

Abstract Matrices in Symbolic Computation

Alan Sexton and Volker Sorge
School of Computer Science
University of Birmingham

aps@cs.bham.ac.uk, vxs@cs.bham.ac.uk

ABSTRACT

We introduce a new data type of *abstract matrices* that allows the description of underspecified matrices containing ellipses and their use as templates for classes of concrete matrices. We present a series of algorithms that fully analyses the structure of abstract matrices and their representation and supports subsequent instantiation to concrete matrices.

Categories and Subject Descriptors

I.1.1 [Symbolic and Algebraic Manipulation]: Expressions and Their Representation; I.1.2 [Symbolic and Algebraic Manipulation]: Algorithms—*Algebraic algorithms*

General Terms

Algorithms

Keywords

Underspecified Matrices, Constraints, Semantic Analysis

1. INTRODUCTION

In every day mathematical practice, matrices are often not fully specified, but rather are of indefinite dimension and contain abbreviations and underspecified parts such as ellipses (usually written as a series of dots indicating the omission of a number of terms). Thus the matrix

$$A = \begin{bmatrix} a_1 & b & \cdots & b \\ 0 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & b \\ 0 & \cdots & 0 & a_n \end{bmatrix} \quad (1)$$

can be considered as a template of the class of all square matrices of the above shape. The concrete instantiations for dimension n where $1 \leq n \leq 4$ are then:

$$[a_1], \quad \begin{bmatrix} a_1 & b \\ 0 & a_2 \end{bmatrix}, \quad \begin{bmatrix} a_1 & b & b \\ 0 & a_2 & b \\ 0 & 0 & a_3 \end{bmatrix}, \quad \begin{bmatrix} a_1 & b & b & b \\ 0 & a_2 & b & b \\ 0 & 0 & a_3 & b \\ 0 & 0 & 0 & a_4 \end{bmatrix}$$

While using underspecified matrices is routine, it has very limited automated support. For example, Maple [1, 3] provides functionality to specify matrices with some predefined

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ISSAC'06, July 9–12, 2006, Genova, Italy.

Copyright 2006 ACM 1-59593-276-3/06/0007 ...\$5.00.

shapes, such as symmetric or hermitian. It also provides the facility to specify customised shapes via user defined functions. However, in all cases, the Maple support is only for fully specified matrices with fixed dimensions and no ellipses.

As a step towards rectifying this deficiency, we introduce a new data type, *Abstract Matrix*, that makes underspecified matrices first class objects in computer algebra. To support this data type, we present a series of algorithms that fully processes an input format for Abstract Matrices that is very similar to that which appears in mathematical texts, but adapted for automated systems, analyse its semantics and produce a structure that supports subsequent incremental refinement via the addition of extra constraints and eventual instantiation to a valid matching concrete matrix.

There are a number of subtleties in the interpretation of underspecified matrices. For example, some matrix expressions that appear in the literature have ellipses that are intended to instantiate into descending sequences of index values such as a_{-1}, a_{-2}, \dots . Another possibility is that for the instantiation $n = 1$ the dimension of matrix is not fixed to 1 but rather the main diagonal contains only the constant a_1 . If one wishes to accommodate those cases, then it is probably not reasonable to disallow the following possible, if unusual, instantiations of (1).

$$\begin{bmatrix} a_1 & b & b \\ 0 & a_0 & b \\ 0 & 0 & a_{-1} \end{bmatrix} \quad \begin{bmatrix} a_1 & b & b \\ 0 & a_1 & b \\ 0 & 0 & a_1 \end{bmatrix}$$

However, while mathematically there may be no problem with allowing any arbitrary integer sequence, it is extremely rare to find anything other than simple increment or decrement by one for matrix ellipsis sequences in mathematical texts. For this reason, we impose the restriction that all such ellipses are restricted to such increments, decrements or may remain constant. Furthermore we assume that every ellipsis is either vertical, horizontal, diagonal or anti-diagonal, and that, for the latter two cases, their vertical and horizontal lengths must be equal. Thus a matrix with a diagonal ellipsis from its top left to its bottom right cells can be deduced to be square. Nevertheless, we can model truly rectangular matrices for our algorithms by specifying them as, for instance, in matrix (2) below.

$$\begin{bmatrix} a_{1,1} & \cdots & a_{1,m} \\ \vdots & \ddots & \vdots \\ a_{n,1} & \cdots & a_{n,m} \end{bmatrix} \quad (2) \quad \begin{bmatrix} a_1^1 & \cdots & \cdots & a_m^1 \\ \vdots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ a_1^n & 0 & \cdots & 0 \end{bmatrix} \quad (3)$$

For ease of computer analysis, and practical usability in Maple (in which we implemented our programs) we require our input to be a matrix structure where all cells must be filled. We employ a *dot* term ‘.’ for cells that would nor-

mally be left blank. We allow for distinguished terms to represent the dot and any of the four ellipsis symbols, and a distinguished function symbol to indicate that a region can be filled with a term that does not change over the different cells of that area (for examples of a fill term, see matrix (4) below). Finally, we refer to all normal terms as *concrete terms* and require all ellipses to be terminated on both ends by concrete terms and to contain, between their end terms, only ellipsis terms of the correct direction. We will use matrix (3) above as running example throughout the paper. Observe that in the term a_n^1 , the 1 is an exponent.

Preliminary reports [8, 7] discussed our approach but here we present the detailed algorithms and formal syntax.

2. OVERVIEW OF THE ALGORITHMS

We first present a high-level overview of our algorithms. The structure of their descriptions also corresponds to the overall layout of the paper.

