Pure, Declarative, and Constructive Arithmetic Relations (Declarative Pearl)*

Oleg Kiselyov[1], William E. Byrd[2], Daniel P. Friedman[2], and Chung-chieh Shan[3]

[1] FNMOC oleg@pobox.com
[2] Indiana University {webyrd,dfried}@cs.indiana.edu
[3] Rutgers University ccshan@cs.rutgers.edu

Abstract. We present *decidable* logic programs for addition, multiplication, division with remainder, exponentiation, and logarithm with remainder over the *unbounded* domain of natural numbers. Our predicates represent *relations* without mode restrictions or annotations. They are fully decidable under the common, DFS-like, SLD resolution strategy of Prolog or under an interleaving refinement of DFS. We prove that the evaluation of our arithmetic goals always terminates, given arguments that share no logic variables. Further, the (possibly infinite) set of solutions for a goal denotes exactly the corresponding mathematical relation. (For SLD without interleaving, and for some infinite solution sets, only half of the relation's domain may be covered.) We define predicates to handle unary (for illustration) and binary representations of natural numbers, and prove termination and completeness of these predicates. Our predicates are written in pure Prolog, without cut (`!`), `var/1`, or other non-logical operators. The purity and minimalism of our approach allows us to declare arithmetic in other logic systems, such as Haskell type classes.

1 Introduction

Logic programming is said to be programming with relations, but arithmetic is often dealt with in a non-relational, restricted way. For example, Prolog's built-in `is/2` predicate for evaluating arithmetic expressions does not allow free (unbound) logic variables in the expressions. Whereas the goal `Z is 6*7` succeeds, binding `42` to `Z`, the related goal `42 is X*Y` is considered erroneous because its multiplicands 'are not sufficiently instantiated.' Multiplication is not treated as a ternary relation between multiplicands and the product because of mode restrictions on the first two arguments. Constraint logic programming (CLP) overcomes this drawback to some extent [1]; for example, disjunctive Datalog [2] treats arithmetic relationally. Indeed, relational handling of arithmetic was one of the motivations for CLP. Unfortunately, this flexibility has a price: CLP restricts

* We thank Ronald Garcia and the anonymous reviewers for many helpful comments.

the arithmetic domain to be finite and changes the evaluation mode of the logic programming system away from Kowalski's 'predicate as function' model [3].

We present fully relational arithmetic on the *unbounded* domain of binary natural numbers for conventional (SLD [4], or SLD with interleaving [5]) logic programming systems. We define predicates for addition, multiplication, division with remainder, and logarithm with remainder. These predicates express the remaining arithmetic operations, including subtraction and exponentiation. These so-called base predicates have *no* mode restrictions or annotations on their arguments, and are implemented in a *pure* logic system without cut (`!`), the `var/1` predicate, or negation. Furthermore, each base predicate terminates under SLD evaluation (and re-evaluation, upon backtracking), provided that the predicate's arguments share no logic variables. The stream of answers produced by (re-)evaluating each arithmetic predicate under SLD with interleaving [5] covers exactly the corresponding mathematical relation; under SLD resolution and some infinite domains, only half of the relation is covered.

In particular, we define the following decidable predicates:

`add/3` such that $\texttt{add}(X, Y, Z)$ can be used to add two numbers X and Y to get Z, to subtract X or Y from Z, to decompose Z into summands, or to compare two naturals. For example, we can determine that the triple $(1, 2, 3)$ is in the ternary addition relation[4] by evaluating either the goal $\texttt{add}(\mathbf{1}, \mathbf{2}, \mathbf{3})$[5] or the goal $\texttt{add}(X, Y, Z), X = \mathbf{1}, Y = \mathbf{2}, Z = \mathbf{3}$. By evaluating $\texttt{add}(X, \mathbf{3}, \mathbf{2})$ we determine, also in finite time, that the addition relation does *not* include *any* triple $(n, 3, 2)$.

`mul/3` such that $\texttt{mul}(X, Y, Z)$ can be used, inter alia, to multiply X and Y, to factor Z, or to generate a stream of triples related by multiplication.

`div/4` such that $\texttt{div}(N, M, Q, R)$ succeeds if and only if $N = M \cdot Q + R < M \cdot (Q + 1)$.[6] For instance, the goal $\texttt{div}(\mathbf{1}, \mathbf{0}, \_, \_)$ to relate the divisor zero to a non-zero dividend fails instantly, without trying to enumerate all natural numbers. The goal $\texttt{div}(\mathbf{5}, M, \mathbf{1}, \_)$ finds all numbers M that divide 5, perhaps unevenly, with a quotient of 1. The answers are 5, 4, and 3. Finally, $\texttt{div}(\mathbf{5}, M, \mathbf{7}, \_)$ fails in finite time rather than diverging.

`log/4` such that $\texttt{log}(N, B, Q, R)$ succeeds if and only if $N = B^Q + R < B^{Q+1}$. We can use `log/4` to perform exponentiation, find logarithms, and find n-th roots.

We prove that these base predicates are decidable and that the arithmetic relations over natural numbers form their universal model (see the *faithfulness* property introduced in §2).

---

[4] We mean a relation that relates triples of numbers (x, y, z) so that $x + y = z$.

[5] **3** means the representation of the binary numeral 3, which in Prolog is encoded as `[1,1]` (§3).

[6] This equation implies that `div/4` fails when M is **0**. Hence `mul/3` is not reducible to `div/4`. Besides, the former is simpler and is part of the implementation of the latter.

1.1 Challenges

We require the predicates to be both effective and efficient. First, the evaluation of base arithmetic goals must terminate. Put differently, it must be effectively computable whether a tuple of naturals is included in or *excluded* from a base arithmetic relation (addition, multiplication, division with remainder, and logarithm with remainder). Further, these computations must finish without taking an exponential amount of time or space with respect to the 'search depth' (corresponding, in the case of binary numbers, to the logarithm of the largest number appearing in the computation). In other words, we wish to maintain the efficiency of depth-first search (DFS) (or DFS with interleaving [5, 6]) of the and-or tree expressing the solution space of our base goals.

