Generic Proof Synthesis for Presburger Arithmetic
Draft

Amine Chaieb and Tobias Nipkow
Institut für Informatik
Technische Universität München

Abstract

We develop in complete detail an extension of Cooper's decision procedure for Presburger arithmetic that returns a proof of the equivalence of the input formula to a quantifier-free formula. For closed input formulae this is a proof of their validity or unsatisfiability. The algorithm is formulated as a functional program that makes only very minimal assumptions w.r.t. the underlying logical system and is therefore easily adaptable to specific theorem provers.

1 Presburger arithmetic

Presburger arithmetic is first-order logic over the integers with $+$ and $<$. Presburger [3] first showed its decidability. We extend Cooper's decision procedure [1] such that a successful run returns a proof of the input formula.

The atomic $\mathcal{PA}$-formulae are defined by *Atom*:

- $\mathit{Atom} ::= T\ \mathcal{R}\ T \mid L\ \mathsf{dvd}\ T$
- $T ::= L \mid V \mid T + T \mid T - T \mid -T \mid L * T$
- $\mathcal{R} ::= < \mid > \mid \leq \mid \geq \mid =$
- $V ::= x \mid y \mid z \mid \ldots$
- $L ::= \ldots \mid -2 \mid -1 \mid 0 \mid 1 \mid 2 \mid \ldots$

We allow the use of $>$, $\leq$, $\geq$ and $=$ in the input language since they can be expressed with formulae based on $<$. We also allow the use of constants and the multiplication with constants since these simulate simply a finite summation. Because '|' is overloaded, we write $d\ \mathsf{dvd}\ t$ to express that d divides t. Throughout the paper a, b, c, d and l denote integer numerals.

2 Notation

2.1 Logic

Terms and formulae follow the usual syntax of predicate calculus. However, there are two levels that we must distinguish. On the programming language level, i.e. the implementation level, the type of formulae is an ordinary first-order recursive datatype. In particular, the formula $\exists x.A$ can be decomposed (e.g. by pattern matching) into the bound variable x and the formula A, which may contain x. On the logic level, we assume that our language allows predicate variables, as is the case in all higher-order systems. On this level $\exists x.A$ is a formula where A does not depend on x, whereas P in $\exists x.P(x)$ is a predicate variable, i.e. a *function* from terms to formulae, which is applied to x, thus expressing the dependence on x. One advantage of the higher-order notation is that substituting x by t is expressed by moving from $P(x)$ to $P(t)$.

In addition to theorems we also have *inference rules* of the form

$$[\![A_1; \ldots; A_n]\!] \Longrightarrow A$$

where the A_i are the premises and A is the conclusion. Logically this is equivalent to $A_1 \to \cdots \to A_n \to A$. The inference rule notation merely increases readability. We assume the following basic functions for manipulating theorems.

If th is the inference rule above, and th_1, ..., th_n are theorems that match the premises $A_1,\ldots,A_n$, then *fwd th* $[th_1, \ldots, th_n]$ is the corresponding instance of A. That is, if $B_1, \ldots, B_n$ are the formulae proved by the theorems $th_1, \ldots, th_n$, and if θ is a most-general unifier of the set of equations $A_1 = B_1, \ldots, A_n = B_n$, then *fwd th* $[th_1, \ldots, th_n]$ yields the theorem $\theta(A)$.

The *free* variables in a theorem th can be instantiated from left to right with terms t_1, ..., t_n by writing $th[t_1, \ldots, t_n]$. For example, if th is the theorem $m \leq m + n{\cdot}n$ then $th[1, 2]$ is the theorem $1 \leq 1 + 2{\cdot}2$.

Function *gen* performs $\forall$-introduction: it takes a variable x and a theorem $P(x)$ and returns the theorem $\forall x.P(x)$.

We assume that the underlying theorem prover provides a function *prove* from formulae to theorems which must be able to prove theorems of the following form: $0 \neq d$ for any non-zero integer numeral d; $0 < d$ for any positive integer numeral d; $d \in B$ for any integer numeral d and any finite set of B of integer numerals; d dvd D where d and D are integer numerals.

