Extending Gödel for Expressing Restricted Quantifications and Arrays

Jonas Barklund  
Computing Science Department  
Uppsala University  
jonas@csd.uu.se

Patricia M. Hill*  
School of Computer Studies  
University of Leeds  
hill@scs.leeds.ac.uk

Abstract

The expressiveness of the declarative language Gödel can be improved by adding to it bounded quantifications, i.e., quantifications over finite domains, and arrays. Many problems can be expressed more concisely using bounded quantifications than using recursion. Arrays are natural for many applications, e.g., in scientific computing, and are conveniently used in bounded quantifications. Treating bounded quantifications differently from other quantifications also reduces floundering, allows efficient sequential execution and enables efficient parallel execution on various architectures. In extending Gödel to allow bounded quantifications, the finiteness condition would be difficult to implement. Thus the extension defined in this paper allows restricted quantifications where the domain can be restricted, but not necessarily finite.

*Work supported by EPSRC grant GR/H/79862
1 Introduction

A bounded quantification is a quantification that ranges over a finite domain [11]. This form of quantification has been introduced to logic programming [2, 3, 12] to enhance expressiveness and enable many repetitive computations to be implemented iteratively rather than by means of recursion. Arrays are a common data structure in mathematics arising in many programming applications. The indices of an array provide an immediate way of referring to specific elements and, through iteration, allow an easy means of processing all the elements. Thus, arrays with bounded quantifications form a useful combination that can significantly increase the expressiveness [2] and improve the efficiency of logic programming [3].

Gödel [8] is a declarative language in the family of logic programming languages. Declarative programming is concerned with writing what is to be computed rather than how it is done. Without sufficient language constructs, a specification of a problem may require unnecessarily complex expressions, obscuring, not only the declarative meaning, but also possible efficient implementation techniques. In this paper we show how, with a fairly minor extension to the Gödel syntax, we can make a significant improvement to the expressiveness of Gödel by allowing both the bounded quantifications and the arrays. We will also show how, for Gödel, bounded quantifications can reduce floundering\(^1\). Gödel is particularly suitable for parallel implementations, since the semantics is declarative and does not depend on the order of computation. It has been shown that parallel implementations of bounded quantifications can obtain good speed-ups over sequential processing [1, 4]. Thus any extension to Gödel that enhances the possible exploitation of parallelism in a problem should be explored.

Bounded quantifications, as defined by Barklund [2], require the domain of the quantifier to be finite. In extending Gödel to allow such quantifications, the finiteness condition would be difficult to implement. Thus the extension defined in this paper allows restricted quantifications where the domain can be restricted, but where there is no necessity that it is finite. (Such quantifications are similar to relativised quantifications [10].) Of course, for reasons of termination, the programmer should ensure that the domain is finite and hence, bounded, whenever the quantification is actually executed.

The paper is organised as follows. In the next section we show how the existing Gödel syntax that allows universal and existential quantifications easily extends to a notation for restricted quantifications. We explain how the system modules already provide predicates that would be suitable for defining bounded quantifications. In Section 3, arithmetical quantifications

\(^1\)Floundering occurs when the goal is insufficiently instantiated for the execution to proceed. Floundering is often caused by the safe computation rule of SLDNF-resolution which does not allow non-ground negative literals to be evaluated.
are introduced and again the necessary extensions to Gödel are defined. Section 4 specifies an additional Gödel system module that would provide an array type together with the standard syntax for arrays and predicates for manipulating them. Section 5 discusses possible implementation techniques. The paper concludes by summarising the advantages of this extension and outlining future work.

2 Universal and Existential Quantification

Before discussing any extensions to Gödel, we describe briefly some relevant points concerning the type system and allowed syntax of the program statements.