The main challenge in defining our predicates is that the domains of natural numbers, and of arithmetic relations in general, are infinite—enumerating them is not an option. The incompleteness of DFS (used in SLD resolution) immediately presents a problem. For example, assuming that `gen/1` is a predicate that generates all (ground) natural numbers in sequence, one may be tempted to implement `mul/3` (separating the generation and testing for clarity) as

```
mul(X,Y,Z) :- gen(A), X=A, gen(B), Y=B, gen(C), Z=C, C is A*B.
```

In addition to its obvious inefficiency, this implementation[7] often diverges under DFS. For example, evaluating $\texttt{mul}(X, Y, \mathbf{1})$ instantiates `A` and `B` to `0` and `C` to `1`, causing the goal `C is A*B` to fail; after backtracking into `gen(B)`, `mul/3` keeps forever instantiating `B` to larger and larger positive numbers, each time resulting in failure of the goal `1 is 0*B`.

One may attempt to fix this problem by using a complete search strategy, such as breadth-first search (BFS) or iteratively-deepening DFS. Even ignoring efficiency concerns, this implementation of `mul/3` is still unacceptable: a complete search strategy will find a solution *if it exists*, but if no solution exists the search will continue forever. For example, although BFS finds the instantiation of Y that satisfies $\texttt{mul}(\mathbf{1}, Y, \mathbf{2})$, the goal $\texttt{mul}(\mathbf{2}, Y, \mathbf{1})$ still diverges. Thus we must devise a termination criterion for `mul/3`.

Devising a termination criterion may seem easy. For example, when searching for X and Y that satisfy the goal $\texttt{mul}(X, Y, \mathbf{5})$, we only need to examine X and Y values up to 5—since the search space is finite, the evaluation of the goal certainly terminates. The problem arises when determining whether the third argument in a specific use of `mul/3` is instantiated to a ground numeral. This task is trivial if we use the 'impure' non-logical features provided by Prolog's reflection facilities, such as the infamous `var/1` predicate. Even if `var/1` were absent from Prolog, it could be emulated using cuts and negation. We disavow such tools—we aim to implement our predicates in a pure subset of Prolog, without cuts, reflection, or any way to distinguish a logic variable. This aim for purity is a challenge that in return makes our approach most elucidating and extensible.

[7] Braßel, Fischer, and Huch [7] describe the drawbacks of this residuation-based approach in functional logic programming.

The final challenge is that the binary representation of a number may not be structurally part of that of its successor. For example, the binary numeral 111 (in decimal, 7) is not structurally part of its successor 1000. This lack of structural inclusion prevents straightforward structural recursion.

1.2 Termination and Solvability

These challenges make the arithmetic predicates tricky to code. It is not obvious that the resulting predicates have the claimed properties, in particular, that they terminate in all modes. We therefore devote much of the paper to proofs. A typical termination theorem we prove assures that evaluating or re-evaluating $\texttt{add}(X, Y, Z)$ terminates, provided that the terms initially associated with X, Y, and Z share no logic variables. Successive re-evaluations of this goal recursively enumerate the stream of unique triples (X, Y, Z) of potentially non-ground terms whose denotation (§2.2) is the domain $\{\, (u, v, w) \in X \times Y \times Z : u+v = w \,\}$. Thus, membership and non-membership are computable for base arithmetic relations.

We guarantee termination only for stand-alone base arithmetic goals but not their conjunctions (see §2.1). This non-compositionality is expected, since conjunctions of arithmetic goals can express Diophantine equations; were such conjunctions guaranteed to terminate, we would be able to solve Hilbert's 10th problem, which is undecidable [8]. We also do not guarantee termination if the goal's arguments share variables. Such a goal can be expressed by conjoining a sharing-free base goal and equalities.

We proceed as follows. In §2 we define addition and multiplication predicates for a *unary* representation of natural numbers. We introduce solution sets, §2.1, to carefully establish termination for these predicates, laying a foundation for our analysis of binary predicates. In §3 we introduce our representation of binary numerals, and in §4 and §5 we define predicates for binary addition and subtraction, and multiplication, respectively. In §6 we briefly describe our predicates for binary division, exponentiation, and logarithm. For lack of space, we relegate our pure Prolog implementation of exponentiation, logarithm, and non-interleaving binary multiplication to the accompanying source code,[8] along with additional proofs, tests, examples, and discussion. We review related work in §7, and conclude in §8. The full version of this paper[9] includes appendices outlining proofs of the properties of solution sets. We have also implemented declarative arithmetic as Haskell type-level relations (type classes) [9].

2 Predicates for Unary Arithmetic

We begin with unary numerals. Unary is simpler than binary, since every unary numeral is structurally part of its successor, so we may use structural recursion. However, membership and non-membership still need to be decidable for the base arithmetic predicates, whose domains are infinite however they are represented.

[8] `http://okmij.org/ftp/Prolog/Arithm/`
[9] `http://okmij.org/ftp/Prolog/Arithm/arithm.pdf`

Hence, the unary case already presents our main challenges. The unary case also lets us introduce and illustrate most of our terminology in this section.

We represent unary numerals as lists of atoms `u`: `[]` denotes zero, `[u]` denotes one, `[u,u]` denotes two, etc.[10] Throughout this section we use the shorthand **n** to indicate our representation of the number n—that is, the list of `u`'s of length n.

Although Prolog is not statically typed, we assume an implicit type of 'unary numerals' and an implicit typing of logic variables. For example, the implicit type of the term $[\mathtt{u}, \mathtt{u} \mid X]$ is 'unary numeral'; the logic variable X has the same type. The type predicate for unary numerals can be expressed as a generator:

```
genu([]).
genu([u|X]) :- genu(X).
```

A term t has the type of 'unary numeral' if the goal `genu`(t) succeeds. When providing denotations for terms and goals we use the implicit type of a logic variable to characterize the variable's domain.

