City University of New York (CUNY)
CUNY Academic Works

Computer Science Technical Reports CUNY Academic Works

2004

TR-2004004: Implementing and Automating Basic Number Theory in MetaPRL Proof Assistant

Yegor Bryukhov

Alexei Kopylov

Vladimir Krupski

Aleksey Nogin

How does access to this work benefit you? Let us know!

More information about this work at: https://academicworks.cuny.edu/gc_cs_tr/240
Discover additional works at: https://academicworks.cuny.edu

This work is made publicly available by the City University of New York (CUNY).
Contact: AcademicWorks@cuny.edu

Implementing and Automating Basic Number Theory in MetaPRL Proof Assistant*

Yegor Bryukhov[1], Alexei Kopylov[2], Vladimir Krupski[3], and Aleksey Nogin[4]

[1] Graduate Center, City University of New York
365 Fifth Avenue, New York, NY 10016
ybryukhov@gc.cuny.edu
[2] Department of Computer Science, Cornell University, Ithaca, NY 14853
kopylov@cs.cornell.edu
[3] Laboratory for Logical Problems of Computer Science
Department of Mathematical Logic and Theory of Algorithms
Faculty of Mechanics and Mathematics, Moscow State University
Vorob'evy Gory, 119899 RUSSIA
krupski@lpcs.math.msu.ru
[4] Department of Computer Science, California Institute of Technology
M/C 256-80, Pasadena, CA 91125
nogin@cs.caltech.edu

Abstract. No proof assistant can be considered complete unless it provides facilities for basic arithmetical reasoning. Indeed, integer theory is a part of the necessary foundation for most of mathematics, logic and computer science. In this paper we present our approach to implementing arithmetic in the intuitionistic type theory of the MetaPRL proof assistant. We focus on creating an axiomatization that would take advantage of the computational features of MetaPRL type theory. Also, we implement the Arith decision procedure as a *tactic* that constructs proofs based on existing axiomatization, instead of being a part of the "trusted" code base.

1 Introduction

MetaPRL [4,6] is the latest system in the PRL family of theorem provers [2,3]. The MetaPRL system combines the properties of an interactive LCF-style tactic-based proof assistant, a logical programming environment, and a formal methods programming toolkit. MetaPRL is also a logical framework that allows for reasoning in different logical theories. Its most extensively developed and most frequently used theory is a variation of the NuPRL intuitionistic type theory [3] (which in turn is based on the Martin-Löf type theory [9]).

* This work was supported in part by the DoD Multidisciplinary University Research Initiative (MURI) program administered by the Office of Naval Research (ONR) under Grant N00014-01-1-0765, the Defense Advanced Research Projects Agency (DARPA), the United States Air Force, the Lee Center, and by NSF Grant CCR 0204193.

Since MetaPRL type theory is so close to NuPRL's one, it is natural to use NuPRL's implementation of arithmetic as a basis for comparison. In NuPRL, a big part of the support for arithmetical reasoning is provided via two decision procedures — Arith [1] and Sup-Inf [14]. As output, these decision procedures do not provide real proofs; instead they only tell if current goal is provable or not according to their knowledge. This approach is at least imperfect because it extends the code base we have to *trust*.

Including such trusted decision procedures in a system that allows formalizing different logical theories, as well as different variations of the same logical theory, can have additional disadvantages. It significantly reduces the flexibility — whenever we want to change or update some aspects of a logical theory being used in such a system, we have to make sure that all the assumptions made by all the trusted decision procedures used remain valid in the updated theory. In MetaPRL, we wanted to avoid using trusted decision procedures, turning them into tactics instead. This way even if such procedure is flawed, or is not fully compatible with the logical theory used, the worst thing could happen is that it will fail to prove the statement it was applied to (of course, we still have to trust our proof checker).

Another goal we had when designing the arithmetical theory for MetaPRL is efficiency. MetaPRL is a highly efficient system; on most proof tasks it is over two orders of magnitude faster than its predecessor, NuPRL. We wanted the new arithmetics implementation to keep up with the efficiency spirit of the rest of the system.