1. **Parsing** to produce an abstract matrix object from an input specification
 - (a) **Input matrix syntax** We define our input syntax using a matrix grammar that allows precise specification of the syntactic structure parsing of the input of our algorithm
 - (b) **Ellipsis analysis** to identify how the underspecified parts of the matrix can change in size
 - i. **Matrix vertex graph construction** to capture ellipsis lengths as constraint variables
 - ii. **Graph analysis** to build constraints relating the ellipsis length constraint variables
 - (c) **Region Finding** Identify regions and their shapes by using a 2-D region finding algorithm
 - (d) **Region analysis** to discover region contents
 - i. **Region boundary points analysis** with an anti-unification algorithm
 - ii. **Interpolation function construction** to capture semantics of content of region
 - iii. **Subterm Constraints** capture possible relationships between identified indexing functions and the structure of regions
2. **Concretisation** Iteratively refine the matrices to a more concrete form by binding constraint variables and propagating the results to produce a refined abstract matrix with fewer free constraint variables
3. **Instantiation** Take an abstract matrix with no free constraint variables and produce a fully concrete matrix by interpolating regions

3. PARSING

3.1 Input Matrix Syntax

In this section we define an isotonic *matrix grammar* to specify the input syntax for the abstract matrix algorithm. Matrix grammars are specialised versions of the more general *array grammars* [6]. The productions of such a grammar replace 2-dimensional patterns of symbols with patterns of other symbols of precisely the same size and geometric shape. A background symbol, #, can also be specified and substituted for. Thus, given a rectangular array of specified size which contains only the background symbol # and one occurrence of the non-terminal start symbol S , the productions replace all occurrences of # and the non-terminals and derive a matrix that is a valid input for our algorithm.

$S \rightarrow T$	(1.1)	$S \rightarrow F$	(1.2)
$F \rightarrow \text{fill}(x)$	(2.1)	$F \rightarrow \cdot$	(2.2)
$F \# \rightarrow F S$	(2.3)	$F \# \rightarrow F S$	(2.5)
$F \rightarrow F$	(2.4)	$F \rightarrow S$	(2.6)
$F \rightarrow S$	(2.4)	$\# \rightarrow S$	(2.6)
$T \rightarrow x$	(3.1)	$T \# \rightarrow T S$	(3.4)
$T \# \rightarrow T S$	(3.2)	$T \rightarrow T$	(3.5)
$T \rightarrow T$	(3.3)	$\# \rightarrow S$	(3.5)
$\# \rightarrow S$	(3.3)	$T \# \rightarrow T D$	(3.8)
$T \# \rightarrow T H$	(3.6)	$\# \rightarrow A$	(3.9)
$T \rightarrow T$	(3.7)		
$\# \rightarrow V$	(3.7)		
$H \rightarrow T$	(4.1)	$H \# \rightarrow \dots H$	(4.3)
$H T \rightarrow \dots T$	(4.2)		
$V \rightarrow T$	(5.1)	$V \rightarrow \vdots$	(5.3)
$V \rightarrow \vdots$	(5.2)	$\# \rightarrow V$	(5.3)
$T \rightarrow T$	(5.2)		
$D \rightarrow T$	(6.1)	$D \# \rightarrow \dots D$	(6.3)
$D T \rightarrow \dots T$	(6.2)		
$A \rightarrow T$	(7.1)	$\# A \rightarrow A \dots$	(7.3)
$T A \rightarrow T \dots$	(7.2)		

Table 1: Grammar rules for the input syntax.

We define the rules of the grammar over the finite alphabet $\{x, \text{fill}(x), \dots, \dots, \dots, \dots, \dots, \dots\}$, where the terminal symbol x stands for an arbitrary term different from the non-terminal symbols. That is, two occurrences of x do not necessarily have to represent the same term. $\text{fill}(x)$ stands for the fill function of some symbol x . The set of non-terminal symbols in the grammar is $\{S, T, F, V, H, D, A\}$, where S is the start symbol. For each $n \times m$ input matrix we have a $n \times m$ start matrix in which all the cells contain # except the top left corner cell which contains the start symbol S .

The input syntax that our algorithm accepts is characterised by the productions given in table 1 and an example derivation for matrix (2) is shown in table 2.

The grammar ensures that sequences of ellipsis symbols are always of the same type and bounded by concrete terms on both sides. However, the more detailed semantic restrictions imposed on the input matrices by our algorithms can of course not be captured in the grammar. We will refer to those in more detail in §3.4.

3.2 Ellipsis Analysis

An ellipsis is a pattern which can, in context, match multiple different areas of varying sizes and contents. The length of the line of cells that an ellipsis can match is constrained, in a mutually interdependent fashion, by the length, position and connectivity of the other ellipses. We capture the length of each ellipsis with a non-negative, integer valued constraint variable, and the relationship between the ellip-

$$\begin{array}{c}
\begin{bmatrix} S & \# & \# \\ \# & \# & \# \\ \# & \# & \# \end{bmatrix} \xrightarrow{1.1} \begin{bmatrix} T & \# & \# \\ \# & \# & \# \\ \# & \# & \# \end{bmatrix} \xrightarrow{3.6} \begin{bmatrix} T & H & \# \\ \# & \# & \# \\ \# & \# & \# \end{bmatrix} \xrightarrow{4.3} \begin{bmatrix} T & \dots & H \\ \# & \# & \# \\ \# & \# & \# \end{bmatrix} \xrightarrow{4.1} \begin{bmatrix} T & \dots & T \\ \# & \# & \# \\ \# & \# & \# \end{bmatrix} \xrightarrow{\begin{matrix} 3.7 \\ 5.3 \\ 5.1 \end{matrix}} \begin{bmatrix} T & \dots & T \\ \vdots & \# & \# \\ T & \# & \# \end{bmatrix} \\
\\
\xrightarrow{3.4} \begin{bmatrix} T & \dots & T \\ \vdots & S & \# \\ T & \# & \# \end{bmatrix} \xrightarrow{\begin{matrix} 1.2 \\ 2.2 \end{matrix}} \begin{bmatrix} T & \dots & T \\ \vdots & \cdot & \# \\ T & \# & \# \end{bmatrix} \xrightarrow{\begin{matrix} 3.7 \\ 5.3 \\ 5.1 \end{matrix}} \begin{bmatrix} T & \dots & T \\ \vdots & \cdot & \# \\ T & \# & T \end{bmatrix} \xrightarrow{\begin{matrix} 3.6 \\ 4.2 \end{matrix}} \begin{bmatrix} T & \dots & T \\ \vdots & \cdot & \# \\ T & \dots & T \end{bmatrix} \xrightarrow{3.1 \times 4} \begin{bmatrix} a_{1,1} & \dots & a_{1,m} \\ \vdots & \cdot & \vdots \\ a_{n,1} & \dots & a_{n,m} \end{bmatrix}
\end{array}$$