We can add unary numerals with a special case of the `append/3` predicate.

```
add([],X,X).
add([u|X],Y,[u|Z]) :- add(X,Y,Z).
```

We can use `add/3` to add numerals: the goal `add`($\mathbf{1}, \mathbf{2}, X$) unifies X with **3**. We can also use `add/3` for subtraction: both `add`($X, \mathbf{1}, \mathbf{3}$) and `add`($\mathbf{1}, X, \mathbf{3}$) unify X with **2**, and the evaluation of `add`($X, \mathbf{3}, \mathbf{1}$) fails finitely. We can further use `add/3` to decompose a number into its summands: for example, `add`($X, Y, \mathbf{3}$) has four solutions, in which X and Y are unified in turn with **0** and **3**, **1** and **2**, **2** and **1**, and **3** and **0**. Finally, passing three distinct uninstantiated logic variables to `add/3` lets us enumerate the domain of addition over the natural numbers. The evaluation of the query `add`(X, Y, Z) in Prolog yields an infinite set of solutions:

```
X = []      Y = _G188  Z = _G188 ;
X = [u]     Y = _G188  Z = [u|_G188] ;
X = [u, u]  Y = _G188  Z = [u, u|_G188] ;
```

and so forth. This stream of solutions represents the infinite addition relation *in two different ways*. First, the goal produces an infinite number of solutions; we can always re-evaluate the goal to get another solution. Second, each solution represents infinitely many triples, all members of the addition relation. For example, the second solution compactly represents the infinitely many triples of naturals (x, y, z) for which x is one and z is the successor of y. Each instantiation of the free logic variable `_G188` to a member of its domain (corresponding to its implicit type) yields a new triple of naturals that is a member of the addition relation. We will make extensive use of this compact representation of infinite domains when proving termination properties of our predicates.

[10] We could just as easily use a more common representation z, $s(z)$, $s(s(z))$, etc. We chose lists for consistency with our binary representation.

2.1 Solution Sets

To formulate propositions about `add/3` and other predicates, we introduce the notion of *solution sets*, which accounts for the search strategy used to run a logic program (unlike other procedural notions of solutions). We use Prolog syntax [4]. We identify a bound logic variable with the term it is bound to, so all variables are free for us, as for Lloyd [4]. Our goals are all pure (contain no `var/1`, negation, or cuts).

We assume an *idempotent* notion of substitution [10]: a substitution θ is a finite map $\{ X_i = t_i \}$ from logic variables X_i to terms t_i such that no t_j contains any X_i. We write the application of the substitution θ to a term t as $t\theta$; this application easily extends to tuples of terms and sets of tuples.

Definition 1 (Conjunction of substitutions). *We define the* conjunction $\theta\xi$ *of two substitutions θ and ξ by treating them as sets of equations $\{ X_i = t_i \}$: we combine both sets (the result may contain two equations for the same X_i) and use unification to solve the resulting equations [10, §2.2.3]. Our conjunction of substitutions is thus commutative and associative. Because of the unification our conjunction is partial: if unification fails we call the original substitutions contradictory.*

We can interpret a goal g as a function from a substitution to a (finite or infinite) stream of substitutions. Both SLD and SLD-interleaving [5] interpret (right-associative and non-commutative) disjunction and conjunction by

$$(g_1; g_2)(\theta) = g_1(\theta) \oplus g_2(\theta),$$
$$(g_1, g_2)(\theta) = g_1(\theta) \star g_2$$

where the *bind* operation $\star$ is defined recursively by

$$[] \star f = [],$$
$$[\theta \mid \vec{\theta}] \star f = (f\theta) \oplus (\vec{\theta} \star f).$$

The difference between SLD and SLD-interleaving is that $\oplus$ above is defined as stream concatenation in SLD but stream interleaving in SLD-interleaving.

Definition 2 (Solution sequence and set). *Given a predicate `g/n` and n terms $t_1, \ldots, t_n$ that may contain logic variables, a* solution *of a goal $g(t_1, \ldots, t_n)$ is an n-tuple $(t'_1, \ldots, t'_n)$ where each t'_i instantiates t_i after the evaluation of the goal succeeds using SLD or SLD-interleaving strategy. A solution can be represented as $(t_1, \ldots, t_n)\theta$ for some substitution θ. A* solution sequence *is a sequence of solutions obtained by evaluating and successively re-evaluating a goal. The sequence is a* solution set *if no two of its members unify with each other.*

A goal that fails has the empty solution set. A goal whose evaluation or re-evaluation does not terminate does not have a solution sequence. In contrast, the goal `genu`(X) does have a solution sequence (which can be infinite) because

its evaluation and re-evaluation always terminates. We assume SLD as the default solution strategy. If we add the clause `add(X,[],X).` as a well-meaning optimization ($X + 0 = X$) after the first clause of `add/3` above, the solution sequence may not necessarily be a solution set: for example, the goal $\mathtt{add}(X, Y, \mathbf{0})$ will have $(\mathbf{0}, \mathbf{0}, \mathbf{0})$ in duplicate.

The notion of solution sequence is constructive, 'proof-theoretic,' to be distinguished from a model of a logic program: each solution has been actually *derived*, in finite time, from the facts and rules at hand using the given solution strategy. A term in a solution may contain free logic variables. We can prove the following properties of solution sets (see Appendix A of our full paper).

Proposition 1. *If a goal $g(t_1, \ldots, t_n)$ has a (finite or infinite) solution set $\{ (t_1, \ldots, t_n)\theta_i \}$ and ξ is a substitution that contradicts only finitely many θ_i, then the goal $g(t_1\xi, \ldots, t_n\xi)$ has a solution set $\{ (t_1, \ldots, t_n)(\theta_i\xi) \}$, omitting the elements where θ_i and ξ are contradictory.*

The solution set of a conjunction of goals $g_1(t_{11}, \ldots, t_{1n})$ and $g_2(t_{21}, \ldots, t_{2n})$ is (as in a natural database join) the set of tuples $\{ (t_{11}, \ldots, t_{1n}, t_{21}, \ldots, t_{2n}) \}$ after the evaluation of the conjunction has succeeded.

Proposition 2. *If the goals g_1 and g_2 have a finite solution set, then the conjunction g_1, g_2 has a finite solution set.*

This proposition is a corollary of the previous proposition. It does not generally hold if the solution set of one of the conjuncts is infinite. For example,

```
mf([],[u|Y]) :- genu(Y).
mf([u|X],Y)  :- mf([u|X],Y).
```

The goal $\mathtt{mf}(X, Y)$, with the variables X and Y free, has the infinite solution set $\{ (\mathbf{0}, \mathbf{i}) : i \in \mathbb{N}^+ \}$. However, the (left or right) conjunction of this goal with $X = \mathbf{1}$ diverges and has no solution set. Similarly, the conjunction of $\mathtt{mf}(X, Y)$ with $X = Y$ has no solution set either. Prop. 1 applies to neither conjunction because the substitutions $\{X = \mathbf{1}\}$ and $\{X = Y\}$ both contradict infinitely many solution-set substitutions (in fact, all of them). The latter conjunction is equivalent to the goal $\mathtt{mf}(X, X)$; such sharing of variables among the arguments of a goal is lethal to the termination guarantees below (such as Prop. 9).