While working on arithmetic code in MetaPRL, we wanted to create an implementation in which as much as possible could be reused between different logical theories of MetaPRL (both existing ones and any that could be added to MetaPRL in the future). However our main focus in this work is adding arithmetic to MetaPRL's implementation of NuPRL type theory.

Since we want all decision procedures to output explicit proofs of arithmetic inferences, we need to have a complete explicit axiomatization of arithmetic (as opposed to having large parts of the axiomatization in the form of trusted code of decision procedures). In this paper we propose such an axiom system and describe a proof constructing procedure (similar to a version of Arith implemented in Coq [7]) which succeeds in the same cases as NuPRL's Arith, while also generating an actual proof.

2 Choosing the Axioms

The first choice we needed to make is whether to define the arithmetical operators and types as primitive, postulating all the relevant axioms, or to define everything trough existing constructors. For example, we could have attempted to implement the type of natural numbers as `Unit List`. The choice we made is to try to pick a set of axioms that would be universally true across all the reasonable ways of defining the arithmetical primitives. These axioms are added to the system as basic postulates of the type theory; however at a later point we could derive them from other type constructors (using MetaPRL's derived rules mechanism [12]).

Before defining the set of axioms, one has to choose either integer or natural numbers as a basic type (and later define the remaining type via the chosen type). Both Arith [1] and Sup-Inf [14] use integers as primitive; defining natural numbers on top of integers is more straightforward than the opposite approach. For these reasons we chose to use integers as a primitive type. We used list of axioms from [1] as a prototype.

Another important choice that we had to make is the style of equality reasoning. There are two types of equalities in PRL type theory:

(A) Two terms can be equal as elements of a certain type. For example, two terms $\lambda x.t_1[x]$ and $\lambda x.t_2[x]$ would be equal as elements of a type $A \rightarrow B$ (written as "$\lambda x.t_1[x] = \lambda x.t_2[x] \in A \rightarrow B$") if for equal inputs of type A, t_1 and t_2 produce equal outputs of type B. Note that two terms could be distinct elements of one type and at the same time be equal in another type — for example, $\lambda x.t_1[x] = \lambda x.t_2[x] \in \mathtt{Void} \rightarrow T$ is true for arbitrary t_1, t_2 and T.

(B) Finer-grained computational/definitional equality[8] specifies that two terms refer to objects that are not just equal, but are actually *identical*. For example, the terms $\lambda x.a[x]\, b$ and $a[b]$ are computationally equal (written as "$\lambda x.a[x]\, b \equiv a[b]$"); terms $1+2$ and 3 are also computationally equivalent.

All things being equal, the equalities of the second kind are easier and more efficient to use. In PRL type theory, we are always allowed to replace a term with a computationally equal one; however we can only replace a term with an equal one when we can prove that the context will tolerate the equalities of the given type. In other words, to be allowed to replace $C[t_1]$ with $C[t_2]$, it is insufficient to be able to prove that $t_1 = t_2 \in T$; we are also required to prove a *well-formedness assumption* stating that C respects the equalities of type T.

Not surprisingly, in our axiomatization we pick computational equality over the equality in a type whenever possible. At the same time we chose to still include the typing assumptions in most of the computational equivalence rules. For example, the commutativity of addition rule is as follows:

$$\frac{\Gamma \vdash a \in \mathbb{Z} \qquad \Gamma \vdash b \in \mathbb{Z}}{\Gamma \vdash (a+b) \;\equiv\; (b+a)} \qquad (add\_Commut)$$

Assumptions look redundant in this rule (at least as long as the $+$ operator is not overloaded). However as we explained above, we want our axioms to stay valid for different formalizations of integers. In particular, in list implementation of natural numbers (with the type of integers being defined as a disjoint union of two list types and addition defined using recursion over lists) the above rule will be provable only in presence of the typing assumptions, as we would need to know that a and b are lists to be able to use their inductive properties.