Gödel is a strongly and statically typed logic programming language; the type system based on many-sorted logic with parametric polymorphism. By strongly typed, we mean that the type of every constant, function, and predicate must be declared together with the required types of any arguments. By statically typed, we mean that all type checking of a program can be done at compile time and the type checking of the goal can be done before execution. Note that, in Gödel, the types of variables are not declared but uniquely inferred from their positions in the program.

Each predicate and proposition is defined by a set of program statements. A statement is a generalisation of a normal clause that allows for arbitrary first order formulas in the body; in particular, universally and existentially quantified expressions are allowed. A goal for a Gödel program is a similar generalisation of a normal goal. Any variable not in the scope of an existential or universal quantifier is assumed to be universally quantified at the left of the statement or goal. We first illustrate the syntax of program statements and goals using universal and existential quantifiers in Gödel and highlight some inadequacies with a few examples.

The first example is a goal that computes the Pythagorean triple and illustrates the explicit use of the existential quantifier.

\[ \text{SOME } [x,z] \ (0 < x < 20 & 0 < z < 20 & x^2 + y^2 = z^2) \].

Note that a computed answer for this goal will only bind the free variable y. To execute this goal, the Integers system module is required. This module provides the integers together with the common functions and relations that operate on the integers. For instance, the binary infix predicate \(<\) for comparing two numbers is provided by Integers.

The next example consists of a program statement that defines the predicate LessAll. This checks that the number in the first argument is less than every element of the list in the second argument and illustrates the explicit use of the universal quantifier.

\[ \text{LessAll}(a,x) \leftarrow \text{ALL}[b] \ (\text{Member}(b,x) \rightarrow a < b) \].
This statement requires the Lists system module which provides the usual
list notation and a number of list processing predicates. For instance, Member
is a predicate in Lists that is true if and only if the first argument is an
element of the list in the second argument. Gödel requires that, in a call to
Member, the second argument is non-variable. Moreover, both arguments in
any call to < should be ground. Thus, in a call to LessAll, if the program
is not to flounder, both its arguments must be ground.

In the definition of the language [8], the procedural semantics for Gödel is
not fully specified. The only requirement for any implementation of Gödel
is that it respects the intended semantics of the logic defined by the comple-
tion of the program and the intended meaning of any control annotations.
However, all the language features of Gödel have a procedural interpretation
which can be used as the basis of an implementation. For example, if the
statements and goal are in normal form, then SLDNF-resolution can be used
to evaluate the goal. A well-known technique for implementing statements
not in normal form, possibly containing quantifiers, is to first transform
them to normal clauses using transformations based on the Lloyd-Topor
transformations [9]. For example, using these transformations, the above
definition for LessAll would be transformed to:

\[
\text{LessAll}(\text{a}, \text{x}) \leftarrow \neg \text{MoreOne}(\text{a}, \text{x}).
\]

\[
\text{MoreOne}(\text{a}, \text{x}) \leftarrow \text{Member}(\text{b}, \text{x}) \land \neg \text{a} < \text{b}.
\]

where MoreOne is a new predicate symbol. (In this example, implemen-
tations could eliminate the negation in the second clause by replacing \( \neg \text{a} < \text{b} \) by \( \text{a} \geq \text{b} \), but, for other relations, this may not be possible.) Although
the resulting code is in normal form, it appears complicated compared to
the original simple program statement.

Consider next the following definition for reversing a list.

\[
\text{ReverseList}(\text{x}1, \text{x}2) \leftarrow
\text{Length}(\text{x}1, \text{l}) \land \text{Length}(\text{x}2, \text{l}) \land
\text{ALL}[\text{i}]
\]

\[
(1 =< \text{i} =< 1 \rightarrow
\text{SOME} [\text{a}] (\text{Suffix}(\text{x}1, \text{i}, [\text{a} | \_]) \land \text{Suffix}(\text{x}2, 1 - \text{i} + 1, [\text{a} | \_])))\).
\]

