A Sequent Calculus for Integer Arithmetic with Counterexample Generation

Philipp Rümmer

Department of Computer Science and Engineering,
Chalmers University of Technology and Göteborg University, Sweden
philipp@chalmers.se

Abstract. We introduce a calculus for handling integer arithmetic in first-order logic. The method is tailored to Java program verification and meant to be used both as a supporting procedure and simplifier during interactive verification and as an automated tool for discharging (ground) proof obligations. There are four main components: a complete procedure for linear equations, a complete procedure for linear inequalities, an incomplete procedure for nonlinear (polynomial) equations, and an incomplete procedure for nonlinear inequalities. The calculus is complete for the generation of counterexamples for invalid ground formula in integer arithmetic. All parts described here have been implemented as part of the KeY verification system.

1 Introduction

We introduce a Gentzen-style sequent calculus for integer arithmetic that is tailored to integrated automated and interactive Java software verification. The calculus was developed for dynamic logic for the Java language [1, Chapter 3] (a classical first-order logic) and integrates well-known as well as new algorithms, with the goal to meet the following features:

- Simplification of arithmetic expressions or formulas with the goal to keep everything small and readable. A calculus for this purpose should always terminate and should not cause proof splitting; completeness is a secondary.
- Transparency and the ability to create human-readable proofs and sequences of simplification steps, otherwise it is difficult for a user to resume interactive proving after a number of automated proof steps. The fastest way to understand a proof goal is often to look at the history that led to the goal.
- Handling of nonlinear arithmetic guided by the user, which is necessary for programs that happen to contain multiplication or division operations. The cost of interactive software verification should be justified by the ability to also handle more complex programs than automatic tools.
- Generation of counterexamples for invalid formulas, which is useful during specification and when proving with induction and invariants.
- Handling of the actual modular Java integers, which in our system is modelled by a mapping to the mathematical integers [1, Chapter 12]. Reasoning in this setting requires good support for simplifying expressions, for instance by

(implicitly) proving the absence of overflows. The methods that we developed to this end are beyond the scope of the paper, but are based on the presented calculus.
– Most importantly: it should be easy to use!

Only some of these points can be solved using external procedures and theorem provers (which are, nevertheless, extremely useful for dealing with simpler proof obligations). As a complementary approach, we have developed a novel calculus for integer arithmetic that is directly implemented in our theorem prover KeY [1] in form of derived (i.e., verified) proof rules. The rules are deliberately kept as elementary as possible and are here presented in sequent calculus notation. The calculus is driven by a proof strategy that controls the rule application and realises the following components: (i) a simplification procedure that works on single terms and formulas and is responsible for normalisation of polynomials (Sect. 2), (ii) a complete procedure for systems of linear equations, based on Gaussian elimination and the Euclidian algorithm (Sect. 3), (iii) a complete procedure for systems of linear inequalities, based on Fourier-Motzkin variable elimination (Sect. 4), (iv) an incomplete procedure for nonlinear (polynomial) equations, based on Gröbner bases (Sect. 5), (v) an incomplete procedure for nonlinear inequalities using cross-multiplication of inequalities and systematic case analysis (Sect. 6).

The development of the method was mostly an engineering process with the goal of handling cases that practically occur in Java program verification. It was successful in the sense that many proofs that before only were possible with the help of external provers can now be handled by KeY alone (e.g., the case study [2]), and that many proofs that before were impossible have become feasible.

We do not consider quantifiers or uninterpreted functions in this paper. The calculus is proof confluent (cf. [3]) and can basically be used in two different modes: (i) for simplification, which disables the handling of nonlinear inequalities, prevents case splits and guarantees termination (Procedure 4 in Sect. 5), and (ii) for proving and countermodel construction, which enables all parts (Procedure 5 in Sect. 6).

Introductory example. We start with an example and show how the following statement can be proven within our calculus (in the "full" mode):[1]

$$11a + 7b \doteq 1 \;\vdash\; \langle \texttt{b=a*c-1; if (c>=a) a=a/b;} \rangle\, \mathit{true} \qquad (1)$$

In Java dynamic logic, this sequent expresses that the program in angle brackets terminates normally, i.e., in particular does not raise exceptions, given the assumption $11a + 7b \doteq 1$. A proof is conducted by rewriting the program following the symbolic execution paradigm [4], whereby the calculus presented in this

[1] On an Intel Pentium M processor with 1.6 GHz, the KeY implementation of the procedure needs about 460 inference steps and 2 seconds to find this proof.