We certainly want all arithmetic relations ($=, <, >$, etc) to be decidable. In PRL type theory, there are two different ways of defining a decidable relation.

The straightforward approach is to define a predicate (e.g. a function returning a *proposition*) on numbers with an additional axiom stating that this predicate happens to be decidable. The alternative is to postulate an existence of a function on numbers that returns a *boolean* result. To better understand the difference between two choices, it is useful to keep in mind that PRL type theory is *constructive*. A PRL proposition P is identified with a type of all *constructive witnesses* for P; there can be many different propositions and the type of all propositions is a pretty complicated one. By contrast, PRL booleans is a 1-bit type containing just the two boolean constants. While equivalent propositions could be very different, the booleans are completely transparent — if two booleans happen to be equivalent, they must be identical. Because of the latter feature of the boolean type, the rule

$$\frac{\Gamma \vdash a \in \mathbb{Z} \qquad \Gamma \vdash b \in \mathbb{Z} \qquad \Gamma \vdash c \in \mathbb{Z}}{\Gamma \vdash (a \ < \ b) \ \equiv \ ((a+c) \ < \ (b+c))} \qquad (\mathit{lt\_addMono})$$

will be valid in different implementations of $\mathbb{Z}$, as long as $<$ relation is defined via a boolean comparison function. But if $<$ is defined directly as a proposition, then the best that can be guaranteed is $(a \ < \ b) \ \Leftrightarrow \ ((a+c) \ < \ (b+c))$, which is significantly weaker.

When implementing support for numerals, we could either take the traditional route of building all the numerals using 0 and successor function, or we could simply *expose* MetaPRL's built-in numbers implementation. First approach looks more reliable (and trustworthy) since one only needs to trust the proof checker; however this approach is unbearably slow when one wants to do actual computation using these numerals. Second approach looks less reliable as built-in arithmetic now has to be trusted; however one might argue that it is not adding any new code to the trusted code base, but instead just exposing what is already a part of the prover. In the end, we decided to implement the second approach.

3 Axioms We Chose

In this section we provide an outline of our axiomatization, with an overview of the classes of axioms and some examples. The full list of axioms may be found in the listing of the MetaPRL theories [5, modules Itt_int_base and Itt_int_ext].

(A) Typing properties:

$$\frac{}{\Gamma \vdash \mathbb{Z}\,\mathrm{Type}} \qquad (\mathit{type\_of\_Int})$$

$$\frac{}{\Gamma \vdash \mathit{number}\{n\} \in \mathbb{Z}} \qquad (\mathit{type\_of\_number})$$

where `number` is the arithmetical *numeral* operator (e.g. `number`$\{n\}$ stands for an arbitrary numeral constant).

(B) Numbers are computationally transparent — two equal integers will necessarily be identical:

$$\frac{\Gamma \vdash a = b \ \in \mathbb{Z}}{\Gamma \vdash a \equiv b} \qquad (\mathit{intCongruence})$$

(C) Reduction of operations (and relations) on integer constants to meta-level operations on integers, e.g.:

$$\frac{}{\Gamma \vdash (\mathit{number}\{i\} + \mathit{number}\{j\}) \ \equiv \ \mathit{number}\{i +_m j\}} \qquad (\mathit{reduce\_add\_meta})$$

where $+_m$ performs addition of numeral constants using underlying internal arithmetic. This rewrite makes possible an evaluation from $1+2$ to 3.

(D) Well-formedness of operations and relations. As with any new operations we have to say term of what type it constructs, e.g.:

$$\frac{\Gamma \vdash a = a' \in \mathbb{Z} \qquad \Gamma \vdash b = b' \in \mathbb{Z}}{\Gamma \vdash (a + a') = (b + b') \in \mathbb{Z}} \qquad (\mathit{add\_wf})$$

(E) Equivalence of propositional and boolean relations. These two rules are actually define $=_b$ for integers via equality in $\mathbb{Z}$:

$$\frac{\Gamma \vdash \uparrow (a =_b b) \qquad \Gamma \vdash a \in \mathbb{Z} \qquad \Gamma \vdash b \in \mathbb{Z}}{\Gamma \vdash a = b \in \mathbb{Z}} \qquad (\mathit{beq\_int2prop})$$

where $\uparrow (t) \ ::= \ (t = \mathit{true} \ \in \ \mathbb{Z})$.