Length is a predicate in the Lists module that is true if and only if the sec-
ond argument is the length of the list in the first argument. For Length to be
selected, one of the arguments must be non-variable. Gödel’s flexible com-
putation rule allows the first two atoms to be executed in any order subject
to the non-variable conditions on the arguments to Length. Thus provided
\text{x}1 or \text{x}2 is given, the first two atoms can be evaluated. Suffix is another
predicate in Lists and is true if the third argument is the suffix of the list
in the first argument with length given by the second argument. Using the
Lloyd-Topor transformations, the definition of ReverseList becomes:
ReverseList(x1, x2) <-
  Length(x1, l) & Length(x2, l) & ~ NotReversed(x1, x2, l).

NotReversed(x1, x2, l) <-
  1 <= i <= 1 &
  ~ Match(x1, x2, l).

Match(x1, x2, l) <-
  Suffix(x1, i, [a\_]) & Suffix(x2, l-i+1, [a\_]).

where NotReversed and Match are new predicate symbols. Not only is this complex but it will flounder if one of the lists x1 and x2 is not ground. Thus it can only be used to check that one list is the reverse of the other and not to generate a reversed list. The problem stems from the fact that the i in ALL [i] ranges over all the integers, whereas only those values of i between 1 and the length of one of the lists need to be considered when evaluating the expression

Suffix(x1, i, [a\_]) & Suffix(x2, l-i+1, [a\_]).

We show here how the universal and existential quantification notation in Gödel can be easily extended to allow the ranges of the quantified variables to be restricted. To illustrate and motivate this extension, we modify the above examples to use this extension. First, we repeat the Pythagorean example to illustrate the extended form of the SOME quantification.

<- SOME [x, z : 0 < x < 20 & 0 < z < 20] x^2 + y^2 = z^2.

This example shows how the interval expressions provided by the Integers module in Gödel naturally define finite bounds for the quantified variables. The next example illustrates the extension for the ALL quantification.

LessAll(a, x) <- ALL [b : Member(b, x)] a < b.

This can be interpreted as: all b which are elements of the list x, must be greater than a. Note that it is clear from this formulation that we are only interested in the set of elements in x; comparing only distinct elements of x with a. If a < b was replaced by a more expensive test and x contained repeated elements, such an optimisation could significantly improve the efficiency. Finally we give a definition of ReverseList using restricted quantifications.

ReverseList(x1, x2) <-
  Length(x1, l) & Length(x2, l) &
  ALL [i : 1 <= i <= 1]
  SOME [a] (Suffix(x1, i, [a\_]) & Suffix(x2, l-i+1, [a\_])).
This can be interpreted as: lists \( x_1 \) and \( x_2 \) have the same length \( l \) and for each \( i \) in the set \( \{1, \ldots, 1\} \), the element in the \((1+i)\)th position in \( x_2 \) is the same as the element in the \( i \)th position in \( x_1 \).

The *restricted quantifications* in this extension of Gödel have the general form:

\[
\begin{align*}
\text{SOME} & \ [x_1, \ldots, x_k : F] \ G \\
\text{ALL} & \ [x_1, \ldots, x_k : F] \ G
\end{align*}
\]

The \( x_1, \ldots, x_k \) are the *quantified variables* and must occur in the formula \( G \). The formula \( F \) is called the *range formula* and \( G \) the *body* of the quantification. The intended semantics of each of these expressions is respectively:

\[
\begin{align*}
\exists x_1, \ldots, & \exists x_k (F \land G) \\
\forall x_1, \ldots, & \forall x_k (F \to G)
\end{align*}
\]

If, for each substitution \( \theta \) such that \( x_1, \ldots, x_k \) are the only free variables in \( F\theta \), the sets \( \{x_i : F\theta\} \) (\( 1 \leq i \leq k \)) are finite, then the quantification is said to be *bounded*. Although, it is desirable that there are only a finite number of values for \( x_1, \ldots, x_k \) such that the current instance of \( F\theta \) is true, it is not practical for this to be a condition.