Proposition 3. *If a goal $g_1(t_1, \ldots, t_n)$ has a finite solution set $\{ (t_1, \ldots, t_n)\theta_i \}$ and the goal $g_2(t'_1\theta_1, \ldots, t'_m\theta_1)$ has an infinite solution set $\{ (t'_1\theta_1, \ldots, t'_m\theta_1)\xi_j \}$, then the conjunction g_1, g_2 has the infinite solution set $\{ (t_1, \ldots, t_n, t'_1, \ldots, t'_m)(\theta_1\xi_j) \}$.*

This proposition describes the incompleteness of SLD: its underlying depth-first search becomes trapped exploring the leftmost infinite branch of the search tree. However, using the SLD-interleaving strategy, we can strengthen the proposition:

Proposition 4. *If a goal $g_1(t_1, \ldots, t_n)$ has a (finite or infinite) solution set $\{ (t_1, \ldots, t_n)\theta_i \}$ and the goal $g_2(t'_1\theta_i, \ldots, t'_m\theta_i)$ has a non-empty (finite or infinite) solution set $\{ (t'_1\theta_i, \ldots, t'_m\theta_i)\xi_{ij} \}$ for each i, then the conjunction g_1, g_2 has the solution set $\{ (t_1, \ldots, t_n, t'_1, \ldots, t'_m)(\theta_i\xi_{ij}) \}$.*

The proof is based on the laws of fair conjunction and disjunction in [5]; details are given in Appendix B of our full paper. We use analogous properties for disjunctions of goals.

2.2 Properties of Addition: Solution Sets of Addition

If t is a term and n is a natural number, then we write $[\mathtt{u}^n \mid t]$ to mean $[\mathtt{u}, \ldots, \mathtt{u} \mid t]$ where $\mathtt{u}, \ldots, \mathtt{u}$ consists of n occurrences of $\mathtt{u}$.

One can easily prove the following propositions:

Proposition 5. *The goal* $\mathtt{add}(X, Y, Z)$*, where* X *and* Y *are instantiated to ground numerals and* Z *is free, has a singleton solution set unifying* Z *with the numeral that is the sum of those corresponding to* X *and* Y*.*

That is, if the first two arguments of `add/3` are instantiated to numerals, the goal is decidable and has one solution.

Proposition 6. *The goal* $\mathtt{add}(X, Y, Z)$ *where* Z *is instantiated to a ground numeral has a finite solution set.*

The proof is an easy induction on the third argument.

Proposition 7. *The goal* $\mathtt{add}(X, Y, Z)$*, where* X *is a ground numeral* $\mathbf{n}$ *and* Y *and* Z *are free, has the singleton solution set* $\{\, (\mathbf{n}, G, [\mathtt{u}^n \mid G]) \,\}$ *where* G *is a free logic variable.*

The proof is by induction on X. The proposition easily extends to the case where Y and Z are arbitrary terms, using Prop. 1.

Proposition 8. *The goal* $\mathtt{add}(X, Y, Z)$*, where the arguments are distinct free logic variables, has an infinite solution set* $\{\, (\mathbf{n}, G, [\mathtt{u}^n \mid G]) : n \in \mathbb{N} \,\}$ *where* G *is a free logic variable. In each solution,* X *is unified to a ground numeral.*

The proof is by induction on X and soundness of SLD resolution.

Definition 3 (Denotation of arithmetic solutions). *The* denotation $[\![\cdot]\!]$ *of an arithmetic term or solution set is defined as follows.*

$$
\begin{aligned}
[\![\mathbf{n}]\!] &= \{n\} \\
[\![t]\!] &= \{\ n \in \mathbb{N} : n \geq m\ \} \\
&\quad \textit{where } t \textit{ is the non-ground term } [\mathtt{u}^m \mid X] \\
&\quad \textit{where } X \textit{ is a free logic variable of the type of unary numerals} \\
[\![(t_1, \ldots, t_n)]\!] &= [\![t_1]\!] \times \cdots \times [\![t_n]\!] \\
[\![S]\!] &= \cup_{s \in S} [\![s]\!] \quad \textit{for a solution set } S \textit{ (whose elements are all disjoint)}
\end{aligned}
$$

The previous propositions along with Prop. 1 let us prove:

Proposition 9. *The goal* $\mathtt{add}(X, Y, Z)$ *where the arguments have no shared logic variables has a solution set with* X *always unified to a ground numeral. The denotation of the solution set is* $\{\, (u, v, w) \in [\![X]\!] \times [\![Y]\!] \times [\![Z]\!] : u + v = w \,\}$*.*

The proof simply invokes Prop. 6 or Prop. 7 if Z or X is ground. If neither Z nor X is ground, then the arguments of the goal, since they share no logic variables, contradict only finitely many solutions in the set of Prop. 8, so we invoke Prop. 1.

The proposition states that the predicate `add/3` is fully decidable: for any arguments sharing no free variables, the predicate decides if the denotation of these arguments is in the domain of addition *or not*. Furthermore, for any subset of the domain of addition of the form $\{\,(u,v,w) \in [\![X]\!] \times [\![Y]\!] \times [\![Z]\!] : u+v=w\,\}$, there is a goal with exactly this denotation. We call such a predicate *faithful*.

2.3 Multiplication

Multiplication of unary numbers may seem as trivial as addition. (We will be developing several versions of multiplication, so we label the `mul` predicates with a numeric version suffix to distinguish them.)

```
mul1([],_,[]).
mul1([u|X],Y,Z) :- mul1(X,Y,Z1), add(Z1,Y,Z).
```

This predicate directly encodes the inductive definition of multiplication: $0 \cdot x = 0$, $(x{+}1){\cdot}y = x{\cdot}y{+}y$. Indeed, the goal `mul1`$(\mathbf{3},\mathbf{2},X)$ has the singleton solution set with X unified with $\mathbf{6}$. However, the goal `mul1`$(X,\mathbf{2},\mathbf{6})$ diverges after producing the first solution, the goal `mul1`$(X,\mathbf{2},\mathbf{5})$ diverges without producing anything, and `mul1`$(X,Y,\mathbf{6})$ overflows the stack after producing three solutions.