Table 2: Derivation of matrix (2) with production rule applications indicated by number.

sis lengths with integer valued linear constraint equations. We define the length of an ellipsis to be the number of cells it matches in a concrete matrix. For this to make sense we think of ellipses as extending from the extreme edges, or corners, of their terminating cells and count their length as the total number of cells they cross. This also implies that the length of a single cell, as we would expect, is 1.

We must find the precise constraint equations relating the ellipsis lengths in order to capture the shape of all possible consistent concrete instantiations of the abstract matrix. This can be done by equating the lengths of different paths between the same points on the matrix, where each step in the path traverses a single concrete cell for a cost of 1, or a single ellipsis for a cost which is represented by its length constraint variable. Since paths can move in two dimensions, we separate the vertical and horizontal components of the path lengths. Since we need to generate all paths, we construct a graph whose vertices correspond to the reachable points in the matrix, i.e. the 4 corner points surrounding each cell which contains a concrete term. It is important to note that the graph vertices correspond to the *corners* of the matrix cells, not to the cells themselves. The edges of this graph correspond to the traversal possibilities afforded by ellipses and concrete terms. The weights on the edges are pairs of traversal costs, a traversal cost being either the integer 1 or an integer constraint variable.

Thus cell (i, j) in the matrix¹ has its upper left corner associated with vertex $\langle i, j \rangle$ in the graph and its lower right corner associated with vertex $\langle i + 1, j + 1 \rangle$. We create the graph vertices lazily and do not create any that have no edges incident on them. We write a weighted edge from $\langle p, q \rangle$ to $\langle r, s \rangle$ with vertical weight v and horizontal weight h as $\langle p, q \rangle \xrightarrow{v,h} \langle r, s \rangle$.

3.2.1 Vertex Connectivity Graph Construction

In *createConnectionGraph*, we construct a minimal initial graph in an adjacency list structure directly from the parsing of the input matrix and add its symmetric closure (with suitable negated edge weights). We make use of two functions, *upleft* and *downright* defined as follows:

	<i>upleft</i> (i, j, t)	<i>downright</i> (i, j, t)
if $t = \dots'$	$(i, j - 1)$	$(i, j + 1)$
if $t = \cdot'$	$(i - 1, j)$	$(i + 1, j)$
if $t = \dots'$	$(i - 1, j - 1)$	$(i + 1, j + 1)$
if $t = \cdot \dots'$	$(i - 1, j + 1)$	$(i + 1, j - 1)$

¹ As is usual in the literature of matrices, (i, j) refers to the cell with row i and column j and the top left cell of the matrix is $(1, 1)$.

Given a cell location and an ellipsis term, these functions return the neighbouring location that is one cell away along the line of the ellipsis in the specified vertical direction, if possible, or the specified horizontal direction otherwise.

The algorithm iterates over each cell in the input matrix. If it contains a concrete term, 3 surrounding edges of weight 1 are added. If it contains an ellipsis term it finds the end cells of the ellipsis and adds the appropriate edge. It keeps track of which cells it has processed so that it only adds one edge for an ellipsis rather than one for each ellipsis term on the ellipsis. Finally it adds the symmetric completion of the graph. For matrix (3), the graph produced, minus the extra reverse edges and with no weights indicated, is shown in figure (1). Error messages are omitted here for space reasons but are present in the implementation.

createConnectionGraph(M)

- 1: let graph $G \leftarrow \emptyset$
- 2: let *visited* $\leftarrow \emptyset$
- 3: for each cell location (i, j) of M , in lexicographic order
- 4: if $M(i, j)$ is a concrete term
- 5: add 3 edges to G : $\langle i, j \rangle \xrightarrow{0,1} \langle i, j + 1 \rangle$,
 $\langle i, j \rangle \xrightarrow{1,0} \langle i + 1, j \rangle$ and
 $\langle i + 1, j \rangle \xrightarrow{0,1} \langle i + 1, j + 1 \rangle$.
- 6: else if $(i, j) \notin \textit{visited}$ and $M(i, j)$ is an ellipsis term t
- 7: add (i, j) to *visited*
- 8: $(p, q) \leftarrow \textit{upleft}(i, j, t)$
- 9: if (p, q) is out of bounds of M
or $M(p, q)$ is not a concrete term
report input error
- 10: $(r, s) \leftarrow \textit{downright}(i, j, t)$
- 11: while (r, s) is in bounds of M and $M(r, s) = t$
- 12: add (r, s) to *visited*
- 13: $(r, s) \leftarrow \textit{downright}(r, s, t)$
- 14: if (r, s) is out of bounds of M
or $M(r, s)$ is not a concrete term
report input error
- 15: create fresh constraint variable e
- 16: case ellipsis of
- 17: horizontal: add to G : $\langle p, q \rangle \xrightarrow{0,e} \langle p, s + 1 \rangle$
- 18: vertical: add to G : $\langle p, q \rangle \xrightarrow{e,0} \langle r + 1, q \rangle$
- 19: diagonal: add to G : $\langle p, q \rangle \xrightarrow{e,e} \langle r + 1, s + 1 \rangle$
- 20: anti-diagonal: add to G : $\langle p, q + 1 \rangle \xrightarrow{e,-e} \langle r + 1, s \rangle$
- 21: else if $(i, j) \notin \textit{visited}$
and $M(i, j)$ is not a dot or fill term
- 22: report input error
- 23: let E be the edges of G
- 24: for each edge $\langle p, q \rangle \xrightarrow{v,h} \langle r, s \rangle$ in E
- 25: add $\langle r, s \rangle \xrightarrow{-v,-h} \langle p, q \rangle$ to G
- 26: return G