Any implementation of restricted quantifications should always respect the semantics given above. Using the range formula, the values of the quantified variables can be computed eagerly or lazily. With eager computation, the body can be computed in parallel, with distinct values for the quantified variables. However, with lazy computation, particularly with the existential quantifier, unnecessary computation of the range formula can be avoided.

Barklund [2] states that a bounded quantification must have a range formula that is “obviously” true for only a finite number of values of the variables. The word “obvious” is clarified by means of examples. For these, the range formulas contain only certain predicates for which the compiler (or run-time pre-processor) has a specialised method for determining the actual number of values. If this number is known in advance of the execution of the evaluation of a bounded quantification, a parallel compiler can evenly distribute the evaluation of the body of the quantification over the available processors [1, 4]. In the extension here for restricted quantifications, arbitrary range formulas are allowed. It is not possible, in general, for a compiler or run-time preprocessor to determine the actual number of values of the quantified variables, or even, whether or not that number is finite although this may be clear to the programmer. Provided the set of values is finite, and these can be enumerated in an acceptable period of time, the quantifications can be executed. Thus, although the bounded quantifications are required to be as defined [2] for maximum efficiency, the more general re-
stricted quantifications that allow arbitrary range formulas further enhance the expressiveness of the language, while compile-time optimisations may still be applicable.

Although not semantically essential, it is expected that \( x_1, \ldots, x_k \) would normally be the only free variables in the current instance of \( F \) before the quantification was selected for processing. For instance, in the ReverseList example, before evaluating the quantification, at least one of the Length atoms should be executed to determine the value of \( l \). A Gödel programmer can use the control declarations provided as part of the language to ensure that, even with user-defined predicates, this is the case.

Gödel has a number of system modules that provide many predicates for performing common tasks. It is straightforward to identify some of these as predicates suitable for range formulas and certain argument positions for the quantified variables. Many of these could guarantee the domain of the quantifier to be finite and hence ensure that the quantification was bounded. Bounded quantifications using these predicates could have specialised compilation improving the efficiency of execution [3], particularly in the case of parallel implementations [1, 4].

The Integers system module provides the Interval predicate. This has three arguments \( m, i, \) and \( n \) and is more commonly denoted by the range expression \( m \leq i \leq n \) (or variations involving \( < \) instead of \( \leq \)). The use of this notation (as illustrated in the examples) as a range formula with \( i \) as the quantified variable is natural and, as the values of \( m \) and \( n \) must be given, any restricted quantification with this as a conjunct of its range formula will be bounded.

Many recursive programs operate on lists and the system module Lists in Gödel provides the usual syntax for lists as well as many common list processing predicates. A number of these are useful in range formulas. The predicates Member and Suffix have already been used in the examples as range formulas. For Member, a variable in the first argument position would normally be the quantified variable while the second argument would be ground. The Suffix predicate used in the range formula for ReverseList together with a similar predicate Prefix can be used in bounded quantifications provided the first argument is ground. (In practice, Suffix is much easier to implement than Prefix and should be preferred wherever possible.) In fact, ReverseList does not use the complete suffix of a list, but only uses Suffix to index an element of the list. As this is a common task, we propose adding the predicate IndexMember to the module Lists. This has three arguments, the first two are as for Member and the third is for the position of the element in the list.

The module Sets provides facilities for defining and manipulating finite sets. With this module, an expression \( \{ \} \) stands for the empty set and an
expression \( \{t_1, \ldots, t_n\} \) stands for the set \( \{t_1, \ldots, t_n\} \). A set provides a natural range for a quantified variable and thus the binary infix predicate \( \text{In} \) that defines set membership is an obvious candidate for range formulae. For example, with the Sets and Lists modules and an appropriate definition of \( \text{Worked} \), the query

\[
\text{Worked} \left(\text{listdays}\right) \ & \ \\
\text{SOME} \ [\text{day} : \text{day} \ \text{In} \ \{\text{Saturday, Sunday}\}] \\
\quad \text{Member}(\text{day}, \text{listdays}).
\]

will find if work was done at the weekend.

