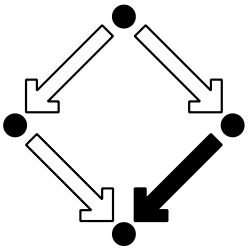
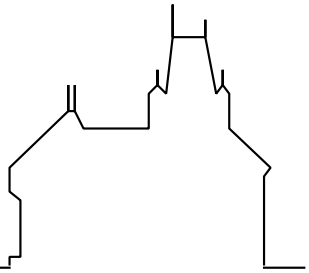


RISC-Linz

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



Austria-Japan Workshop on Symbolic Computation and Software Verification

Temur KUTSIA, Mircea MARIN (eds.)

Linz, Austria
July 1, 2007

RISC-Linz Report Series No. 07-09

Supported by: Austrian Science Foundation (FWF), Japan Society for the Promotion of Science (JSPS).

Preface

Austria-Japan Workshops on Symbolic Computation and Software Verification aim at strengthening cooperation between the research groups at the Research Institute for Symbolic Computation (RISC) of the Johannes Kepler University of Linz, University of Tsukuba, and Kyoto University working on theory and practice of symbolic computation for reliable software development. Symbolic computation methods include those of computational logic and computer algebra. Reliable software development involves verification and synthesis of programs/algorithms, and verification and validation of XML documents that is closely related to knowledge management.

This report collects the extended abstracts of the talks given at the first workshop, organized by RISC on July 1, 2007 in Linz, Austria. The members of the partner research groups present position statements, reports on their recent work, or work-in-progress on applications of symbolic computation methods in reliable software development. The workshop has been collocated with the RISC Summer 2007.

July 2007

Temur Kutsia
Mircea Marin

Organization

The first Austria-Japan Workshop on Symbolic Computation and Software Verification is organized by the Research Institute for Symbolic Computation, Johannes Kepler University of Linz on July 1, 2007.

Workshop Organizers

Temur Kutsia	RISC, Johannes Kepler University, Austria
Mircea Marin	University of Tsukuba, Japan

Speakers

Bruno Buchberger	Yukiyoshi Kameyama	Koji Nakazawa
Madalina Erascu	Laura Kovács	Florina Piroi
Martin Giese	Mircea Marin	Masahiko Sato
Tetsuo Ida	Yasuhiko Minamide	

Sponsoring Institutions

Austria-Japan workshops in Symbolic Computation and Software Verification are supported by the Austrian Science Fund (FWF) and JSPS in the frame of JSPS-FWF Bilateral Joint Projects.

Table of Contents

Austria-Japan Workshop on Symbolic Computation and Software Verification

Ordinals and Sets: Two Sides of the Same Coin

Masahiko Sato

Graduate School of Informatics
Kyoto University, Japan
masahiko@kuis.kyoto-u.ac.jp

We introduce a new way of looking at ordinals and sets. Owing to the great success of Zermelo-Fraenkel set theory ZF as a foundational system for describing mathematics, it is now almost taken for granted that the notion of *set* is the most basic notion in mathematics. Indeed, formally speaking, every mathematical object is a set in ZF. In this paper, however, we show that another view of sets is possible. This is done by introducing the notion of *infon*. Intuitively speaking, an infon is a sequence of bits whose length is arbitrary but bounded by an ordinal. Using the notion of infon, which we believe to be more fundamental than that of set, we will show that it is possible to view an infon either as an ordinal or as a set. Thus the totality of infons, so to speak, makes up a coin whose one side consists of ordinals and the other side consists of sets.

Checking the Balancedness of a Context-Free Language in Polynomial Time (Extended Abstract)

Akihiko Tozawa[1] and Yasuhiko Minamide[2]

[1] IBM Research
Tokyo Research Laboratory, IBM Japan, Ltd.
[2] Department of Computer Science
University of Tsukuba, Japan
minamide@score.cs.tsukuba.ac.jp

1 Introduction

We develop an algorithm which decides in polynomial time whether or not the language of a context-free grammar is balanced. The problem is related to the static analysis of programs generating XML strings [4]. The algorithm was developed by the authors in [7], and this extended abstract gives a slightly revised account of the algorithm.

Let A be a base alphabet. Then, we introduce a paired alphabet consisting of two sets $\acute{A}$ and $\grave{A}$:

$$\acute{A} = \{\, \acute{a} \mid a \in A \,\} \qquad \grave{A} = \{\, \grave{a} \mid a \in A \,\}$$

where $\acute{A}$ and $\grave{A}$ correspond to the set of start tags and the set of end tags, respectively. We consider that $\acute{a}$ and $\grave{a}$ match. We write Σ for $\acute{A} \cup \grave{A}$. Then the fundamental notion on a string over a paired alphabet is whether it is balanced. For example, $\acute{a}\acute{b}\grave{b}\acute{c}\grave{c}\grave{a}$ and $\acute{a}\grave{a}\acute{b}\grave{b}$ are balanced, but $\acute{a}\grave{b}$ and $\acute{a}\acute{b}\grave{b}$ are not. This notion of balanced strings corresponds to well-formed documents in XML. We develop an algorithm that decides whether or not the language of a given context-free grammar (CFG) G is balanced. To our knowledge, the best known algorithm for this problem requires exponential time. However this is not optimal. We will prove that this problem in actually in PTIME.

This PTIME algorithm consists of two steps. The first step is to check the balancedness of the language as a grammar of a single kind of parentheses. We here use a fixpoint algorithm based on the algorithms by Knuth [3], and Berstel and Boasson [1]. However, we give a finer analysis of the number of iterations needed to reach fixpoint, which at first glance seems to be exponential to the size of the grammar, but in fact it is linear. The second step is to check each matched letters are of the same kind. Consider a CFG G with a singleton language. Such a CFG is sometimes called a straight line program (SLP). Assume that in this G, we have a production rule $I \to XY$ whose X and Y derive strings $\phi \in \acute{A}^*$ and $\psi \in \grave{A}^*$, respectively, of the same length. Now clearly, the singleton language

of G is balanced if ϕ is identical to the reverse of ψ by ignoring $\acute{}$ and $\grave{}$ signs. Plandowski has shown that the problem of deciding the equivalence of two SLPs, and hence the balancedness of this $L(G)$, is in PTIME [5]. We later show how to apply Plandowski's algorithm to the problem for general CFGs.

2 PTIME Balancedness Check

The algorithm has two steps. In the first step, we only check *shapes* of strings. The language of a grammar is shape-balanced iff it is balanced by treating it as if using a single pair of parentheses, e.g., $\acute{a}\grave{b}$ is not balanced, but shape-balanced. In this step, we also pick up a *maximal* string in the set of strings produced from each nonterminal. The second step is the check of *color-balancedness*, i.e., we never see corresponding $\acute{a}$ and $\grave{b}$ such that $a \neq b$.

We assume a grammar $G = (\Sigma, V, R, I)$ such that for each $X \in V$, exactly one production rule is defined in R and has its form either $X \to \alpha\beta$ or $X \to \alpha|\beta$ where $\alpha, \beta \in \Sigma \uplus V$. By a notation $C[]$, we mean a string containing a hole, which is filled by ϕ as $C[\phi]$.

2.1 Checking Balancedness of Shapes

We say a string ϕ is shape-balanced if repeatedly removing all matching $\acute{a}\grave{b}$ from ϕ gives an empty string. Any string $\phi \in \Sigma^*$ is reduced to the following form with this reduction.

$$\grave{a}_i \cdots \grave{a}_1 \acute{b}_1 \cdots \acute{b}_j$$

We identify the shape of ϕ by $c(\phi)$ and $d(\phi)$ defined as $c(\phi) = j - i$ and $d(\phi) = i$. A string ϕ is shape-balanced iff $c(\phi) = d(\phi) = 0$. The language $L(G)$ of a grammar G is shape-balanced if all strings in the language are shape-balanced.

If a grammar has a shape-balanced language, each language corresponding to its nonterminal X has constant $c(\phi)$, i.e., $c(\phi) = c(\psi)$ if $X \stackrel{*}{\Rightarrow} \phi, \psi$. We can easily check this condition in polynomial time by exploiting $c(\phi_1\phi_2) = c(\phi_1) + c(\phi_2)$. Hereafter in this paper, we assume that a CFG satisfies this condition and write $c(X)$ for that constant for X.

Example 1. Consider the following grammar generating $\{ (\grave{a}\grave{b}\acute{b}\acute{a})^n \grave{a}\acute{a} \mid n \geq 0 \}$.

$$\begin{array}{lll} X \to X_1 | X_2 & X_1 \to \grave{a}\acute{a} & X_2 \to Y Z_0 \\ Z_0 \to Z_1 X & Y \to \grave{a}\grave{b} & Z_1 \to \acute{b}\acute{a} \end{array}$$

We have $c(X) = c(X_1) = c(X_2) = 0$, $c(Z_0) = c(Z_1) = 2$, and $c(Y) = -2$.

The second observation is that for each nonterminal X there is a bound m such that $X \stackrel{*}{\Rightarrow} \phi$ implies $d(\phi) \leq m$. To see this, consider the derivation $I \stackrel{*}{\Rightarrow} C[X]$. This $C[]$ must be in the form $\psi_0\acute{a}_1\psi_1 \cdots \acute{a}_i\psi_i[]\zeta_j\grave{b}_j \cdots \grave{b}_1\zeta_0$ with $\psi_0, \ldots, \psi_i$, $\zeta_0, \ldots, \zeta_j$ shape-balanced. Since $c(C[\phi]) = d(C[\phi]) = 0$ for any $X \stackrel{*}{\Rightarrow} \phi$, we have $c(\phi) = j - i$ and $d(\phi) \leq i$. The bound of $d(\phi)$ implies the existence of, not

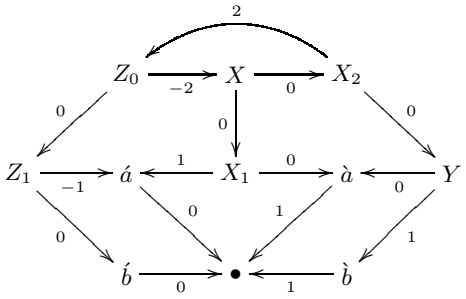
necessarily unique, element ϕ_X such that $X \stackrel{*}{\Rightarrow} \phi_X$ with maximum depth $d(\phi_X)$. To check the balancedness of a grammar, we compute $max\{\ d(\phi) \mid X \stackrel{*}{\Rightarrow} \phi\ \}$ and ϕ_X with the maximum depth for each X by applying a graph algorithm. The graph is constructed based on the following property $c(\phi)$ and $d(\phi)$ given by Knuth.

$$d(\phi_1\phi_2) = max(d(\phi_1), d(\phi_2) - c(\phi_1))$$

We construct a labeled directed graph (W, E). The set of vertexes W is $\Sigma \uplus V \uplus \{\bullet\}$. An edge is (α, l, β) where $\alpha, \beta \in W$ and the label l is a pair of a natural number and a context over $\Sigma \uplus V$. We include (α, l, β) in E if one of the following conditions holds.

- $\alpha \to \beta \in R$ and $l = (0, [])$
- $\alpha \to \beta\beta' \in R$ and $l = (0, []\beta')$
- $\alpha \to \beta'\beta \in R$ and $l = (-c(\beta'), \beta'[])$
- $\alpha = \acute{a}$, $\beta = \bullet$, and $l = (0, [])$.
- $\alpha = \grave{a}$, $\beta = \bullet$, and $l = (1, [])$.

Example 2. For the grammar above, we obtain the following graph by the construction where the context components of the labels are omitted.

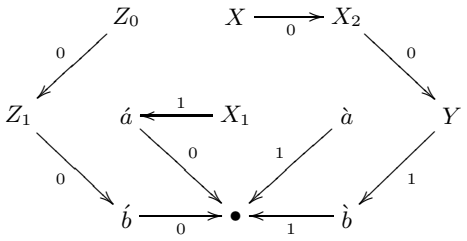


Proposition 1. *Let G be a CFG with $c(I) = 0$. Then, G is shape-balanced iff a longest path from I to $\bullet$ exists in its graph and has length 0.*

Because we assume a CFG is reduced, there is a path from I to any node and a path from any node to $\bullet$. Thus, the length of paths from I to $\bullet$ is bounded iff the graph has no positive cycle. Moreover, a longest path to $\bullet$ exists for any node in the graph if it has no positive cycle.

We can compute the longest paths from all nodes to $\bullet$ and construct a tree of longest paths in $O(|E||W|)$ by Bellman-Ford algorithm. Bellman-Ford algorithm computes shortest paths from a single source to all nodes for a graph where edges may have negative weights. It can also check weather the graph has negative cycle. By converting the sign of weights, we can apply this algorithm to our problem. The following is a tree of longest paths which is constructed by the

algorithm for the CFG above.



2.2 Straight Line Programs for ϕ_X

If we concern only about the shape-balancedness, it is enough to compute only shapes of ϕ_X. This can be done by Bellman-Ford algorithm as we explained in the previous section. However for the color-balancedness, we need to compute ϕ_X with maximum $d(\phi_X)$. Unfortunately, in the worst case, they are not of polynomial length, so that we cannot directly represent them as true strings. However, there is a more compact representation called a straight line program.

Definition 1. *A straight line program (SLP) is an acyclic CFG without alternatives in production rules.*

We shall give an algorithm to find each ϕ_X as an SLP.

First, we construct a set of production rules U of size linear to the size of the grammar with the following properties.

- U satisfies the conditions of SLPs.
- For each $Y \in V$, $Y \stackrel{*}{\Rightarrow} \phi$ in U for some ϕ.

Second, we construct a set of rules P based on the tree of longest paths obtained by Bellman-Ford algorithm. We introduce a fresh nonterminal $\overline{X}$ for each nonterminal X and, for each edge from α to $\beta(\neq \bullet)$ in the tree, add an production rule as follows.

- add $\overline{\alpha} \to \overline{\beta}$ for an edge with label $(n, [])$.
- add $\overline{\alpha} \to \beta'\overline{\beta}$ for an edge with label $(n, \beta'[])$.
- add $\overline{\alpha} \to \overline{\beta}\beta'$ for an edge with label $(n, []\beta')$.

where we let $\overline{\sigma} = \sigma$. This process does not construct a cycle in P.

By construction, the following SLP only has the size linear to the original grammar G.

Algorithm 2.1 We obtain the SLP for ϕ_X as $(\Sigma, V \uplus \overline{V}, U \uplus P, \overline{X})$.

Example 3. We obtain the following grammar for the CFG in the previous section.

$$
\begin{array}{lll}
\overline{X} \to \overline{X_2} & \overline{X_1} \to \grave{a}\acute{a} & \overline{X_2} \to \overline{Y}Z_0 \\
\overline{Z_0} \to \overline{Z_1}X & \overline{Y} \to \grave{a}\grave{b} & \overline{Z_1} \to \acute{b}\acute{a} \\
X \to X_1 & X_1 \to \grave{a}\acute{a} & X_2 \to YZ_0 \\
Z_0 \to Z_1X & Y \to \grave{a}\grave{b} & Z_1 \to \acute{b}\acute{a}
\end{array}
$$

By eliminating useless productions for ϕ_X, we obtain the following grammar generating $\grave{a}\grave{b}\acute{b}\acute{a}\grave{a}\acute{a}$.

$$\begin{array}{llll} \overline{X} \to \overline{X_2} & \overline{X_2} \to \overline{Y} Z_0 & \overline{Y} \to \grave{a}\grave{b} & \\ X \to X_1 & X_1 \to \grave{a}\acute{a} & Z_0 \to Z_1 X & Z_1 \to \acute{b}\acute{a} \end{array}$$

2.3 Checking Color-Balancedness

The remaining step of the balancedness check is to confirm that each pair of coupled parentheses in strings is of the same color, i.e., of the same base letter in A. A color-balanced string ϕ is factorized as $\phi_{-n}\grave{a}_n\phi_{n-1}\cdots\grave{a}_1\phi_0\acute{b}_1\phi_1\cdots\acute{b}_m\phi_m$ such that $\phi_{-n},\ldots,\phi_m$ are balanced. Even an arbitrary string can be similarly factorized by allowing $\phi_{-n},\ldots,\phi_m$ to be shape-balanced. According to this factorization, we define

$$\rho(\phi) = \grave{a}_n\cdots\grave{a}_1\acute{b}_1\cdots\acute{b}_m$$

Next we define an ordering $\sqsubseteq$ as the minimal one satisfying

$$\phi\psi \sqsubseteq \phi\grave{a}\acute{a}\psi$$

for $\phi \in \grave{A}^*$ and $\psi \in \acute{A}^*$. We extend this to a quasi-ordering $\phi \sqsubseteq \psi \Leftrightarrow \rho(\phi) \sqsubseteq \rho(\psi)$. Note that $\phi \sqsubseteq \psi$ implies $\phi \leq \psi$. Now, the remaining part of the algorithm is fairly simple.

Algorithm 2.2 (Balancedness Check) Assume that we already computed a maximal ϕ_X for each nonterminal of the given grammar $G = (\Sigma, V, R, I)$. For each $X \in V$, we check the following:

- ϕ_X is color-balanced.
- $\phi_\alpha\phi_\beta$ is color-balanced and $\phi_X \sqsupseteq \phi_\alpha\phi_\beta$, if $X \to \alpha\beta \in R$.
- $\phi_X \sqsupseteq \phi_\alpha$ and $\phi_X \sqsupseteq \phi_\beta$, if $X \to \alpha|\beta \in R$.

where we let $\phi_\sigma = \sigma$.