2.2 Programming language

All algorithms are expressed in generic functional programming notation. We assume the following special features. Lambda-abstraction uses a bold $\boldsymbol{\lambda}$ (in contrast to the ordinary λ on the logic level) and permits pattern-matching (where most uses are of the form $\boldsymbol{\lambda}[].t$, where [] is the empty list).

P	$\mathcal{A}(P)$	$\mathcal{B}(P)$	$P_{-\infty}$	$P_{+\infty}$
$A \wedge B$	$\mathcal{A}(A) \cup \mathcal{A}(B)$	$\mathcal{B}(A) \cup \mathcal{B}(B)$	$A_{-\infty} \wedge B_{-\infty}$	$A_{+\infty} \wedge B_{+\infty}$
$A \vee B$	$\mathcal{A}(A) \cup \mathcal{A}(B)$	$\mathcal{B}(A) \cup \mathcal{B}(B)$	$A_{-\infty} \vee B_{-\infty}$	$A_{+\infty} \vee B_{+\infty}$
$0 < x + a$	$\emptyset$	$\{-a\}$	$\bot$	$\top$
$0 < -x + a$	$\{a\}$	$\emptyset$	$\top$	$\bot$
$0 = x + a$	$\{1 - a\}$	$\{-1 - a\}$	$\bot$	$\top$
$\neg(0 = x + a)$	$\{-a\}$	$\{-a\}$	$\neg\bot$	$\neg\top$
_	$\emptyset$	$\emptyset$	P	P

Figure 1: Definition of $\mathcal{A}(P)$, $\mathcal{B}(P)$, $P_{-\infty}$ and $P_{+\infty}$

Because formulae (a concrete recursive type) and theorems (some abstract type) are quite distinct, we need a way to refer to the formula proved by some theorem. This is done by pattern matching: a theorem can be matched against the pattern th **as** 'f', where th is a theorem variable and f a formula pattern, thus binding the formula variables in f. For example, matching the theorem $0 = 0 \wedge 1 = 1$ against the pattern th **as** '$A \wedge B$' binds th to the given theorem, A to the term $0 = 0$ and B to the term $1 = 1$.

3 Cooper's algorithm

The input to Cooper's algorithm is a formula $\exists x.P$, where P is a quantifier-free $\mathcal{PA}$-formula. The algorithm consists in the following steps.

Normalization N: Put the formula in negation normal form (NNF).

L: Replace negated inequalities $\neg(s < t)$ by $0 < s - t + 1$ and then transform each atomic formulae A to have the form

$$d \mathrel{\mathcal{R}} l_1 \cdot x_1 + \cdots + l_n \cdot x_n + c$$

where

- $x_i = x_j$ iff $i = j$
- $x_1 = x$ iff x occurs in A
- $d = 0$ if $\mathcal{R} \neq \mathsf{dvd}$
- $d > 0$ if $\mathcal{R} = \mathsf{dvd}$

U: After calculating $l = lcm\{c \mid c \cdot x \text{ occurs in an atomic formula}\}$ generate an equivalent formula where all atoms have been multiplied by an appropriate constant such that the coefficient of the bound variable x is l or $-l$ everywhere. Using (4) these coefficients can be replaced by -1 or 1. Furthermore, the coefficients of x inside atomic formulae involving dvd and $=$ can be set to 1 since following theorems hold:

$$\begin{aligned} 0 = -1 \cdot x + t &\leftrightarrow 0 = 1 \cdot x + (-1 \cdot t) \\ d \mathsf{\ dvd\ } -1 \cdot x + t &\leftrightarrow d \mathsf{\ dvd\ } 1 \cdot x + (-1 \cdot t) \end{aligned}$$

Calculation From the normalized formula P calculate

$$\delta = lcm\{d \mid d \text{ dvd } t \text{ occurs in } P \wedge x \text{ occurs in } t\} \tag{1}$$

and $\mathcal{A}(P)$, $\mathcal{B}(P)$, $P_{-\infty}$ and $P_{+\infty}$ as defined in Fig. 1. Note that the definition implicitly depends on the bound variable x: the final line applies in case the subformula under consideration contains dvd or does not contain x.