In *getGeneralisedPositions*, we assume (possibly empty) lists of vertices *Top*, *Bottom*, *Left* and *Right*, that contain vertices of the connection graph that are on the corresponding sides of the input matrix (these are trivially computable from the input). Further, we assume a search function $firstIn(X, \phi(\cdot))$, which searches for the first element x in the list X such that $\phi(x)$ and returns \perp if no matching elements are found. We use $v.w$ and $w.h$ to refer to the vertical and horizontal components respectively of a weight pair w .

For each connected component, the algorithm takes one vertex and, if necessary, adds constraint variables to capture its distance to the four sides. It then adds equations to relate the width and height of the matrix to the distances from the vertex to the sides of the matrix, thus providing a generalised position for the vertex. The generalised positions of all remaining vertices in the component can be calculated from the generalised position of the first and returned in P .

```

getGeneralisedPositions(Q, V, W)
1: create fresh constraint variables,  $e_{Width}$  and  $e_{Height}$ 
2: for each list  $L$  in  $V$ 
3:   let  $f \leftarrow$  the first element of  $L$ 
4:   let  $x \leftarrow firstIn(Top, W(\cdot, f) \neq \perp)$ 
5:   if  $x = \perp$  then let  $t \leftarrow$  a fresh constraint variable
6:   else let  $t \leftarrow W(x, f).v$ 
7:   let  $x \leftarrow firstIn(Left, W(\cdot, f) \neq \perp)$ 
8:   if  $x = \perp$  then let  $l \leftarrow$  a fresh constraint variable
9:   else let  $l \leftarrow W(x, f).h$ 
10:  let  $x \leftarrow firstIn(Bottom, W(f, \cdot) \neq \perp)$ 
11:  if  $x = \perp$  then let  $b \leftarrow$  a fresh constraint variable
12:  else let  $b \leftarrow W(f, x).v$ 
13:  let  $x \leftarrow firstIn(Top, W(f, \cdot) \neq \perp)$ 
14:  if  $x = \perp$  then let  $r \leftarrow$  a fresh constraint variable
15:  else let  $r \leftarrow W(f, x).h$ 
16:  add  $l + r - e_{Width} = 0$  and  $t + b - e_{Height} = 0$  to  $Q$ 
17:  for each vertex  $x$  in  $L$ 
18:    let  $P(x) \leftarrow (t + W(f, x).v, l + W(f, x).h)$ 
19: return  $Q, P$ 

```

3.3 Region Finding

Regions are contiguous areas of the matrix whose contents are filled by a single parameterised term which we call a *generalised term*. The parameterisation determines how the generalised term changes with its relative position within the region and with changes in the size of the various regions in the whole matrix. Computation of generalised terms is discussed in §3.4. There are three regions in matrix (1); the lower left triangle of constant zero terms, the upper right triangle of constant b terms and the diagonal of a_α terms where α is a parameter whose value is to be determined for each cell in the region. We capture the semantics of the matrix as a whole by identifying the regions that the matrix is composed of together with the constraints that relate the sizes, shapes and contents of each region.

Note that, while the diagonal region in matrix (1) can grow in only one dimension, the triangular regions are 2-dimensional. Regions which are 2-dimensional do not have to be triangular, c.f. matrix (2), but can be any shape bounded by arbitrary, non-crossing, closed polylines of ellipses and cells containing concrete terms, or even by the matrix boundaries itself in the case of fill regions. Furthermore there can be 1-dimensional, or *linear*, regions as well as 0-dimensional, regions, which contain a single concrete term.

We can identify two types of 2-dimensional regions. These are *dot regions* that, in the input matrix, contain dot and/or

fill terms and *tight triangle regions* — i.e. regions consisting of three ellipses where each ellipsis is exactly 3 cells long (a concrete term, an ellipsis and another concrete term) and shares its terminal concrete cells with the other two ellipses. Thus the lower right triangle in matrix (3) is a tight triangle region, while the upper left is not but is a dot region. The following theorem says that these are the only types of 2-dimensional regions and we take advantage of this fact in the algorithm to find all regions in the matrix, *findRegions*.