Finally, we need to confirm that for strings given as SLPs, both the color-balancedness and the check of $\phi \sqsubseteq \psi$ are decidable in PTIME. Fortunately, as noted in the introduction, we can use Plandowski's algorithm deciding the equivalence of two SLPs in PTIME.

Let ϕ^o be strings created from ϕ just by taking letters in $\acute{A}$ and removing $\acute{}$ sign. Let ϕ^c be strings created from ϕ similarly for $\grave{A}$ but in addition, by reversing the obtained string. For example, $(\grave{a}\grave{b}\acute{c})^o = c$ and $(\grave{a}\grave{b}\acute{c})^c = ba$. We use $\phi[j,k]$ to denote a substring of ϕ starting from its j-th letter and ending before the k-th letter. We also use an extension of SLP called a composition system (CS). A CS is an SLP that allows occurrences of nonterminals in the form $X[j,k]$ in the rhs of productions. It is known that the equivalence problem of two CSs also has a PTIME algorithm, since a CS can always be converted back into a polynomial-size SLP [2, 6].

We developed in [7] an algorithm that computes CSs for an SLP generating ϕ such that $I^c \stackrel{*}{\Rightarrow} \rho(\phi)^c$ and $I^o \stackrel{*}{\Rightarrow} \rho(\phi)^o$. Then, we can check color-balancedness and $\phi \sqsubseteq \psi$ by applying the algorithm checking CS equivalence to these CSs.

Example 4. For the grammar of ϕ_X, we obtain the following production rules for $\overline{X}^o$.

$$\begin{array}{llll} \overline{X} \to \overline{X_2} & \overline{X_2} \to \overline{Y}[0,0-1]Z_0 & \overline{Y} \to \grave{a}\grave{b} & \\ X \to X_1 & X_1 \to a & Z_0 \to Z_1[0,2-1]X & Z_1 \to ba \end{array}$$

This is simplified as follows.

$$\overline{X} \to \overline{X_2} \quad \overline{X_2} \to Z_0 \quad X \to X_1 \quad X_1 \to a \quad Z_0 \to Z_1[0,1]X \quad Z_1 \to ba$$

References

1. J. Berstel and L. Boasson. Formal properties of XML grammars and languages. *Acta Informatica*, 38(9):649–671, 2002.
2. Ch. Hagenah. *Gleichungen mit regulären Randbedingungen über freien Gruppen*. PhD thesis, Universität Stuttgart, Fakultät Informatik, 2000.
3. D. E. Knuth. A characterization of parenthesis languages. *Information and Control*, 11(3):269–289, 1967.
4. Y. Minamide and A. 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.
5. W. Plandowski. Testing equivalence of morphisms on context-free languages. In *Algorithms – ESA '94 (Utrecht)*, volume 855 of *LNCS*, pages 460–470. Springer, 1994.
6. S. Schleimer. Polynomial-time word problems, 2006. Available at http://front.math.ucdavis.edu/math.GR/0608563.
7. A. Tozawa and Y. Minamide. Complexity results on balanced context-free languages. In *Proc. of Tenth International Conference on Foundations of Software Science and Computational Structures*, volume 4423 of *LNCS*, pages 346–360, 2007.

Polymorphic Typed Calculi for Delimited Continuations and their CPS Transformation (Extended Abstract)

Yukiyoshi Kameyama[1] and Kenichi Asai[2]

[1] Computer Science Department
University of Tsukuba, Japan
kameyama@acm.org
[2] Ochanomizu University, Japan
asai@is.ocha.ac.jp

Control operators for delimited continuations (aka "partial continuations") enable one to manipulate the control of programs in a concise manner without transforming them into continuation-passing style (CPS). In particular, shift and reset, introduced by Danvy and Filinski, have strong connection to CPS, and thus most of the control effects compatible with CPS can be expressed using shift and reset. They have been used, for example, to program back-tracking, A-normalization in direct style, let-insertion in partial evaluation, and type-safe "printf" in direct style.

Despite the increasing interest in the use of delimited continuations operators, there has been little work that formulates these control operators without sacrificing their expressive power, and investigates the basic properties such as type soundness; the first type system for shift and reset by Danvy and Filinski is restricted to monomorphic types, and Gunter et al's type system for `cupto` operator is restricted to a fixed answer type for each prompt. As such, none of the above type systems can type check, for instance, the "printf" program written with shift and reset.

We present a type system with predicative polymorphism (let-polymorphism in ML), as an extension of the monomorphic type system by Danvy and Filinski. We show, among others, strong type soundness, existence of principal types together with a type inference algorithm, and strong normalization for the subcalculus without the fixpoint operator. The polymorphism does not break the semantic foundation of the monomorphic type system: CPS translation is naturally defined for our polymorphic calculus and preserves types and equivalence.

Unrestricted polymorphism in the presence of control operators leads to an unsound type system. We employ a simple criterion: "only pure expressions can be made polymorphic", where an expression is pure if it has no control effects. The notion of "purity" has a rather concise representation in our type system compared with "weak type variables" or effect-and-type system approaches, and we can construct a sound and complete type inference algorithm as a straightforward extension of Hindley-Milner type inference.

It is easy to extend our type system to accommodate impredicative polymorphism (Girard-Reynolds' second order lambda calculus). Two different evaluation strategies are in consideration: the "standard" strategy does not evaluate

an expression under type abstraction (which is regarded as a value), and the "ML-like" one evaluates an expression under type abstraction. For each case, we obtain strong type soundness as well as we can define type-and-equality preserving CPS translations.

References

1. K. Asai and Y. Kameyama, Polymorphic Delimited Continuations. Submitted for publication.

Organizational Tools for Formal Mathematics in *Theorema*

Florina Piroi[1,2]

[1] Johann Radon Institute for Computational and Applied Mathematics (RICAM)
Austrian Academy of Sciences
[2] Research Institute for Symbolic Computation (RISC)
Johannes Kepler University Linz, Austria
f.piroi@risc.uni-linz.ac.at

Organizational problems in Mathematical Knowledge Management (MKM) become apparent when large mathematical knowledge bases are being built up. At the moment, the largest such collection is Mizar [17]. Among other existing ones we mention MBase [11], the FDL project [1], the NIST library [12], Helm [2], the libraries of the theorem provers Isabelle [14], PVS [13], IMPS [9], Coq [3]. In principle, the tools we describe below are applicable to all the enumerated MKM systems.

In this extended abstract we illustrate the functionality and usage of tools for organizing formal knowledge in the case of the *Theorema* system. These tools refer to syntactic annotation of formal knowledge, knowledge reuse, finding knowledge, and proving with library support. Most of these tools fall into the area of label management. (The exceptions will be marked so later in this document.) The original idea behind the organizational tools in *Theorema* is due to Bruno Buchberger.

What is Label Management? In the overall view of MKM expressed by Bruno Buchberger in the preface of the first conference on MKM and the subsequent special issue of the journal AMAI, see [5, 7], label management belongs to the area of organizational tools for MKM.

In traditional mathematical texts formula labels are ubiquitous but a systematic management of labels is, normally, not considered to be important nor is it feasible for documents on paper. In the computer assisted build-up of formalized mathematical knowledge bases, the systematic design and processing of structured labels (i.e., individual labels like "(1)" or "(associativity)", hierarchical section headings, key words like "definition" and "theorem", names of files etc.) becomes vital for the automated structuring and re-structuring of collections of formulae as input to formal reasoning tools. Consequently, we need algorithmic tools that handle (i.e., manage) all types of labels and allow us to partition and combine, structure and re-structure mathematical knowledge bases.

We emphasize that, in our view, labels do not intend to have any logical meaning or functionality. Their functionality is purely organizational. This is in contrast to "annotations" as, for example, in [16, 8] or [10], which convey at least part of the semantics. Labels, in our view, only help in addressing, referencing, selecting individual formulae in knowledge bases and in partitioning and re-combining (small and big) collections of formulae.

The implementation of our tools is done in the frame of MATHEMATICA [18] and *Theorema* [4, 6] which is built on top of MATHEMATICA. The tools take as input MATHEMATICA notebooks which contain comments in text cells and predicate logic formulae in input cells. Also, in these notebooks, labels of various kinds (section heading, key words like "definition", "theorem", etc., and individual labels) can be attached to formulae and groups of formulae with the help of a particularly designed MATHEMATICA stylesheet. (We note here that key words like "definition, "theorem", again, are nothing else than a kind of labels: They do not add any logical meaning to the formulae they wrap. A given formula may be a definition in one exploration situation and a theorem or an algorithm in some other exploration.) Such a stylesheet is a special kind of notebook that defines the styles to be used in other notebooks ([18] Section 2.10). With the help of this stylesheet, the label management tools can identify, select, and re-combine formal parts of documents and use them, for example, as input to automated reasoners. We call the notebooks that use this stylesheet '*Theorema* notebooks', and a 'library' is a collection of such notebooks.

The organizational tools guarantee that the notebook labels are unique.

For the convenience of the user these tools can be accessed also by a new *Theorema* toolbar, called 'Library Utilities'.

Systematic Generation of Labels. A typical *Theorema* notebook has a notebook title and a notebook label. Formulae, in input cells, are grouped under section and subsection headings. (Hierarchically grouping cells in sections, subsections, etc., is a feature of MATHEMATICA notebooks.) The headings of the cell groups, the notebook title, the notebook label, and labels of individual formulae are the components used for generating and attaching unique composite labels to all formulae and groups of formulae in the *Theorema* notebook. If the notebook label is not present in the document, one is generated from the notebook title in such a way that every notebook in the notebook library has a unique identifier. For this reason, the notebook title is a mandatory element in the *Theorema* notebooks. User given notebook labels are checked against the list of existing notebook labels. The user is notified when the notebook label is already used by another *Theorema* notebook.

From the notebook title, notebook label, section headings, etc. provided by the user our tool automatically generates composite labels for each section, subsection, etc., and individual formula in the notebook. (We remark here that any other information present in the notebook, for example comments posted in MATHEMATICA "text cells", is skipped by the algorithmic process.)

In MATHEMATICA, notebooks are represented as MATHEMATICA expressions. The label generating routine takes as input the MATHEMATICA expression of a *Theorema* notebook. The notebook expression has a recursive structure, reflecting the grouping and sub-grouping of cells in the notebook.

Correspondingly the label generating routine proceeds recursively. The labels are created by concatenation operations, where the operands are the text of the heading and notebook counters. Eventual user-given labels are taken

into account, too. An example of a label that was generated by this routine is: "BN : Tuples.Axioms of Equality and Equality on Tuples".

Further, each of the labels is prepended the notebook label so that any label we look at, in the *Theorema* notebook, contains the notebook label as a substring. Because the notebook label is unique among notebook labels in the library and because within one *Theorema* notebook any two generated labels are different, we are sure that any given (group of) formulae in the whole library is uniquely identified by the label generated and attached to it.

Knowledge Reuse. When we compose a new *Theorema* notebook we often want to use existing formalized knowledge, stored in other *Theorema* notebooks. For this we implemented an 'Include' command with the following structure $\texttt{Include}[\texttt{Label}_1, \texttt{Label}_2, \ldots, \texttt{Label}_n, \mathit{Options}]$ where $\texttt{Label}_1, \ldots, \texttt{Label}_n$ are composite labels of (groups of) formulae in the notebook library. The command understands two options: "IncludeStyle" (which can take the values "Dynamic" or "Static") and "KeepStructure" (with the values True or False). Based on the labels given as parameters, the command either creates links to the knowledge the labels refer to (' IncludeStyle → "Dynamic" '), or copies the knowledge into the current notebook (' IncludeStyle → "Static" ') in which case new unique labels will be generated for them. In the latter case, by the option "KeepStructure", we can chose to keep or not the grouping structure the knowledge had in the originating *Theorema* notebook. (When dynamic inclusion is used, any value of "KeepStructure" is ignored.) In both cases, in the *Theorema* notebook being composed, corresponding cells are inserted after the ones that contain the 'Include' command.

The dynamic inclusion is useful for minimizing duplication of formalized mathematical content, while static inclusion gives us the possibility to concentrate knowledge dispersed in various notebooks in the library in one *Theorema* notebook and use this notebook independent of the library.

Knowledge Search. With the help of the generated composite labels, groups of formulae and individual formulae in *Theorema* notebooks can be referenced, can be used for composing new *Theorema* notebooks and knowledge bases, can be given as input to formal reasoners. Each of these operations presumes the availability of some kind of search mechanism to extract the requested knowledge from the library. The set of organizational tools available in *Theorema* provide mechanisms for textual search on the labels of the formulae, search on constant symbols occurring in the formulae, and search by the structure of formulae. Only the first search tool belongs to the area of label management. We give below examples of procedure calls for each of the three search kinds:

$\texttt{GetFormulaeByLabel}[\text{"}*?\texttt{ommutativity}*\text{"}, \{\text{"}\texttt{MinMax}\text{"}\}]$
$\texttt{GetFormulaeBySymbols}[\{\texttt{.}[=, +]\}, \{\text{"}\texttt{MinMax}\text{"}\}, \texttt{FilterType} \rightarrow \text{"}\texttt{All}\text{"}]$
$\texttt{GetFormulaeByStructure}[\texttt{.}[\_\_ \leq \_\_], \{\text{"}\texttt{MinMax}\text{"}\}, \texttt{SearchInSubformulae} \rightarrow \texttt{True}]$

The first command will search for formulae that have the string "commutativity"

within their textual label, the second command will search for formulae containing both symbols '= and '+, and the third command will search for formulae which have the outermost symbol ' $\leq$. All will search within the *Theorema* notebook with the label "MinMax". (An empty list at this position in the command will cause the tool to perform the search within the whole existing library.) The function calls above return a (possible empty) list of formulae satisfying the given search criteria.

The textual search by labels mechanism can be used for reasoning with library support. The various reasoners (provers, simplifiers, and solvers) of *Theorema* can be called by instructions of the following structure

$$\texttt{Reason}[\texttt{Goal}, \texttt{using} \rightarrow \texttt{KnowledgeBase}, \texttt{by} \rightarrow \texttt{ReasoningMethod}],$$

where '`Reason`' is '`Prove`', '`Compute`', or '`Solve`'; '`KnowledgeBase`' is given by $\langle \texttt{Label}_1, \ \ldots \ , \texttt{Label}_n \rangle$ and $\texttt{Label}_1, \ \ldots \ , \ \texttt{Label}_n$ are composite labels of formulae or collections of formulae in the notebook library. Each of the reasoning commands has been enhanced with a simplified version of the `LabelLookup` tool to extract from the notebook library the (groups of) formulae referred by the given labels and use them in the reasoning process.

Instead of Conclusions. The tools described above were designed with the additional intention to reduce the burden of annotating formal knowledge to a necessary minimum. The only requirement users should take into account is to separate the informal from the formal content of their theories. In MATHEMATICA this separation is done by the different styles of the cells in a notebook: Formulae are to be typed into input cells, comments into text cells, etc. Using these distinctions, the label generation routine can correctly identify and extract the ingredients needed to generate the labels for each (group of) formulae.

The intended users of the described tools are working mathematicians who want to build up and explore mathematical theories within the *Theorema* system. For this reason we have focused on creating tools that are easy and intuitive to use, releasing the users of the burden of annotating and labeling.

The idea that originated this work is due to Bruno Buchberger, a first implementation of it ([15]), and further implementations were done by the author.

References

1. S. Allen, M. Bickford, R. Constable, R. Eaton, C. Kreitz, and L. Lorigo. FDL: A prototype formal digital library, 2002. `http://www.nuprl.org/FDLproject/02cucs-fdl.html`.
2. A. Asperti, L. Padovani, C. Sacerdoti Coen, F. Guidi, and I. Schena. Mathematical knowledge management in HELM. *Annals of Mathematics and Artificial Intelligence*, 38(1):27–46, 2003.
3. Y. Bertot and P. Castèran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

4. B. Buchberger. Theorema: A Proving System Based on Mathematica. *The Mathematica Journal*, 8(2):247–252, 2001.
5. B. Buchberger and O. Caprotti, editors. *First International Workshop on Mathematical Knowledge Management (MKM 2001)*, 24-26 Sept 2001.
6. B. Buchberger, C. Dupré, T. Jebelean, F. Kriftner, K. Nakagawa, D. Văsaru, and W. Windsteiger. The *Theorema* project: A progress report. In M. Kerber and M. Kohlhase, editors, *Proc. of Calculemus'2000*, pages 98–113, St. Andrews, UK, 2000.
7. B. Buchberger, G. Gonnet, and M. Hazewinkel, editors. *Mathematical Knowledge Management*, volume 38 of *Special Issue of Annals of Mathematics and Artificial Intelligence*. Kluwer Academic Publishers, Dordrecht, Netherlands, May 2003.
8. O. Caprotti and D. Carlisle. OpenMath and MathML: Semantic mark up for mathematics. *ACM Crossroads, Special Issue on Markup Languages*, 6(2), 1999. ACM Press.
9. W. M. Farmer, J. D. Guttman, and F. J. Thayer. IMPS: An Interactive Mathematical Proof System. *J. Automated Reasoning*, 11(2):213–248, 1993.
10. M. Kohlhase. OMDoc: An infrastructure for OpenMath content dictionary information. *ACM SIGSAM Bulletin*, 34(2):43–48, 2000.
11. M. Kohlhase and A. Franke. MBase: Representing knowledge and context for the integration of mathematical software systems. *Journal of Symbolic Computation*, 23(4):365–402, 2001.
12. D. W. Lozier. NIST digital library of mathematical functions. pages 105–119.
13. S. Owre, J. Rushby, N. Shankar, and D. Stringer-Calvert. PVS: An experience report. In D. Hutter, W. Stephan, P. Traverso, and M. Ullman, editors, *Applied Formal Methods FM–Trends '98*, volume 1641 of *LNCS*, pages 338–345. Springer–Verlag, Germany, 2004.
14. L. Paulson. Isabelle: the next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.
15. F. Piroi. *Tools for Using Automated Provers in Mathematical Theory Exploration*. PhD thesis, RISC, Johannes Kepler University, Linz, Austria, August 2004.
16. P. Sandhu. *The MathML Handbook*. Charles River Media, 2002.
17. A. Trybulec and H. A. Blair. Computer aided reasoning. In R. Parikh, editor, *Proc. of the conference Logic of Programs*, volume 193 of *LNCS*, pages 406–412. Springer, 1985.
18. S. Wolfram. *The Mathematica Book*. Wolfram Media Inc., 5th edition, 2003.