Result Apply either (2) or (3) in Cooper's Theorem. The right-hand side is quantifier-free. The choice of which of the two equivalences to apply is normally determined by the relative size of $\mathcal{A}$ and $\mathcal{B}$.

Theorem 1 (Cooper [1]) *If P is a normalized Presburger formula then*

$$(\exists x.P(x)) \;\leftrightarrow\; \bigvee_{j=1}^{\delta} P_{-\infty}(j) \vee \bigvee_{j=1}^{\delta} \bigvee_{b \in \mathcal{B}(P(x))} P(b+j) \tag{2}$$

$$(\exists x.P(x)) \;\leftrightarrow\; \bigvee_{j=1}^{\delta} P_{+\infty}(j) \vee \bigvee_{j=1}^{\delta} \bigvee_{a \in \mathcal{A}(P(x))} P(a-j) \tag{3}$$

4 Theorem extraction model

Many of our proofs are performed by the following generic function:

thm decomp t =
let
  (*ts*,*recomb*) = *decomp t*
in *recomb* (*map* (*thm decomp*) *ts*)

It takes a problem decomposition function of type $\alpha \to \alpha\ list \times (\beta\ list \to \beta)$ and a problem t of type α, decomposes t into a list of subproblems ts and a recombination function *recomb*, solves the subproblems recursively, and combines their solution into an overall solution.

In our applications, problems are formulae to be proved, solutions are theorems, and termination will be guaranteed because all decompositions yield syntactically smaller terms.

As an example we look at a generic function for quantifier elimination. More precisely, we present a function *qelim* which eliminates all quantifiers from a first-order formula provided it is given a function *qe* which can eliminate a single existential quantifier. That is, if *qe* applied to a formula $\exists x.P$, where P is quantifier free, yields a theorem $(\exists x.P) \leftrightarrow Q$, where Q is quantifier free, then *qelim* applied to any first-order formula A yields a theorem $A \leftrightarrow B$, where B is quantifier free.

decomp_qe qe P =

case P **of**
  $A \wedge B \Rightarrow ([A, B], \mathit{fwd}\ \mathit{cong}_\wedge)$
  $A \vee B \Rightarrow ([A, B], \mathit{fwd}\ \mathit{cong}_\vee)$
  $A \rightarrow B \Rightarrow ([A, B], \mathit{fwd}\ \mathit{cong}_\rightarrow)$
  $A \leftrightarrow B \Rightarrow ([A, B], \mathit{fwd}\ \mathit{cong}_\leftrightarrow)$
  $\neg A \Rightarrow ([A], \mathit{fwd}\ \mathit{cong}_\neg)$
  $\exists x.A \Rightarrow ([A], \boldsymbol{\lambda}[th\ \textbf{as}\ \text{`\_} \leftrightarrow B\text{'}].\ \textbf{let}\ \mathit{lift} = \mathit{fwd}\ \mathit{cong}_\exists\ [\mathit{gen}\ x\ \mathit{th}]$
    $\textbf{in}\ \mathit{fwd}\ \mathit{trans}\ [\mathit{lift}, \mathit{qe}(\exists x.B)])$
  $\forall x.A \Rightarrow ([\exists x.\neg A], \mathit{fwd}\ \mathit{qe}_\forall)$
  $\_ \Rightarrow ([], \boldsymbol{\lambda}[].\ \mathit{refl}[P])$

$\mathit{qelim}\ \mathit{qe} = \mathit{thm}\ (\mathit{decomp\_qe}\ \mathit{qe})$

Elimination proceeds from the innersmost quantifier to the outermost one. The function is theory independent. It uses a number of predicate calculus tautologies which can be found in the appendix. Note that *decomp_qe qe* terminates altough $\exists x.\neg A$ is not syntactically smaller than $\forall x.A$: simply unfold the definition of *decomp_qe qe* twice in that branch to reach a syntactically smaller term.