THEOREM 1. *There are only two types of 2-dimensional regions: tight triangle regions and dot regions.*

PROOF SKETCH. Tight triangles are 2-dimensional and do not contain any dot or fill terms. Any other 2-dimensional region has to contain at least three ellipses; otherwise the return path to enclose the region has to be formed by single cells which constrain the ellipses to fixed lengths if they do not lie on a straight line with each other (in which case they do not have a two dimensional extent). Given that an ellipsis has to be at least 3 cells long in the input matrix, and can only lie in a vertical, horizontal or diagonal direction, a simple case analysis shows that any closed loop containing 3 or more ellipses must have a dot or fill term inside it. \square

In *findRegions*, M is the input matrix and the local variables E and C are used to collect the sets of ellipses and concrete cells which have been dealt with by the corresponding point in the algorithm. Angle brackets, $\langle \cdot \rangle$, are used to indicate record structures.

```

findRegions(M)
1: let  $D \leftarrow getDotGroups(M)$ 
2: let  $\langle R_D, E, C \rangle \leftarrow getDotRegions(M, D)$ 
3: let  $\langle R_T, E, C \rangle \leftarrow getMinTriangleRegions(M, E, C)$ 
4: let  $\langle R_L, C \rangle \leftarrow getLinearRegions(M, E, C)$ 
5: let  $R_S \leftarrow getSingleCellRegions(M, C)$ 
6: return  $R_D \cup R_T \cup R_L \cup R_S$ 

```

The set of dot groups in the input matrix is returned by *getDotGroups*. A single dot group is a maximal set of horizontally or vertically connected input matrix cell locations, each of which contains a dot or a fill term. Horizontally or vertically adjacency of locations are tested by *adjacent*.

```

getDotGroups(M)
1: let  $D \leftarrow \emptyset$ 
2: for each location  $c$  in  $M$  containing a dot or fill term
3:   let  $C \leftarrow \{d \in D \mid \exists x \in d. adjacent(x, c)\}$ 
4:   let  $D \leftarrow (D \setminus C) \cup \{c\} \cup C$ 
5: return  $D$ 

```

In *getDotRegions*, the dot groups are used to identify regions which contain dot terms. If the dot group contains a fill term then the region is a fill region. For each dot group, a start point on the boundary of the group is found by taking the topmost, leftmost cell in the group and applying a straightforward case analysis to find a boundary point in the neighbourhood of that cell together with a direction suitable for a clockwise traversal of the boundary at that point. Then boundary is followed in a clockwise direction to find all the boundary edges of the region. As it is possible that two disconnected dot groups are actually part of the same region (e.g. where the region is “pinched” but not closed off in the middle), more than one copy of some boundaries may have been found. In such cases, the duplicate boundaries are removed. Testing for duplicates is easy as one polyline boundary will be a cyclic permutation of the other. All boundary polylines are rotated into a canonical order (beginning with the least boundary point location

in lexicographic row/column order) before comparison and only the first two points need be checked.

In *getMinTriangleRegions*, tight triangle regions are found by a simple exhaustive search of appropriately connected ellipses that are each three cells long in the input matrix. Component ellipses and concrete cells are marked as used are added to the corresponding parameter sets.

In *getLinearRegions*, linear regions are produced from any unused ellipsis. The terminal concrete cell of each such ellipsis is added to the C concrete cell set.

In *getSingleCellRegions*, single cell regions are produced from any concrete cell not in C .

The result is a set of region boundaries from the input matrix. From this we construct a list of partial region structures which will be completed into full region structures in the region analysis phase, c.f. §3.4. To describe the partial region structures, we use angle brackets, as before, for records, and square brackets for lists. The partial region structure is represented as a list containing a fill term set and a list of boundary points. The fill term set contains the fill term if this is a fill region and is empty otherwise. The boundary point list length will be 3 or more for 2-dimensional, 2 for linear and 1 for single cell regions. Each boundary point is a list of 3 elements. The first element is the direction of traversal of the boundary to this boundary point. This is used by the instantiation algorithm (§5) to identify the interior of a region. These directions are recorded in the structures using constants but are described in the examples below using short arrows. The second element is the generalised position of the boundary point, and the third is the concrete term that occurs at that position. The partial region structure for the upper left triangle of matrix (3) would be:

$$[\emptyset, [[\uparrow, \langle 1, 1 \rangle, a_1^1], [\rightarrow, \langle p_1, 1 \rangle, a_m^1], [\swarrow, \langle 1, q_1 \rangle, a_i^n]]]$$

Thus it is not a fill region, it has 3 boundary points which can be traversed by going from the top left corner rightwards to the top right, down left to the bottom left and up back to the top left again.

3.4 Region Analysis

Once we have found all the regions, classified them as 0, 1 or 2-dimensional, and know their relative location as indicated by the generalised positions of their boundaries, we can start analysing the region contents. While single points do not have to be further analysed, for other regions we have to determine what their content is. This is done in three steps: find a suitable generalisation of the boundary terms of a region (§3.4.1), interpolate a region with respect to the boundary terms (§3.4.2), and relate the structure of the boundary terms to the lengths of the ellipses (§3.4.3).

For the following analysis we use the assumptions already mentioned in the introduction. Namely, that in the ellipses we have simple indexing functions over the integers, with an increase of -1, 0, or 1. These are semantic restrictions we impose on the type of abstract matrices that can be processed by the remainder of our algorithms. All algorithms up to this point have required only that the input matrix matches the grammar in §3.1. From this point on, further semantic restrictions apply.

3.4.1 Anti-Unification Algorithm

In order to compute the actual content of a region we compare all the boundary points of that region to establish the general form of the terms the region is composed of and to

find the indexing functions in those terms and their respective ranges. We employ a first-order anti-unification algorithm to compute a suitable generalisation of the boundary terms in the form of a least general generalisation or anti-unifier that unifies with all the boundary terms of a region. Our algorithm is, in spirit, very similar to the one given by Huet in [4], but since we are not in a strict logical setting, we can omit some of the classical anti-unification rules. In particular, we do not have a formal distinction between constants and free variables in our terms and therefore do not have to deal with possible unification between terms. However, we add some rules that deal with special cases arising in arithmetic. While our algorithm works on sets of terms, representing the boundary elements of a region, for simplicity we describe the anti-unification here for pairs of terms only. Its generalisation to term sets is straightforward.