Strong Cut-Elimination and CPS-Translations

Koji Nakazawa

Graduate School of Informatics
Kyoto University, Japan
knak@kuis.kyoto-u.ac.jp

Abstract. We show that continuation passing style translations (CPS-translations) can be used to prove strong cut-elimination of sequent calculi.

One of the simplest methods to prove strong normalizability (SN) of calculi is to give a reduction preserving translation from the system to another for which SN is already proved. For a very easy example, SN of the Church-style simply typed λ-calculus is easily proved by reducing to SN of the Curry-style via the forgetful function $\lambda x : A.m = \lambda.|M|$, which preserves one-step reduction relation and typability.

For calculi with control operators (or type systems corresponding classical natural deduction), such as Parigot's $\lambda\mu$-calculus, CPS-translations have been used to prove SN, since it translates programs with control operators to programs without them [2, 12, 6, 3] (these proofs, in fact, contain errors due to the same problem, and we give a correction for them in [10, 8]). Furthermore, in [4], de Groote introduced a CPS-translation which maps a λ-calculus $\lambda^{\rightarrow\wedge\vee}$ with conjunction and disjunction to $\lambda^{\rightarrow}$, and he proved SN of $\lambda^{\rightarrow\wedge\vee}$ by reducing it to the well-known result, SN of $\lambda^{\rightarrow}$. In his proof, since the CPS-translation collapses the permutative conversion for disjunction, we must separately show SN of permutative conversion. In [8], a modified CPS-translation, *continuation and garbage passing translation*(*CGPS-translation*), for $\lambda^{\rightarrow\wedge\vee}$ is given, and SN of the system is proved. Since the CGPS-translation preserves one or more steps reduction including permutative conversion, the proof is simpler than [3].

In this talk, we prove SN of a local-step cut-elimination procedure of an intuitionistic sequent calculus by a CGPS-translation, which is the result of [11]. The proof of SN consists of two parts: (1) proving SN of a subsystem $\mathtt{LJ}_{\mathrm{p}}$ of the sequent calculus by a CGPS-translation, (2) reducing SN of the sequent calculus to SN of the $\mathtt{LJ}_{\mathrm{p}}$. For (2), we adopt the method of [1], which was introduced for systems with explicit substitutions. We can see that the sequent calculus is isomorphic to a natural deduction with general elimination rules [13] and explicit substitutions. In this correspondence, the subsystem $\mathtt{LJ}_{\mathrm{p}}$ corresponds to the natural deduction with general elimination rules, for which SN has been already proved by Joachimski and Matthes [9] by inductive characterization of SN terms.

Another example of strong cut-elimination proof of sequent calculi by CGPS-translations is given by Espirito Santo et al. in [5]. They proved SN of several extensions of the Herbelin style sequent calculus λ [7] by CGPS-translation.

References

1. R. Bloo and K.H. Rose. Preservation of strong normalization in named lambda calculi with explicit substitution and garbage collection. In *Computer Science in the Netherlands (CSN'95)*, 62–72, 1995.
2. P. de Groote. A simple calculus of exception handling. In M. Dezani-Ciancaglini and G. Plotkin, Eds., *Typed Lambda Calculi and Applications, TLCA'95, Lecture Notes in Comput. Sci.* 902, pp. 201–215, 1995.
3. P. de Groote. Strong normalization of classical natural deduction with disjunction. In S. Abramsky, Ed., *Typed Lambda Calculi and Applications, 5th International Conference, TLCA 2001, Lecture Notes in Comput. Sci.* 2044, pp.182–196, 2001.
4. P. de Groote. On the Strong Normalisation of Intuitionistic Natural Deduction with Permutation-Conversions. *Inform. and Comput.* 178, pp. 441–464, 2002
5. J. Espirito Santo and R. Matthes and L. Pinto. Continuation-Passing Style and Strong Normalisation for Intuitionistic Sequent Calculi. In S. Ronchi Della Rocca, Ed., *Typed Lambda Calculi and Applications, 8th International Conference, TLCA 2007, Lecture Notes in Comput. Sci.* 4583, 2007.
6. K. Fujita. Domain-free $\lambda\mu$-calculus. *Theoretical Informatics and Applications* 34(6), pp. 433–466, 2000.
7. H. Herbelin. A λ-calculus structure isomorphic to Gentzen-style sequent calculus. In *Selected Papers from the 8th International Workshop on Computer Science Logic (CSL'94), Lecture Notes in Computer Science* 933, pp. 61–75, 1994.
8. S. Ikeda and K. Nakazawa. Strong normalization proofs by CPS-translations. Information Processing Letters 99:163–170, 2006.
9. F. Joachimski and R. Matthes. Short proofs of normalization for the simply-typed λ-calculus, permutative conversions and Gödels's T. *Archive for Mathematical Logic* 42, pp. 59–87, 2003.
10. K. Nakazawa and M. Tatsuta. Strong normalization proof with CPS-translation for second order classical natural deduction. *J. Symbolic Logic,* 68(3), pp. 851–859, 2003. Corrigendum is available in *J. Symbolic Logic,* 68(4), pp. 1415–1416, 2003.
11. K. Nakazawa. An isomorphism between cut-elimination procedure and proof reduction In S. Ronchi Della Rocca, Ed., *Typed Lambda Calculi and Applications, 8th International Conference, TLCA 2007, Lecture Notes in Comput. Sci.* 4583, 2007.
12. M. Parigot. Strong normalization for second order classical natural deduction. *J. of Symbolic Logic* 62(4), pp. 1461–1479, 1997.
13. J. von Plato. Natural deduction with general elimination rules. *Archive for Mathematical Logic* 40:541–567, 2001.

Matching with Membership Constraints for Sequence Variables (Work in Progress)

Mircea Marin

Department of Computer Science
University of Tsukuba, Japan
mmarin@cs.tsukuba.ac.jp

Abstract. We describe a sound and complete matching algorithm for sequences of terms built over flexible arity function symbols. The patterns for such sequences of terms may contain first-order variables for terms and sequences, and second order variables of two kinds: for function symbols, and for second-order expressions that denote sequences with an arbitrary number of hole occurrences that can be filled with sequences. The admissible bindings of first- and second-order sequence variables can be specified by membership constraints with regular expressions.

1 Preliminaries

In general, if A is a set, then we denote by $A^\star$ the monoid of strings of elements of A with neutral element ε, and by A^+ the set $A^\star \setminus \{\varepsilon\}$. We write $a_1 \ldots a_n$ for the string of $A^\star$ made of the elements $a_1, \ldots, a_n$ of A in this order. Also, we write $A \uplus B$ for the union of two disjoint sets A and B, and $|A|$ for the number of elements of a finite set A.

We consider an alphabet made of six mutually disjoint sets of symbols: a finite set $\mathcal{F}$ of *function symbols*, denoted by f, g, possibly subscripted; a countably infinite set $\mathcal{V}_i$ of *individual variables*, denoted by x, y; a countably infinite set $\mathcal{V}_{\mathrm{h}}^u$ of *unconstrained hedge variables*, denoted by $\overline{x}$, $\overline{y}$, possibly subscripted; a countably infinite set $\mathcal{V}_f$ of *function variables*, denoted by X, Y, possibly subscripted; a countably infinite set $\mathcal{V}_{\mathrm{h}}^{\mathrm{k}}$ of *constrained hedge variables*, denoted by $\overline{H}$, possibly subscripted; and a countably infinite set $\mathcal{V}_{\mathrm{c}}$ of *contextual variables*, denoted by $\overline{C}$, possibly subscripted. In addition, we consider the special constant symbol $\bullet \notin \mathcal{F} \uplus \mathcal{V}_i \uplus \mathcal{V}_f \uplus \mathcal{V}_{\mathrm{h}}^u \uplus \mathcal{V}_{\mathrm{h}}^{\mathrm{k}} \uplus \mathcal{V}_{\mathrm{c}}$, called the *hole*.

A *hedge variable* is an element of the set $\mathcal{V}_{\mathrm{h}} := \mathcal{V}_{\mathrm{h}}^u \cup \mathcal{V}_{\mathrm{h}}^{\mathrm{k}}$, *sequence variable* is an element of the set $\mathcal{V}_s := \mathcal{V}_{\mathrm{h}} \uplus \mathcal{V}_{\mathrm{c}}$, and a *constrained variable* is an element of the set $\mathcal{V}_{\mathrm{k}} := \mathcal{V}_{\mathrm{h}}^{\mathrm{k}} \cup \mathcal{V}_{\mathrm{c}}$. A *variable* is an element of the set $\mathcal{V} := \mathcal{V}_i \uplus \mathcal{V}_f \uplus \mathcal{V}_s$.

The set $\mathcal{T}(\mathcal{F}, \mathcal{V})$ of *terms* and the set $\mathcal{SE}(\mathcal{F}, \mathcal{V})$ of *sequence elements* are defined by

$$\begin{aligned} t &::= \bullet \mid x \mid \underline{f}(s) \mid \underline{X}(s) && \textit{terms} \\ se &::= t \mid \overline{x} \mid \overline{C}(\underline{s}) \mid \overline{H} && \textit{sequence elements} \end{aligned}$$

with $s \in \mathcal{S}(\mathcal{F},\mathcal{V})$, $\underline{s} \in \mathcal{S}_p(\mathcal{F},\mathcal{V})$, where

$$\mathcal{S}(\mathcal{F},\mathcal{V}) := \mathcal{SE}(\mathcal{F},\mathcal{V})^\star.$$
$$\mathcal{S}_p(\mathcal{F},\mathcal{V}) := \{se_1 \cdots se_n \in \mathcal{SE}(\mathcal{F},\mathcal{V}) \mid se_i \in \mathcal{T}(\mathcal{F},\mathcal{V}) \text{ for some } 1 \le i \le n\}.$$

We call the elements of $\mathcal{S}(\mathcal{F},\mathcal{V})$ *sequences*, and the elements of $\mathcal{S}_p(\mathcal{F},\mathcal{V})$ *proper sequences*. A *hole-free term* is a term without occurrences of $\bullet$. A *hedge* is a sequence without occurrences of $\bullet$, and a *contextual hedge* is a sequence with at least one occurrence of $\bullet$. Thus, a sequence is either a hedge or a contextual hedge. We write $\mathcal{H}(\mathcal{F},\mathcal{V})$ for the set of hedges, and $\mathcal{C}(\mathcal{F},\mathcal{V})$ for the set of contextual sequences. A *hedge language* is an element of the powerset $\mathcal{L}_\mathrm{h} := \mathcal{P}(\mathcal{H}(\mathcal{F},\mathcal{V}))$, and a language of contextual hedges is an element of the powerset $\mathcal{L}_\mathrm{c} := \mathcal{P}(\mathcal{C}(\mathcal{F},\mathcal{V}))$. We denote hedges by h possible primed or subscripted, and contextual hedges by c possibly primed or subscripted.

Given two sequences s and s', we denote by $s[s']$ the result of replacing all occurrences of $\bullet$ in s with s'. $s[s']$ is obviously a sequence, and if $s' \in \mathcal{H}(\mathcal{F},\mathcal{V})$ then $s[s'] \in \mathcal{H}(\mathcal{F},\mathcal{V})$ too.

A *substitution* is a mapping θ from $\mathcal{V}$ such that

- $\theta(x)$ is a hole-free term if $x \in \mathcal{V}_i$,
- $\theta(v)$ is a hedge if $v \in \mathcal{V}_\mathrm{h}$,
- $\theta(X)$ is a λ-term of the form $\lambda\bullet.q(\bullet)$ with $q \in \mathcal{F} \cup \mathcal{V}_f$ if $X \in \mathcal{V}_f$, and
- $\theta(\overline{C})$ is a λ-term of the form $\lambda\bullet.c$ with $c \in \mathcal{C}(\mathcal{F},\mathcal{V})$ if $\overline{C} \in \mathcal{V}_\mathrm{c}$.

and the set $Dom(\theta) := \{v \in \mathcal{V}_i \cup \mathcal{V}_\mathrm{h} \mid \theta(v) \neq v\} \cup \{v \in \mathcal{V}_f \cup \mathcal{V}_\mathrm{c} \mid \theta(v) \neq \lambda\bullet.v(\bullet)\}$, called the *domain* of θ, is finite. Thus, we regard the elements of $\mathcal{V}_i \cup \mathcal{V}_\mathrm{h}$ as first-order variables, and the elements of $\mathcal{V}_f \cup \mathcal{V}_\mathrm{c}$ as second-order variables. If V is a set of variables and θ_1, θ_2 are substitutions, then we write $\theta_1 = \theta_2\,[V]$ iff $\theta_1(v) = \theta_2(v)$ for all $v \in V$.

The *instantiation* of $s \in \mathcal{S}(\mathcal{F},\mathcal{V})$ by θ is the sequence $s\theta$ defined by

$$\begin{array}{lll} x\theta = \theta(x) & f(s)\theta = f(s\theta), & X(s)\theta = \theta(X)\cdot(s\theta), \\ \varepsilon\theta = \varepsilon, & (se_1 \ldots se_n)\theta = se_1\theta \ldots se_n\theta & \bullet\theta = \bullet, \\ \overline{C}(\underline{s})\theta = \theta(\overline{C})\cdot(\underline{s}\theta), & \overline{H}\theta = \theta(\overline{H}) & \overline{x}\theta = \theta(\overline{x}). \end{array}$$

where "$\cdot$" denotes β-reduction: $(\lambda\bullet.s)\cdot s' = s[s']$ for all $s, s' \in \mathcal{S}(\mathcal{F},\mathcal{V})$. If M is a set of sequences then we define $M\theta := \{s\theta \mid s \in M\}$. Similarly, we define the following operations for $s \in \mathcal{S}(\mathcal{F},\mathcal{V})$ and $L, M \in \mathcal{P}(\mathcal{S},\mathcal{V})$:

- $s.L :=$ the result of replacing every occurrence of $\bullet$ in s with an element of L. Different occurrences of $\bullet$ in s may be replaced with different sequences from L.
- $L.M := \bigcup_{s\in L} s.M$. We call his operation *language concatenation.*
- $L^* = \bigcup_{k=0}^{\infty} L^k$ where $L^0 = \{\bullet\}$ and $L^{n+1} = L^n \cup L.L^n$. We call this operation the *compositional closure* of L.
- $L* = \bigcup_{k=0}^{\infty} L_k$ and $L+ = \bigcup_{k=1}^{\infty} L_k$ where $L_0 = \{\varepsilon\}$ and $L_{n+1} = L_n \cup \{s\,s' \mid s \in L, s' \in L_n\}$. We call this operation the *sequential closure* of L.

Notice that, if θ is a substitution, $t \in \mathcal{T}(\mathcal{F},\mathcal{V})$, $\underline{s} \in \mathcal{S}_p(\mathcal{F},\mathcal{V})$, $h \in \mathcal{H}(\mathcal{F},\mathcal{V})$ and $c \in \mathcal{C}(\mathcal{F},\mathcal{V})$ then $t\theta \in \mathcal{T}(\mathcal{F},\mathcal{V})$, $\underline{s}\theta \in \mathcal{S}_p(\mathcal{F},\mathcal{V})$, $h\theta \in \mathcal{H}(\mathcal{F},\mathcal{V})$, and $c\theta \in \mathcal{C}(\mathcal{F},\mathcal{V})$. An immediate consequence of these properties is that the composition of substitutions is a substitution too.