$$\frac{\Gamma \vdash a = b \in \mathbb{Z}}{\Gamma \vdash (a =_b b) \equiv \mathit{true}} \qquad (\mathit{beq\_int\_is\_true})$$

As it was said we decided to define boolean versions of $=$, $<$, etc as primitive and express propositional inequalities using boolean ones. However equality is so fundamental in PRL type theory that we decided to have both boolean and propositional equality as primitives and have rules that constitute their equivalence.

(F) Ring axioms — commutativity, associativity of $+$ and $*$, distributivity, properties of 0 and 1, e.g.:

$$\frac{\Gamma \vdash a \in \mathbb{Z}}{\Gamma \vdash (a + 0) \equiv a} \qquad (\mathit{add\_Id})$$

Here type condition on a is necessary because the rule establishes bidirectional equivalence relation and we, of course, do not want to allow replacing arbitrary term a with $a + 0$.

(G) Axioms of $<$-order ($<$ is irreflexive, transitive, asymmetric, discrete), connection between $<$ and arithmetic operations, e.g.:

$$\frac{\Gamma \vdash a \in \mathbb{Z} \qquad \Gamma \vdash b \in \mathbb{Z}}{\Gamma \vdash ((a \ <_b \ b) \ \wedge_b \ (b \ <_b \ a)) \equiv \mathit{false}} \qquad (\mathit{lt\_Reflex})$$

$$\frac{\Gamma \vdash a \in \mathbb{Z} \qquad \Gamma \vdash b \in \mathbb{Z}}{\Gamma \vdash ((a <_b b) \vee_b (b <_b a) \vee_b (a =_b b)) \equiv true} \quad (lt\_Trichot)$$

$$\frac{\Gamma \vdash a \in \mathbb{Z} \qquad \Gamma \vdash b \in \mathbb{Z}}{\Gamma \vdash (a <_b b) \equiv (((a + 1) =_b b) \vee_b ((a + 1) <_b b))} \quad (lt\_Discret)$$

First rule constitutes irreflexivity of $<$ relation, second constitutes that $<$ is a linear order and the third one constitutes discreteness of integers. These rules define properties of $<_b$ — the boolean version of $<$ relation; propositional version of $<$ and other inequalities are defined via it below, their properties are derivable from properties of $<_b$.

(H) Induction and definition of primitive recursion over $\mathbb{Z}$:

$$\frac{\begin{array}{ll} \Gamma;\ n : \mathbb{Z};\ \Delta[n];\ m : \mathbb{Z};\ v : m < 0;\ z : C[m+1] & \vdash C[m] \\ \Gamma;\ n : \mathbb{Z};\ \Delta[n] & \vdash C[0] \\ \Gamma;\ n : \mathbb{Z};\ \Delta[n];\ m : \mathbb{Z};\ v : 0 < m;\ z : C[m-1] & \vdash C[m] \end{array}}{\Gamma;\ n : \mathbb{Z};\ \Delta[n] \vdash C[n]} \quad (intElimination)$$

This rule is our formulation of the induction principle. We cover both negative and positive numbers in a single rule, so we have two separate induction steps.

The next rewrite is a part of definition of `ind` — the primitive recursion operation. We have two more (for zero and positive cases) rewrites to define `ind`.

$$\frac{\Gamma \vdash x < 0}{\Gamma \vdash \begin{array}{l} \mathtt{ind}\{x; i, j.down[i;j]; base; k, l.up[k;l]\} \equiv \\ \qquad down[x; \mathtt{ind}\{(x+1); i, j.down[i;j]; base; k, l.up[k;l]\}] \end{array}} \quad (reduce\_ind\_down)$$