5 Proof synthesis for Presburger arithmetic

Presburger arithmetic can be implemented by quantifier elimination applied to Cooper's algorithm:

$$\mathit{presburger} = \mathit{qelim}\ \mathit{cooper}$$

We will now implement Cooper's algorithm as in §3.

$\mathit{cooper}\ x\ P =$
let
  nlthm **as** $\text{`\_} \leftrightarrow N\text{'} = \mathit{nlstep}\ x\ P$ $\{P(x) \leftrightarrow N(x)\}$
  $\mathit{enlthm} = \mathit{fwd}\ \mathit{cong}_\exists\ (\mathit{gen}\ x\ \mathit{nlthm})$ $\{(\exists x.P(x)) \leftrightarrow (\exists x.N(x))\}$
  uthm **as** $\text{`\_} \leftrightarrow (\exists x.U)\text{'} = \mathit{ustep}\ x\ N$ $\{(\exists x.N(x)) \leftrightarrow (\exists x.U(x))\}$
  $\mathit{normthm} = \mathit{fwd}\ \mathit{trans}\ [\mathit{enlthm}, \mathit{uthm}]$ $\{(\exists x.P(x)) \leftrightarrow (\exists x.U(x))\}$
  $\mathit{cpthm} = \mathit{cooper\_thm}\ x\ U$ $\{(\exists x.U(x)) \leftrightarrow Q\}$
in $\mathit{fwd}\ \mathit{trans}\ [\mathit{normthm}, \mathit{cpth}]$ $\{(\exists x.P(x)) \leftrightarrow Q\}$

The intermediate theorems are annotated with comments that show the formula that they prove. This is in contrast to the pattern-matching (**as**) notation in the programming language where P etc are first-order variables. In the following subsections we discuss the implementations of the individual steps.

Note that the Q in the equivalence $(\exists x.P(x)) \leftrightarrow Q$ may but need not be $True$ or $False$. If Q is closed, *cooper_thm* below will ensure that it is in fact $True$ or $False$. Otherwise the truth value of Q may depend on its free variables.

5.1 Proving the N and L-step

We obtain NNF via our workhorse *thm*; alternative implementation techniques include rewriting with DeMorgan rules.

decomp_nnf lf P =
case P **of**
$A \wedge B \Rightarrow ([A, B],\ fwd\ cong_{\wedge})$
$A \vee B \Rightarrow ([A, B],\ fwd\ cong_{\vee})$
$A \rightarrow B \Rightarrow ([A, B],\ fwd\ nnf_{\rightarrow})$
$A \leftrightarrow B \Rightarrow ([A, B],\ fwd\ nnf_{\leftrightarrow})$
$\neg\neg p \Rightarrow ([p],\ fwd\ nnf_{\neg\neg})$
$\neg(A \wedge B) \Rightarrow ([A, B],\ fwd\ nnf_{\neg\wedge})$
$\neg(A \vee B) \Rightarrow ([A, B],\ fwd\ nnf_{\neg\vee})$
$\neg(A \rightarrow B) \Rightarrow ([A, B],\ fwd\ nnf_{\neg\rightarrow})$
$\neg(A \leftrightarrow B) \Rightarrow ([A, B],\ fwd\ nnf_{\neg\leftrightarrow})$
$\_ \Rightarrow ([], \boldsymbol{\lambda}[].\ lf\ P)$

nlstep x P = *thm* (*decomp_nnf* (*proveL x*)) *P*

Atomic formulae are immediately transformed via function *proveL* which takes a variable x and an atomic formula A and returns the theorem $A \leftrightarrow A'$ where A' is the result of performing the L-step (see §3) on A. This requires the manipulation of individual (in)equalities between linear terms, essentially just permuting subterms. How this is best handled depends on the infrastructure of the underlying theorem prover: rewriting or quantifier-free linear arithmetic are possible implementation tools. Hence we refrain from giving a generic solution for the L-step prover.

