium on Logic in Computer Science, pp. 193–202, 1999.
8. Frege, G., *Begriffsschrift, eine der arithmetishen nachgebildete Formelsprache des reinen Denkens*, Halle, 1879. (English translation in [15])
9. Gabbay, M.J. and Pitts, A.M., A new approach to abstract syntax involving binders, in *14th Annual Symposium on Logic in Computer Science*, pp. 214–224, IEEE Computer Society Press, 1999.
10. Gabbay, M.J. and Pitts, A.M., A new approach to abstract syntax involving binders, *Formal Aspects of Computing*, 2002.
11. Gentzen, G., Untersuchungen über das logische Schließen, I, Mathematische Zeitschrift, **39**, pp. 175–210, 1935, English translation in [29, pp. 68 – 131].
12. Gödel, K., Über formal unentscheidbare Sätze der Pricipia mathematica und verwandter Systeme I, *Monatshefte für Mathematik und Physik*, **38**, pp. 173–198, 1931, English translation in [15].
13. Gordon, A.D., A mechanisation of name-carrying syntax up to alpha-conversion, in Joyce, J.J. and Seger, C.-J.H. (eds.), *Higher-order Logic Theorem Proving and its Applications, Proceedings, 1993*, **780**, Lecture Notes in Computer Science, pp. 414–426, Springer, 1994.
14. Harper, R., Honsell, R. and Plotkin, G., A framework for defining logics, *Journal of the ACM*, **40**, pp. 143–184, 1993.
15. Heijenoort, J.v. (ed.), *From Frege to Gödel, A Source Book in Mathematical Logic, 1879 – 1931*, Harvard University Press, 1977.
16. McCarthy, J., Recursive Functions of Symbolic Expressions and their Computation by Machine (Part 1), *Comm. ACM*, **3**, 184–195, 1960.
17. McCarthy, J., A basis for a mathematical theory of computation, in P. Braffort and D. Hirschberg (eds.), *Computer Programming and Formal Systems*, pp. 33 – 70, North-Holland, 1963.
18. McKinna, J. and Pollack, R., Some Lambda Calculus and Type Theory Formalized, *Journal of Automated Reasoning*, **23**, pp. 373–409, 1999.
19. Nordström, B., Petersson, K. and Smith, J.M., *Programming in Martin-Löf's Type Theory*, Oxford University Press, 1990.
20. Pitts, A.M., Nominal logic: a first order theory of names and binding, *Information and Computation*, **186**, pp. 165–193, 2003.
21. Pitts, A.M., Alpha-Structural Recursion and Induction J. ACM, **53**, pp. 459 – 506, 2006.
22. Quine, W., *Mathematical Logic (Revised Edition)*, MIT Press, 1951.

23. Sato M. and Hagiya, M., Hyperlisp, *Proceedings of the International Symposium on Algorithmic Language*, 251-269, North-Holland, 1981.
24. Sato M., Theory of Symbolic Expressions, I, *Theoretical Computer Science*, **22**, 19-55, 1983.
25. Sato M., Theory of Symbolic Expressions, II, *Publ. RIMS, Kyoto Univ.*, **21**, 455–540, 1985.
26. Sato M., An Abstraction Mechanism for Symbolic Expressions, 1991, in V. Lifschitz ed., *Artificial Intelligence and Mathematical Theory of Computation (Papers in Honor of John McCarthy)*, Academic Press, 381-391, 1991.
27. Sato M., Theory of Judgments and Derivations, in Arikawa, S. and Shinohara, A. eds., *Progress in Discovery Science*, Lecture Notes in Artificial Intelligence **2281**, pp. 78 – 122, Springer, 2002.
28. Sato M., A Framework for Checking Proofs Naturally, *Journal of Intelligent Information Systems*, to appear.
29. Szabo, M.E. (ed.), *The collected papers of Gerhard Gentzen*, North-Holland, 1969.
30. Urban, C., Berghfer, S. and Norrish, M., *Barendregt's Variable Convention in Rule Induction*, in *Proc. of the 21st International Conference on Automated Deduction (CADE)*, Lecture Notes in Artificial Intelligence, **4603**, pp. 35–50, 2007.
31. Vestergaard, R., *The Primitive Proof Theory of the λ-Calculus*, Ph.D Thesis, Heriot-Watt University, 2003.
32. Whitehead, A.N. and Russell, B., *Principia mathematica, vol. 1*, Cambridge University Press, 1910.

Author Index