$$\begin{array}{lr}
5c \mathrel{\dot{\geq}} -7e-8,\ e \mathrel{\dot{\leq}} -1,\ c \mathrel{\dot{\leq}} -1,\ c \mathrel{\dot{\geq}} 7e+2,\ 7ce \doteq -2c+1 \ \vdash & (13) \\ \hline
ce \mathrel{\dot{\geq}} -c-e-1,\ e \mathrel{\dot{\leq}} -1,\ c \mathrel{\dot{\leq}} -1,\ c \mathrel{\dot{\geq}} 7e+2,\ 7ce \doteq -2c+1 \ \vdash & (12) \\ \hline
e \mathrel{\dot{\leq}} -1,\ c \mathrel{\dot{\leq}} -1,\ c \mathrel{\dot{\geq}} 7e+2,\ 7ce \doteq -2c+1 \ \vdash & (11) \\ \hline
\cdots \qquad c \mathrel{\dot{\leq}} -1,\ c \mathrel{\dot{\geq}} 7e+2,\ 7ce \doteq -2c+1 \ \vdash & (10) \\ \hline
\ldots,\ c \mathrel{\dot{\geq}} 7e+2,\ 7ce \doteq -2c+1 \ \vdash & (9) \\ \hline
a \doteq 7e+2,\ b \doteq -11e-3,\ c \mathrel{\dot{\geq}} 7e+2 \ \vdash\ 7ce+2c-1 \not\doteq 0 & (8) \\ \hline
\cdots \qquad a \doteq 7e+2,\ b \doteq -11e-3,\ c \mathrel{\dot{\geq}} 7e+2 \ \vdash\ \{b := 7ce+2c-1\}\langle\, \texttt{a=a/b;}\,\rangle\ \mathit{true} & (7) \\ \hline
a \doteq 7e+2,\ b \doteq -11e-3 \ \vdash\ \{b := 7ce+2c-1\}\langle\, \texttt{if (c>=a) a=a/b;}\,\rangle\ \mathit{true} & (6) \\ \hline
a \doteq 7e+2,\ b \doteq -11e-3 \ \vdash\ \{b := a \cdot c-1\}\langle\, \texttt{if (c>=a) a=a/b;}\,\rangle\ \mathit{true} & (5) \\ \hline
a \doteq 7e+2,\ b \doteq -11e-3,\ d \doteq 3e+1 \ \vdash\ \langle\, \texttt{b=a*c-1; if (c>=a) a=a/b;}\,\rangle\ \mathit{true} & (4) \\ \hline
3a \doteq 7d-1,\ b \doteq -2a+d \ \vdash\ \langle\, \texttt{b=a*c-1; if (c>=a) a=a/b;}\,\rangle\ \mathit{true} & (3) \\ \hline
7b \doteq -11a+1 \ \vdash\ \langle\, \texttt{b=a*c-1; if (c>=a) a=a/b;}\,\rangle\ \mathit{true} & (2) \\ \hline
11a+7b \doteq 1 \ \vdash\ \langle\, \texttt{b=a*c-1; if (c>=a) a=a/b;}\,\rangle\ \mathit{true} & (1)
\end{array}$$

Fig. 1. Proof tree for the introductory example

paper is permanently applied on the *path condition* (in (1), $11a + 7b \doteq 1$) and the *symbolic variable assignment* (in (1), the identity).

The complete proof is shown in Fig. 1. As first step, all formulas are normalised: we choose an arbitrary well-ordering $<_r$ on the variables in the problem ($a <_r b <_r c$) and move big variables to the left and small variables to the right of the relations $\doteq$, $\dot{\leq}$, $\dot{\geq}$, resulting in (2). We then concentrate on the equation in (2), in order to (eventually) turn the leading coefficient 7 into a 1, by means of the extended Euclidian algorithm (cf. [5]). A basis transformation is performed that replaces b with a fresh variable d (such that $a <_r b <_r c <_r d$). One can minimise the coefficient of $11a$ by choosing $b \doteq -2a + d$ and replace the occurrence of b in the original equation with $-2a + d$ (afterwards, the equation is again normalised, sequent (3)). Because the leading coefficient of the first equation is still not 1, a second basis transformation $a \to 2d + e$ is performed (with $d <_r e$). This turns the leading coefficients of all equations into 1 (sequent (4)).

We can now leave out the equation $d \doteq 3e + 1$, because d does not occur in the sequent anymore. No further inferences are possible in the path condition and the first statement of the program is executed, *updating* the variable assignment accordingly (for simplicity, we assume that no overflows are possible). The assignment $b := 7ce + 2c - 1$ is written in front of the program in (5) and is rewritten and simplified using the equations in (6). The next program statement causes the proof to split on the condition $c \mathrel{\dot{\geq}} a$. One branch ($c \mathrel{\dot{<}} a$) can immediately be closed because the program contains no further statements. On the other branch (7), we obtain a new assumption $c \mathrel{\dot{\geq}} a$ that can be simplified.

The execution of the last assignment yields a new proof obligation (8) in order to prevent division by zero. We prove by contradiction and normalise the new equation in (9) (and also leave out the first two equations, which are no longer needed for the proof). Because all other possibilities fail in the resulting situation, a case split on the sign of one of the "independent" variables c or e is performed. Here, we will choose c and consider the cases $c \dot{\leq} -1$, $c \doteq 0$, and $c \dot{\geq} 1$. The case $c \doteq 0$ contradicts $7ce \doteq -2c + 1$, and the other two cases can be handled in essentially the same way, so we show only the first one in (10).

By transitivity, from the two inequalities in (10) the inequality $7e + 2 \dot{\leq} -1$ can be derived, which is rounded to $e \dot{\leq} -1$ in (11). No further linear inference steps are possible, but we can at this point deduce properties of product ce by *cross-multiplying* the inequalities $e \dot{\leq} -1$ and $c \dot{\leq} -1$, which yields the new inequality $0 \dot{\leq} (-c-1) \cdot (-e-1)$ in (12). After multiplying this inequality with 7, it can in (13) be rewritten using the equation $7ce \doteq -2c + 1$ and turned into $-2c + 1 \dot{\geq} 7 \cdot (-c - e - 1)$.

Now, a contradiction can be derived by reasoning about linear inequalities. From $5c \dot{\geq} -7e - 8$ and $c \dot{\leq} -1$ we derive $7e \dot{\geq} -3$, which is rounded to $e \dot{\geq} 0$ and a contradiction to $e \dot{\leq} -1$.

2 Normalisation of Arithmetic Expressions

Before starting a derivation and permanently during a proof, our calculus normalises (atomic) formulas. This was already demonstrated in the introductory example, and in a proof tree we denote such simplification steps with SIMP. We always fully expand polynomial expressions and represent them as a sum of monomials $\alpha_1 \cdot m_1 + \cdots + \alpha_n \cdot m_n$, in which $\alpha_1, \ldots, \alpha_n$ are non-zero integer literals and $m_1, \ldots, m_n$ are pairwise distinct products of variables (possibly 1 as the empty product, and possibly 0 as the empty sum). Full expansion is in general obviously a bad idea, but we found that it is a reasonable approach in interactive Java program verification that in the vast majority of cases improves the readability of formulas.