The problem is left-recursion in the second clause: the goal `mul1`$(X,\mathbf{2},Z)$ with free X and Z requires evaluating `mul1`$(X',\mathbf{2},Z')$ again with free X' and Z'. Reordering goals in the body of the second clause eliminates left-recursion:

```
mul2([],_,[]).
mul2([u|X],Y,Z) :- add(Z1,Y,Z), mul2(X,Y,Z1).
```

Now `mul2`$(X,\mathbf{2},\mathbf{6})$ has a singleton solution set and `mul2`$(X,\mathbf{2},\mathbf{5})$ fails finitely. However, whereas before `mul1`$(\mathbf{3},\mathbf{2},X)$ had a singleton solution set (with $X=\mathbf{6}$), this version `mul2` diverges after the first solution, so the goal has no solution set.

Interestingly, simply swapping the arguments to `add/3` fixes the problems.

```
mul3([],_,[]).
mul3([u|X],Y,Z) :- add(Y,Z1,Z), mul3(X,Y,Z1).
```

Now `mul3`$(X,\mathbf{2},\mathbf{6})$ and `mul3`$(\mathbf{3},\mathbf{2},X)$ both have singleton solution sets, and `mul3`$(X,\mathbf{2},\mathbf{5})$ fails finitely. The reason is important. Evaluating `mul2`$(\mathbf{3},\mathbf{2},Z)$ requires evaluating `add`$(Z_1,\mathbf{2},Z)$ whereas evaluating `mul3`$(\mathbf{3},\mathbf{2},Z)$ requires evaluating `add`$(\mathbf{2},Z_1,Z)$. Although both addition goals denote the same relation, the former has the infinite solution set $\{\,(\mathbf{0},\mathbf{2},\mathbf{2}),(\mathbf{1},\mathbf{2},\mathbf{3}),(\mathbf{2},\mathbf{2},\mathbf{4}),\dots\,\}$ whereas the latter has the singleton solution set $\{(\mathbf{2},G,[\mathtt{u}\,|\,G])\}$ (Prop. 7). That makes all the difference, as we prove below.

However, `mul3` is not perfect. Evaluating `mul3`$(X,Y,\mathbf{6})$ overflows the stack, because it requires evaluating `add`$(Y,Z_1,\mathbf{6})$, which gives $Z_1=\mathbf{6}, Y=\mathbf{0}$ as one solution. This solution causes a recursive call `mul3`$(X',Y,\mathbf{6})$, same as the original

call. Thus we must treat zero multiplicands separately, taking care to avoid overlapping solutions so that the solution sequence remains a solution set.

```
mul([],_,[]).
mul([u|_],[],[]).
mul([u|X],[u|Y],Z) :- add([u|Y],Z1,Z), mul(X,[u|Y],Z1).
```

This pattern of fixing one problem only to see another problem emerge is quite common, which is why we need proofs.

Proposition 10. *The goal* $\texttt{mul}(X, Y, Z)$*, where* X *and* Y *are instantiated to ground numerals and* Z *is free, has the singleton solution set that unifies* Z *with the numeral that is the product of the numerals for* X *and* Y*.*

The proof is an induction on X using Prop. 2 and Prop. 7.

Proposition 11. *The goal* $\texttt{mul}(X, Y, Z)$*, where* Z *is instantiated to a ground numeral, has a finite solution set.*

The proof depends on Prop. 2 and Prop. 6: each solution of $\texttt{add}([\texttt{u} \mid Y], Z_1, Z))$ instantiates Z_1 to a ground numeral smaller than the numeral for Z.

Proposition 12. *The goal* $\texttt{mul}(X, Y, Z)$*, where* Y *is instantiated to a positive ground numeral* $\mathbf{n}$ *and* X *and* Z *are free, has the solution set* $\{\, (\mathbf{i}, \mathbf{n}, \mathbf{i} \cdot \mathbf{n}) : i \in \mathbb{N} \,\}$*. If* Y *is* $\mathbf{0}$*, the solution set is finite:* $\{\, (\mathbf{0}, \mathbf{0}, \mathbf{0}), ([\texttt{u} \mid X], \mathbf{0}, \mathbf{0}) \,\}$*.*

The proof is by induction on Y and Prop. 7.

Proposition 13. *Under the SLD-interleaving strategy, the goal* $\texttt{mul}(X, Y, Z)$*, where the arguments share no logic variables, has a solution set that denotes* $\{\, (u, v, w) \in [\![X]\!] \times [\![Y]\!] \times [\![Z]\!] : u \cdot v = w \,\}$*.*

The proof depends on Props. 9, 12 and 4. Thus under SLD with interleaving, our `mul/3` predicate is faithful to the multiplication relation on naturals.

Without interleaving, the goal $\texttt{mul}(X, Y, Z)$ has the solution set $\{(\mathbf{0}, G, \mathbf{0}), ([\texttt{u} \mid G'], \mathbf{0}, \mathbf{0}), (\mathbf{1}, \mathbf{1}, \mathbf{1}), (\mathbf{2}, \mathbf{1}, \mathbf{2}), (\mathbf{3}, \mathbf{1}, \mathbf{3}), \ldots\}$. The denotation of this solution set obviously does not cover the entire multiplication relation: the second argument gets 'stuck' on 1. Under SLD, then, our predicate `mul` covers only an infinitesimal part of the domain of multiplication. We can do better: we define a predicate `semimul` that has the *same* termination properties as `mul` but covers *half* of the domain of multiplication, namely $Y \leq X$, whichever arguments are instantiated.

We first define the predicate `less/2`, corresponding to the less-than relation.

```
less([],[_|_]).
less([u|X],[u|Y]) :- less(X,Y).
```

The `semimul/3` predicate is then as follows.

```
semimul([],_,[]).
semimul([u|_],[],[]).
semimul([u|X],[u|Y],Z) :- less(X,Z), less(Y,[u|X]),
                          mul([u|X],[u|Y],Z).
```

Proposition 14. *The goal* `semimul`(X, Y, Z)*, where the arguments are free logic variables, has a solution set that denotes* $\{\,(u, v, w) \in \mathbb{N}^3 : v \leq u,\ u \cdot v = w\,\}$.