(I) Expression of subtraction via negation, $>, <=, >=$ via $<$, propositional relations via boolean relations (except equality), e.g.:

$$(a - b) ::= (a + (-b))$$

$$(a < b) ::= (\uparrow (a <_b b))$$

$$(a \leq_b b) ::= (\neg_b (b <_b a))$$

(J) Inductive definition of integer division and remainder operations, e.g.:

$$\frac{\Gamma \vdash 0 \leq a \qquad \Gamma \vdash a < b \qquad \Gamma \vdash a \in \mathbb{Z} \qquad \Gamma \vdash b \in \mathbb{Z}}{\Gamma \vdash (a \,\%\, b) \equiv a} \quad (rem\_baseReduce)$$

4 Automation of Arithmetic Reasoning

As we have mentioned in the introduction, NuPRL has two decision procedures (Arith and Sup-Inf) automating the arithmetical reasoning. They both work with hypotheses and conclusion [5] in the form of quantifier free Presburger formulas [13], which are essentially arithmetic relations among linear forms; all non-linear subterms (after conversion of every polynomial to its canonical form) are considered to be variables of linear forms.

Arith is very limited in its proof-power but it is relatively fast. Depending on the kind of equation it gets as an input, it runs in either polynomial or exponential time (only $\neq$ in hypotheses and $=$ in conclusion add exponential part). Arith proves simple inequalities; specifically, it can prove inequalities that logically follow from hypotheses by associativity and commutativity of addition and multiplication, properties of 0 and 1, reflexivity, transitivity and weak monotonicity of inequalities. Weak monotonicity is the ge_addMono rule:

$$\frac{\Gamma \vdash a \in \mathbb{Z} \qquad \Gamma \vdash b \in \mathbb{Z} \qquad \Gamma \vdash c \in \mathbb{Z}}{\Gamma \vdash (a \ \geq_b \ b) \ \equiv \ ((a+c) \ \geq_b \ (b+c))} \qquad (\mathit{ge\_addMono})$$

with restriction that c has to be a numeral constant (or be reducible to one).

A big advantage of Arith is that it uses proof by contradiction [6] and constructs contradictory inequality that follows from assumptions (together with negated conclusion). This is what allows us to construct an actual proof from axioms based on Arith algorithm.

Sup-Inf is much more powerful. When used over rational numbers, Sup-Inf is complete and can provide counterexamples in case a proof fails. Sup-Inf has exponential complexity with respect to the number of equations.

As opposed to Arith, Sup-Inf algorithm does not provide a straightforward migration path that would have allowed turning it into a proof-building tactic. However Mayr's initial investigations [10] suggest that it should be possible to achieve this transformation; even if in a less direct manner.

Since Arith tactic implementation is clearly simpler and more straightforward than Sup-Inf, Arith seems to be a better choice for the initial testing of our axiomatization. As a result, we decided to start our work on proof automation with implementation of Arith procedure. Currently we have a working implementation that supports $+$, $-$(both unary and binary),$*$ and $=$,$\neq$,$<$,$>$,$<=$,$>=$ with arbitrary number of nested negation around them; the only unsupported operations are division and remainder.

5 Implementation of Arith

Our implementation provides the user with two main proof procedures — `normalizeC` and `arithT` [7]:

[5] The PRL type theory is formulated in a single-conclusion sequent form.

[6] Since it only involves decidable relations, the proof it generates is still valid in intuitionistic theory.

[7] The **C** and **T** suffixes is the MetaPRL convention for marking the class of a tool. **C** is used for functions that perform term rewriting — we call them **c**onversions and

The `normalizeC` rewriting procedure is used to rewrite polynomials. When applied to a polynomial term, `normalizeC` converts it to its canonical form (e.g. normalizes it). When applied to a term that has polynomial subterms in it, `normalizeC` will normalize all the polynomial subterms of the given term. For example, if a proof assumption has a form of an equality, one can apply `normalizeC` to the whole assumption and it will normalize both sides of the equality.