5.2 Proving the U-step

The U-step takes a variable x and a formula P and first proves $P \leftrightarrow P'$, where P' is the formula obtained from P by multiplying each atomic formula with some constant such that the coefficient of x becomes $l = lcm\{c \mid c{\cdot}x \text{ occurs in } P\}$ or $-l$.

Then we instantiate the generic theorem *unitycoeff*

$$(\exists x.\ A(l{\cdot}x)) \leftrightarrow (\exists x.\ l \ \mathsf{dvd}\ x \wedge A(x)) \tag{4}$$

Now we give a function that returns the proof of adjusting the coefficient of x in P to l or $-l$.

decomp_ac x l P =
case P **of**
$a\ R\ c{\cdot}x + t \Rightarrow$
let
$m = \lfloor \frac{l}{c} \rfloor$
$k =$ **if** $R =$ '$<$' **then** $|m|$ **else** m

$x' = (\lfloor \frac{m}{k} \rfloor \cdot l) \cdot x$
in
case R **of**
'$<$' $\Rightarrow$
let
$pre = prove(0 < k)$
th **as** '$A \leftrightarrow B$' $= fwd\ (ac_<[k, a, c, x, t])\ pre$
in $([], \boldsymbol{\lambda}[].fwd\ trans\ [th, proveL\ x\ B])$
'$=$' $\Rightarrow$
let
$pre = prove(0 \neq k)$
th **as** '$A \leftrightarrow B$' $= fwd\ (ac_=[k, a, c, x, t])\ pre$
in $([], \boldsymbol{\lambda}[].fwd\ trans\ [th, proveL\ x\ B])$
'dvd' $\Rightarrow$
let
$pre = prove(0 \neq k)$
th **as** '$A \leftrightarrow B$' $= fwd\ (ac_{\mathsf{dvd}}[k, a, c, x, t])\ pre$
in $([], \boldsymbol{\lambda}[].fwd\ trans\ [th, proveL\ x\ B])$
$\neg A \Rightarrow ([A], fwd\ cong_\neg)$
$A \wedge B \Rightarrow ([A, B], fwd\ cong_\wedge)$
$A \vee B \Rightarrow ([A, B], fwd\ cong_\vee)$
$\_ \Rightarrow ([], \boldsymbol{\lambda}[].\ refl[P])$

$ustep\ x\ P =$
let
$l = termlcm\ x\ P$
$acth$ **as** '$\_ \leftrightarrow Q(l \cdot x)$' $= thm\ (decomp\_ac\ x\ l)\ P$
in $fwd\ trans\ [fwd\ cong_\exists\ [gen\ x\ acth], unitycoeff\ [Q, l]]$

Note that the right-hand sides of $ac_<$, $ac_=$ and ac_{dvd} are not of the normal form established by the L-step, which is why their use is followed by an application of *proveL* to re-establish that normal form.

In the penultimate line of *ustep* we cheat a bit to avoid excessive technicalities. For a start, we assume that on the right-hand side of the '$\leftrightarrow$' all occurrences of the quantified variable are either $l \cdot x$ or $-1 \cdot (l \cdot x)$ — the negative form needs to be created explicitly from $(-l) \cdot x$. As a result, x is indeed multiplied by l everywhere. The second cheat is that we have taken the liberty to employ a simple form of higher-order matching: matching the pattern $Q(l \cdot x)$, where x is considered bound, against a formula f succeeds iff all occurrences of x in f are of the form $l \cdot x$, in which case function Q is the result of λ-abstracting over all the occurrences of $l \cdot x$. Although on the programming language level formulae are a first-order data type and do not directly support higher-order matching, this simple instance of it is readily implemented.