Sorting Terms. We put polynomial expressions into a canonical form by ordering the factors in a monomial and the monomials in a polynomial. The ordering $<_r$ that is used in both cases is a strict monomial ordering [6, 7]:

– We assume that a graded monomial ordering $<_r$ [6, 7] on products of variables is given, i.e., a well-ordering (a total, well-founded ordering) with the properties: (i) $\deg m < \deg m'$ implies $m <_r m'$, and (ii) $m <_r m'$ implies $x \cdot m <_r x \cdot m'$ for all variables x. In practice, we define $<_r$ as a graded lexicographic ordering: we assume that a well-ordering $<_r$ on variables[2] is given and then define $c_1 \cdots c_n <_r d_1 \cdots d_k$ if and only if $n < k$ or $n = k$ and $\{\{c_1, \ldots, c_n\}\} <_r \{\{d_1, \ldots, d_k\}\}$ (in the multiset extension of $<_r$, cf. [9]).

[2] In reality, instead of variables we have to deal with arbitrary terms whose head-symbol is not $+$ or $\cdot$, which are compared with a lexicographic path ordering [8].

– We extend $<_r$ by constructing a well-ordering on integer literals: $0 <_r 1 <_r -1 <_r 2 <_r -2 <_r 3 <_r \cdots$.
– We extend $<_r$ on monomials by $\alpha \cdot m <_r \alpha' \cdot m'$ if and only if $m <_r m'$ or $m = m'$ (modulo associativity and commutativity of $\cdot$) and $\alpha <_r \alpha'$.
– We extend $<_r$ on polynomials by $\alpha_1 m_1 + \cdots + \alpha_n m_n <_r \alpha'_1 m'_1 + \cdots + \alpha'_k m'_k$ if and only if $\{\{\alpha_1 m_1, \ldots, \alpha_n m_n\}\} <_r \{\{\alpha'_1 m'_1, \ldots, \alpha'_n m'_n\}\}$ (again using the multiset extension of $<_r$).

For sake of brevity, we will also compare arbitrary terms with $<_r$ and implicitly assume that the terms are first normalised.

Normalisation of Formulas. Atomic formulas are always written in the form $\alpha s * t$ with $* \in \{\dot{\leq}, \dot{=}, \dot{\geq}\}$, employing equivalences like $s \dot{<} t \Leftrightarrow s + 1 \dot{\leq} t$, and transformed so that the left-hand side αs is the $<_r$-greatest monomial of the polynomial $\alpha s - t$ and $\alpha > 0$. Furthermore, all inequalities are moved to the antecedent, and in case $\alpha s - t$ is a constant polynomial an equation or inequality is directly replaced with *true* or *false*.

We always demand that the coefficients of non-constant terms in an equation or inequality are coprime (do not have non-trivial factors in common), and otherwise divide all coefficients by the greatest common divisor. This also detects that equations like $2y \dot{=} 1 - 6c$ are unsolvable and equivalent to *false*, and that an inequality like $2y \dot{\leq} 1 - 6c$ can be simplified and rounded to $y \dot{\leq} -3c$ thanks to the discreteness of the integers.

Finally, we add a simple subsumption check for inequalities that eliminates an inequality $s \dot{\leq} t$ from the antecedent in case there is a second inequality $s \dot{\leq} t - \beta$ with $\beta \geq 0$ (correspondingly for $\dot{\geq}$).

3 Equation Handling: Gaussian Variable Elimination

In contrast to many decision procedures or SMT provers, equation and inequality handling for integers are kept separate in our system. The initial reason for this was that we believe that a reduction of equations to inequalities is not an option for interactive proving. Much later we became aware that we also can design more efficient, elegant and practical calculi for linear integer equations than for inequalities, which afterwards justifies the decision. We believe that this is also an important insight when working with the modular Java arithmetic, where the handling of such equations is essential. The sequent calculus described in this section is based on Gaussian elimination and the Euclidian algorithm.[3] It is complete, does not involve proof splitting, and is fast for all problems and benchmarks that we so far have looked at.

[3] The calculus is in parts inspired by [5, Chapter 4.5.2], but in contrast to [5] we perform both row and column operations.

Row Operations. The primary rule of the calculus reduces an expression with the help of an equation in the antecedent. The application of the rule is only allowed if s' is not a subterm of $s \doteq t$ (u is an arbitrary term):[4]

$$\frac{\Gamma, s \doteq t \ \vdash\ \phi[s' + u \cdot (s - t)], \Delta}{\Gamma, s \doteq t \ \vdash\ \phi[s'], \Delta}\ \textsc{red} \qquad \text{if} \ \ s' + u \cdot (s - t) <_r s'$$

Example 1. We show how the rules RED and SIMP are used to solve a system of linear equations (with the ordering $x <_r y$):

$$\cfrac{\cfrac{\cfrac{*}{x \doteq -5,\ y \doteq -1 \ \vdash\ x \doteq -5}}{3y \doteq x + 2,\ y \doteq -1 \ \vdash\ x \doteq -5}\ \textsc{red, simp}}{\cfrac{3y \doteq x + 2,\ 5y - (3y - x - 2) \doteq x \ \vdash\ x \doteq -5}{3y \doteq x + 2,\ 5y \doteq x \ \vdash\ x \doteq -5}\ \textsc{red}}\ \textsc{simp}$$

Column Operations. It is well-known that this kind of reduction alone does not yield a complete calculus for integer equations. An example is the formula $11a + 7b \doteq 1$ in the introductory example, for which no reduction steps are possible. To obtain a complete calculus, we also perform *column operations*—referring to the usual matrix representation of the Gaussian elimination method. Assuming that no more applications of RED are possible in a sequent, and given an equation $\alpha x \doteq s$ of the antecedent, we introduce a fresh unknown x' and perform a basis transformation $x \to u + x'$:

$$\frac{\Gamma, \alpha \cdot (u + x') \doteq s,\ x \doteq u + x' \ \vdash\ \Delta}{\Gamma, \alpha x \doteq s \ \vdash\ \Delta}\ \textsc{col-red}$$

if: x a variable, $\alpha > 1$, $(s - \alpha u) = \min_{<_r} \{s - \alpha u' \mid u' \ \text{a term}\}$,
x' a fresh variable, $<_r$-smaller than all previous symbols

The term u is chosen such that the difference $s - \alpha u$ becomes $<_r$-minimal. One subsequent application of SIMP will thus turn the new equation $\alpha(u + x') \doteq s$ into a formula $\beta y \doteq t$ with $\beta <_r \alpha$. Likewise, βy is $<_r$-smaller than the left-hand sides of other equations $\beta' y = t'$, because RED was applied exhaustively prior to COL-RED. This ensures the overall termination of the procedure (Lem. 1 below) and allows to continue with reduction steps as long as linear equations are present whose left-hand side has a non-unit-coefficient.