We write $vars(s)$ for the set of variables that occur in the sequence s, and say that s is *ground* if $vars(s) = \emptyset$. θ is *ground* if $\theta(v)$ is ground for all $v \in Dom(\theta)$. The substitution with empty domain is denoted by ϵ, and called *empty substitution*. We adopt the convention to represent a substitution θ by the set

$$\begin{array}{l}\{v \mapsto \theta(v) \mid v \in Dom(\theta) \cap (\mathcal{V}_i \cup \mathcal{V}_{\mathrm{h}})\} \cup \\ \quad \{X \mapsto q \mid X \in Dom(\theta) \cap \mathcal{V}_f \wedge \theta(X) = \lambda\bullet.q(\bullet)\} \cup \\ \quad \{\overline{C} \mapsto c \mid \overline{C} \in Dom(\theta) \cap \mathcal{V}_s \wedge \theta(\overline{C}) = \lambda\bullet.c\}.\end{array}$$

Regular Expressions. We propose a special kind of expressions to be used in specifications of constraints for the admissible bindings of constrained variables. We call these expressions *regular*. We distinguish two kinds of regular expressions: for hedges, and for contextual hedges. They are defined by the following grammars:

$\mathtt{Re} ::= \mathtt{He} \mid \mathtt{Ce}$		*regular expressions.*
$\mathtt{He} ::= \boldsymbol{\varepsilon} \mid \overline{x} \mid \overline{H} \mid \mathtt{He} + \mathtt{He} \mid \mathtt{He}* \mid \mathtt{pHe}$		*regular hedge expressions.*
$\mathtt{Ce} ::=$		*regular contextual expressions*:
	$\bullet$	
$\mid$	$\mathtt{Ce} + \mathtt{Ce}$	union
$\mid$	$f(\mathtt{Ce}) \mid X(\mathtt{Ce}) \mid \overline{C}(\mathtt{Ce})$	applications
$\mid$	$\mathtt{Ce}.\mathtt{Ce}$	composition
$\mid$	$\mathtt{Ce}\,\mathtt{Ce} \mid \mathtt{Ce}\,\mathtt{He} \mid \mathtt{He}\,\mathtt{Ce}$	concatenations
$\mid$	$\mathtt{Ce}^*$	compositional iteration
$\mid$	$\mathtt{Ce}+$	non-empty sequential iteration
$\mathtt{pHe} ::=$		*proper regular hedge expressions*:
	x	
$\mid$	$\mathtt{pHe} + \mathtt{pHe}$	union
$\mid$	$f(\mathtt{He}) \mid X(\mathtt{He}) \mid \overline{C}(\mathtt{pHe})$	applications
$\mid$	$\mathtt{Ce}.\mathtt{pHe}$	composition
$\mid$	$\mathtt{pHe}\,\mathtt{He} \mid \mathtt{He}\,\mathtt{pHe}$	concatenations

We will abbreviate $(\ldots(\mathtt{Re}_1 + \mathtt{Re}_2) + \ldots) + \mathtt{Re}_n$ by $\mathtt{Re}_1 + \ldots + \mathtt{Re}_n$ or $\sum_{i=1}^{n} \mathtt{Re}_i$.

Every $\mathtt{He}$ has an associated language $[\![\mathtt{He}]\!] \in \mathcal{L}_{\mathrm{h}}$, and every $\mathtt{Ce}$ has an associated language $[\![\mathtt{Ce}]\!] \in \mathcal{L}_{\mathrm{c}}$. These languages are defined by mutual recursion:

$$
\begin{array}{ll}
[\![\boldsymbol{\varepsilon}]\!] = \{\varepsilon\}, & \\
[\![h]\!] = \{h\} & \text{if } h \in \{\bullet\} \cup \mathcal{V}_i \cup \mathcal{V}_{\mathrm{h}}, \\
[\![\mathtt{Ce}_1 + \mathtt{Ce}_2]\!] = [\![\mathtt{Ce}_1]\!] \cup [\![\mathtt{Ce}_2]\!], & \\
[\![\mathtt{He}_1 + \mathtt{He}_2]\!] = [\![\mathtt{He}_1]\!] \cup [\![\mathtt{He}_2]\!], & \\
[\![q(\mathtt{Re})]\!] = \{q(s) \mid s \in [\![\mathtt{Re}]\!]\} & \text{if } q \in \mathcal{F} \cup \mathcal{V}_f \cup \mathcal{V}_{\mathrm{c}}, \\
[\![\mathtt{Ce}.\mathtt{Re}]\!] = [\![\mathtt{Ce}]\!].[\![\mathtt{Re}]\!], & \\
[\![\mathtt{Re}_1\ \mathtt{Re}_2]\!] = \{s_1\, s_2 \mid s_1 \in [\![\mathtt{Re}_1]\!], s_2 \in [\![\mathtt{Re}_2]\!]\}, & \\
[\![\mathtt{Ce}^*]\!] = [\![\mathtt{Ce}]\!]^*,\ [\![\mathtt{Ce}+]\!] = [\![\mathtt{Ce}]\!]+, \text{ and } [\![\mathtt{He}*]\!] = [\![\mathtt{He}]\!]*. &
\end{array}
$$

Notice that the following relations hold for any regular expression $\mathtt{Re}$:

$$
\begin{array}{c}
[\![\mathtt{Re}\ \boldsymbol{\varepsilon}]\!] = [\![\mathtt{Re}]\!], \quad [\![\boldsymbol{\varepsilon}\ \mathtt{Re}]\!] = [\![\mathtt{Re}]\!], \quad [\![\bullet.\mathtt{Re}]\!] = [\![\mathtt{Re}]\!], \\
[\![\overline{C}(\bullet).\mathtt{Re}]\!] = [\![\overline{C}(\mathtt{Re})]\!], \quad [\![\mathtt{Ce}.\bullet]\!] = [\![\bullet]\!].
\end{array}
$$

These equalities show that, when $\mathtt{Re}$ is used as language specifier, then it is harmless to identify it with its normal form with respect to the transformation system $\{\mathtt{Re}\,\boldsymbol{\varepsilon} \to \mathtt{Re},\ \boldsymbol{\varepsilon}\,\mathtt{Re} \to \mathtt{Re},\ \bullet.\mathtt{Re} \to \mathtt{Re},\ \overline{C}(\bullet).\mathtt{Re} \to \overline{C}(\mathtt{Re}),\ \mathtt{Ce}.\bullet \to \mathtt{Ce}\}$. Therefore, from now on we assume that regular expressions are always represented in this normal form, and that this normalization operation is performed implicitly.

We define the *compositional complexity* $\omega(\mathtt{Re})$ of $\mathtt{Re}$ as the number of occurrences of the composition operator "." in $\mathtt{Re}$:

$$
\begin{array}{ll}
\omega(h) := 0, & \omega(\mathtt{Ce}.\mathtt{Re}) := 1 + \omega(\mathtt{Ce}) + \omega(\mathtt{Re}) \\
\omega(\mathtt{Re}_1\ \mathtt{Re}_2) := \omega(\mathtt{Re}_1) + \omega(\mathtt{Re}_2), & \omega(\mathtt{Re}_1 + \mathtt{Re}_2) := \omega(\mathtt{Re}_1) + \omega(\mathtt{Re}_2), \\
\omega(q(\mathtt{Re})) := \omega(\mathtt{Re}), & \omega(\mathtt{He}*) := \omega(\mathtt{He}), \\
\omega(\mathtt{Ce}^*) = \omega(\mathtt{Ce}+) := \omega(\mathtt{Ce}). &
\end{array}
$$

where $h \in \{\boldsymbol{\varepsilon}, \bullet\} \cup \mathcal{V}_i \cup \mathcal{V}_{\mathrm{h}}$ and $q \in \mathcal{F} \cup \mathcal{V}_f \cup \mathcal{V}_{\mathrm{c}}$.

For our further considerations, it is useful to define a notion of substitution instantiation for regular expressions that is compatible with language instantiation, i.e., $[\![\mathtt{Re}\theta]\!] = [\![\mathtt{Re}]\!]\theta$ for any regular expression $\mathtt{Re}$ and substitution θ. To achieve this, we define first an interpretation of sequences as regular expressions: Given a sequence s, we define the regular expression $\ulcorner s \urcorner$ as follows:

$$
\begin{array}{l}
\ulcorner \varepsilon \urcorner = \boldsymbol{\varepsilon} \\
\ulcorner se_1 \ldots se_{n-1}\, se_n \urcorner = \ulcorner se_1 \urcorner (\ulcorner se_2 \urcorner \ldots (\ulcorner se_{n-1} \urcorner \ulcorner se_n \urcorner) \cdots) \ \text{ if } n \geq 2 \\
\ulcorner f(s) \urcorner = f(\ulcorner s \urcorner) \qquad \ulcorner v \urcorner = v \text{ if } v \in \{\bullet\} \cup \mathcal{V}_i \cup \mathcal{V}_{\mathrm{h}} \\
\ulcorner X(s) \urcorner = X(\ulcorner s \urcorner) \qquad \ulcorner \overline{C}(s) \urcorner = \overline{C}(\ulcorner s \urcorner).
\end{array}
$$

Next, we extend the notion of substitution instantiation to regular expressions in the following way:

- $\boldsymbol{\varepsilon}\theta = \boldsymbol{\varepsilon}$, $h\theta = \ulcorner \theta(h) \urcorner$ if $h \in \{\bullet\} \cup \mathcal{V}_i \cup \mathcal{V}_{\mathrm{h}}$, $(\mathtt{Ce}_1 + \mathtt{Ce}_2)\theta = \mathtt{Ce}_1\theta + \mathtt{Ce}_2\theta$,
- $(\mathtt{He}_1 + \mathtt{He}_2)\theta = \mathtt{He}_1\theta + \mathtt{He}_2\theta$, $f(\mathtt{Re})\theta = f(\mathtt{Re}\theta)$,
- $X(\mathtt{Re})\theta = q(\mathtt{Re}\theta)$ where $q \in \mathcal{F} \cup \mathcal{V}_f$ such that $\theta(X) = \lambda\bullet.q(\bullet)$,
- $\overline{C}(\mathtt{Re})\theta = \ulcorner \theta(\overline{C}) \cdot \bullet \urcorner.\mathtt{Re}\theta$, $(\mathtt{Ce}.\mathtt{Re})\theta = (\mathtt{Ce}\theta).(\mathtt{Re}\theta)$, $(\mathtt{Re}_1\ \mathtt{Re}_2)\theta = (\mathtt{Re}_1\theta)\,(\mathtt{Re}_2\theta)$,
- $\mathtt{Ce}^*\theta = (\mathtt{Ce}\theta)^*$, $\mathtt{He}*\theta = \mathtt{He}\theta*$, and $\mathtt{Ce}+\theta = \mathtt{Ce}\theta+$.

It is easy to see that for any regular hedge expression $\mathtt{He}$, proper regular hedge expression $\mathtt{pHe}$, regular contextual expression $\mathtt{Ce}$, and substitution θ, we have

- $\mathtt{He}\theta$ is a regular hedge expression, and $[\![\mathtt{He}\theta]\!] = [\![\mathtt{He}]\!]\theta$.
- $\mathtt{pHe}\theta$ is a proper regular hedge expression, and $\varepsilon \notin [\![\mathtt{He}\theta]\!]$.
- $\mathtt{Ce}\theta$ is a regular contextual expression, and $[\![\mathtt{Ce}\theta]\!] = [\![\mathtt{Ce}]\!]\theta$.

1.1 Regular matching problems

A *matching equation* is a pair of hedges $h \ll h'$ where h' is ground. θ is a *solution* of $h \ll h'$ if $h\theta = h'$.

A *membership constraint* for $\overline{C} \in \mathcal{V}_{\mathrm{c}}$ (respectively $\overline{H} \in \mathcal{V}_{\mathrm{h}}^{\mathrm{k}}$) is an expression of the form $\overline{C}\,\mathtt{in}\,(\mathtt{Ce}, u)$ (respectively $\overline{H}\,\mathtt{in}\,(\mathtt{He}, u)$) where $u \in \{0, 1\}$. A substitution θ is a *solution* of $\overline{C}\,\mathtt{in}\,(\mathtt{Ce}, u)$ if $\theta(\overline{C}) = \lambda\bullet.c$ for some $c \in [\![\mathtt{Ce}\theta]\!]$, and $c \neq \bullet$ if $u = 1$. θ is a *solution* of $\overline{H}\,\mathtt{in}\,(\mathtt{He}, u)$ if $\theta(\overline{H}) \in [\![\mathtt{He}\theta]\!]$ and $\theta(\overline{H}) \neq \varepsilon$ if $u = 1$. Thus, u in a membership constraint acts like a flag to control whether $\overline{C}$ (respectively $\overline{H}$) is allowed to be bound to $\bullet$ (to ε) or not. θ is a *solution* of a set $\mathcal{K}$ of membership constraints, notation $\theta \in \mathit{Sol}(\mathcal{K})$, if it is a solution of every membership constraint of $\mathcal{K}$.

A set $\mathcal{K}$ of membership constraints induces a binary relation $\succcurlyeq$ on $\mathit{vars}(\mathcal{K}) \cap \mathcal{V}_{\mathrm{k}}$, which is defined by: $\overline{X}_2 \succcurlyeq \overline{X}_1$ iff there exists $\overline{X}_1\,\mathtt{in}\,(\mathtt{Re}, u) \in \mathcal{K}$ with $\overline{X}_2 \in \mathit{vars}(\mathtt{Re})$. We define $\mathit{dvars}_{\mathcal{K}}(\overline{X}) := \{\overline{Y} \mid \overline{Y} \succcurlyeq^+ \overline{X}\}$ where $\succcurlyeq^+$ is the transitive closure of $\succcurlyeq$, and say that $\mathcal{K}$ is *acyclic* iff $\overline{X} \notin \mathit{dvars}_{\mathcal{K}}(\overline{X})$ for all $\overline{X} \in \mathit{vars}(\mathcal{K}) \cap \mathcal{V}_s$.

In this paper we are interested in solving systems of matching equations and membership constraints of the form $\langle E \mid \mathcal{K}\rangle$ where E is a sequence of matching equations and $\mathcal{K}$ is an acyclic set of membership constraints such that:

k_1. $\mathit{vars}(E) \cap \mathcal{V}_{\mathrm{k}} \subseteq \mathit{vars}(\mathcal{K}) \cap \mathcal{V}_{\mathrm{k}}$.
k_2. $\mathit{vars}(\mathcal{K}) \cap \mathcal{V}_{\mathrm{k}} = \{\overline{X} \mid \exists \overline{X}\,\mathtt{in}\,(\mathtt{Re}, u) \in \mathcal{K}\}$.
k_3. $\mathcal{K}$ contains exactly one membership constraint for every $\overline{X} \in \mathit{vars}(\mathcal{K}) \cap \mathcal{V}_{\mathrm{k}}$.

We call such a system a *regular matching problem* (RMP). Solving $\langle E \mid \mathcal{K}\rangle$ means to compute all substitutions θ that are solutions of all equations of E and of all membership constraints of $\mathcal{K}$. We write $\mathit{Sol}(P)$ for the set of solutions of an RMP P. If E is empty, then we write $\Box_{\mathcal{K}}$ instead of $\langle \mid \mathcal{K}\rangle$.

We conclude this section with two remarks.

1. Acyclicity of $\mathcal{K}$ implies that $\Box_{\mathcal{K}}$ is always a satisfiable RMP, and $\mathcal{K}$ is a generic representation of $\mathit{Sol}(\Box_{\mathcal{K}})$. This means that $\Box_{\mathcal{K}}$ can be regarded as a solved form of $\mathcal{K}$.
2. Since $\mathcal{F}$ is finite, the concepts of individual variable, unconstrained hedge variable, and function variable are redundant because:
   - Every individual variable x can be replaced by $\overline{H} \in \mathcal{V}_{\mathrm{h}}^{\mathrm{k}}$ with the membership constraint $\overline{H}\,\mathtt{in}\,((\sum_{f\in\mathcal{F}} f(\bullet))^*.(\sum_{f\in\mathcal{F}} f()), 0)$,
   - Every function variable X can be replaced by $\overline{C} \in \mathcal{V}_{\mathrm{c}}$ with the membership constraint $\overline{C}\,\mathtt{in}\,(\sum_{f\in\mathcal{F}} f(\bullet), 0)$,
   - Every unconstrained hedge variable $\overline{x}$ can be replaced by $\overline{H} \in \mathcal{V}_{\mathrm{h}}^{\mathrm{k}}$ with the membership constraint $\overline{H}\,\mathtt{in}\,\varepsilon + (\bullet\,\bullet)^*.((\sum_{f\in\mathcal{F}} f(\bullet))^*.(\sum_{f\in\mathcal{F}} f()), 0)$.

   However, we prefer to keep them as convenient abbreviations.

2 The Matching Procedure

Our matching procedure is described by a transformation system $\mathcal{U}$ consisting of schematic representations of transformation rules of the form $P \Rightarrow_\sigma P'$ where P and P' are RMPs and σ is a substitution. $\mathcal{U}$ has two important properties:

Soundness: If $P \Rightarrow_\sigma P' \in \mathcal{U}$ and $\theta' \in Sol(P')$ then $\sigma\theta' \in Sol(P)$.
Local completeness: If $P \neq \Box_\mathcal{K}$ and $\theta \in Sol(P)$ then there exists $P \Rightarrow_\sigma P' \in \mathcal{U}$ and $\theta' \in Sol(P')$ such that $\theta = \sigma\theta'\,[vars(P)]$.

To solve an RMP P, we search for all derivations $P \Rightarrow_{\sigma_1} P_1 \Rightarrow_{\sigma_2} \ldots \Rightarrow_{\sigma_n} P_n = \Box$. We abbreviate such derivations by $P \Rightarrow^n_\theta P_n$, or simply $P \Rightarrow^*_\sigma P_n$, where $\sigma = \sigma_1 \cdots \sigma_n$. The transformation system $\mathcal{U}$ consists of 27 transformation rules which can be split into two groups:

1. $\mathcal{U}_\mathrm{i} = \{t, iv, fv, uhv, rd\}$. The transformation rules of this group are independent of the membership constraints of the unification problem.
2. $\mathcal{U}_\mathrm{d} = \{$hv1, hv2, hv3, hv4, cv1, cv2, ciq, hiq, ac1, ac2, cf1, cf2, hf1, hf2, thf, icc, ich, ihh, c$*$, c*, c+1, c+2$\}$.These rules are driven by the structure of the membership constraint associated with the constrained variable of the head of the leftmost outermost sequence element of a matching equation.