Before we give the anti-unification algorithm, we define some necessary concepts: Let Σ be a finite signature containing constants and n -ary function symbols and $T(\Sigma)$ be the set of terms over the signature Σ . For the algorithm we choose $T(\Sigma)$ to be the set of concrete terms occurring in the input matrix, which corresponds to the possible instantiations of the terminal symbol x in the input grammar §3.1. This automatically determines Σ . Let \mathcal{V} be a set of variable symbols, such that $\mathcal{V} \cap \Sigma = \emptyset$. We define a substitution σ as a set $\sigma = \{v_1 \mapsto t_1, \dots, v_n \mapsto t_n\}$, where $v_1, \dots, v_n \in \mathcal{V}$, $v_i \neq v_j$ for $i \neq j$, and $t_1, \dots, t_n \in T(\Sigma)$, whose application $t\sigma$ replaces all occurrences of v_i by t_i in t for $i = 1, \dots, n$. We can now define the anti-unification algorithm as a recursive function that, for terms $s, t \in T(\Sigma)$, computes their least general generalisation Φ together with substitutions σ and τ such that $\Phi\sigma = s$ and $\Phi\tau = t$ as well as a set X of all unification variables used in the two substitutions. The algorithm fails if no first order anti-unifier can be computed.

- 1: let $\mathcal{V} \leftarrow \{\}$, $\sigma \leftarrow \{\}$, $\tau \leftarrow \{\}$
- 2: $\Phi \leftarrow \text{Anti-unify}(s, t)$
- 3: if $s = t$ then return s
- 4: if $s = c_1$ and $t = c_2$ with $c_1 \neq c_2$ each constants then
create new $v \notin \mathcal{V}$
 $\mathcal{V} \leftarrow \mathcal{V} \cup \{v\}$, $\sigma \leftarrow \sigma \cup \{v \mapsto s\}$, $\tau \leftarrow \tau \cup \{v \mapsto t\}$
return v
- 5: if $s = f(s_1, s_2, \dots, s_n)$ and $t = f(t_1, t_2, \dots, t_n)$ then
return $f(\text{Anti-unify}(s_1, t_1), \dots, \text{Anti-unify}(s_n, t_n))$
- 6: if $s = f(s_1, s_2, \dots, s_n)$ and $t = g(t_1, t_2, \dots, t_n)$
and $f \neq g$ then fail
- 7: if $s = \hat{\ }(s', s'')$ and $t = 1$ and s'' is a constant then
create new $v \notin \mathcal{V}$
 $\mathcal{V} \leftarrow \mathcal{V} \cup \{v\}$, $\sigma \leftarrow \sigma \cup \{v \mapsto s''\}$, $\tau \leftarrow \tau \cup \{v \mapsto 0\}$
return $\hat{\ }(s', v)$
- ...
- 8: end

Note that the above algorithm does not contain a case for variables as the input terms are variable free. Step 4 is the generalisation step, which ensures that for each generalised term a new variable is created. Step 5 only recursively descends the term, whereas step 6 makes sure that no functional expressions are generalised. The remaining cases, starting in step 7, deal with some arithmetic exceptions. Additional cases take care of the case symmetric to step 7 (i.e., $s = 1$ and $t = \hat{\ }(s', s'')$) as well as of exceptions for other arithmetic functions and their symmetric and possibly commutative cases such as for t^1 , $0 * t$, $1 * t$, $t + 1$. In order to deal easily with arithmetic functions, we rewrite all terms into prefix form in a preprocessing step. Thus in the overall

parsing algorithm we first extract the boundary terms from a given region structure and put all terms in infix form before applying the anti-unification algorithm to them. Upon successful completion of the algorithm we update the region structure by adding the anti-unifier and replacing each boundary term with the respective substitutions.

As an example, we consider the region from the previous section. While our algorithm does not require the explicit exponents 1, for clarity of the example we consider boundary terms a_1^1, a_m^1, a_1^n , which are represented in prefix form as $\wedge(a(1), 1)$, $\wedge(a(m), 1)$, and $\wedge(a(1), n)$. Anti-unification yields $\wedge(a(v_1), v_2)$ as least general generalisation, together with substitutions $\sigma_1 = \{v_1 \mapsto 1, v_2 \mapsto 1\}$, $\sigma_2 = \{v_1 \mapsto m, v_2 \mapsto 1\}$, $\sigma_3 = \{v_1 \mapsto 1, v_2 \mapsto n\}$, as well as $\mathcal{V} = \{v_1, v_2\}$. The updated region structure will then be of the form

$$[a_{v_1}^{v_2}, [\uparrow, \langle 1, 1 \rangle, \{v_1 \mapsto 1, v_2 \mapsto 1\}], \\ [\rightarrow, \langle p_1, 1 \rangle, \{v_1 \mapsto m, v_2 \mapsto 1\}], \\ [\swarrow, \langle 1, q_1 \rangle, \{v_1 \mapsto 1, v_2 \mapsto n\}]]]$$

3.4.2 Computing Interpolation Functions

Next we compute interpolation functions that determine the content of a region. Each unification variable in a region acts as a separate indexing function, whose range is determined by the respective substitutions for the boundary terms. The task is to find one function that computes the right intermediate values for each indexing function. If we view the value of the indexing function as a third dimension, the coordinates of a boundary term are then given by its generalised position together with the instantiation of the unification variable under consideration for each such variable. The problem is then to fit a 2-dimensional plane through a set of points in 3-dimensional space. The following algorithm returns an interpolation function in the index coordinates x, y by solving the determinant equation that represents the three-point form of the plane equation, with respect to the value of the indexing variables in step 3.