We do not apply the rule COL-RED to nonlinear equations, due to the experience that the basis transformations performed by COL-RED cause more harm than good in the nonlinear setting. This is because the usage of a good monomial ordering $<_r$ becomes far more important than in the linear setting (COL-RED effectively alters the ordering by introducing a new smallest variable, possibly in a harmful way). We further discuss this issue in Sect. 5.

[4] In the rule, we write $\phi[s']$ in the succedent to denote that the term s' can occur in an arbitrary position in the sequent, in particular also in the antecedent.

Procedure 1. *Apply* SIMP *with the highest priority,* RED *with second-highest priority, and* COL-RED *with the lowest priority.*

Lemma 1. *Procedure 1 terminates (for sequents containing arbitrary equations and inequalities). For sequents that only contain linear equations, it is complete and proof confluent.*

Example 2. If a proof branch does not get closed by this procedure, the remaining equations are an explicit description of all solutions (counterexamples) of the equations:

$$\dfrac{\begin{array}{c} x_0 \doteq 125x_3'' - 4,\ x_1 \doteq 25x_3'' - 1,\ x_2 \doteq 20x_3'' - 1,\ x_3 \doteq 16x_3'' - 1, \\ x_0' \doteq 16x_3'',\ x_3' \doteq -3x_3'' \end{array} \vdash}{\begin{array}{c} \vdots \\ x_0 \doteq 5x_1 + 1,\ 4x_1 \doteq 5x_2 + 1,\ 4x_2 \doteq 5x_3 + 1\ \vdash \end{array}}$$

The equations that define x_0' and x_3' can be removed afterwards, because these symbols do not occur in the original problem and have no impact on its validity. A concrete counterexample is obtained by assigning arbitrary values to the variables that only occur in the right-hand sides of equations (x_3'').

4 Handling of Linear Inequalities: Fourier-Motzkin Variable Elimination and Case Splits

Although Fourier-Motzkin variable elimination (cf. [10]) generally has a high complexity, it is one of the most popular methods to handle linear inequalities and used in proof assistants like PVS [11], Coq [12] or ACL2 [13, 14]. We found Fourier-Motzkin to be a suitable base method both for linear and nonlinear inequality handling: most reasoning during verification is rather shallow and most inequalities only share symbols with a small number of other inequalities (sparse constraints), which is a situation where Fourier-Motzkin works well. At the same time, the Fourier-Motzkin elimination rule is suited for interactive proving due to its simplicity and the fact that it directly works on integers, in contrast to more efficient linear programming techniques. The full procedure given in this section is complete over the integers, but it involves proof splitting and does usually not terminate for invalid sequents, which means that it cannot (directly) be used as a simplifier for interactive proving. We therefore also identify a subset of the method that does not cause splitting and always terminates, but which is no longer complete (which hardly ever matters in practice). An example for a program that can be verified using the incomplete procedure (together with axioms for division, modulo and Java arithmetic) is shown in Fig. 2.

The Incomplete Procedure. As equations have already been handled in the previous section, we can implement Fourier-Motzkin with a single rule for "cancelling" two inequalities:

$$\frac{\Gamma, \alpha s \dot{\geq} t, \beta s \dot{\leq} t', \beta t \dot{\leq} \alpha t' \ \vdash \ \Delta}{\Gamma, \alpha s \dot{\geq} t, \beta s \dot{\leq} t' \ \vdash \ \Delta}\ \textsc{fm-elim} \qquad\qquad \text{if } \alpha > 0,\ \beta > 0$$

The resulting inequality $\beta t \dot{\leq} \alpha t'$ does no longer contain the monomial s and is therefore $<_r$-smaller than both previous inequalities (after a subsequent application of SIMP). To ensure termination, the rule must never be applied twice on a proof branch to the same pair of inequalities.

The performance of Fourier-Motzkin can be improved by adding a rule that turns two inequalities into an equation, based on the law of anti-symmetry:

$$\frac{\Gamma, s \doteq t \ \vdash \ \Delta}{\Gamma, s \dot{\leq} t, s \dot{\geq} t \ \vdash \ \Delta}\ \textsc{anti-symm}$$

Procedure 2. *Apply Procedure 1 (linear equations) with the highest priority, the rule* ANTI-SYMM *with second highest priority and the rule* FM-ELIM *with lowest priority.*

Lemma 2. *The procedure obtained in this way terminates when applied to a sequent containing arbitrary equations and inequalities.*

The Complete Procedure. Fourier-Motzkin is complete for rationals, but incomplete for integers. Our calculus is already more complete than pure Fourier-Motzkin due to the normalisation from Sect. 2 (rounding of inequalities) and the different equation handling of Procedure 1, which are enough to handle many cases that occur in practice (e.g., to show the inconsistency of $4x \dot{\geq} 5 \wedge 4x \dot{\leq} 7$). Making the calculus actually complete has therefore not been of great importance for us. The following approach to this end is rather simplistic, but it has a counterexample generation property that is practically more relevant.