The transformation rules of $\mathcal{U}$ are given in Sections 2.1–2.2. Some transformation rules of $\mathcal{U}$ will rely on the solvability of a slightly more general class of RMPs, denoted by $\mathrm{RMP_a}$. $\mathrm{RMP_a}$ is obtained from RMP by extending the language of hedges with the *anonymous variable* constructs $\_\_$ and $\_\_\_$ for hedges. Formally, this means that we extend the grammar for sequence elements with the clauses

$$se ::= \_ \mid \_\_\_$$

and extend the notion of substitution instantiation with $\_\theta = \_\_$ and $\_\_\_\theta = \_\_\_$. An $\mathrm{RMP_a}$ is a system $P = \langle h_1 \ll h'_1, \ldots, h_n \ll h'_n \mid \mathcal{K}\rangle$ which satisfies conditions 1-2 of RMP and also the condition that $\_\_$ and $\_\_\_$ do not occur in $h'_1, \ldots, h'_n$. Intuitively, every $P \in \mathrm{RMP_a}$ is a generic representation of the set of RMPs producible from P by replacing the occurrences of $\_$ with non-empty ground hedges and the occurrences of $\_\_\_$ with possibly empty ground hedges. To capture this intuition, we proceed as follows:

1. We associate to every $Q \in \mathrm{RMP_a}$ the set $\overline{Q}$ of RMPs producible from Q by replacing the occurrences of $\_$ with non-empty ground hedges and the occurrences of $\_\_\_$ with possibly empty ground hedges.. Different occurrences of $\_\_$ or $\_\_\_$ may be replaced by different hedges.
2. We define $Sol(Q) := \bigcup_{P \in \overline{Q}} Sol(P)$.

$\mathrm{RMP_a}$-s can be solved by a straightforward adjustment of the $\mathcal{U}$-based matching procedure for RMP problems: we add two transformation rules that simply discard the equations $\_\_\_ \ll h$ and $\_\_ \ll h$ with $h \neq \varepsilon$. In Sect. 2.3 we define a system of transformation rules $\mathcal{U}'$ that, in addition to solving $\mathrm{RMP_a}$-s, it accumulates the ground hedges denoted by occurrences of anonymous variables. The transformation rules of $\mathcal{U}'$ are of the form $\langle P \mid S\rangle \Rightarrow_\sigma \langle P' \mid S'\rangle$ where $P, P' \in \mathrm{RMP_a}$,

σ is the substitution computed by the transformation step, and S, S' are lists of ground hedges. To solve an RMP_a P, we compute all derivations

$$\langle P_0 \mid S_0\rangle = \langle P \mid [\,]\rangle \Rightarrow_{\sigma_1} \langle P_1 \mid S_1\rangle \Rightarrow_{\sigma_1} \ldots \Rightarrow_{\sigma_n} \langle P_n = \Box_{\mathcal{K}_n} \mid \mathcal{S}_n\rangle,$$

abbreviated $\langle P_0 \mid S_0\rangle \Rightarrow_\sigma^n \langle \Box_{\mathcal{K}_n} \mid S_n\rangle$, or simply $\langle P_0 \mid S_0\rangle \Rightarrow_\sigma^* \langle \Box_{\mathcal{K}_n} \mid S_n\rangle$, where $[\,]$ denotes the empty list and $\sigma = \sigma_1 \ldots \sigma_n$.

Like $\mathcal{U}$, system $\mathcal{U}'$ is also designed to be sound and locally complete. For $\mathcal{U}'$, soundness means "If $\langle Q \mid S\rangle \Rightarrow_\sigma \langle Q' \mid S'\rangle \in \mathcal{U}'$ and $\theta' \in \mathit{Sol}(Q')$ then $\sigma\theta' \in \mathit{Sol}(Q)$,"and local completeness means "If $Q \neq \Box_\mathcal{K}$, S is a list of ground hedges, and $\theta \in \mathit{Sol}(Q)$, then there exists $\langle Q \mid S\rangle \Rightarrow_\sigma \langle Q' \mid S'\rangle$ and a solution θ' of Q' such that $\theta = \sigma\theta'\,[\mathit{vars}(Q)]$." Moreover, if $\langle E \mid \mathcal{K}\rangle$ is an RMP_a, then $\mathit{Sol}(\langle E \mid \mathcal{K}\rangle) = \{\langle\sigma\sigma' \mid \langle E \mid \mathcal{K}\rangle \mid [\,]\rangle \Rightarrow_\sigma^* \langle \Box_{\mathcal{K}_n} \mid S\rangle, \sigma' \in \mathit{Sol}(\mathcal{K}_n)\}$, where S is the list of hedges denoted by the occurrences of $\_\_$ in $E\sigma$, enumerated by a leftmost innermost traversal of $E\sigma$.

2.1 $\mathcal{U}$: Constraint-independent transformation rules

(t) *Trivial equation.*
$\langle \varepsilon \ll \varepsilon, E \mid \mathcal{K}\rangle \Rightarrow_\epsilon \langle E \mid \mathcal{K}\rangle$.

(iv) *Individual variable elimination*
$\langle x\, h_1 \ll t\, h_2,\, E \mid \mathcal{K}\rangle \Rightarrow_{\sigma=\{x\mapsto t\}} \langle h_1\sigma \ll h_2, E\sigma \mid \mathcal{K}\sigma\rangle$.

(fv) *Function variable elimination.*
$\langle X(h_1)\, h_2 \ll f(h_3)\, h_4, E \mid \mathcal{K}\rangle \Rightarrow_{\sigma=\{X\mapsto f\}} \langle h_1\sigma \ll h_3,\, h_2\sigma \ll h_4, E\sigma \mid \mathcal{K}\sigma\rangle$.

(uhv) *Unconstrained hedge variable elimination.*
$\langle \overline{x}\, h \ll h_1\, h_2, E \mid \mathcal{K}\rangle \Rightarrow_{\sigma=\{\overline{x}\mapsto h_1\}} \langle h\sigma \ll h_2, E\sigma \mid \mathcal{K}\sigma\rangle$.

(rd) *Rigid decomposition.*
$\langle f(h_1)\, h_2 \ll f(h_3)\, h_4, E \mid \mathcal{K}\rangle \Rightarrow_\epsilon \langle h_1 \ll h_3,\, h_2 \ll h_4, E \mid \mathcal{K}\rangle$.

2.2 $\mathcal{U}$: Constraint-dependent transformation rules

In the specifications of some of these transformation rules we will use the notation $h|_{\bullet\leftarrow h_1 \ldots h_n}$ where h is a ground hedge with n hole occurrences. This expression denotes the hedge produced by replacing the occurrences of $\bullet$ in h with $h_1, \ldots, h_n$ in the order of their leftmost innermost occurrence in h, respectively.

(hv1) *Hedge variable elimination 1.*
$\langle \overline{H}\, h \ll \varepsilon,\, E \mid \{\overline{H}\,\mathtt{in}\,(\boldsymbol{\varepsilon}, 0)\} \uplus \mathcal{K}\rangle \Rightarrow_{\sigma=\{\overline{H}\mapsto\varepsilon\}} \langle h\sigma \ll \varepsilon, E\sigma \mid \mathcal{K}\sigma\rangle$.

(hv2) *Hedge variable elimination 2.*
$\langle \overline{H}\, h \ll t,\, E \mid \{\overline{H}\,\mathtt{in}\,(x, u)\} \uplus \mathcal{K}\rangle \Rightarrow_{\sigma=\{\overline{H}\mapsto t, x\mapsto t\}} \langle h\sigma \ll \varepsilon, E\sigma \mid \mathcal{K}\sigma\rangle$.

(hv3) *Hedge variable elimination 3.*
$\langle \overline{H}\, h \ll h_1\, h_2,\, E \mid \{\overline{H}\,\mathtt{in}\,(\overline{x}, u)\} \uplus \mathcal{K}\rangle \Rightarrow_{\sigma=\{\overline{H}\mapsto h_1, \overline{x}\mapsto h_1\}} \langle h\sigma \ll h_2, E\sigma \mid \mathcal{K}\sigma\rangle$
where $h_1 \neq \varepsilon$ if $u = 1$.

(hv4) *Hedge variable elimination 4.*
$\langle \overline{H}\, h \ll \varepsilon,\, E \mid \{\overline{H}\,\mathtt{in}\,(\mathtt{He*}, 0)\} \uplus \mathcal{K}\rangle \Rightarrow_{\sigma=\{\overline{H}\mapsto\varepsilon\}} \langle h\sigma \ll \varepsilon, E\sigma \mid \mathcal{K}\sigma\rangle$.

(cv1) *Contextual variable elimination 1.*
$\langle \overline{C}(h_1)\, h_2 \ll h,\, E \mid \{\overline{C}\,\mathtt{in}\,(\bullet, 0)\} \uplus \mathcal{K}\rangle \Rightarrow_{\sigma=\{\overline{C}\mapsto\bullet\}} \langle h_1\sigma\, h_2\sigma \ll h,\, E\sigma \mid \mathcal{K}\sigma\rangle$.

(cv2) *Contextual variable elimination 2.*
$\langle \overline{C}(h_1)\, h_2 \ll h,\ E \mid \{\overline{C}\ \mathtt{in}\ (\mathtt{Ce}^*, 0)\} \uplus \mathcal{K}\rangle \Rightarrow_{\sigma=\{\overline{C}\mapsto\bullet\}} \langle h_1\sigma\, h_2\sigma \ll h,\ E\sigma \mid \mathcal{K}\sigma\rangle.$

(ciq) *Contextual imitation for term-like constraint.*
$\langle \overline{C}(h_1)\, h_2 \ll f(h), E \mid \{\overline{C}\ \mathtt{in}\ (q(\mathtt{Ce}), u)\} \uplus \mathcal{K}\rangle$
$\Rightarrow_\sigma \langle \overline{C}_1(h_1\sigma)\, h_2\sigma \ll h, E\sigma \mid \{\overline{C}_1\ \mathtt{in}\ (\mathtt{Ce}, 0)\} \cup \mathcal{K}\sigma\rangle$
if $q \in \{f\}\cup\mathcal{V}_f$, $\overline{C}_1$ is fresh, and $\sigma = \begin{cases} \{\overline{C} \mapsto f(\overline{C}_1(\bullet))\} & \text{if } q = f \in \mathcal{F}, \\ \{\overline{C} \mapsto f(\overline{C}_1(\bullet)),\ X \mapsto f\} & \text{if } q = X \in \mathcal{V}_f. \end{cases}$

(hiq) *Hedge imitation for term-like constraint.*
$\langle \overline{H}\, h_1 \ll f(h), E \mid \{\overline{H}\ \mathtt{in}\ (q(\mathtt{He}), u)\} \uplus \mathcal{K}\rangle$
$\Rightarrow_\sigma \langle \overline{H}_1 \ll h, h_1\sigma \ll \varepsilon,\ E\sigma \mid \{\overline{H}_1\ \mathtt{in}\ (\mathtt{He}, 0)\} \cup \mathcal{K}\sigma\rangle$
if $q \in \{f\}\cup\mathcal{V}_f$, $\overline{H}_1$ is fresh, and $\sigma = \begin{cases} \{\overline{H} \mapsto f(\overline{H}_1)\} & \text{if } q = f \in \mathcal{F}, \\ \{\overline{H} \mapsto f(\overline{H}_1),\ X \mapsto f\} & \text{if } q = X \in \mathcal{V}_f. \end{cases}$

(ac1) *Abstraction of compositional constraint 1.*
$\langle \overline{C}(h_1)\, h_2 \ll h,\ E \mid \{\overline{C}\ \mathtt{in}\ (\mathtt{Ce}_1.\mathtt{Ce}_2, u)\} \uplus \mathcal{K}\rangle \Rightarrow_\epsilon$
$\langle \overline{C}(h_1)\, h_2 \ll h \mid \{\overline{C}\ \mathtt{in}\ (\overline{C}_1(\mathtt{Ce}_2), u), \overline{C}\ \mathtt{in}\ (\mathtt{Ce}_1, 0)\} \cup \mathcal{K}\rangle$
where $\overline{C}_1$ is fresh.

(ac2) *Abstraction of compositional constraint 2.*
$\langle \overline{H}\, h_1 \ll h, E \mid \{\overline{H}\ \mathtt{in}\ (\mathtt{Ce}.\mathtt{He},\ u)\} \uplus \mathcal{K}\rangle \Rightarrow_\epsilon$
$\langle \overline{H}\, h_1 \ll h \mid \{\overline{H}\ \mathtt{in}\ (\overline{C}(\mathtt{He}), u), \overline{C}\ \mathtt{in}\ (\mathtt{Ce}, 0)\} \cup \mathcal{K}\rangle$
where $\overline{C}$ is fresh.

(cf1) *Contextual flex constraint 1.*
$\langle \overline{C}(h)\, h' \ll h'', E \mid \{\overline{C}\ \mathtt{in}\ (\overline{C}_0(\mathtt{Ce}), u)\} \uplus \mathcal{K}\rangle \Rightarrow_{\theta=\sigma\cup\{\overline{C}\mapsto\sigma(\overline{C}_0)|^{\bullet\leftarrow\overline{C}_1(\bullet)\ \dots\ \overline{C}_n(\bullet)}_n\}}$
$\langle \overline{C}_1(h\theta) \ll h_1, \dots, \overline{C}_n(h\theta) \ll h_n, h'\theta \ll h_{n+1}, E\theta \mid \mathcal{K}'\theta \cup \bigcup_{i=1}^{n} \{\overline{C}_i\ \mathtt{in}(\mathtt{Ce}\sigma, 0)\}\rangle$
if $h'' \neq \varepsilon$, $\langle\langle \overline{C}_0(\_)\ \_\_ \ll h'' \mid \mathcal{K}\rangle \mid [\ ]\rangle \Rightarrow^*_\sigma \langle \Box_{\mathcal{K}'} \mid [h_1, \dots, h_n, h_{n+1}]\rangle, \overline{C}_1, \dots, \overline{C}_n$ are fresh, and $\sigma(\overline{C}_0) \neq \bullet$.

(cf2) *Contextual flex constraint 2.*
$\langle \overline{C}(h)\, h' \ll h'', E \mid \{\overline{C}\ \mathtt{in}\ (\overline{C}_0(\mathtt{Ce}), u)\} \uplus \mathcal{K}\rangle \Rightarrow_{\theta=\sigma\cup\{C\mapsto\overline{C}_1(\bullet)\}}$
$\langle \overline{C}_1(h\theta)\, h'\theta \ll h'', E\theta \mid \mathcal{K}'\theta \cup \{\overline{C}_1\ \mathtt{in}\ (\mathtt{Ce}\sigma, u)\}\rangle$
if $h'' \neq \varepsilon$, $\langle\langle \overline{C}_0(\_)\ \_\_ \ll h'' \mid \mathcal{K}\rangle \mid [\ ]\rangle \Rightarrow^*_\sigma \langle \Box_{\mathcal{K}'} | S\rangle$, $\overline{C}_1$ is fresh and $\sigma(\overline{C}_0) = \bullet$.

(hf1) *Hedge flex constraint 1.*
$\langle \overline{H}\, h \ll h', E \mid \{\overline{H}\ \mathtt{in}\ (\overline{C}(\mathtt{pHe}), u)\} \uplus \mathcal{K}\rangle \Rightarrow_{\theta=\sigma\cup\{\overline{H}\mapsto\sigma(\overline{C})|^{\bullet\leftarrow\overline{H}_1\ \dots\ \overline{H}_n}\}}$
$\langle \overline{H}_1 \ll h_1, \dots, \overline{H}_n \ll h_n, h\theta \ll h_{n+1}, E\theta \mid \mathcal{K}'\theta \cup \bigcup_{i=1}^{n} \{\overline{H}_i\ \mathtt{in}\ (\mathtt{pHe}\sigma, 0)\}\rangle$
if $h' \neq \varepsilon$, $\langle\langle \overline{C}(\_\_)\ \_\_ \ll h' \mid \mathcal{K}\rangle \mid [\ ]\rangle \Rightarrow^*_\sigma \langle \Box_{\mathcal{K}'} \mid [h_1, \dots, h_n, h_{n+1}]\rangle, \overline{H}_1, \dots, \overline{H}_n$ are fresh, and $\sigma(\overline{C}) \neq \bullet$.

(hf2) *Hedge flex constraint 2.*
$\langle \overline{H}\, h \ll h', E \mid \{\overline{H}\ \mathtt{in}\ (\overline{C}(\mathtt{pHe}), u)\} \uplus \mathcal{K}\rangle \Rightarrow_{\theta=\sigma\cup\{\overline{H}\mapsto\overline{H}_1\}}$
$\langle \overline{H}_1\, h\theta \ll h', E\theta \mid \mathcal{K}'\theta \cup \{\overline{H}_1\ \mathtt{in}\ (\mathtt{pHe}\sigma, 0)\}\rangle$
if $h' \neq \varepsilon$, $\langle\langle \overline{C}(\_\_)\ \_\_ \ll h' \mid \mathcal{K}\rangle \mid [\ ]\rangle \Rightarrow^*_\sigma \langle \Box_{\mathcal{K}'} \mid S\rangle, \overline{H}_1$ is fresh and $\sigma(\overline{C}) = \bullet$.