From a declarative point of view, the extended syntax does not introduce any extra expressiveness. In fact, as an alternative to allowing such an extension to the syntax, the compiler could be sufficiently clever in detecting if there is an intended range formula and implement bounded quantification techniques in such cases. However, we believe, for most programmers, this syntax will be natural and they will welcome the chance to indicate intended optimisations, knowing that in this case the program is less likely to flounder.

A similar extension to the syntax of first order logic is already allowed in the case of the \( \text{IF} \ C \ \text{THEN} \ T \ \text{ELSE} \ E \) construct in Gödel [8, page 44]. (and illustrated in the Gödel example in the next section). In this case, by expressing the logic in this form, the programmer can indicate that the test \( C \) need only be evaluated once. If \( C \) is true, then \( T \) is evaluated and if \( C \) is false, then \( E \) is evaluated.

3 Arithmetic Quantifications

Tennent [11] has pointed out that, in a computational context, quantification can be generalised to allow for quantifiers other than the universal and existential quantifiers of predicate calculus. In this section we are interested in another form of this generalisation called \textit{arithmetical quantifications} which have been introduced into logic programming [2]. To clarify the underlying idea, consider the mathematical expression:

\[
\sum_{i=1}^{k} f(i)
\]

This can be viewed as a \( \sum \) quantification with the quantified variable \( i \), range formula \( 1 \leq i \leq k \), and body expression \( f(i) \). The major difference between this arithmetical quantification and the restricted universal and existential quantification described in the previous section is that the value of the arithmetical quantification will be a number whereas the universal and existential quantification have Boolean values.

To illustrate the ideas in this section, we give three simple example definitions of predicates in Gödel for computing numerical values. In each case
it is assumed that the Integers module is available. The first example computes the factorial \( \Pi_{i=1}^{n} i \) of a number \( n \geq 0 \).

\[
\text{Factorial}(0,1).
\]

\[
\text{Factorial}(n,n\times) \leftarrow
n \geq 0 \& \text{Factorial}(n-1,x).
\]

The next example computes the Euclidean inner product \( \sum_{i=1}^{k} x_{i}y_{i} \) of two vectors \( x_{1}, \ldots, x_{k} \) and \( y_{1}, \ldots, y_{k} \) of the same length \( k \). The vectors are represented as two lists of the same length.

\[
\text{InnerProduct}([],[],0).
\]

\[
\text{InnerProduct}([a|as],[b|bs],a\times b + s) \leftarrow
\text{InnerProduct}(as,bs,s).
\]

Finally, we give a definition of the greatest common divisor of a pair of numbers.

\[
\text{Gcd}(i,j,d) \leftarrow
\text{CommonDivisor}(i,j,d) \&
\sim \text{SOME} [e] \left( \text{CommonDivisor}(i,j,e) \& e > d \right).
\]

\[
\text{CommonDivisor}(i,j,d) \leftarrow
\text{IF} (i = 0 | j = 0)
\text{THEN}
\quad d = \text{Max}(\text{Abs}(i),\text{Abs}(j))
\text{ELSE}
\quad 1 =< d =< \text{Min}(\text{Abs}(i),\text{Abs}(j)) \&
\quad i \mod d = 0 \&
\quad j \mod d = 0.
\]

The definition, although computable, is not designed for efficiency and is an example of the use of Gödel as a language for writing executable specifications. In spite of this, we claim that the statement defining \( \text{Gcd} \) is not the most natural way to specify a greater common divisor. The easiest way to do this is to say that it is the maximum of all common divisors (together with a suitable definition of \( \text{CommonDivisor} \)). Furthermore, if the above specification is implemented directly, then the program needs to evaluate each common divisor several times.