Example: The canonical form of $((b * 2 * (a + c)) - (a * b)) + 1$ is $1 + (a * b) + (2 * (b * c))$

In traditional decision procedures normalization step will be usually performed based on some representation of arithmetic terms that is internal to the procedure. In our case we perform an in-theory normalization — as we normalize, we prove every step, and eventually we build a proof of the equality between the original polynomial and the normalized one. In-theory normalization has a higher complexity since commutativity and associativity rules normally only allow swapping neighboring subterms. In-theory normalization also means having to work within a pretty restrictive set of allowed transformations; this makes it noticeably trickier than the "unsupervised" normalization with dedicated representation for arithmetic terms.

The canonical form of a polynomial is achieved by the following steps:

(A) Get rid of subtraction.
(B) Open parentheses using distributivity, move parentheses to the right using associativity of addition and multiplication, preform the basic simplifications (such as $0 \cdot a \to 0$, $1 \cdot a \to a$).
(C) In every monomial, sort (using the commutativity axiom) multipliers in increasing order, with numerical constants pushed to the left (We put coefficients first because later we have to reduce similar monomials). Multiply the constants if there is more than one numeral in one monomial; but if monomial does not have a constant multiplier at all, put 1 in front of it for uniformity.
(D) Sort monomials in increasing order, reducing similar monomials on the fly. As in previous step, numerals are pulled to the left (i.e. considered to be the least in the sort order).
(E) Get rid of zeros and ones in the resulting term.

The `arithT` proof search procedure implements Arith [1]:

(A) First it checks whether the conclusion of the goal sequent is an arithmetic fact, and if so, moves it into hypotheses in negated form (using reasoning by contradiction).
(B) Next, `arithT` converts all negative arithmetic facts in hypotheses to positive ones (it adds new hypotheses, and also leaves the original ones intact). Since there may be several nested negations, this step will be applied repetitively.

conversionals; **T** is used for the class functions capable of performing arbitrary proof search — **t**actics and **t**acticals.

(C) After that it converts all positive arithmetic facts in hypotheses into $\geq$-inequalities.

(D) Now every $\geq$-inequality is normalized — we use `normalizeC` to normalize the polynomials on both sides of every inequality.

(E) Then it tries to find the contradictory inequality that logically follows from that normalized $\geq$-inequalities and proves this implication. This problem is reduced to search for positive cycle in a directed graph. If successful, the resulting inequality will be derived from hypotheses.

(F) Finally, false is derived from found inequality, thus completing the proof by contradiction.

6 Ongoing Work and Future Directions

In this work we were able to come up with a very computation–oriented axiomatization of basic arithmetic. The initial experience of being able to use this axiomatization as a basis for creating LCF-style tactics for automating arithmetical reasoning was positive as well. However; we are still in somewhat early stages of this work — a lot more proof automation is needed and we now see several areas where we could improve the existing implementation as well.

In the nearest future we are going to investigate several of these challenges.

We are planning to further evaluate the usefulness and value of computational rewrites in our implementation of Arith— both from performance and proof size viewpoints. Currently, computational rewrites seem to be the right choice for polynomial normalization; however it is not as clear now how convenient they are going to be, for example, when trying to reduce different forms of inequalities to some canonical form.

Current implementation of arithmetical proof automation is neither easily extendable nor flexible. Currently we use a hardcoded set of rules and rewrites, and we would like to replace it with a more flexible declarative code. Two obvious points of improvement are polynomial normalization and reduction of different inequalities to one type. The standard approach for building extendable algorithms in MetaPRL is resources and resource annotation mechanism [11, Section 4.3] that allow splitting complicated proof search procedures into derived rules and short declarative annotations on those rule.

Currently arithmetical reasoning uses only one resource-driven procedure — `reduceC`. In arithmetical theories it is used to perform rewrites that simplify terms (e.g. $a+0 \rightarrow a$), but the `reduceC` is not specific to arithmetic, it is capable of performing a very large variety of simplifications and reductions (such as, for example, β-reduction, $\lambda x.t[x]\,a \rightarrow t[a]$). However, `reduceC` can not be used for transformations like commutativity ($a + b \longleftrightarrow b + a$), since this transformation does not has any clear directional properties.