Our calculus becomes complete by performing a systematic case analysis, i.e., by doing proof splitting, in a way similar to Gomory's cutting-planes (cf. [10]). This is realised by the following rule for investigating the borderline case of an inequality:

$$\frac{\Gamma, s \dot{<} t \ \vdash \ \Delta \quad \Gamma, s \doteq t \ \vdash \ \Delta}{\Gamma, s \dot{\leq} t \ \vdash \ \Delta}\ \textsc{strengthen}$$

There is a corresponding rule for $\dot{\geq}$. The application of these rules does obviously not terminate in general, but it does for valid sequents (of linear inequalities), provided that a fair application strategy[5] is used and the rule is combined with

[5] In the presence of subsumption checks (Sect. 2), we consider a strategy as fair if STRENGTHEN is eventually applied to each inequality or to any subsuming inequality.

```
/*@
  @ normal_behavior
  @ requires -Decimal.PRECISION < f && f < Decimal.PRECISION
  @          && e + intPart < 32767 && -32768 < e + intPart;
  @ requires -Decimal.PRECISION < decPart && decPart < Decimal.PRECISION;
  @ modifiable intPart, decPart;
  @ ensures intPart * Decimal.PRECISION + decPart ==
  @         (\old(intPart) + e) * Decimal.PRECISION + \old(decPart) + f;
  @ ensures -Decimal.PRECISION < decPart && decPart < Decimal.PRECISION;
  @*/
public void add(short e, short f) {
  intPart += e;
  if ( intPart > 0 && decPart < 0 ) {
    intPart--; decPart = (short)( decPart + PRECISION );
  } else if ( intPart < 0 && decPart > 0 ) {
    intPart++; decPart = (short)( decPart - PRECISION ); }
  decPart += f;
  if ( intPart > 0 && decPart < 0 ) {
    intPart--; decPart = (short)( decPart + PRECISION );
  } else if ( intPart < 0 && decPart > 0 ) {
    intPart++; decPart = (short)( decPart - PRECISION );
  } else {
    short retenue = 0; short signe = 1;
    if ( decPart < 0 ) {
      signe = -1; decPart = (short)( -decPart ); }
    retenue = (short)( decPart / PRECISION );
    decPart = (short)( decPart % PRECISION );
    retenue *= signe; decPart *= signe; intPart += retenue;
  } }
```

Fig. 2. Addition method of class `Decimal` taken from [15], where it was verified using the Loop tool and PVS [11]. This method is part of the JavaCard Purse applet by Gemplus [16]. Using the KeY implementation of our calculus, it takes about 200 seconds and 26000 rule applications to automatically verify that the method adheres to its specification, reasoning about the modular arithmetic of Java.

Procedure 2. For an invalid sequent, a fair strategy eventually produces goals in which all inequalities have been replaced with equations and where Procedure 1 can take over and produce a counterexample.

Case distinctions are also necessary to handle equations in the succedent:

$$\frac{\Gamma \vdash s \mathrel{\dot{\leq}} t, \Delta \quad \Gamma \vdash s \mathrel{\dot{\geq}} t, \Delta}{\Gamma \vdash s \doteq t, \Delta} \text{ SPLIT-EQ}$$

Procedure 3. *Apply Procedure 2 (the incomplete method) with the highest priority, the rule* SPLIT-EQ *with second highest priority, and the rule* STRENGTHEN *with lowest priority and in a fair manner.*

Lemma 3. *This procedure is complete and proof confluent, and it eventually produces a counterexample for an invalid sequent.*

Example 3. Consider the following example taken from [17]: Because Procedure 2 is not able to derive a contraction, we apply STRENGTHEN to $x \mathrel{\dot{\leq}} 2$ and obtain two cases $x \doteq 1$, $x \doteq 2$ (thanks to ANTI-SYMM), the second of which leads to a counterexample:

$$
\dfrac{\dfrac{\begin{array}{cc}
\dfrac{*}{y \mathrel{\dot{\geq}} 1,\ y \mathrel{\dot{\leq}} 0,\ x \doteq 1 \ \vdash}\ \textsc{fm-elim} & \dfrac{y \doteq 1,\ x \doteq 2 \ \vdash}{y \mathrel{\dot{\geq}} 1,\ y \mathrel{\dot{\leq}} 1,\ x \doteq 2 \ \vdash}\ \textsc{anti-symm} \\
\vdots & \vdots \\
4y \mathrel{\dot{\geq}} x+1,\ 4y \mathrel{\dot{\leq}} x+2,\ x \doteq 1 \ \vdash & 4y \mathrel{\dot{\geq}} x+1,\ 4y \mathrel{\dot{\leq}} x+2,\ x \doteq 2 \ \vdash
\end{array}}{\vdots}}{4y \mathrel{\dot{\geq}} x+1,\ 4y \mathrel{\dot{\leq}} x+2,\ x \mathrel{\dot{\leq}} 2,\ x \mathrel{\dot{\geq}} 1 \ \vdash}\ \textsc{strengthen}
$$

5 Handling of Nonlinear Polynomial Equations: Pseudo-Reduction and Gröbner Bases

The validity of equations or inequalities over arbitrary (possibly nonlinear) polynomials over the integers is known to be undecidable [18]. This means that all rules and procedures that we give from now on can never be complete and have been employed or developed with the aim of handling the common cases: when verifying programs, a large amount of the occurring nonlinear proof obligations can and should be taken care of automatically by incomplete calculi. The most important step to this end is to normalise nonlinear expressions (Sect. 2). We describe a comparatively cheap extension—that does not cause any proof splitting and is suited for interactive proving—of Procedure 1 to deal with nonlinear equation.

Pseudo-Reduction. As in Sect. 3, the primary rule for rewriting with (nonlinear) equations is RED. Because we do not apply the rule COL-RED to nonlinear equations, however, there are cases where equations $\alpha s \doteq t$ with $\alpha > 1$ remain in the antecedent that cannot be simplified further. In the sequent $x \mathrel{\dot{\geq}} 1,\ y \mathrel{\dot{\geq}} 1,\ 2z^2 \doteq y \ \vdash\ xz^2 \mathrel{\dot{\leq}} xy$, for instance, none of the rules so far can be applied. In order to handle such cases, we introduce a further reduction rule that is based on pseudo-division and works by first multiplying the target expression with a constant (cf. [5]). The rule must only be applied if $\alpha s \doteq t$ and $u \cdot t \doteq \alpha t'$ are different equations:

$$
\dfrac{\Gamma, \alpha s \doteq t \ \vdash\ \phi[u \cdot t \doteq \alpha t'], \Delta}{\Gamma, \alpha s \doteq t \ \vdash\ \phi[s' \doteq t'], \Delta}\ \textsc{pseudo-red} \qquad \text{if}\ \ \deg s > 1,\ \alpha > 1,\ s' = u \cdot s
$$

There are similar rules for inequalities $s' \mathrel{\dot{\leq}} t'$, $s' \mathrel{\dot{\geq}} t'$. We apply PSEUDO-RED only if the left-hand side of the equation $\alpha s \doteq t$ is nonlinear and $\alpha > 1$. Otherwise, the normal reduction rule RED can be used, possibly after turning α into 1 with help of COL-RED.