The proof depends on the goal `less`(X, Z), which asserts the trivial inequality $x < (x+1) \cdot (y+1)$ for all $x, y \in \mathbb{N}$. With free X and Z, the goal `less`(X, Z) has an infinite solution set whose solutions each instantiate X to a ground numeral. The goal `less`$(Y, [\mathtt{u} \mid X])$ then has a *finite* solution set that grounds Y as well. The last goal has thus a singleton solution set, by Prop. 10.

These attempts to define decidable multiplication even for the seemingly trivial unary case show the difficulties that become more pronounced as we move to binary arithmetic. We rely on a finite representation of infinite domains, precise instantiatedness analysis, and reasoning about SLD using search trees.

3 Binary Numerals

We represent numerals as lists of binary digits in little-endian order (least significant bit first), with zero represented as the empty list. A zero bit is denoted by `o` (lower-case 'oh') and a one bit is denoted by `l` (lower-case 'el')—to distinguish a number from its representation. For example, the terms `[]`, `[l]`, `[o,l]`, `[l,l]`, `[o,o,l]` represent 0 through 4. The last bit of a positive numeral *must* be `l`. Our code below takes special care to maintain this well-formedness condition. It is trivial to convert between this and Prolog's native representations of integers.

We often use the auxiliary one-clause predicates `zero/1`, `pos/1`, and `gtl/1`:

```
zero([]).
pos([_|_]).
gtl([_,_|_]).
```

The goal `zero`(N) succeeds if N is zero. The goal `pos`(N) succeeds if N is positive. The goal `gtl`(N) succeeds if N is at least two.

The predicate `genb/1` below expresses the implicit type of binary numerals.

```
genb([]).
genb([l|X]) :- genb(X).
genb([o|X]) :- pos(X), genb(X).
```

The presence of `pos`(X) in the last clause ensures that the last bit of a positive numeral is `l`. The code below contains similar guarding occurrences of `pos/1`.

Recall that one challenge of binary arithmetic is that a numeral is not structurally part of its successor. However, such a notion of inclusion exists (with the attendant induction principle) if we consider the 'length' of a binary number, i.e., the number of bits in its binary representation. More precisely, the length $\|n\|$ of a numeral n is $\lfloor \log_2 n \rfloor + 1$ if $n > 0$, and 0 if $n = 0$. We define `lessl/2` by

```
lessl([],[_|_]).
lessl([_|X],[_|Y]) :- lessl(X,Y).
```

It has the meaning and form of the unary `less` in §2.3—and the same termination properties—but compares the length of binary numbers rather their magnitude.

4 Addition and Subtraction

Our treatment of addition is inspired by hardware full-adders and multi-bit adders, as found in a digital computer's arithmetic logic unit [11]. A one-bit full-adder `full1_adder`$(C_{\text{in}}, A, B, S, C_{\text{out}})$ relates two input bits A, B and the incoming carry bit C_{in} with the sum bit S and the outgoing carry bit C_{out}, according to the equation $C_{\text{in}}+A+B = S+2C_{\text{out}}$. In Prolog, we define `full1_adder/5` by enumerating eight facts: `full1_adder(o,o,o,o,o).`, `full1_adder(o,l,o,l,o).`, etc. The multi-bit adder `fulln_adder`(C_{in}, A, B, S) relates the incoming carry bit C_{in} (either `o` or `l`), two binary numbers A, B, and their sum S, according to the equation $C_{\text{in}} + A + B = S$. It is defined by recursively combining one-bit adders as in a ripple-carry adder of digital logic, only in our case, the summands' bitwidths need not be the same or limited.

```
fulln_adder(o,A,[],A).
fulln_adder(o,[],B,B) :- pos(B).
fulln_adder(l,A,[],R) :- fulln_adder(o,A,[l],R).
fulln_adder(l,[],B,R) :- pos(B), fulln_adder(o,[l],B,R).

fulln_adder(Cin,[l],[l],R) :- R = [R1,R2],
        full1_adder(Cin,l,l,R1,R2).
fulln_adder(Cin,[l],[BB|BR],[RB|RR]) :- pos(BR), pos(RR),
        full1_adder(Cin,l,BB,RB,Cout),
        fulln_adder(Cout,[],BR,RR).
fulln_adder(Cin,A,[l],R) :- gtl(A), gtl(R),
        fulln_adder(Cin,[l],A,R).

fulln_adder(Cin,[AB|AR],[BB|BR],[RB|RR]) :-
        pos(AR), pos(BR), pos(RR),
        full1_adder(Cin,AB,BB,RB,Cout),
        fulln_adder(Cout,AR,BR,RR).
```

The first four clauses above deal with the cases of a summand being zero. The next three clauses handle the cases of a summand being one. The last clause adds numbers at least two bits wide. We take care to keep clauses from overlapping, so the solution sequence of a `fulln_adder` goal is a solution set. The splitting of the cases and the many occurrences of `pos/1` are necessary to keep all numerals in a solution 'well-typed': If logic variables in a solution are instantiated according to their implicit types, we never see a list whose last element unifies with `o`.

This multi-bit adder expresses binary addition, subtraction, and ordering:

```
add(A,B,C) :- fulln_adder(o,A,B,C).
sub(A,B,C) :- add(B,C,A).
less(A,B)  :- pos(X), add(A,X,B).
```

Our predicate `add/3` for binary numerals satisfies Prop. 5 (by induction on $4C_{\text{in}} + 3A + B$) and Prop. 6. However, due to carry propagation (clauses 4 and 6 of `fulln_adder`), Prop. 7 and Prop. 8 no longer accurately describe the solution

sets of binary `add`: the goal `add`(X, Y, Z), with either X or Y ground and the other arguments being distinct free variables, has an infinite solution set.

Prop. 9 holds (without X being always ground) only in the case of SLD with interleaving; for SLD, we can develop a binary addition predicate that covers half of the domain of addition, as in §2.3. The proofs begin by unrolling the recursion in clauses 3, 4, and 7 of `fulln_adder` (splitting cases for clauses 3 and 4), so that `fulln_adder` invokes itself recursively only for shorter arguments.