(thf) *Trivial hedge flex constraint.*
$\langle \overline{H}\, h \ll h',\ E \mid \{\overline{H}\ \mathtt{in}\ (\overline{H}_1, u_1), \overline{H}_1\ \mathtt{in}\ (\mathtt{He}, u_2)\} \uplus \mathcal{K}\rangle \Rightarrow_{\sigma=\{\overline{H}\mapsto\overline{H}_1\}}$
$\langle \overline{H}_1\, h\sigma \ll h', E\sigma \mid \{\overline{H}_1\ \mathtt{in}\ (\mathtt{He}\sigma, \max\{u_1, u_2\})\} \cup \mathcal{K}\sigma\rangle.$

(icc) *Imitation of concatenation of contextual sequences.*

$\langle \overline{C}(h_1)\, h_2 \ll h, E \mid \{\overline{C}\ \mathtt{in}\ (\mathtt{Ce}_1\ \mathtt{Ce}_2, u)\} \uplus \mathcal{K}\rangle \Rightarrow_{\sigma=\{\overline{C}\mapsto \overline{C}_1(\bullet)\,\overline{C}_2(\bullet)\}}$
$\langle \overline{C}_1(h_1\sigma)\,\overline{C}_2(h_1\sigma)\, h_2 \ll h, E\sigma \mid \{\overline{C}_1\ \mathtt{in}\ (\mathtt{Ce}_1, 0), \overline{C}_2\ \mathtt{in}\ (\mathtt{Ce}_2, 0)\} \cup \mathcal{K}\sigma\rangle$

where $h \neq \varepsilon$ and $\overline{C}_1, \overline{C}_2$ are fresh.

(ich) *Imitation of contextual sequence-hedge concatenation.*

$\langle \overline{C}(h_1)\, h_2 \ll h, E \mid \{\overline{C}\ \mathtt{in}\ (\mathtt{Ce}\ \mathtt{He}, u)\} \uplus \mathcal{K}\rangle \Rightarrow_{\sigma=\{\overline{C}\mapsto \overline{C}_1(\bullet)\,\overline{H}\}}$
$\langle \overline{C}_1(h_1\sigma)\,\overline{H}\, h_2\sigma \ll h, E\sigma \mid \{\overline{C}_1\ \mathtt{in}\ (\mathtt{Ce}, u_1), \overline{H}\ \mathtt{in}\ (\mathtt{He}, u_2)\} \cup \mathcal{K}\sigma\rangle$

$\langle \overline{C}(h_1)\, h_2 \ll h, E \mid \{\overline{C}\ \mathtt{in}\ (\mathtt{He}\ \mathtt{Ce}, u)\} \uplus \mathcal{K}\rangle \Rightarrow_{\sigma=\{\overline{C}\mapsto \overline{H}\,\overline{C}_1(\bullet)\}}$
$\langle \overline{H}\,\overline{C}_1(h\sigma)\, h_2\sigma \ll h, E\sigma \mid \{\overline{C}_1\ \mathtt{in}\ (\mathtt{Ce}, u_1), \overline{H}\ \mathtt{in}\ (\mathtt{He}, u_2)\} \cup \mathcal{K}\sigma\rangle$

where $h \neq \varepsilon$, $\overline{C}_1, \overline{H}$ are fresh, and $u_1, u_2 \in \{0, 1\}$ such that $u_1 + u_2 = u$.

(ihh) *Imitation of hedge-hedge concatenation.*

$\langle \overline{H}\, h \ll h', E \mid \{\overline{H}\ \mathtt{in}\ (\mathtt{He}_1\ \mathtt{He}_2, u)\} \uplus \mathcal{K}\rangle \Rightarrow_{\sigma=\{\overline{H}\mapsto \overline{H}_1\,\overline{H}_2\}}$
$\langle \overline{H}_1\,\overline{H}_2\, h\sigma \ll h', E\sigma \mid \{\overline{H}_1\ \mathtt{in}\ (\mathtt{He}_1, u_1), \overline{H}_2\ \mathtt{in}\ (\mathtt{He}_2, u_2)\} \cup \mathcal{K}\sigma\rangle$

where $\overline{H}_1, \overline{H}_2$ are fresh, and $u_1, u_2 \in \{0, 1\}$ such that $u_1 + u_2 = u$.

(c*) *Constraint for compositional iteration.*

$\langle \overline{C}(h_1)\, h_2 \ll h', E \mid \{\overline{C}\ \mathtt{in}(\mathtt{Ce}^*, u)\} \uplus \mathcal{K}\rangle \Rightarrow_{\epsilon}$
$\langle \overline{C}(h_1)\, h_2 \ll h', E \mid \{\overline{C}\ \mathtt{in}\ (\overline{C}_1(\mathtt{Ce}^*), u), \overline{C}_1\ \mathtt{in}\ (\mathtt{Ce}, 1)\} \cup \mathcal{K}\rangle$

where $h' \neq \varepsilon$ and $\overline{C}_1$ is fresh.

(c∗) *Constraint for sequential iteration.*

$\langle \overline{H}\, h_1 \ll h, E \mid \{\overline{C}\ \mathtt{in}\ (\mathtt{He}*, u)\} \uplus \mathcal{K}\rangle \Rightarrow_{\sigma=\{\overline{H}\mapsto \overline{H}_1\,\overline{H}_2\}}$
$\langle \overline{H}_1\,\overline{H}_2\, h_1\sigma \ll h, E\sigma \mid \{\overline{H}_1\ \mathtt{in}\ (\mathtt{He}, 1), \overline{H}_2\ \mathtt{in}\ (\mathtt{He}*, 0)\} \cup \mathcal{K}\sigma\rangle$

where $h_1 \neq \varepsilon$ and $\overline{H}_1, \overline{H}_2$ are fresh.

(c+1) *Constraint for non-empty sequential iteration 1.*

$\langle \overline{C}(h_1)\, h_2 \ll h,\ E \mid \{\overline{C}\ \mathtt{in}\ (\mathtt{Ce}+, u)\} \uplus \mathcal{K}\rangle \Rightarrow_{\epsilon}$
$\langle \overline{C}(h_1)\, h_2 \ll h, E \mid \{\overline{C}\ \mathtt{in}\ (\mathtt{Ce}, u)\} \cup \mathcal{K}\rangle.$

(c+2) *Constraint for non-empty sequential iteration 2.*

$\langle \overline{C}(h_1)\, h_2 \ll h, E \mid \{\overline{C}\ \mathtt{in}\ (\mathtt{Ce}+, u)\} \uplus \mathcal{K}\rangle \Rightarrow_{\sigma=\{\overline{C}\mapsto \overline{C}_1(\bullet)\,\overline{C}_2(\bullet)\}}$
$\langle \overline{C}_1(h_1\sigma)\,\overline{C}_2(h_1\sigma)\, h_2\sigma \ll h, E\sigma \mid \{\overline{C}_1\ \mathtt{in}\ (\mathtt{Ce}, 0), \overline{C}_2\ \mathtt{in}\ (\mathtt{Ce}+, 0)\} \cup \mathcal{K}\sigma\rangle$

where $\overline{C}_1, \overline{C}_2$ are fresh.

2.3 The transformation system $\mathcal{U}'$

$\mathcal{U}' = \{\alpha' \mid \alpha \in \mathcal{U}\} \cup \{\mathtt{any0}, \mathtt{any1}\}$ where transformation rule α' is defined by

$$\langle Q \mid S\rangle \Rightarrow_\sigma \langle Q' \mid S\rangle$$

for all $\alpha \in \mathcal{U}$ when $Q \Rightarrow_\sigma Q'$ is an α-rule of $\mathcal{U}$.
Rule any0 is defined by $\langle\langle \_\_ \ll h, E \mid \mathcal{K}\rangle \mid S\rangle \Rightarrow_\epsilon \langle\langle E \mid \mathcal{K}\rangle \mid S{:}h\rangle$.
Rule any1 is defined by $\langle\langle \_ \ll h, E \mid \mathcal{K}\rangle \mid S\rangle \Rightarrow_\epsilon \langle\langle E \mid \mathcal{K}\rangle \mid S{:}h'\rangle$ where $h' \neq \varepsilon$.
($S{:}h$ denotes the list produced by appending hedge h at the end of list S.)

2.4 Main Properties of systems $\mathcal{U}$ and $\mathcal{U}'$

To simplify the analysis of our transformation systems, we will introduce a few more auxiliary notions. We denote by $\ell(\pi)$ the label of a transformation step $\pi = P \Rightarrow_\sigma P' \in \mathcal{U}$. If π' is $\langle P \mid S\rangle \Rightarrow_\sigma \langle P' \mid S'\rangle$, then $\ell(\pi') = \mathtt{any}$ if π' is an instance of transformation rule `any`, and $\ell(\pi') = \ell(\pi)$ if π' corresponds to a transformation step $\pi = P \Rightarrow_\sigma P' \in \mathcal{U}$.

We define the *complexity* $\kappa(\pi)$ of a transformation step $\pi \in \mathcal{U} \cup \mathcal{U}'$ as follows:

- $\kappa(\pi) = 1$ if $\pi = P \Rightarrow_\sigma P'$ or $\pi = \langle P \mid S\rangle \Rightarrow_\sigma \langle P' \mid S'\rangle$ with $\ell(\pi) \notin \{$cf1, cf2, hf$\}$,
- $\kappa(\pi) = 1 + \sum_{i=1}^{m} c(\pi_i)$ if $\ell(\pi) \in \{$cf1, cf2, hf$\}$ and π_i are the transformation steps of the derivation $\langle \overline{C}_0(\_) \_\_ \ll h' \mid [\,]\rangle \Rightarrow^m \langle \Box_{\mathcal{K}'} \mid S'\rangle$ mentioned in the side condition of those transformation steps.

If Π is a derivation consisting of the (possibly empty) sequence of transformation steps $\pi_1; \ldots; \pi_m$, then we define $\kappa(\Pi) = \sum_{i=1}^{m} \kappa(\pi_i)$.

Suppose $\mathcal{K}$ is an acyclic set of membership constraints that satisfies conditions (k_2)–(k_3) mentioned on page 21. We define the *size* of a regular expression $\mathtt{Re}$ with respect to an acyclic set $\mathcal{K}$ of membership constraints, as follows:

- $sz_\mathcal{K}(\boldsymbol{\varepsilon}) = sz_\mathcal{K}(\bullet) = 0$, and $sz_\mathcal{K}(v) = 1$ if $v \in \mathcal{V}_\mathrm{h}^u \cup \mathcal{V}_i$,
- $sz_\mathcal{K}(q(\mathtt{Re})) := sz_\mathcal{K}(\mathtt{Re}) + 1$ if $q \in \mathcal{F} \cup \mathcal{V}_f$,
- $sz_\mathcal{K}(\overline{C}(\mathtt{Re})) := sz_\mathcal{K}(\mathtt{Ce}) + sz_\mathcal{K}(\mathtt{Re}) + 1$ where $\mathtt{Re}_1$ is the unique regular expression such that $\overline{C}\,\mathtt{in}\,(\mathtt{Ce}, u) \in \mathcal{K}$,
- $sz_\mathcal{K}(\overline{H}) := sz_\mathcal{K}(\mathtt{He})$ where $\mathtt{He}$ is the unique regular expression such that $\overline{H}\,\mathtt{in}\,(\mathtt{He}, u) \in \mathcal{K}$,
- $sz_\mathcal{K}(\mathtt{Ce.Re}) = sz_\mathcal{K}(\mathtt{Ce\,Re}) = sz_\mathcal{K}(\mathtt{Re\,Ce}) := sz(\mathtt{Ce}) + sz(\mathtt{Re}) + 1$,
- $sz_\mathcal{K}(\mathtt{He}_1\,\mathtt{He}_2) = sz_\mathcal{K}(\mathtt{He}_1 + \mathtt{He}_2) := sz_\mathcal{K}(\mathtt{He}_1) + sz_\mathcal{K}(\mathtt{He}_2) + 1$,
- $sz_\mathcal{K}(\mathtt{Ce}_1 + \mathtt{Ce}_2) := sz_\mathcal{K}(\mathtt{Ce}_1) + sz_\mathcal{K}(\mathtt{Ce}_2) + 1$,
- $sz_\mathcal{K}(\mathtt{Ce}^*) = sz_\mathcal{K}(\mathtt{Ce+}) := sz_\mathcal{K}(\mathtt{Ce}) + 1$, and $sz_\mathcal{K}(\mathtt{He}*) := sz_\mathcal{K}(\mathtt{He}) + 1$.

If $\overline{X} \in vars(\mathcal{K}) \cap \mathcal{V}_\mathrm{k}$, then the *size of* $\overline{X}$ *in* $\mathcal{K}$ is $sz_\mathcal{K}(\overline{X}) := sz_\mathcal{K}(\mathtt{Re})$ where $\mathtt{Re}$ is the unique regular expression such that $\overline{X}\,\mathtt{in}(\mathtt{Re}, u) \in \mathcal{K}$.

Example 1. If

$$\mathcal{K} = \{\overline{C}_1\,\mathtt{in}\,((f(g(\boldsymbol{\varepsilon})\,\bullet)^*, 0), \overline{H}\,\mathtt{in}\,(\overline{x}\,g(x))*, 1), \overline{C}_2\,\mathtt{in}\,(\bullet + \overline{C}_1(\bullet\,\overline{H})\mathtt{+}, 1)\}$$

then

$$\begin{aligned}
sz_\mathcal{K}(\overline{C}_1) = sz_\mathcal{K}((f(g(\boldsymbol{\varepsilon})\,\bullet)^*) &= 1 + sz_\mathcal{K}(f(g(\boldsymbol{\varepsilon})\,\bullet)) = 2 + sz_\mathcal{K}(g(\boldsymbol{\varepsilon})\,\bullet) = 4,\\
sz_\mathcal{K}(\overline{H}) = sz_\mathcal{K}((\overline{x}\,g(x))*) &= 1 + sz_\mathcal{K}(\overline{x}\,g(x)) = 2 + sz_\mathcal{K}(\overline{x}) + sz_\mathcal{K}(g(x)) = 5,\\
sz_\mathcal{K}(\overline{C}_2) = sz_\mathcal{K}(\bullet + \overline{C}_1(\bullet\,\overline{H})\mathtt{+}) &= 1 + sz_\mathcal{K}(\bullet) + (1 + sz_\mathcal{K}(\overline{C}_1(\bullet\,\overline{H})))\\
&= 2 + (1 + sz_\mathcal{K}(\overline{C}_1) + sz_\mathcal{K}(\bullet\,\overline{H})) = 13.
\end{aligned}$$

□

Let now h be a hedge. We say that a symbol position p of h is *persistent* with respect to $\mathcal{K}$ if the symbol of h at position p, denoted by $h|_p$, satisfies the conditions (a) $h|_p \notin \mathcal{V}_{\mathrm{h}}^u$, and (b) if $h|_p = \overline{X} \in \mathcal{V}_{\mathrm{k}}$ then there exists $\overline{X}\,\mathtt{in}\,(\mathtt{Re}, 1) \in \mathcal{K}$. Intuitively, the position p of h is persistent with respect to $\mathcal{K}$ if $h|_p\theta$ persists in any instantiation $h\theta$ of h with $\theta \in \mathit{Sol}(\mathcal{K})$. We write $\mathit{pers}_{\mathcal{K}}(h)$ for the set of persistent positions of h with respect to $\mathcal{K}$, and define the *minimal persistent position* of h w.r.t. $\mathcal{K}$ as follows:

$$\min\nolimits_{\mathcal{K}}^{\mathit{pers}}(h) := \begin{cases} \min_{\prec} \mathit{pers}_{\mathcal{K}}(h) & \text{if } \mathit{pers}_{\mathcal{K}}(h) \neq \emptyset, \\ \infty & \text{if } \mathit{pers}_{\mathcal{K}}(h) = \emptyset. \end{cases}$$

where $\min_{\prec} A$ denotes the minimal position of A with respect to the prefix order $\prec$ on positions. The *regular size* of h w.r.t. $\mathcal{K}$ is the multiset

$$\mathit{regsz}_{\mathcal{K}}(h) := \{\mathit{sz}_{\mathcal{K}}(\overline{X}) \mid h|_p = \overline{X} \in \mathcal{V}_{\mathrm{k}} \wedge p \preceq \min\nolimits_{\mathcal{K}}^{\mathit{pers}}(h)\}.$$

Example 2. Let $\mathcal{K}$ be as in Example 1, and

$$h_1 = \overline{x}\,\overline{C}_1(\overline{C}_1(\overline{y}\,\overline{C}_2(f(x_1))))\,x_2 \quad h_2 = \overline{y}\,\overline{y}\,\overline{C}_1(\overline{H}()\,f()) \quad h_3 = \overline{y}\,f(x) \quad h_4 = \overline{y}\,\overline{x}.$$

Then $\min_{\mathcal{K}}^{\mathit{pers}}(h_1) = 2{\cdot}1{\cdot}2{\cdot}0$, $h_1|_{2\cdot1\cdot2\cdot0} = \overline{C}_2$, $\min_{\mathcal{K}}^{\mathit{pers}}(h_2) = 3{\cdot}1{\cdot}0$, $h_1|_{3\cdot1\cdot0} = \overline{H}$, $\min_{\mathcal{K}}^{\mathit{pers}}(h_3) = 2{\cdot}0$, $h_3|_{2\cdot0} = f$, and $\min_{\mathcal{K}}^{\mathit{pers}}(h_4) = \infty$. The regular sizes of these hedges are $\mathit{regsz}_{\mathcal{K}}(h_1) = \{\mathit{sz}_{\mathcal{K}}(\overline{C}_1), \mathit{sz}_{\mathcal{K}}(\overline{C}_1), \mathit{sz}_{\mathcal{K}}(\overline{C}_2)\} = \{5, 5, 14\}$, $\mathit{regsz}_{\mathcal{K}}(h_2) = \{\mathit{sz}_{\mathcal{K}}(\overline{C}_1), \mathit{sz}_{\mathcal{K}}(\overline{H})\} = \{5, 4\}$, $\mathit{regsz}_{\mathcal{K}}(h_3) = \mathit{regsz}_{\mathcal{K}}(h_4) = \{\}$. □

With there preparations, we are ready to prove some important properties of our transformation systems. First, we show that $\mathcal{U}$ and $\mathcal{U}'$ are sound by proving a slightly stronger result:

If Π is $P \Rightarrow_\sigma^* P'$ or $\langle P \mid S\rangle \Rightarrow_\sigma^* \langle P' \mid S'\rangle$ and $\theta' \in \mathit{Sol}(P')$ then $\sigma\theta' \in \mathit{Sol}(P)$.