Gröbner Bases. Rewriting with nonlinear equations using the rules RED and PSEUDO-RED is not confluent and is not able to decide ideal membership in a ring of polynomials. Ideal membership is an approximation of semantic entailment of (nonlinear) equations that we can practically decide: we complete the set of antecedent equations by computing a Gröbner basis [6].

The simplest way to generate a Gröbner basis is to saturate the antecedent with "S-polynomial"-equations by considering all critical pairs of existing integer equations—the Buchberger algorithm [6]. Our calculus produces a non-reduced Gröbner basis over the field of rational numbers that only consists of polynomial equations with integer coefficients, which are easier to compute and almost as useful for reduction as actual Gröbner bases over the integers. Given two equations with overlapping left-hand sides, S-polynomials are added as follows:

$$\frac{\Gamma, s \doteq t, s' \doteq t', s'_r \cdot t \doteq s_r \cdot t' \;\vdash\; \Delta}{\Gamma, s \doteq t, s' \doteq t' \;\vdash\; \Delta}\ \text{S-POLY} \qquad \begin{array}{l} s = \gcd(s, s') \cdot s_r, \\ s' = \gcd(s, s') \cdot s'_r, \\ 0 < \deg s_r < \deg s, \\ 0 < \deg s'_r < \deg s' \end{array}$$

Similarly to the Fourier-Motzkin elimination rule, this rule must not be applied repeatedly for the same pair of equations to ensure termination. The performance of this naive implementation of Buchberger's algorithm is not comparable with more advanced methods, of course. We have yet to find, however, a verification problem where this would be a problem.

Procedure 4. *Apply Procedure 1 (linear equations) with highest priority, the rule* PSEUDO-RED *with second highest priority, the rule* S-POLY *with third highest priority, and Procedure 2 (linear inequalities) with lowest priority.*

Lemma 4. *Procedure 4 terminates when applied to a sequent containing arbitrary equations and inequalities.*

6 Handling of Nonlinear Polynomial Inequalities: Cross-Multiplication and Case Splits

The handling of nonlinear polynomial inequalities is realised as an extension of the linear inequality handling (Sect. 4). In order to apply linear reasoning to nonlinear arithmetic, we generate linear approximations of products and incrementally strengthen the precision of the approximations through case distinctions. Likewise, case splits are used to ensure the *existence* of linear approximations. Our method has been developed as a heuristic, and we do not have an exact description of the fragment of nonlinear arithmetic that it can handle. The main application areas where the method has proven to be extremely useful are correctness proofs for lemma rules that can be loaded by the prover KeY [1], and the verification of programs with the actual modular integer semantics of Java.

Similarly to the approach in ACL2 [14, 19] (and using their terminology), the primary rule to handle nonlinear inequalities is *cross-multiplication:*

$$\frac{\Gamma, s \mathrel{\dot{\le}} t, s' \mathrel{\dot{\le}} t', 0 \mathrel{\dot{\le}} (t-s)\cdot(t'-s') \ \vdash\ \Delta}{\Gamma, s \mathrel{\dot{\le}} t, s' \mathrel{\dot{\le}} t' \ \vdash\ \Delta}\ \textsc{cross-mult}$$

There are corresponding rules for $\dot{\ge}$ and for mixed pairs of inequalities. As usual in order to ensure termination, CROSS-MULT must not be applied repeatedly to the same pair of inequalities.

We can give a geometric interpretation of cross-multiplication: for two linear inequalities $x \mathrel{\dot{\le}} \alpha$, $y \mathrel{\dot{\le}} \beta$, cross-multiplication introduces a linear approximation of the product (the bilinear term) xy. In this particular case, the right-hand side of the new inequality $xy \mathrel{\dot{\ge}} \beta x + \alpha y - \alpha\beta$ is the greatest plane that bounds the expression xy from below (under the assumptions $x \mathrel{\dot{\le}} \alpha$, $y \mathrel{\dot{\le}} \beta$). More generally, the result of cross-multiplication is a bound on the value of a monomial in terms of $<_r$-smaller monomials. Deriving such bounds is, in practical cases, often sufficient to prove statements in nonlinear arithmetic.

Restricting Cross-Multiplication. An unrestricted application of the rule CROSS-MULT can produce arbitrarily many inequalities and does not terminate. As a heuristic, we only use CROSS-MULT if the product $s \cdot s'$ already occurs as a factor within a left-hand side of an equation or inequality (ignoring the coefficient of $s \cdot s'$). Although this is not strong enough to ensure termination, it guarantees that the total degree of occurring monomials is bounded. We found this heuristic to work reasonably well for most cases (a counterexample is Ex. 5 below).

Case Splits. For two reasons, it is crucial to combine cross-multiplication with case distinctions: (i) nonlinear monomials over the complete set of integers do in general not have linear bounds (observe, for instance, that the term xy is not bounded from above or below by any linear expression in x and y). (ii) case distinctions are in general the only way to strengthen linear bounds (again, consider the term xy under the assumptions $x \mathrel{\dot{\le}} \alpha$, $y \mathrel{\dot{\le}} \beta$, for which no more precise linear lower bound exists than $\beta x + \alpha y - \alpha\beta$).