5 Multiplication

Binary multiplication may seem an obvious generalization of unary multiplication. The most complex case is to multiply an odd number by a positive one, $(2N+1)M = 2(NM)+M$, because it involves addition. One may think that the approach in `mul` in §2.3 will work here. Alas, binary addition does not satisfy Prop. 7, so we must turn to a much less obvious solution.

```
mul([],M,[]).
mul(N,[],[]) :- pos(N).
mul([1],M,M) :- pos(M).
mul([0|NR],M,[0|PR]) :- pos(M), pos(NR), pos(PR), mul(NR,M,PR).
mul([1|NR],M,P) :- pos(M), pos(NR), gtl(P),
        lessl3(P1,P,[1|NR],M),
        mul(NR,M,P1), add([0|P1],M,P).
```

This solution relies on a seemingly contrived predicate `lessl3`:

```
lessl3([],[_|_],_,_).
lessl3([_|P1R],[_|PR],[],[_|MR]) :- lessl3(P1R,PR,[],MR).
lessl3([_|P1R],[_|PR],[_|NR],M)  :- lessl3(P1R,PR,NR,M).
```

The goal `lessl3`(P_1, P, N, M) relates four numerals such that $\|P_1\| < \min(\|P\|, \|N\| + \|M\| + 1)$. As long as the arguments of the goal, *however instantiated*, share no logic variables, the goal has a solution set. In any solution, P_1 is a numeral whose bits may be free but whose length is fixed. We call such numerals *L-instantiated*. Moreover, whenever P_1, P, or both N and M are L-instantiated, the solution set is finite.

In the code for binary `mul` above, `lessl3` occurs in a clause that is selected when multiplying an odd number $2N+1$ (where $N > 0$) by a positive number M to yield P. Under these conditions, clearly $\lfloor \log_2 NM \rfloor$ is less than both $\lfloor \log_2 P \rfloor$ and $\lfloor \log_2(2N+1) \rfloor + \lfloor \log_2 M \rfloor + 2$, so the constraint imposed by `lessl3` does not affect the declarative meaning of `mul`. The guarantee of `lessl3` mentioned above lets us prove Prop. 10 and Prop. 11 for binary multiplication. For Prop. 11, `lessl3` ensures that P_1 is L-instantiated, so we can use induction to prove that `mul`(N, M, P) with an L-instantiated P has a finite solution set.

The important role of `lessl3` can be informally explained as follows. Given the goal `mul`(X, Y, Z), when X and Y are instantiated but Z is free, or when Z is instantiated but X and Y are free, the search space is finite even though the

free variables can be instantiated an infinite number of ways: the length of the product limits the lengths of the multiplicands, and vice versa. The predicate `lessl3` enforces these limits, even though our code cannot distinguish these two cases by determining which arguments are instantiated.

The `mul` predicate also satisfies Prop. 13 (the proof again relies on the properties of `lessl3`). As in the unary case, using SLD without interleaving, we cannot cover the whole domain of natural multiplication—but we can cover half of it. The technique is essentially the same as explained for unary multiplication.

6 Division, Exponentiation, and Logarithm

Our predicate for division with remainder $\texttt{div}(N, M, Q, R)$ relates four natural numbers such that $N = M \cdot Q + R < M \cdot (Q + 1)$. The implementation is quite complex so as to finitely fail as often as possible: for example, not only when the divisor M is zero, but also in the case $\texttt{div}([\texttt{o} \mid X], [\texttt{o}, \texttt{1}], Q, [\texttt{1}])$ with *free* X and Q, as no even number divided by 2 gives the remainder 1. Our algorithm is akin to long division as taught in elementary school, only done right-to-left. At each step, we determine at least one bit of the quotient in finite time.

First, we handle the easy case when the divisor M is bigger than the dividend N, so $Q = 0$ and $N = R$.

```
div(N,M,[],R) :- N = R, less(N,M).
```

Otherwise, Q must be positive. The second easy case is when N is at least M and has the same length as M.

```
div(N,M,[1],R) :- samel(N,M), add(R,M,N), less(R,M).
```

This code relies on the auxiliary predicate `samel`, which holds if its two arguments are binary numerals with the same number of digits.

```
samel([],[]).    samel([_|X],[_|Y]) :- samel(X,Y).
```

The main case of division is when N has more digits than M. The key is to represent $N = M \cdot Q + R < M \cdot (Q + 1)$ as the conjunction of the two relations

$$2^{l+1}R_1 = MQ_2 + R - N_2 \qquad \text{and} \qquad N_1 = MQ_1 + R_1$$

where

$$N = 2^{l+1}N_1 + N_2, \qquad Q = 2^{l+1}Q_1 + Q_2, \qquad l = \|R\|,$$

and N_2 and Q_2 are at most $l + 1$ long. Given l, then, we can decide the first relation. Because $0 \leq R_1 < M$, we can invoke $\texttt{div}(N_1, M, Q_1, R_1)$ recursively to decide the second relation. That gives us a convenient induction principle: either N_1 is zero and so Q_1 and $MQ_2 + R - N_2$ are both zero, or N_1 is positive and shorter than N. These two cases correspond to the disjunction below.

```
div(N,M,Q,R) :- lessl(M,N), less(R,M), pos(Q),
                split(N,R,N1,N2), split(Q,R,Q1,Q2),
                (
                    N1 = [],
                    Q1 = [],
                    sub(N2,R,Q2M),
                    mul(Q2,M,Q2M)
                ;
                    pos(N1),
                    mul(Q2,M,Q2M),
                    add(Q2M,R,Q2MR),
                    sub(Q2MR,N2,RR),
                    split(RR,R,R1,[]),
                    div(N1,M,Q1,R1)
                ).
```

The calls to `lessl/2` and `less/2` at the beginning of this code ensure that `M` and then `R` are both L-instantiated. The code also relies on the predicate `split/4` to 'split' a binary numeral at a given length: The goal `split`(N, R, N_1, N_2) holds if $N = 2^{l+1}N_1 + N_2$ where $l = \|R\|$ and $N_2 < 2^{l+1}$. The goal should be invoked only when R is L-instantiated, which is the case in the `div` code above. The goal has a finite solution set, in which N_2 is L-instantiated in every solution.

```
split([],       _, [],    []).
split([o,B|N], [], [B|N], []).
split([l|N],   [], N,     [l]).

split([o,B|N], [_|R], N1, [])     :- split([B|N],R,N1,[]).
split([l|N],   [_|R], N1, [l])    :- split(N,R,N1,[]).
split([B|N],   [_|R], N1, [B|N2]) :- pos(N2), split(N,R,N1,N2).
```

Our predicate `log`(N, B, Q, R) relates four numbers such that $N = B^Q + R < B^{Q+1}$, so it implements exponentiation, logarithm, and n-th root. Our implementation uses an upper bound on R, namely $RB < N(B-1)$, and upper and lower bounds on Q, namely $(\|B\|-1)Q < \|N\|$ and $\|N\|-1 < \|B\|(Q+1)$. These bounds constrain the search just as `lessl3` does for binary multiplication in §5. Because the base-2 case is so simple, we treat it separately.