Proof. The proof proceeds by induction on the complexity of Π. The claim is obvious when $\kappa(\Pi) = 0$. If $\kappa(\Pi) \geq 1$, then we can write

$$\Pi = P \Rightarrow_\theta \underbrace{P'' \Rightarrow_{\sigma'}^* P'}_{\Pi'} \quad \text{or} \quad \Pi = \langle P \mid S\rangle \Rightarrow_\theta \underbrace{\langle P'' \mid S''\rangle \Rightarrow_{\sigma'}^* \langle P' \mid S'\rangle}_{\Pi'}.$$

If $\theta' \in \mathit{Sol}(P')$ then, by induction hypothesis for Π' we conclude $\sigma'\theta' \in \mathit{Sol}(P'')$. It remains to show that $\theta\sigma'\theta' \in \mathit{Sol}(P)$. We proceed by case distinction on $\ell(\pi)$, where π is the first transformation step of Π.

- The cases when $\ell(\pi) \notin \{\mathrm{cf1}, \mathrm{cf2}, \mathrm{hf}, \mathrm{any1}, \mathrm{any2}\}$ are straightforward to verify.
- If $\ell(\pi) \in \{\mathtt{any1}, \mathtt{any2}\}$, then π is $\langle \underbrace{\langle a \ll h, E \mid \mathcal{K}\rangle}_{P} \mid S\rangle \Rightarrow_{\theta=\epsilon} \langle \underbrace{\langle E \mid \mathcal{K}\rangle}_{P''} \mid S{:}h\rangle$ where $a \in \{\_\_, \_\_\}$ and $h \neq \varepsilon$ if $a = \_\_$. In this case, $\theta\sigma'\theta' = \epsilon\sigma'\theta' = \sigma'\theta' \in \mathit{Sol}(P'') = \mathit{Sol}(P)$.

– If $\ell(\pi) = \text{cf1}$ then π is either $P \Rightarrow_\theta P''$ or $\langle P \mid S\rangle \Rightarrow_\theta \langle P'' \mid S\rangle$ with

$$P = \langle \overline{C}(h) \ll h', E \mid \{\overline{C}\,\mathtt{in}\,(\overline{C}_0(\mathtt{Ce}), u)\} \uplus \mathcal{K}\rangle,$$
$$P'' = \langle \overline{C}_1(h\theta) \ll h_1, \ldots, \overline{C}_n(h\theta) \ll h_n,\, h'\theta \ll h_{n+1}, E\theta \mid \mathcal{K}'\theta \cup \textstyle\bigcup_{i=1}^{n}\{\overline{C}_i\,\mathtt{in}\,(\mathtt{Ce}\sigma, 0)\}\rangle,$$
$$\theta = \sigma \cup \{\overline{C} \mapsto \sigma\left(\overline{C}_0\right)|_{\bullet\leftarrow\overline{C}_1(\bullet)\ldots\overline{C}_n(\bullet)}\} \text{ where } \sigma(\overline{C}_0) \neq \bullet$$

and $\Pi'' = \underbrace{\langle \overline{C}_0(\_)\_\_ \ll h' \mid \mathcal{K}\rangle}_{P_1} \Rightarrow^*_\sigma \langle \Box_{\mathcal{K}'} \mid [h_1, \ldots, h_{n+1}]\rangle$. $\sigma'\theta' \in Sol(P'')$ implies $\theta\sigma'\theta' \in Sol(\langle \overline{C}_1(h) \ll h_1, \ldots, \overline{C}_n(h) \ll h_n, h' \ll h_{n+1}, E \mid \mathcal{K}'\rangle)$. It remains to show that $\theta\sigma'\theta' \in Sol(\langle \overline{C}(h) \ll h' \mid \{\overline{C}\,\mathtt{in}\,(\overline{C}_0(\mathtt{Ce}), u)\} \uplus \mathcal{K}\rangle)$. The induction hypothesis for Π'' yields that $\sigma\theta'' \in Sol(P_1)$ for all $\theta'' \in Sol(\mathcal{K}')$. Since $\theta\sigma'\theta' \in Sol(\mathcal{K}')$, we get $\sigma\theta\sigma'\theta' \in Sol(P_1)$, and thus $\sigma\theta\sigma'\theta' \in Sol(\mathcal{K})$. Since σ is a restriction of θ, we conclude that $\theta = \sigma\theta$, and thus $\theta\sigma'\theta' = \sigma\theta\sigma'\theta' \in Sol(\mathcal{K})$. Moreover,

- $\overline{C}\theta\sigma'\theta' = \sigma(C_0)|_{\bullet\leftarrow\overline{C}_1\sigma'\theta'\ldots\bullet\leftarrow\overline{C}_n\sigma'\theta'} \in [\![\overline{C}_0(\mathtt{Ce})\theta\sigma'\theta']\!]$, and so $\theta\sigma'\theta' \in Sol(\overline{C}\,\mathtt{in}\,(\overline{C}_0(\mathtt{Ce}), 0)))$. Also, if $\sigma(C_0) \neq \bullet$ then $\overline{C}\theta\sigma'\theta' \neq \bullet$ and thus $\theta\sigma'\theta' \in Sol(\overline{C}\,\mathtt{in}\,(\overline{C}_0(\mathtt{Ce}), u)))$ regardless of the value of u.
- $\overline{C}(h)\theta\sigma'\theta' = \sigma(C_0)|_{\bullet\leftarrow\overline{C}_1(h\theta)\sigma'\theta'\ldots\bullet\leftarrow\overline{C}_n(h\theta)\sigma'\theta'} = \sigma(C_0)|_{\bullet\leftarrow h_1\ldots\bullet\leftarrow h_n} = h'$, and thus $\theta\sigma'\theta' \in Sol(\overline{C}(h) \ll h')$ too.

We conclude that $\theta\sigma'\theta' \in Sol(P)$.

– The cases when $\ell(\pi) \in \{\text{cf1}, \text{hf1}, \text{hf2}\}$ can be proved in a similar was as the previous case. □

Next, we will prove the following important property of $\mathcal{U}$:

> (T) There exists a terminating order $\triangleright$ on RMPs and RMP_a-s such that $P \triangleright P'$ whenever $P \Rightarrow_\sigma P'$.

The specification of $\triangleright$ relies on the following complexity measures of RMPs and RMP_a-s:

– $m_1(\langle h_1 \ll h'_1, \ldots, h_n \ll h'_n \mid \mathcal{K}\rangle) :=$ the multiset $\{sz(h'_1), \ldots, sz(h'_n)\}$ where $sz(h')$ is the *size* of h', i.e., the total number of symbols that make up h'.
– $m_2(P) := |vars(P) \cap \mathcal{V}^u_\text{h}|$.
– $m_3(h_1 \ll h'_1, \ldots, h_n \ll h'_n \mid \mathcal{K}) := regsz_\mathcal{K}(h_1\ \ldots\ h_n)$.
– $m_4(\langle E \mid \{\overline{X}_1\,\mathtt{in}\,(\mathtt{Re}_1, u_1), \ldots, \overline{X}_n\,\mathtt{in}\,(\mathtt{Re}_n, u_n)\}\rangle) :=$ the multiset $\{\omega(\mathtt{Re}_1), \ldots, \omega(\mathtt{Re}_n)\}$.
– $m_5(\langle E \mid \mathcal{K}\rangle) := |\mathcal{K}|$,
– $m_i(\langle P \mid S\rangle) = m_i(P)$ for all $1 \leq i \leq 5$.

It is not hard to verify that if $P \Rightarrow_{\alpha,\sigma} P'$ then the complexity measures of P relate to the complexity measures of P' as indicated in Fig. 1.

Let $\geq$ be the natural order on $\mathbb{N}$, $\geq^m$ be the extension of $\geq_\mathbb{N}$ to multisets of non-negative numbers, and $>_{lex(\geq^m,\geq,\geq^m,\geq^m,\geq)}$ the lexicographic combination of $\geq^m, \geq, \geq^m, \geq^m, \geq$ in this order. Figure 1 shows that, if we define $m(P) := (m_1(P), m_2(P), m_3(P), m_4(P), m_5(P))$ and

$P \Rightarrow_{\alpha,\sigma} P'$ or $\langle P \mid S\rangle \Rightarrow_{\alpha,\sigma} P' \mid S'$	m_1	m_2	m_3	m_4	m_5
$\alpha \in$ {t,iv,fv,rd,hv2,hv4,ciq,hiq,cf1,hf1,any0,any1}	$>$				
$\alpha \in$ {uhv,hv3}	$\geq$	$>$			
$\alpha \in$ {hv1,hv4,cv1,cv2,cf2,hf2,icc,ich,ihh,c*,c`*`,c`+1`,c`+2`}	$\geq$	$\geq$	$>$		
$\alpha \in$ {ac}	$\geq$	$\geq$	$\geq$	$>$	
$\alpha \in$ {thf}	$\geq$	$\geq$	$\geq$	$\geq$	$>$

Fig. 1. Variation of complexity measures of RMPs and RMP$_\mathrm{a}$-s during $\mathcal{U}$- and $\mathcal{U}'$-transformation steps.

$$P \triangleright P' \text{ iff } m(P) >_{lex(\geq^m,\geq,\geq^m,\geq^m,\geq)} m(P'),$$

then $P \triangleright P'$ whenever $P \Rightarrow_{\alpha,\sigma} P'$. Since $\triangleright$ is a terminating order, we conclude that our $\mathcal{U}$-based matching procedure is also terminating.

3 Conclusion

The transformation rules of $\mathcal{U}$ simulate a leftmost outermost pattern matching process. As long as the matching instance of the pattern can be computed directly by an analysis of the leftmost outermost pattern symbol, and possibly its associated membership constraint, the process proceeds as expected, in accordance with the small step operational semantics defined by the transformation rules of $\mathcal{U}\setminus$\{cf1, cf2, hf1, hf2\}. The pattern matching strategy is slightly different when we encounter a second order (hedge or contextual) variable $\overline{X}$ at leftmost outermost position, whose associated constraint is of the form $\overline{X}$ `in` $(\overline{C}(\texttt{Re}), u)$. In this case, we compute first the matching substitution for $\overline{C}$ and delay the matching of the corresponding instance of $\overline{X}$ below the hole occurrences of the instance of $\overline{C}$. These situations are handled by the big step operational semantics defined by the transformation rules of {cf1, cf2, hf1, hf2}.

Towards Practical Reflection for Formal Mathematics

Martin Giese and Bruno Buchberger

Research Institute for Symbolic Computation (RISC)
Johannes Kepler University Linz, Austria
martin.giese@risc.uni-linz.ac.at
bruno.buchberger@risc.uni-linz.ac.at

Abstract. We describe a design for a system for mathematical theory exploration that can be extended by implementing new reasoners using the logical input language of the system. Such new reasoners can be applied like the built-in reasoners, and it is possible to reason about them, e.g., proving their soundness, within the system. This is achieved in a practical and attractive way by adding reflection, i.e., a representation mechanism for terms and formulae, to the system's logical language, and some knowledge about these entities to the system's basic reasoners. The approach has been evaluated using a prototypical implementation called Mini-Tma. It will be incorporated into the Theorema system.

Introduction

Mathematical theory exploration consists not only of inventing axioms and proving theorems. Amongst other activities, it also includes the discovery of algorithmic ways of computing solutions to certain problems, and reasoning about such algorithms, e.g., to verify their correctness. What is rarely recognized is that it also includes the discovery and validation of useful techniques for proving theorems within a particular mathematical domain. In some cases, these reasoning techniques might even be algorithmic, making it possible to implement and verify a specialized theorem prover for that domain.

While various systems for automated theorem proving have been constructed over the past years, some of them specially for mathematics, and some of them quite powerful, they essentially treat theorem proving methods as a built-in part of the services supplied by a system, in general allowing users only to state axioms and theorems, and then to construct proofs for the theorems, interactively or automatically. An extension and adaptation of the theorem proving capabilities themselves, to incorporate knowledge about appropriate reasoning techniques in a given domain, is only possible by stepping back from the theorem proving activity, and modifying the theorem proving software itself, programming in whatever language that system happens to be written.

We consider this to be limiting in two respects:

– To perform this task, which should be an integral part of the exploration process, the user needs to switch to a different language and a radically

different way of interacting with the system. Usually it will also require an inordinate amount of insight into the architecture of the system.
– The theorem proving procedures programmed in this way cannot be made the object of the mathematical studies inside the system: e.g., there is no simple way to prove the soundness of a newly written reasoner within the system. It's part of the system's code, but it's not available as part of the system's knowledge.

Following a proposal of Buchberger [3, 4], and as part of an ongoing effort to redesign and reimplement the Theorema system [5], we will extend that system's capabilities in such a way that the definition of and the reasoning about new theorem proving methods is possible seamlessly through the same user interface as the more conventional tasks of mathematical theory exploration.

In this paper, we briefly outline our approach as it has been implemented by the first author in a prototype called Mini-Tma, a Mathematica [9] program which does not share any of the code of the current Theorema implementation. Essentially the same approach will be followed in the upcoming new implementation of Theorema.

A more complete account of our work, including a discussion of some foundational issues, and various related work, is given in [6].

The Framework

We start from the logic previously employed in the Theorema system, namely higher-order predicate logic with sequence variables. Sequence variables [8] represent sequences of values, and are written like $\overline{xs}$. Amongst many other uses, sequence variables happen to be convenient in statements about terms and formulae, since term construction in our logic is a variable arity operation.

In order to make reasoning about syntactic structures as attractive as possible, we build a *quotation mechanism* into the logic. For the representation of symbols, we require the signature to contain a *quoted version* of every symbol. Designating quotation by underlining, we write the quoted version of a as $\underline{\mathsf{a}}$, the quoted version of f as $\underline{\mathsf{f}}$, etc. For compound terms, we reuse the function application brackets $[\cdots]$ for term construction. This allows us to write $\underline{\mathsf{f}}[\underline{\mathsf{a}}]$ as quoted form of $f[a]$. To further simplify reading and writing of quoted expressions, Mini-Tma allows underlining a whole sub-expression as a shorthand for recursively underlining all occurring symbols.

For variable binding operators, like quantifiers, we prefer an explicit representation to, e.g., higher-order abstract syntax. The only derivation from a straight-forward representation is that we restrict ourselves to λ-abstraction as the only binding operator. Thus $\underset{x}{\forall}\, \mathsf{p}[x]$ is represented as

$$\underline{\mathsf{ForAll}}[\underline{\lambda}[\underline{x}, \underline{\mathsf{p}}[\underline{x}]]]$$

where $\underline{\lambda}$ is an ordinary (quoted) symbol, that does not have any binding properties.

The logical input language can be used to write programs. The standard computation mechanism interprets conditional equations as conditional rewriting rules. While this is the basic computation mechanism, we shall see later on that it is possible to define new computation mechanisms in Mini-Tma.

Mini-Tma does not include a predefined concept of *proving mechanism*. Theorem provers are simply realized as computation mechanisms that simplify a formula to True if they can prove it, and return it unchanged (or maybe partially simplified) otherwise.

Defining Reasoners

Since reasoners are just special computation mechanisms in Mini-Tma, we are interested in how to add a new computation mechanism to the system. This is done in two steps: first, using some existing computation mechanism, we define a function that takes a (quoted) term and a set of (quoted) axioms, and returns another (quoted) term. Then we tell the system that the defined function should be usable as computation mechanism with a certain name.