In general, the rewriting task needs to be better classified and partitioned. Currently we only have two major groups of computational transformations — reductions and simplifications performed by `reduceC` and normalizations performed by `normalizeC`. However it is clear, that in many cases a much more fine-grain control is needed. In particular, it appears to be necessary to start distinguishing between reductions and simplifications.

In the introduction we mentioned that efficiency is an important goal for the MetaPRL community. We did not yet have a chance of doing any comprehensive performance evaluation of our implementation of Arith, but this is something we are hoping to be able to do in the near future.

And, of course, we are planning to keep adding new proof automation to the system. We are planning to continue our investigation of possible approaches to turning Sup-Inf into a tactic (possibly building on Mayr's work [10]).

References

1. T. Chan. An algorithm for checking PL/CV arithmetic inferences. In G. Goos and J. Hartmanis, editors, *An Introduction to the PL/CV Programming Logic*, volume 135 of *Lecture Notes in Computer Science*, appendix D, pages 227–264. Springer-Verlag, 1982.
2. Robert L. Constable. On the theory of programming logics. In *Proceedings of the 9th Annual ACM Symposium on the Theory of Computing,* Boulder, CO., pages 269–85, May 1977.
3. Robert L. Constable, Stuart F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the NuPRL Proof Development System.* Prentice-Hall, NJ, 1986.
4. Jason Hickey, Aleksey Nogin, Robert L. Constable, Brian E. Aydemir, Eli Barzilay, Yegor Bryukhov, Richard Eaton, Adam Granicz, Alexei Kopylov, Christoph Kreitz, Vladimir N. Krupski, Lori Lorigo, Stephan Schmitt, Carl Witty, and Xin Yu. MetaPRL — A modular logical environment. In David Basin and Burkhart Wolff, editors, *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003)*, volume 2758 of *Lecture Notes in Computer Science*, pages 287–303. Springer-Verlag, 2003.
5. Jason J. Hickey, Brian Aydemir, Yegor Bryukhov, Alexei Kopylov, Aleksey Nogin, and Xin Yu. A listing of MetaPRL theories. `http://metaprl.org/theories.pdf`.
6. Jason J. Hickey, Aleksey Nogin, Alexei Kopylov, et al. MetaPRL home page. `http://metaprl.org/`.
7. Daniel Hirschkoff. *Nodesat*, an arithmetical tactic for the Coq proof assistant. Technical Report 96-61, CERMICS, Noisy-le-Grand, April 1996.
8. Douglas J. Howe. Equality in lazy computation systems. In *Proceedings of the 4th IEEE Symposium on Logic in Computer Science*, pages 198–203, Asilomar Conference Center, Pacific Grove, California, June 1989. IEEE, IEEE Computer Society Press.
9. Per Martin-Löf. Constructive mathematics and computer programming. In *Proceedings of the Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, Amsterdam, 1982. North Holland.
10. S. Tobias Mayr. Generating primitive proofs from SupInf. Communicated to the NuPRL group at Cornell University, 1997.
11. Aleksey Nogin. *Theory and Implementation of an Efficient Tactic-Based Logical Framework.* PhD thesis, Cornell University, Ithaca, NY, August 2002.

12. Aleksey Nogin and Jason Hickey. Sequent schema for derived rules. In Victor A. Carreño, Cézar A. Muñoz, and Sophiène Tahar, editors, *Proceedings of the 15th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2002)*, volume 2410 of *Lecture Notes in Computer Science*, pages 281–297. Springer-Verlag, 2002.
13. M. Presburger. Über de vollständigkeit eines gewissen systems der arithmetik ganzer zahlen, in welchen, die addition als einzige operation hervortritt. In *Comptes Rendus du Premier Congrès des Mathématicienes des Pays Slaves*, pages 92–101. Warsaw, 1927.
14. Robert E. Shostak. On the SUP-INF method for proving Presburger formulas. *Journal of the Association for Computing Machinery*, 24(4):529–543, October 1977.