The auxiliary function *termlcm* computes the *lcm* of all coefficients of x in P. Its implementation is trivial and thus not shown.

5.3 Cooper's Theorem

The main complication in the proof synthesis for Cooper's algorithm is the proof of Cooper's theorem itself. Based on the work Michael Norrish[2], the task can be simplified by generalizing the theorem from the specific δ, $P_{-\infty}$ and $\mathcal{B}(P)$ to arbitrary ones subject to certain assumptions:

$$\begin{aligned} \mathit{cooper}_{-\infty}: \quad & [\![\ 0 < \delta; \\ & \exists z.\forall x.x < z \to P(x) \leftrightarrow P_{-\infty}(x); \\ & \forall x.\neg(\exists j \in [\delta].\exists b \in \mathcal{B}.P(b+j)) \to P(x) \to P(x-\delta); \\ & \forall x,k.P_{-\infty}(x) \leftrightarrow P_{-\infty}(x-k{\cdot}\delta)\ ]\!] \\ & \Longrightarrow (\exists x.P(x)) \leftrightarrow (\exists j \in [\delta].P_{-\infty}(j) \vee \exists j \in [\delta].\exists b \in \mathcal{B}.P(b+j)) \end{aligned}$$

Note that we have replaced the indexed disjunctions by existential quantifiers. The notation $[\delta]$ is short for the set $\{1,\ldots,\delta\}$. Function *bset* implements $\mathcal{B}$ in Fig. 1 and function *delta* computes δ as in (1). Their implementation is trival because deduction-free. Hence it is not shown.

We present only the $-\infty$ variant of the theorem and the algorithm since the $+\infty$ one is analogous.

Function *cooper_thm* is essentially an application of $\mathit{cooper}_{-\infty}$:

$\mathit{cooper\_thm}\ x\ U =$
let
  $B = \mathit{bset}\ x\ U$
  $D = \mathit{delta}\ x\ U$
  $\mathit{prem1} = \mathit{prove}(0 < D)$
  $\mathit{prem2} = \mathit{iff\_minf}\ x\ U \qquad \{\exists z.\forall x.\ x < z \to (U(x) \leftrightarrow U_{-\infty}(x))\}$
  $\mathit{prem3} = \mathit{iff\_modd}\ x\ D\ U \qquad \{\forall x,k.\ U_{-\infty}(x) \leftrightarrow U_{-\infty}(x - k \cdot D)\}$
  $\mathit{prem4} = \mathit{notB}\ x\ D\ B\ U$
    $\{\neg(\exists j \in [D], b \in B.U(b+j)) \to U(x) \to U(x-D)\}$
  $\mathit{cpthm} = \mathit{fwd}\ \mathit{cooper}_{-\infty}\ [\mathit{prem1},\mathit{prem2},\mathit{prem3},\mathit{prem4}]$
in $\mathit{expand}_{\exists}\ \mathit{cpthm}$

At the end we apply a function $\mathit{expand}_{\exists}$ which takes some theorem and returns an equivalent one where all occurrences of $\exists x \in I$, where I is some finite set, have been expanded into finite disjunctions. Typically this is performed by rewriting and we do not discuss the details. We assume that at the same time rewriting also evaluates all ground arithmetic and logical expressions. This means that if $U(x)$ is closed (except for x), then the whole right-hand side of *cpthm* is ground and can be rewritten to either $True$ or $False$.

Premise $0 < \delta$ is proved directly by *prove*. The other premises can once again be proved via *thm*, as we will see now.