- 1: given $[\langle p_1, q_1 \rangle, \{v_1 \mapsto r_1, \dots, v_n \mapsto r_n\}],$
 $[\langle p_2, q_2 \rangle, \{v_1 \mapsto s_1, \dots, v_n \mapsto s_n\}],$
 $[\langle p_3, q_3 \rangle, \{v_1 \mapsto t_1, \dots, v_n \mapsto t_n\}]$
- 2: for $i \leftarrow 1, \dots, n$ do
- 3: $f_i(x, y) \leftarrow solve \left(\begin{vmatrix} x - p_1 & y - q_1 & z - r_i \\ p_2 - p_1 & q_2 - q_1 & s_i - r_i \\ p_3 - p_1 & q_3 - q_1 & t_i - r_i \end{vmatrix} = 0, z \right)$
- 4: return $[f_1(x, y), \dots, f_n(x, y)]$

Should a region have more than three boundary terms, we must ensure that the computed interpolation functions are actually compatible with the remaining points. This is done in a subsequent validation step, which takes every remaining boundary term and tries to solve the plane equation for each unification variable instantiation. Should the equation not be solvable for one of the instantiations, the region is not consistently interpolatable and we report failure. If we consider again our triangular region from matrix (3), the algorithm will compute $f_1(x, y) = \frac{xm - x - m + p_1}{p_1 - 1}$, $f_2(x, y) = \frac{ny - y - n + q_1}{q_1 - 1}$ for the unification variables v_1, v_2 , respectively.

While the above algorithm works for 2-dimensional regions, in order to interpolate lines given by single ellipses we need to compute parametric line equations. We omit the details here for lack of space.

When the interpolation functions for a region have been successfully computed, they are added to the region structure.

3.4.3 Adding Subterm Constraints

In a final step we now find possible relations between the boundary terms of a region and the ellipsis length variables of the boundary. In $a_1^1 \dots a_m^1$ we can view the subscript as an indexing function from 1 to m . If m is greater or less than 1 there has to be a relationship between m and the length of the ellipsis, whereas, if $m = 1$ we have a constant term a_1^1 and the ellipsis can still be of arbitrary length as indicated in the introduction. We capture these potential relationships between indexing functions and ellipsis lengths with a disjunction of conditional constraints, which we call *subterm constraints*. The set of all subterm constraints for the ellipses of an abstract matrix is computed as follows:

- 1: let $\mathcal{S} = \emptyset$
- 2: for each ellipsis with length variable e
- 3: let p, q be the start and endpoint of e with anti-unifier Φ and associated substitutions $\sigma_p = \{v_1 \mapsto s_1, \dots, v_n \mapsto s_n\}$, $\sigma_q = \{v_1 \mapsto t_1, \dots, v_n \mapsto t_n\}$
- 4: for $i \leftarrow 1 \dots n$ do
- 5: add $\left\{ \begin{array}{l} s_i = t_i \Rightarrow e = e \\ \vee s_i < t_i \Rightarrow e = t_i - s_i \\ \vee s_i > t_i \Rightarrow e = s_i - t_i + 1 \end{array} \right\}$ to \mathcal{S}
- 6: return \mathcal{S}

Line 5 introduces the subterm constraints. They are given in the form of production rules that produce a new structural constraint if the left hand side of the implication holds. They represent the three mutually exclusive possibilities that we allow for indexing functions: indices either increase by 1 (case $s_i < t_i$), decrease by 1 ($s_i > t_i$), or are constant ($s_i = t_i$), in which case the length of the ellipsis e is independent of the range of the indexing function. Subterm constraints only “fire” if either both s_i and t_i are fixed or if $s_i = t_i$. In both cases the resulting structural constraint will not contain any reference to structural constraint variables.

At this point, the structure is a complete, fully analysed abstract matrix $\langle \mathcal{C}, \mathcal{R}, \mathcal{S} \rangle$, with a set of structural constraints \mathcal{C} , a set of regions \mathcal{R} , and a set of subterm constraints \mathcal{S} .

4. CONCRETISATION

The process of concretisation is an incremental one that, in one step, adds a new equality constraint for either a structural or a subterm constraint variable and returns a modified abstract matrix with that constraint integrated and consequent modifications applied. Typically, each concretisation step will remove at least one unknown. The iterative concretisation algorithm takes an abstract matrix $A = \langle \mathcal{C}, \mathcal{R}, \mathcal{S} \rangle$ plus an assignment for a constraint variable from $\mathcal{C} \cup \mathcal{S}$ as input and returns an abstract matrix $A' = \langle \mathcal{C}', \mathcal{R}', \mathcal{S}' \rangle$ where \mathcal{C}' is the modified version of \mathcal{C} , etc.

- 1: given $\mathcal{C}, \mathcal{S}, \mathcal{R}$
- 2: Concretise(x, r)
- 3: if x is a structural constraint variable
- 4: Add $x = r$ to \mathcal{C}
- 5: $\mathcal{C} \leftarrow solve(\mathcal{C})$
- 6: if $\mathcal{C} = \emptyset$ then fail
- 7: else for each region in \mathcal{R} validate the interpolation function wrt. the new constraint store \mathcal{C}
- 8: if x is a sub-term constraint variable
- 9: replace all occurrences of x in \mathcal{S} and \mathcal{R} by r
- 10: for each sub-term constraint that fires structural
- 11: constraint $x' = r'$ do Concretise(x', r')

Concretisation works in two phases. If a structural variable is assigned, the set \mathcal{C} is simplified and checked for con-

sistency. Moreover, we have to check for all regions whether they are still legally interpolatable under the new structural conditions. The subterm constraints are, however, not affected (lines 3–7). On the other hand, if a subterm variable is assigned, we first have to substitute all its occurrences both in the structural constraints and the region structures. We then check whether any of the subterm constraints is solved and thus yields a new structural constraint. This structural constraint is then integrated in another application of the concretisation algorithm (8–11).