For instance, to define a reasoner that shifts parentheses in sums to the right, e.g., transforming the term Plus[Plus[a, b], Plus[c, d]] to Plus[a, Plus[b, Plus[c, d]]], we start by defining a function that will transform *representations* of terms, i.e., $\underline{\text{Plus}}[\underline{\text{Plus}}[\underline{\text{a}}, \underline{\text{b}}], \underline{\text{Plus}}[\underline{\text{c}}, \underline{\text{d}}]]$ to $\underline{\text{Plus}}[\underline{\text{a}}, \underline{\text{Plus}}[\underline{\text{b}}, \underline{\text{Plus}}[\underline{\text{c}}, \underline{\text{d}}]]]$. We can do this with the following definition:

$$
\begin{array}{l}
\text{Axioms}\,[\text{"shift parens"}, \text{any}[s, t, t_1, t_2, acc, l, ax, comp], \\
\quad \textsf{simp}[t, ax, comp] = \textsf{add-terms}[\textsf{collect}[t, \{\}]] \\
\quad \textsf{collect}\,[\underline{\text{Plus}}[t_1, t_2], acc] = \textsf{collect}[t_1, \textsf{collect}[t_2, acc]] \\
\quad \textsf{is-symbol}[t] \Rightarrow \textsf{collect}[t, acc] = \textsf{cons}[t, acc] \\
\quad \textsf{head}[t] \neq \underline{\text{Plus}} \Rightarrow \textsf{collect}[t, acc] = \textsf{cons}[t, acc] \\
\quad \textsf{add-terms}[\{\}] = \underline{0} \\
\quad \textsf{add-terms}[\textsf{cons}[t, \{\}]] = t \\
\quad \textsf{add-terms}[\textsf{cons}[s, \textsf{cons}[t, l]]] = \underline{\text{Plus}}[s, \textsf{add-terms}[\textsf{cons}[t, l]]] \\
]
\end{array}
$$

The main function is simp, its arguments are the term t, the set of axioms ax, and another computation mechanism *comp*, which may be used to recursively evaluate subterms. Given these axioms, we can ask the system to simplify a term:

$$
\begin{array}{l}
\text{Compute}[\textsf{simp}[\underline{\text{Plus}}[\underline{\text{Plus}}[\underline{\text{a}}, \underline{\text{b}}], \underline{\text{Plus}}[\underline{\text{c}}, \underline{\text{d}}]]], \{\}, \{\}], \text{by} \rightarrow \text{ConditionalRewriting}, \\
\qquad \text{using} \rightarrow \{\text{Axioms}[\text{"shift parens"}], \ldots\}]
\end{array}
$$

We are passing in dummy arguments for ax and *comp*, since they will be discarded anyway. Mini-Tma will answer with the term $\underline{\text{Plus}}[\underline{\text{a}}, \underline{\text{Plus}}[\underline{\text{b}}, \underline{\text{Plus}}[\underline{\text{c}}, \underline{\text{d}}]]]$.

We can now make simp known to the system as a computation mechanism. After typing

$$
\begin{array}{l}
\text{DeclareComputer}[\text{ShiftParens}, \textsf{simp}, \text{by} \rightarrow \text{ConditionalRewriting}, \\
\qquad \text{using} \rightarrow \{\text{Axioms}[\text{"shift parens"}, \ldots]\}
\end{array}
$$

the system recognizes a new computation mechanism named ShiftParens. When we ask it to

$$\text{Compute}[\mathsf{Plus}[\mathsf{Plus}[\mathsf{a},\mathsf{b}],\mathsf{Plus}[\mathsf{c},\mathsf{d}]],\text{by} \rightarrow \text{ShiftParens}]$$

we will receive the answer $\mathsf{Plus}[\mathsf{a},\mathsf{Plus}[\mathsf{b},\mathsf{Plus}[\mathsf{c},\mathsf{d}]]]$.

Reasoning About Logic

To prove statements about the terms and formulae of the logic, we need a prover that supports structural induction on terms, or *term induction* for short. An interesting aspect is that terms in Mini-Tma, like in Theorema, can have variable arity, and arbitrary terms can appear as the heads of complex terms. Sequence variables are very convenient in dealing with the variable length argument lists. Still, an induction schema needs to be employed that simultaneously performs induction over the depth of terms, and over the lengths of argument lists. Using the mechanism outlined in the previous section, were have implemented a simple term induction prover within Mini-Tma's logical input language.

Reasoning About Reasoners

At this point, it should come as no surprise that it is possible to use Mini-Tma to reason about reasoners written in Mini-Tma. The first application that comes to mind is proving the soundness of new reasoners: they should not be able to prove incorrect statements. Other applications include completeness for a certain class of problems, proving that a simplifier produces output of a certain form, etc.

Our current approach to proving soundness is to prove that anything a new reasoner can prove is true with respect to a model semantics. Or, for a simplifier that simplifies t to t', that t and t' have the same value with respect to the semantics. This approach has also been taken in the ACL2 system [7] and its predecessors.

Similarly to ACL2, we supply a function $\mathsf{eval}[t,\beta]$ that recursively evaluates a term t under some assignment β that provides the meaning of symbols. To prove the soundness of ShiftParens, we have to show

$$\mathsf{eval}[\mathsf{simp}[t, ax, comp], \beta] = \mathsf{eval}[t, \beta]$$

for any term t, any ax and $comp$ and any β with $\beta[\underline{\mathsf{0}}] = \mathsf{0}$ and $\beta[\underline{\mathsf{Plus}}] = \mathsf{Plus}$. With a strengthening of the induction hypothesis, and some adaptation of the induction prover, Mini-Tma can prove this statement automatically.

Ultimately, we intend to improve and extend the presented approach, so that it will be possible to successively perform the following tasks within a single framework, using a common logical language and a single interface to the system:

1. define and prove theorems about the concept of Gröbner bases [1],
2. implement an algorithm to compute Gröbner bases,

3. prove that the implementation is correct,
4. implement a new theorem prover for statements in geometry based on coordinatization, and which uses our implementation of the Gröbner bases algorithm,
5. prove soundness of the new theorem prover, using the shown properties of the Gröbner bases algorithm,
6. prove theorems in geometry using the new theorem prover, in the same way as other theorem provers are used in the system.

Though the case studies performed so far are comparatively modest, we are convinced that the outlined approach can be extended to more complex applications.

References

1. B. Buchberger. Ein algorithmisches Kriterium für die Lösbarkeit eines algebraischen Gleichungssystems. *Aequationes Math.*, 4:374–383, 1970. English translation published in [2].
2. B. Buchberger. An algorithmic criterion for the solvability of algebraic systems of equations. In B. Buchberger and F. Winkler, editors, *Gröbner Bases and Applications, 33 Years of Gröbner Bases*, London Mathematical Society Lecture Notes Series 251. Cambridge University Press, 1998.
3. B. Buchberger. Lifting knowledge to the state of inferencing. Technical Report TR 2004-12-03, Research Institute for Symbolic Computation, Johannes Kepler University, Linz, Austria, 2004.
4. B. Buchberger. Proving by first and intermediate principles, November 2, 2004. Invited talk at Workshop on Types for Mathematics / Libraries of Formal Mathematics, University of Nijmegen, The Netherlands.
5. B. Buchberger, A. Crăciun, T. Jebelean, L. Kovács, 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.
6. M. Giese and B. Buchberger. Towards practical reflection for formal mathematics. Technical Report 07-05, RISC, Johannes Kepler University, 2007.
7. W. A. Hunt Jr., M. Kaufmann, R. B. Krug, J Strother Moore, and E. W. Smith. Meta reasoning in ACL2. In Joe Hurd and Th. F. Melham, eds., *Proc. Theorem Proving in Higher Order Logics, TPHOLs 2005, Oxford, UK*, volume 3603 of *LNCS*, pages 163–178. Springer, 2005.
8. T. Kutsia and B. Buchberger. Predicate logic with sequence variables and sequence function symbols. In A. Asperti, G. Bancerek, and A. Trybulec, editors, *Proc. 3rd Intl. Conf. on Mathematical Knowledge Management, MKM'04*, volume 3119 of *LNCS*, pages 205–219. Springer Verlag, 2004.
9. S. Wolfram. *The Mathematica Book*. Wolfram Media, 2003.

Automated Polynomial Invariant Generation over the Rationals for Imperative Program Verification in Theorema *

Laura Kovács

Research Institute for Symbolic Computation (RISC)
Johannes Kepler University Linz, Austria
kovacs@risc.uni-linz.ac.at

Extended Abstract

One of the most rigorous way to imperative program verification is by use of simple formal calculations based on versions of Hoare logic, namely involving either weakest precondition or strongest postcondition calculations [10, 8, 19]. Such calculations allow us to capture the semantics of small program components and in the process derive specifications that can be used to assess whether a given program fragment has the required functionality.

The interesting parts of the code are characterized by (nested) loops or recursions. For these parts, formal program verification *is* an appropriate tool. Verification of correctness of loops needs additional information, so-called annotations, expressing conditions at certain intermediate points of the program. These describe relationships between variables which are meant to "hold" during execution. That means, whenever control passes through that point (i. e. the program point is reached), the assertion should evaluate to `True`.

Assertions could be inserted at any point of the program, but in fact they are only necessary in loops as socalled loop invariants (inductive assertions). During execution, loop invariants have to evaluate to `True` before and after each iteration. A loop has many invariants. An appropriate invariant is an assertion that captures all relevant invariant properties of a loop.

Annotating a program is often non–trivial and needs a good understanding of how the algorithm works. The idea of the invariants is mostly identical to the basic design idea, and therefore most of the properties established during imperative program verification are either loop invariants or depend crucially on invariants. The effectiveness of automated formal verification is thus sensitive to the situation when invariants, even trivial one, can be deduced automatically. It is agreed [11] that finding automatically such annotations is in general impractical - thus most systems will just ask the user for the appropriate expression.

* The program verification project is supported by BMBWK (Austrian Ministry of Education, Science, and Culture), BMWA (Austrian Ministry of Economy and Work) and by MEC (Romanian Ministry of Education and Research) in the frame of the e-Austria Timişoara project. The *Theorema* system is supported by FWF (Austrian National Science Foundation) - SFB project F1302.

However, in most of the practical situations, finding the expression, or at least giving some useful hints, is quite feasible. For practical applications this may be very helpful to the user, since explicitly stated program invariants can help programmers by identifying program properties that must be preserved when modifying code.

Research into methods for automatically generating loop invariants goes a long way, starting with the works [12, 13]. However, success was somewhat limited for cases where only few arithmetic operations (mainly additions) among program variables were involved. Recently, due to the increased computing power of hardware, as well as advances in methods for symbolic manipulation and automated theorem proving, the problem of automated invariant generation is once again getting considerable attention [2, 3], based essentially on two main approaches, namely using either *static* either *dynamic* techniques for invariant discovery.

The *dynamic method* [9, 1, 6] executes a program on a collection of inputs and infers invariants from captured variable traces. In other words, dynamic invariant detection runs a program, observes the values that the program computes, and then reports properties that were true over the observed executions. Thus, the accuracy of the inferred invariant depends in part on the quality and completeness of the test cases; additional test cases might provide new data from which more accurate invariants can be inferred.

Contrarily to the dynamic approach, the *static method* of invariant generation operates on the program text, not on the test runs, therefore has the advantage that the reported properties are true for any program run [18, 16, 17, 14]. Our work is integrated in this particular branch of invariant generation. Theoretically, by the static approach one can detect sound invariants, particularly by applying methods from *abstract interpretation* [7] and polynomial algebra [4].

Using the ideas from static techniques for invariant generation, in our work algebraic and combinatorial based approaches are combined to reason about imperative loops with assignments, sequencing and conditionals. More precisely, by combining advanced techniques from algorithmic combinatorics, symbolic summation, computer algebra and computational logic, a framework is realized for generating polynomial loop invariants for imperative programs operating on numbers.

Polynomial identities found by an automatic analysis are useful for program verification, as they provide non-trivial valid assertions about the program, and thus significantly simplify the verification task. Finding valid polynomial identities (i.e., invariants) has applications in many classical data flow analysis problem [15], e.g., constant propagation, discovery of symbolic constants, discovery of loop induction variables, etc.

Analyzing the code of loops, we express the values of the loop variables at the current loop iteration in terms of their previously computed values. In other words, the assignment statements from a loop body form a system of *recurrence equations* describing the behavior of loop variables. A *solution to these recurrences*, that is a *closed form* solution of the loop variables, would allow us

to express their values as functions of the loop iteration. Hence, finding closed form solutions for recurrence equations can be considered a major challenge in reasoning about imperative loops, for automatically inferring invariant relations among the loop variables.

Following this idea, the key steps of the our method for invariant generation, providing thus also a possibility of proving automatically correctness of deterministic imperative programs, are:

(i) Assignment statements from a loop body are extracted which are used to generate a system of *recurrence equations* describing the behavior of the loop's variables that are changed at each iteration;
(ii) Methods from algorithmic combinatorics are used to *exactly solve* the recurrence equations, yielding the *closed form* for each loop variable;
(iii) *Algebraic dependencies* among possible exponentials of algebraic numbers occurring in the closed forms of the loop variables are derived using algebraic and combinatorial methods.
The result of these steps is that every program variable can be expressed as a polynomial in terms of the initial values of variables (when the loop is entered), loop counter, and some new variables that are polynomially related;
(iv) Loop counters are then eliminated by polynomial methods to derive a finite set of polynomial identities among the program variables as invariants. From this finite set, any polynomial identity serving a loop invariant can be derived.

In our invariant generation approach, A certain family of loops, called *"P-solvable"*, is defined (to stand for polynomial-solvable), for which the value of each program variable can be expressed as a polynomial in terms of the initial values of variables, the loop counter, and some new variables that are polynomially related among them. For such loops, we developed a systematic method for generating polynomial equations as loop invariants. Further, if the body of these loops include only assignments and conditionals, the method is *complete*. Namely it generates a set of polynomial equations as invariants from which any polynomial equation serving as an invariant can be derived. Many nontrivial algorithms working on numbers can be shown to be in fact implemented using P-solvable loops.

Exploiting the symbolic manipulation capabilities of the computer algebra system *Mathematica* [20], our method is implemented in a new software package called *Aligator*. By using several combinatorial packages developed at RISC, *Aligator* includes solving special classes of recurrence relations (that are Gosper-summable or C-finite) and generating polynomial dependencies among algebraic exponential sequences, as well as manipulating polynomial relations based on the theory of Gröbner basis. Using *A*ligator, a complete set of polynomial invariants is successfully generated for numerous imperative programs working on numbers.

The automatically obtained invariant assertions are subsequently used for the partial correctness verification of programs, by generating the appropriate verification conditions as first-order logical formulae. Based on Hoare logic and

the weakest precondition strategy, this verification process is supported in an imperative verification environment implemented in the *Theorema* system [5]. *Theorema* seems to be convenient for such an integration given that it is built on top of the computer algebra system *Mathematica* and it includes automated methods for theorem proving in predicate logic, domain specific reasoning as well as induction proving.

Acknowledgments. The author wishes to thank prof. Deepak Kapur (University of New Mexico), prof. T. Jebelean (RISC-Linz) and M. Kauers (RISC-Linz) for their help and comments.

References

1. J. H. Andrews. Testing Using Log File Analysis: Tools, Methods and Issues. In *Proc. of 13th Annual Int. Conference on Automated Software Engineering (ASE'98)*, 1998.
2. S. Bensalem, Y. Lakhnech, and H. Saidi. Powerful Techniques for the Automatic Generation of Invariants. In *Proc. of SAS 1996*, volume 1102 of *LNCS*, pages 323–335, 1996.
3. N. Bjørner, A. Browne, and Z. Manna. Automatic Generation of Invariants and Intermediate Assertions. *Theoretical Computer Science*, 173(1):49–87, 1997.
4. B. Buchberger. Gröbner-Bases: An Algorithmic Method in Polynomial Ideal Theory. In *Multidimensional Systems Theory - Progress, Directions and Open Problems in Multidimensional Systems*, pages 184–232, 1985.
5. B. Buchberger, A. Crăciun, T. Jebelean, L. Kovács, 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.
6. J. E. Cook and A. L. Wolf. Discovering models of software processes from evet-based data. *ACM Transactions on Software Engineering and Methodology*, 7(3):215–249, 1998.
7. P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints among Variables of a Program. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 84–97, Tucson, Arizona, 1978.
8. E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
9. M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically Discovering Likely Program Invariants to Support Program Evaluation. *IEEE*, TSE 27(2), 2001.
10. R. W. Floyd. Assigning Meanings to Programs. In *Proc. Symphosia in Applied Mathematics 19*, pages 19–37, 1967.
11. G. Futschek. *Programmentwicklung und Verifikation*. Springer, 1989.
12. S. M. German and B. Wegbreit. A Synthesizer of Inductive Assertions. In *IEEE Transactions on Software Engineering*, pages 68–75, March 1975. 1(1):68-75.
13. M. Karr. Affine Relationships Amomg Variables of Programs. *Acta Informatica*, 6:133–151, 1976.
14. L. Kovacs. Finding Polynomial Invariants for Imperative Loops in the Theorema System. Technical Report 06-03, RISC-Linz, Austria, 2006.
15. M. Müller-Olm, M. Petter, and H. Seidl. Interprocedurally Analyzing Polynomial Identities. In *Proc. of STACS 2006*, Marseille, France, 2006.

16. M. Müller-Olm and H. Seidl. Computing Polynomial Program Invariants. *Indormation Processing Letters*, 91(5):233–244, 2004.
17. E. Rodriguez-Carbonell and D. Kapur. Generating All Polynomial Invariants in Simple Loops. *J. of Symbolic Computation*, 42(4):443–476, 2007.
18. S. Sankaranaryanan, H. B. Sipma, and Z. Manna. Non-Linear Loop Invariant Generation using Gröbner Bases. In *Proc. of POPL 2004*, Venice, Italy, 2004.
19. G. Winskel. *The Formal Semantics of Programming Languages. An Introduction*. The MIT Pres, 1994.
20. S. Wolfram. *The Mathematica Book*. Wolfram Media, 2003.