5.4 Proving the premises of Cooper's Theorem

$$\exists z.\forall x.x < z \rightarrow (P(x) \leftrightarrow P_{-\infty}(x))$$

is synthesized by

decomp_iff_minf x F =
case F **of**
$A \wedge B \Rightarrow ([A, B], \mathit{fwd}\ \mathit{iff}_{-\infty\wedge})$
$A \vee B \Rightarrow ([A, B], \mathit{fwd}\ \mathit{iff}_{-\infty\vee})$
$0 < 1{\cdot}y + s \mid y = x \Rightarrow ([], \boldsymbol{\lambda}[].\ \mathit{iff}_{-\infty<}[s])$
$0 < -1{\cdot}y + s \mid y = x \Rightarrow ([], \boldsymbol{\lambda}[].\ \mathit{iff}_{-\infty<-}[s])$
$0 = 1{\cdot}y + s \mid y = x \Rightarrow ([], \boldsymbol{\lambda}[].\ \mathit{iff}_{-\infty=}[s])$
$\neg(0 = 1{\cdot}y + s) \mid y = x \Rightarrow ([], \boldsymbol{\lambda}[].\ \mathit{iff}_{-\infty\neq}[s])$
$d\ \mathsf{dvd}\ 1{\cdot}y + s \mid y = x \Rightarrow ([], \boldsymbol{\lambda}[].\ \mathit{iff}_{-\infty\mathsf{dvd}}[s])$
$\neg(d\ \mathsf{dvd}\ 1{\cdot}y + s) \mid y = x \Rightarrow ([], \boldsymbol{\lambda}[].\ \mathit{iff}_{-\infty\neg\mathsf{dvd}}[s])$
$\_ \Rightarrow ([], \boldsymbol{\lambda}[].\ \mathit{iff}_{-\infty}[\lambda x.F])$

iff_minf x F = *thm* (*decomp_iff_minf* x) F

Syntax: we employ Haskell's guarded patterns: $0 < 1{\cdot}y+s \mid y = x$ represents the pattern $0 < 1{\cdot}y + s$ guarded by the condition $y = x$.

$$\forall x, k.P_{-\infty}(x) \leftrightarrow P_{-\infty}(x - k{\cdot}\delta)$$

is synthesized by

decomp_iff_modd x D F =
case F **of**
$A \wedge B \Rightarrow ([A, B], \mathit{fwd}\ \mathit{iff}_{-\delta\wedge})$
$A \vee B \Rightarrow ([A, B], \mathit{fwd}\ \mathit{iff}_{-\delta\vee})$
$0 < 1{\cdot}x + t \Rightarrow ([], \boldsymbol{\lambda}[].\ \mathit{iff}_{-\delta}[False])$
$0 < -1{\cdot}x + t \Rightarrow ([], \boldsymbol{\lambda}[].\ \mathit{iff}_{-\delta}[True])$
$\neg(0 = 1{\cdot}x + t) \Rightarrow ([], \boldsymbol{\lambda}[].\ \mathit{iff}_{-\delta}[True])$
$0 = 1{\cdot}x + t \Rightarrow ([], \boldsymbol{\lambda}[].\ \mathit{iff}_{-\delta}[False])$
$d\ \mathsf{dvd}\ 1{\cdot}x + t \Rightarrow ([], \boldsymbol{\lambda}[].\ \mathit{iff}_{-\delta\mathsf{dvd}}[d, D, t])$
$\neg(d\ \mathsf{dvd}\ 1{\cdot}x + t) \Rightarrow ([], \boldsymbol{\lambda}[].\ \mathit{iff}_{-\delta\neg\mathsf{dvd}}[d, D, t])$
$\_ \Rightarrow ([], \boldsymbol{\lambda}[].\ \mathit{iff}_{-\delta}[F])$;

iff_modd x D F = *thm* (*decomp_iff_modd* x D) F

$$\forall x.\neg(\exists j \in [\delta].\exists b \in \mathcal{B}.P(b + j)) \rightarrow P(x) \rightarrow P(x - \delta)$$

is synthesized by

decomp_notB x D B P F =
case F **of**
$A \wedge B \Rightarrow ([A, B], \mathit{fwd}\ \mathit{notB}_{\wedge})$
$A \vee B \Rightarrow ([A, B], \mathit{fwd}\ \mathit{notB}_{\vee})$