5. INSTANTIATION

Once an abstract matrix has been fully concretised, i.e., all the sub-term constraints have been removed and each structural constraint is of the form $e = k$ where $k \in \mathbb{Z}$, it can be instantiated to a fully concrete matrix. The instantiation algorithm determines for each cell in the concrete matrix which region is responsible for its concrete instantiation. It exploits the fact that we know exactly the size of the matrix and can compute for each single cell and ellipsis the concrete elements at the corresponding position in the concrete matrix (steps 4–6 in the algorithm below). This outlines the boundaries of the regions in the instantiated matrix and enables us to compute the index pairs for the interior of each region. Subsequently we can use the indices to compute the content of each interior cell with the interpolation functions of the region.

```

1: given abstract matrix  $A$  with now concrete size  $n \times m$ 
2: initialise an array  $B[n][m] \leftarrow 0$ 
3: let  $C \leftarrow []$  be an empty list
4: for each single cell at position  $(i, j)$  in  $A$  set  $B[i][j] \leftarrow 1$ 
5: for each ellipsis  $e \in A$ 
6:   compute the points on  $e$ 
7:   for each point at position  $(i, j)$  on  $e$  set  $B[i][j] \leftarrow 1$ 
8: for  $i \leftarrow 1 \dots n$ 
9:   for  $j \leftarrow 1 \dots m$ 
10:    if  $B[i][j] = 1$  do nothing
11:    else if  $C = []$  then  $C = [(i, j)]$ 
12:    else if there is sublist  $l \in C$  such that
         $l$  contains a point  $(i', j')$  adjacent to  $(i, j)$ 
13:      add  $(i, j)$  to  $l$  in  $C$ 
14:    else add  $[(i, j)]$  to  $C$ 
15: for each region  $r \in A$ 
16:   find all  $l \in C$  such that a point in  $l$  is inside  $r$ 

```

Observe that for step 16 in the above algorithm we use the information provided by the boundary directions in the region structures. As a result of the algorithm we get for each region the indices of its boundaries as well as a list of indices that constitute points on the inside of that region. The latter do not all have to be in a single list only, but can be given in several lists, in case the region consists of several interior regions that are only connected by parallel ellipses. In a final step we can now compute the actual terms in each region by applying the interpolation function to the indices and instantiating the anti-unifier of that region appropriately and thus assemble a fully concrete matrix.

We briefly illustrate the instantiation algorithm for the upper triangle region of matrix (3). Assume that we fix m to be 1 and n to be 4. The latter automatically fixes the length of the ellipses in the triangle to be 4 and thereby also the generalised positions $p_1 = q_1 = 4$. This leads to the simplified functions $f_1(x, y) = \frac{1x - x - 1 + 4}{4 - 1} = 1$, $f_2(x, y) = \frac{4y - y - 4 + 4}{4 - 1} = y$, that is, all the indices will be 1 and the exponents will

be increasing downwards. The instantiation algorithm then first computes all the index pairs for the boundaries, which are $[(1, 1), (2, 1), (3, 1), (4, 1), (1, 2), (3, 2), (1, 3), (2, 3), (1, 4)]$. The list of interior points computed afterwards is $[(2, 2)]$ and instantiation of these values as (x, y) pairs into the interpolation functions f_1 and f_2 yields the expected triangle.

6. CONCLUSIONS

While we have not yet done a formal complexity analysis of our algorithms, our current Maple implementation suggests that the majority of the algorithms is fairly fast even on large and complex example. A notable exception is the algorithm generating the structural constraint equations, which will need further optimisation. Further redundancy could be removed by using more elaborate constraint resolution techniques during the concretisation algorithm. The current Maple implementation is also intended as a first step towards a more comprehensive computational treatment of abstract matrices. In particular we intend to develop the algorithms for elementary operations on and with abstract matrices, such as matrix addition, multiplication, etc. Some preliminary work in this direction has been done by Fateman in Macsyma [2], in which indefinite matrices can be subjected to some basic algebraic manipulations. While his matrices are indefinite in size, their elements are fixed to one particular functional expression and cannot be of arbitrary composition. Although Fateman presents some ideas on how to enhance the display of indefinite matrices by using ellipses, the work does not deal with having unspecified elements and ellipses as input in the first place.

Related to our work is also a network based parsing algorithm presented by Kanahori and Suzuki in [5] for the analysis of matrix structures in the context of optical character recognition of mathematical texts. It can analyse structural elements of a matrix and compute a grid representation of the matrix using a system of simultaneous equations. A current goal is to incorporate Kanahori and Suzuki's system as a front-end for our algorithms.

Acknowledgements. We would like to thank the anonymous referees, whose detailed and insightful comments have helped us to make significant improvements to this paper.

7. REFERENCES

- [1] *Maple 10 User Manual*. Maplesoft, 2005.
- [2] R. Fateman. Manipulation of matrices symbolically. Available from <http://http.cs.berkeley.edu/~fateman/papers/symmat2.pdf>, 2003.
- [3] A. Heck. *Maple Manuals*. Springer, 3rd edition, 2003.
- [4] G. Huet. *Résolution d'équations dans les langages d'ordre 1, 2, ..., ω*. PhD thesis, Univ. de Paris VII, 1976.
- [5] T. Kanahori and M. Suzuki. A recognition method of matrices by using variable block pattern elements generating rectangular areas. *GREC-02, LNCS 2390*, p. 320–329. Springer, 2002.
- [6] A. Mercer and A. Rosenfeld. An array grammar programming system. *CACM*, 16(5):299–305, 1973.
- [7] A. Sexton and V. Sorge. Processing textbook-style matrices. *MKM'05, LNCS 3863*. Springer, 2006.
- [8] A. Sexton and V. Sorge. Semantic analysis of matrix structures. In *ICDAR'05*, p. 1141–1145. IEEE Computer Society, 2005.