To account for (i), we introduce a rule that splits over the sign of the value of a term. We apply this rule for variables x that occur in the left-hand side of equations or inequalities:

$$\frac{\Gamma, x \mathrel{\dot{<}} 0 \ \vdash\ \Delta \quad \Gamma, x \doteq 0 \ \vdash\ \Delta \quad \Gamma, x \mathrel{\dot{>}} 0 \ \vdash\ \Delta}{\Gamma \ \vdash\ \Delta}\ \textsc{sign-cases}$$

Ternary splits are motivated by the observation that the case $x \doteq 0$ usually is easy to handle (significantly easier than the original problem), while at the same time a strict inequality $x \dot{>} 0$ appears to be of much greater use in cross-multiplication than $x \dot{\geq} 0$ (and correspondingly for $x \dot{<} 0$). In our experience, the rule SIGN-CASES outperforms binary cuts.

Point (ii) is accommodated by using the rule STRENGTHEN from Sect. 4, which we apply to linear inequalities in order to incrementally restrict the domain of a variable. For the example above, after strengthening the inequality $x \dot{\leq} \alpha$ to $x \dot{\leq} \alpha - 1$, we can also derive a better bound $\beta x + (\alpha - 1)y - \alpha\beta + \beta$ for the value of xy.

Procedure 5. *Apply Procedure 4 (equations handling and the incomplete procedure for linear inequalities) with the highest priority, the rule* SPLIT-EQ *with second highest priority, and the rules* CROSS-MULT, SIGN-CASES *and* STRENGTHEN *with the lowest priority and in a fair manner.*

Example 4. We give three further examples that can be proven using Procedure 5 (the last two ones are taken from [14, 19]). In practice, it can often be observed that Procedure 5 is able to solve nonlinear equational problems that cannot be proven using Procedure 4 (only using Gröbner bases).

$$xy \doteq 0 \ \vdash \ x \doteq 0,\ y \doteq 0 \qquad x^2 \doteq 2 \ \vdash \qquad 0 \dot{<} ab,\ 0 \dot{<} cd,\ 0 \dot{<} ac \ \vdash \ 0 \dot{<} bd$$

Example 5. A valid sequent that is not provable due to the restriction on the application of CROSS-MULT is $ac \dot{\leq} bd - 1,\ de \dot{\leq} a,\ c \dot{\geq} 1,\ ce \doteq b \ \vdash$. The problem can be solved by cross-multiplying $de \dot{\leq} a$ and $c \dot{\geq} 1$.

Lemma 5. *When applied to an invalid sequent (containing arbitrary equations and inequalities), Procedure 5 will eventually produce a counterexample.*

7 Related Work

Most similar to our approach is the arithmetic handling in ACL2 [13, 14], which also employs Fourier-Motzkin for linear and cross-multiplication for nonlinear arithmetic. Concerning differences, ACL2 runs arithmetic handling as a purely automated procedure, supports also rationals, does not have separate procedures for equations and does not seem to perform a systematic case analysis.

An method for handling linear equations and inequalities similar to our approach (but lacking counterexample generation) is described in [17] and implemented in the Tecton tool. Related is also [20] about the extension of linear reasoning to nonlinear reasoning.

Higher-order proof assistants usually support integer arithmetic and are so general that arbitrary procedures can be implemented on top of them, often targeting mathematical proofs. In comparison, we tried to develop a simple calculus/procedure specifically for Java verification that works "out of the box" and requires little expertise. The PVS proof assistant [11] can handle linear integer arithmetic and can simplify nonlinear expressions (involving multiplication and division) to some degree, but does (apparently) not go as far as our approach or ACL2. The Coq system [12] implements an incomplete version of the Omega method for deciding Presburger arithmetic (linear integer arithmetic with quantifiers) that essentially boils down to Fourier-Motzkin. Coq can also simplify ring expressions like polynomials. For HOL light [21], a number of tactics and decision procedures for arithmetic have been implemented, including Cooper's method for deciding Presburger arithmetic, handling of congruences and simplification of polynomial expressions.

Linear arithmetic is one of the most important theories supported by SMT solvers (which generally provide incomparably better performance for linear arithmetic than our implementation based on a general theorem prover framework), see [22] for a list. To the best of our knowledge, no SMT solver offers support for nonlinear arithmetic similar to our approach or ACL2. SMT solvers typically use linear programming techniques like Simplex, combined with methods like branch-and-bound or Gomory's cutting planes to realise completeness on the integers.

8 Conclusions and Future Work

We have presented the main components of a proof procedure for linear and nonlinear integer arithmetic, represented as sequent calculus rules together with application strategies. The procedure is completely implemented, and the soundness of the implementation is verified in the prover KeY itself. In addition to the calculus shown here, KeY also supports division and modulo operations and provides further methods like polynomial division. Based on this, we have formalised the Java semantics of integer operations.

For the future, we are considering a more efficient stand-alone implementation of the calculus, possibly based on the DPLL(T) framework. As a more conceptual extension, we plan to combine the calculus with free-variable reasoning for handling quantifiers. The general approach for this is described in [23], but needs to be investigated more carefully. Finally, we would like to add support for bit-wise operations (as they can be found in Java).

Acknowledgements. I want to thank Wolfgang Ahrendt and Richard Bubel for many inspiring discussions and comments on this paper. Thanks are also due to the anonymous referees for helpful comments.

References

1. Beckert, B., Hähnle, R., Schmitt, P.H., eds.: Verification of Object-Oriented Software: The KeY Approach. LNCS 4334. Springer-Verlag (2007)