We consider extending the syntax of the Integers module in Gödel with the four quantifiers: \( \text{SUM}, \text{PRODUCT}, \text{MAX}, \) and \( \text{MIN} \). To illustrate and motivate this extension, we repeat the above examples using these quantifiers. The factorial example illustrates the \( \text{PRODUCT} \) quantifier. Here the body \( i \) is trivial.
Factorial(n, x) ← x = PRODUCT [i: 1 <= i <= n] i.
Alternatively, as the quantification denotes a value, it can be written:
Factorial(n, PRODUCT [i: 1 <= i <= n] i).
In the inner product example, there are two quantified variables a and b.
InnerProduct(x, y, s) ←
Length(x, l) & Length(y, l) &
s = SUM [a, b : Suffix(x, i, [a|_]) & Suffix(y, i, [b|_])] a * b.
We illustrate the use of the MAX quantifier with the greatest common divisor example.
Gcd(x, y, MAX [e : CommonDivisor(i, j, e)] e).
Here we assume the same definition of CommonDivisor as that given above. Not only is this definition closer to the intended declarative reading, only one computation of each common divisor is indicated.
Finally, to illustrate the use of the MIN quantifier, we present a revised definition of LessAll.
LessAll(a, x) ← a < MIN [b : Member(b, x)] b.
In general, we propose extending the Gödel syntax with arithmetic quantifications which will have the form:

\[ \mathcal{Q}[x_1, \ldots, x_k : F] E \]

where \( \mathcal{Q} \) is one of the quantifiers SUM, PRODUCT, MAX, or MIN and \( x_1, \ldots, x_k \) are the quantified variables occurring freely in the formula \( F \) and term \( E \). \( F \) is called the range formula and \( E \) the body of the quantification. The intended values of each of these expressions is as follows.

\[
\begin{align*}
\text{SUM} & \quad [x_1, \ldots, x_k : F] E = \sum_F E \\
\text{PRODUCT} & \quad [x_1, \ldots, x_k : F] E = \prod_F E \\
\text{MAX} & \quad [x_1, \ldots, x_k : F] E = \text{the maximum element of the set} \{ E : F \} \\
\text{MIN} & \quad [x_1, \ldots, x_k : F] E = \text{the minimum element of the set} \{ E : F \}
\end{align*}
\]

It would be expected that an implementation would require \( x_1, \ldots, x_k \) to be the only free variables in the current instance of \( F \) before computing the value of the expression.

For arithmetic, there are three system modules \texttt{Integers}, \texttt{Rationals}, and \texttt{Floats} providing, respectively, the integers, rationals, and floating point numbers, respectively. Each of these modules provides a comprehensive set of functions and predicates that operate on their respective types. We have considered here just the \texttt{Integers} module, but most of the preceding discussion applies equally to the corresponding functions and predicates in the \texttt{Rationals} and \texttt{Floats} modules.
4 Arrays

An array, as an explicit data structure, was first introduced to logic programming by Eriksson & Rayner [6]. It has been shown [2, 3, 6] that a facility for defining and manipulating arrays in a language would be useful and improve its expressiveness. Efficient (constant-time) mechanisms for updating an array have been investigated for logic programming [6]. In Prolog, a limited tool for expressing arrays is provided by the built-in predicates arg and functor. These (or equivalent predicates) are not supported by Gödel. In Gödel, the type and module system facilitate abstract data types, providing a natural mechanism for defining an array abstract data type. However, the type system has the two conditions, head and transparency that prevent the definition of a generic array with an arbitrary number of dimensions. To define a generic array, the Gödel type system would have to be extended and it has been shown [7] that such extensions must be made with some care if the intended completion semantics of Gödel is to be preserved. Thus, in this paper we describe a more modest extension that allows for arrays of dimension 1, 2, and 3 only. These dimensions have been chosen since, in practice, most arrays in computing have dimension no more than 3 and they serve to illustrate the technique. It will be seen that it is straightforward to define arrays of any fixed number of dimensions.