7 Related Work

We first presented arithmetic predicates over binary natural numbers (including division and logarithm) in a book [12]. The book used miniKanren [13], which like its big sister Kanren [6] is an embedding of logic programming in Scheme. That presentation had no detailed explanations, proofs, or formal analysis, which is the focus of this paper.

Braßel, Fischer, and Huch's paper [7] appears to be the only previous description of declarative arithmetic. It is a practical paper couched in the programming

language Curry. It argues for declaring numbers and their operations in the language itself, rather than using external numeric data types and operations. It also uses a little-endian binary encoding of natural numbers (later extended to signed integers). Our encodings differ, however, in that our representation of natural numbers includes zero.

Whereas our implementation of arithmetic uses a pure logic programming language, Braßel, Fischer, and Huch use a non-strict functional-logic programming language. Therefore, our implementations use wildly different strategies and are not directly comparable. Also, we implement the logarithm relation.

Braßel, Fischer, and Huch leave it to future work to prove termination of their predicates. In contrast, the thrust of this paper is to formulate and prove decidability of our predicates under DFS or DFS with interleaving. To ensure decidability and completeness under generally incomplete strategies like DFS, our implementation is much more complex.

We present no benchmarks, but we are encouraged by Braßel, Fischer, and Huch's conclusion that declarative arithmetic is practical—and, in fact, has been used satisfactorily as part of a Haskell-based Curry system.

Our notion of solution sets can be defined in terms of Lloyd's SLD-trees [4]. Unlike Lloyd, we use solution sets not to show SLD incomplete. Rather, we use them to characterize the solutions generated by programs using SLD. Our solution sets are coarser, and thus easier to reason with, than the *partial trace semantics* often used to study termination of constraint logic programs [14].

Our approach is minimalist and pure; therefore, its methodology can be used in other logic systems, specifically, Haskell type classes. Hallgren [15] first implemented (unary) arithmetic in such a system, but with restricted modes. Kiselyov [16, §6] treats decimal addition more relationally. Kiselyov and Shan [9] first demonstrated all-mode arithmetic relations for arbitrary binary numerals, so to represent numerical equality and inequality constraints in the type system. Their type-level declarative arithmetic library enabled resource-aware programming in Haskell with expressive static guarantees.

8 Conclusions

In a pure logic programming system, we have declared decidable arithmetic of unrestricted unary and binary natural numbers: addition, multiplication, division with remainder, exponentiation, and logarithm with remainder. The declared relations have unlimited domain and are free from any mode restrictions or annotations. We have proven that our arithmetic predicates are fully decidable and faithfully represent the corresponding arithmetic relations. Our technique can be easily extended to full integers (i.e., a tuple of the sign and a natural number).

The gist of our approach is to limit the search space by a balancing act of computing *bidirectional* bounds from arguments of unknown instantiatedness. For example, the key to decidable multiplication is to limit the search using bounds that the inputs place on each other, without testing whether any input is instantiated. We also rely on the ability to finitely represent infinite domains

using logic variables, so that goals with infinite denotations may have only a finite solution set. Our notion of solution sets and the associated proof techniques are not specific to arithmetic and SLD; rather, they appear applicable to logic programming in other domains using a variety of search strategies.

Bibliography

[1] Apt, K.R.: Principles of Constraint Programming. Cambridge University Press (2003)

[2] Eiter, T., Gottlob, G., Mannila, H.: Disjunctive Datalog. ACM Transactions on Database Systems **22** (1997) 364–418

[3] Kowalski, R.: Predicate logic as programming language. In Rosenfeld, J.L., ed.: Information Processing, North-Holland (1974) 569–574

[4] Lloyd, J.W.: Foundations of Logic Programming. Springer-Verlag (1987)

[5] Kiselyov, O., Shan, C.c., Friedman, D.P., Sabry, A.: Backtracking, interleaving, and terminating monad transformers. In: Proceedings of the International Conference on Functional Programming. (2005) 192–203

[6] Friedman, D.P., Kiselyov, O.: A declarative applicative logic programming system. `http://kanren.sourceforge.net/` (2005)

[7] Braßel, B., Fischer, S., Huch, F.: Declaring numbers. In: Workshop on Functional and (Constraint) Logic Programming. (2007) 23–36

[8] Matiyasevich, Y.V.: Hilbert's Tenth Problem. MIT Press (1993)

[9] Kiselyov, O., Shan, C.c.: Lightweight static resources: Sexy types for embedded and systems programming. In: Draft Proceedings of Trends in Functional Programming, Seton Hall University (2007) TR-SHU-CS-2007-04-1.

[10] Baader, F., Snyder, W.: Unification theory. In Robinson, A., Voronkov, A., eds.: Handbook of Automated Reasoning. Elsevier (2001) 445–532

[11] Hennessy, J.L., Patterson, D.A.: Computer Architecture: A Quantitative Approach. Third edn. Morgan Kaufmann (2002)

[12] Friedman, D.P., Byrd, W.E., Kiselyov, O.: The Reasoned Schemer. MIT Press (2005)

[13] Byrd, W.E., Friedman, D.P.: From variadic functions to variadic relations: A miniKanren perspective. In: 7th Scheme and Functional Programming Workshop, University of Chicago (2006) 105–117 TR-2006-06.

[14] Colussi, L., Marchiori, E., Marchiori, M.: On termination of constraint logic programs. In: Principles and Practice of Constraint Programming. Volume 976 of LNCS. (1995) 431–448

[15] Hallgren, T.: Fun with functional dependencies. `http://www.cs.chalmers.se/~hallgren/Papers/wm01.html` (2001)

[16] Kiselyov, O.: Number-parameterized types. The Monad.Reader **5** (2005)