2. Mostowski, W.: Fully verified JavaCard API reference implementation. In: 4th International Verification Workshop. (2007) To appear.
3. Fitting, M.C.: First-Order Logic and Automated Theorem Proving. 2nd edn. Springer-Verlag, New York (1996)
4. King, J.C.: Symbolic execution and program testing. Communications of the ACM **19** (1976) 385–394
5. Knuth, D.E.: The Art of Computer Programming: Seminumerical Algorithms. Addison-Wesley (1997) Third edition.
6. Buchberger, B.: An algorithmical criterion for the solvability of algebraic systems. Aequationes Mathematicae **4** (1970) 374–383 (German).
7. Buchberger, B.: A critical-pair/completion algorithm for finitely generated ideals in rings. In: Proceedings of the Symposium "Rekursive Kombinatorik" on Logic and Machines: Decision Problems and Complexity, London, UK, Springer-Verlag (1984) 137–161
8. Dershowitz, N.: Termination of rewriting. J. Symb. Comput. **3** (1987) 69–116
9. Dershowitz, N., Manna, Z.: Proving termination with multiset orderings. Commun. ACM **22** (1979) 465–476
10. Schrijver, A.: Theory of Linear and Integer Programming. Wiley (1986)
11. Owre, S., Rajan, S., Rushby, J., Shankar, N., Srivas, M.: PVS: Combining specification, proof checking, and model checking. In Alur, R., Henzinger, T.A., eds.: Proceedings, CAV. Volume 1102 of LNCS., Springer (1996) 411–414
12. Dowek, G., Felty, A., Herbelin, H., Huet, G., Murthy, C., Parent, C., Paulin-Mohring, C., Werner, B.: The Coq proof assistant user's guide. Rapport Techniques 154, INRIA, Rocquencourt, France (1993) Version 5.8.
13. Kaufmann, M., Moore, J.S.: ACL2: An industrial strength version of nqthm. In: Compass'96: Eleventh Annual Conference on Computer Assurance, Gaithersburg, Maryland, National Institute of Standards and Technology (1996)
14. Warren A. Hunt, J., Krug, R.B., Moore, J.S.: Linear and nonlinear arithmetic in ACL2. In Geist, D., Tronci, E., eds.: CHARME. Volume 2860 of Lecture Notes in Computer Science., Springer (2003) 319–333
15. Breunesse, C.B., Jacobs, B., van den Berg, J.: Specifying and verifying a decimal representation in java for smart cards. In: Proceedings of the 9th International Conference on Algebraic Methodology and Software Technology, London, UK, Springer-Verlag (2002) 304–318
16. : Gemplus purse applet. (`http://www.gemplus.com/smart/r_d/publications/case-study/`)
17. Kapur, D., Nie, X.: Reasoning about numbers in tecton. In: International Symposium on Methodologies for Intelligent Systems, Charlotte, North Carolina. (1994)
18. Matijasevic, Y.: Enumerable sets are diophantine (Russian). Dokl. Akad. Nauk SSSR **191** (1970) 279–282 Translation in Soviet Math Doklady, Vol 11, 1970.
19. Warren A. Hunt, J., Krug, R.B., Moore, J.S.: Integrating nonlinear arithmetic into into ACL2. In: Fifth International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2004). (2004)
20. Kapur, D., Cyrluk, D.: Reasoning about nonlinear inequality constraints: a multi-level approach. In: Proceedings of a workshop on Image understanding workshop, San Francisco, CA, USA, Morgan Kaufmann Publishers Inc. (1989) 904–915
21. Harrison, J.: The HOL light manual (1.1) (2000)
22. Ranise, S., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org` (2006)
23. Rümmer, P., Shah, M.A.: Proving programs incorrect using a sequent calculus for Java Dynamic Logic. In: International Conference on Tests And Proofs (TAP). LNCS, Springer (2007) To appear.

A Proofs (-Sketches)

Proof. (Lem. 1) Termination: the termination of SIMP and RED is immediate. We call the left-hand sides x of equations $x \doteq s$ (x a variable) in the antecedent "defined variables," and all other variables "independent variables." When applying RED exhaustively, each defined variable will eventually occur in exactly one place in the sequent (namely, in the defining equation).

For proving termination when COL-RED is added, we show that the leading coefficients $\alpha > 1$ of equations $\alpha x \doteq s$ constantly get smaller. We introduce a well-founded ordering on the set of multisets over $\mathbb{N} \cup \{\infty\}$ by lexicographic comparison: for $a_1 \leq \cdots \leq a_n$, $b_1 \leq \cdots \leq b_m$, we define:

$$\{\{a_1, \ldots, a_n\}\} <_m \{\{b_1, \ldots, b_m\}\} \quad \text{iff} \\ n < m \text{ or } (n = m \text{ and } (a_1, \ldots, a_n) <_{\text{lex}} (b_1, \ldots, b_m))$$

For a sequent and an independent variable x, we then consider the divisors $\gcd(\alpha_1, \ldots, \alpha_n) \in \mathbb{N} \cup \{\infty\}$, where $\alpha_1, \ldots, \alpha_n$ are all coefficients of equations $\alpha_i x \doteq s_i$ in the antecedent (we define $\gcd() = \infty$). The multiset of such gcds for all independent variables gets $<_m$-smaller for each application of COL-RED, and it gets $<_m$-smaller or stays the same when RED is applied (each time potentially followed by an application of SIMP). This proves termination.

Completeness and proof confluence: assume that no further rules can be applied, but the proof branch at hand is not closed. This implies that the coefficient of the left-hand side of all equations is 1 (otherwise, SIMP or COL-RED can be applied), and that no left-hand side term occurs in two places in the sequent (otherwise, RED can be applied). Due to the fact that 0 is the only polynomial whose value is constantly 0 (and correspondingly for tuples of polynomials), there is a countermodel for the equations in the succedent (a valuation of the independent variables). We extend this valuation on the defined variables according to the equations in the antecedent. When investigating RED and COL-RED, it can be seen that this countermodel also is a countermodel of the original sequent.

Proof. (Lem. 2) To see that the application of FM-ELIM terminates, consider the multiset of pairs of inequalities in the antecedent to which FM-ELIM can but has not yet been applied. Pairs of inequalities can be compared lexicographically using $<_r$, and multisets of pairs can be compared using the multiset extension of this ordering. As the multiset gets smaller in this well-founded ordering each time FM-ELIM is applied, termination is guaranteed.

The rule ANTI-SYMM can introduce new equations. Such a new equation is either trivially true and is eliminated, or it is a contradiction and the proof branch is closed, or it reduces the number of independent variables by one. In the last case, Fourier-Motzkin basically has to start over once Procedure 1 has done its job, but this can only happen a finite number of times.