We propose adding a system module Arrays which we present below. A module is divided into an export part which declares symbols that can be used by any importing module and a local part giving their implementation. We present here both parts.

EXPORT Arrays.

IMPORT Lists.

CONSTRUCTOR Array/1.

PREDICATE Dimension : Integer * List(Integer) * Array(a).
PREDICATE ListArray : List(a) * List(Integer) * Array(a).
PREDICATE Element : Array(a) * List(Integer) * a.
PREDICATE Vector : Array(a) * List(Integer) * Array(a).
PREDICATE Matrix : Array(a) * List(Integer) * Array(a).

This export part defines types Array(t) for any type t for the elements of an array. The constants and functions for constructing the type are in the local part of the module. This ensures that the only way to construct or examine a term of such a type is by means of predicates exported by the Arrays module. These predicates are Dimension, ListArray, Element, Vector, and Matrix. Dimension gives the number of dimensions of the
array and the size in each dimension. It can also be used in reverse to create an array with variable elements but given sizes for each of the dimensions.\(^2\)

`ListArray` can be used to construct an array from a single list of its elements and the sizes in each dimension or, in reverse, to obtain a list of elements and the sizes in each dimension from a given array. The predicate `Element` can either be used to obtain the value of an element of an array at a given position or to bind a variable in that position to the given value. Similarly, `Vector` or `Matrix` obtain, respectively the one- or two-dimensional array in the given position.

In a similar way to the syntax provided by the `Lists` and `Sets` modules, it is intended that the `Arrays` provides the standard syntax for an element of an array. Thus, an expression of the form `x[i]` defines the `i+1`’th element of `x` if `x` is a vector, the `i+1`’th vector of `x` if `x` is a matrix, and the `i+1`’th matrix of `x` if `x` is a three-dimensional array. In addition, for arrays with more than one dimension, the notation `x[i,j]` is short for `x[i][j]` and (for three-dimensional arrays) `x[i,j,k]` is short for `x[i][j][k].`

We illustrate the use of these predicates and the array notation with two examples. The first computes the inner product of two vectors.

\[
\text{InnerProduct}(x,y, \sum [i : 0 =< i < 1] x[i] \ast y[i]) \leftarrow \text{Dimension}(1,[1],x) \& \text{Dimension}(1,[1],y).
\]

The second transposes a matrix.

\[
\text{Transpose}(x,y) \leftarrow \text{Dimension}(2,[1,m],x) \& \text{Dimension}(2,[m,1],y) \& \text{AllDim}(1,[m],x).
\]

The following version of the local part of `Arrays` illustrates just one possible implementation and shows the feasibility of this module. It is not meant to restrict any actual implementation of the `Arrays` module.

**LOCAL Arrays.**

**FUNCTION** `Array1` : `List(a) \ast List(Integer) \rightarrow Array(a)`;

`Array2` : `List(Array(a)) \ast List(Integer) \rightarrow Array(a)`;

`Array3` : `List(Array(a)) \ast List(Integer) \rightarrow Array(a)`.

\[
\text{Dimension}(1,[1],\text{Array1}(x,[1])) \leftarrow \text{Length}(x,1).
\]

\[
\text{Dimension}(2,[1,m],\text{Array2}(x,[1,m])) \leftarrow \text{Length}(x,1) \& \text{AllDim}(1,[m],x).
\]

\(^2\)Our arrays are 0-based; the components are numbered from 0.
Dimension(3,[1,m,n],Array3(x,[1,m,n])) <-
    Length(x,1) &
    AllDim(2,[m,n],x).

PREDICATE AllDim : Integer * List(Integer) * List(Array(a)).
% AllDim(d,l,x) -- Every element of list x is an array
% of d dimensions whose sizes are given by given by l.

AllDim(_,_,[]).
AllDim(d,l,[x|y]) <-
    Dimension(d,l,x) &
    AllDim(d,l,y).

Element(Array1(x,_,[i],y) <- IndexMember(y,x,i+1).
Element(Array2(x,_,[i,j],y) <-
    IndexMember(z,x,i+1) &
    Element(z,[j],y).
Element(Array3(x,_,[i,j,k],y) <-
    IndexMember(z,x,i+1) &
    Element(z,[j,k],y).

Vector(Array2(x,_,[i],y) <- IndexMember(y,x,i+1).
Vector(Array3(x,_,[i,j],y) <-
    IndexMember(z,x,i+1) &
    Vector(z,[j],y).

Matrix(Array3(x,_,[i],y) <- IndexMember(y,x,i+1).

PREDICATE IndexMember : a * List(a) * Integer.
% IndexMember(x,y,i) -- y is the i'th element
% of list x.

IndexMember(x,[x|_],1).
IndexMember(w,[_|z],i) <-
    i > 1 &
    IndexMember(w,z,i-1).

ListArray(x,[l],Array1(x,[l])) <-
    Length(x,1).
ListArray(x,[l|m],Array2(z,[l,m])) <-
    ListsArrays(x,[m],z) &
    Length(z,1).
To conclude this section, we present a program that uses the ALL quantification, the SUM quantification, with the Arrays module. This program contains the predicate IntSimp which defines an approximation to the integral

\[ \int_a^b f(x)dx \]

using Simpson's method. In the program, \( n \) is the number of intervals and \( i \) the resulting approximation of the integral. We assume that there is a module FunctR containing the definition of the relation \( R(x,y) \), which is true if and only if \( f(x) = y \).

MODULE SimpsonsMethod.

IMPORT Arrays, FunctR, Rationals.

IntSimp(a,b,n,i) <-
    w = (b-a)/n &
    Dimension(1, [2*n+1], g) &
    ALL[j : 0 <= j < 2*n+1] R(a+j*w/2, g[j]) &
    i = (w/6) * SUM[j : 0 <= j < n]
        (g[2*j] + 4*g[2*j+1] + g[2*j+2]).

5 Implementation

In order to achieve efficient execution of bounded quantifications on a sequential computer, the language processor must have a construct for definite iteration. For example, Warren's abstract Prolog Machine [13] does not contain any such construct but can easily be extended with instructions supporting it [3], as has been done in the Luther abstract machine [5]. The current implementation of Gödel compiles Gödel programs to Prolog programs, so by translating Gödel programs with bounded quantifications to Prolog programs with bounded quantifications, they will be run efficiently. When translating Gödel programs for a system without support for definite iteration, they can be instead be translated to tail-recursive programs [4] and run with reasonable efficiency.

It has been shown how Prolog programs with bounded quantifications can be run on both SIMD parallel computers [1] and MIMD parallel computers with shared memory [4]. By the same token it is thus possible to run Gödel programs with bounded quantifications on these parallel computers by translation to Prolog programs with bounded quantifications.

6 Conclusion

It has been shown how, with a small but fairly straightforward extension to the syntax of Gödel and the addition of a system module, both bounded quantifications and arrays can be expressed. Furthermore, this extension also allows restricted quantifications which are a generalisation of bounded quantifications that allows arbitrary range formulas. This is completely declarative and gives three advantages:

- **Expressive** – Many Gödel programs can be expressed more naturally using an extension providing restricted quantifications and arrays.

- **Executable** – Many Gödel programs will flounder and cannot run using (unrestricted) quantifications whereas, if restricted quantifications are employed, they will execute.

- **Efficiency** – With range formulas using only certain predicates defined in system modules, restricted quantifications will be known to be bounded. In this case, special implementation techniques can ensure significant efficiency improvements.
An implementation of these extensions for Gödel is being developed.

References