$0 < 1 \cdot y + s \mid y = x \Rightarrow ([], \boldsymbol{\lambda}[].\ fwd\ (notB_{>}[s, B, P, D])\ [prove(-s \in B)])$
$0 < -1 \cdot y + s \mid y = x \Rightarrow ([], \boldsymbol{\lambda}[].\ fwd\ (notB_{<}[D, P, B, s])\ [prove(0 < D)])$
$0 = 1 \cdot y + s \mid y = x \Rightarrow$
$\quad ([], \boldsymbol{\lambda}[].\ fwd\ (notB_{=}[D, s, B, P])\ [prove(0 < D), prove(-s - 1 \in B)])$
$\neg(0 = 1 \cdot y + s) \mid y = x \Rightarrow$
$\quad ([], \boldsymbol{\lambda}[].\ fwd\ (notB_{\neq}[D, s, B, P])\ [prove(0 < D), prove(-s \in B)])$
$d|1 \cdot y + s \mid y = x \Rightarrow ([], \boldsymbol{\lambda}[].\ fwd\ (notB_{\mathsf{dvd}}[d, D, P, B, s])\ [prove(d\ \mathsf{dvd}\ D)])$
$\neg(d\ \mathsf{dvd}\ 1 \cdot y + s) \mid y = x \Rightarrow$
$\quad ([], \boldsymbol{\lambda}[].\ fwd\ (notB_{\neg\mathsf{dvd}}[d, D, P, B, s])\ [prove(d\ \mathsf{dvd}\ D)])$
$\_ \Rightarrow ([], \boldsymbol{\lambda}[].\ notB_{id}[P, D, B, F])$

$notB\ x\ D\ B\ P\ = fwd\ notB_E\ [thm\ (decomp\_notB\ x\ D\ B\ (\lambda x.P))\ P]$

References

[1] D.C. Cooper. Theorem proving in arithmetic without multiplication. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 7, pages 91–100. Edinburgh University Press, 1972.

[2] Michael Norrish. Complete integer decision procedures as derived rules in HOL. In D.A. Basin and B. Wolff, editors, *Theorem Proving in Higher Order Logics, TPHOLs 2003*, volume 2758 of *Lect. Notes in Comp. Sci.*, pages 71–86. Springer-Verlag, 2003.

[3] Mojzesz Presburger. Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In *Comptes Rendus du I Congrès de Mathématiciens des Pays Slaves*, pages 92–101, 1929.

A Some Theorems

refl: $P \leftrightarrow P$

trans: $[\![P \leftrightarrow Q; Q \leftrightarrow R]\!] \Longrightarrow P \leftrightarrow R$

$notB_{\wedge}$:
$[\![\forall x.Q(x) \wedge \neg(\exists j \in [\delta].\exists b \in B.Q(b+j)) \rightarrow P_1(x) \rightarrow P_1(x-\delta);$
$\forall x.Q(x) \wedge \neg(\exists j \in [\delta].\exists b \in B.Q(b+j)) \rightarrow P_2(x) \rightarrow P_2(x-\delta)]\!]$
$\Longrightarrow \forall x.Q(x) \wedge \neg(\exists j \in [\delta].\exists b \in B.Q(b+j)) \rightarrow (P_1(x) \wedge P_2(x))$
$\rightarrow (P_1(x-\delta) \wedge P_2(x-\delta))$

$notB_{\vee}$:
$[\![\forall x.Q(x) \wedge \neg(\exists j \in [\delta].\exists b \in B.Q(b+j)) \rightarrow P_1(x) \rightarrow P_1(x-\delta);$
$\forall x.Q(x) \wedge \neg(\exists j \in [\delta].\exists b \in B.Q(b+j)) \rightarrow P_2(x) \rightarrow P_2(x-\delta)]\!]$

$\Longrightarrow (\forall x.Q(x) \wedge \neg(\exists j \in [\delta].\exists b \in B.Q(b+j)) \rightarrow (P_1(x) \vee P_2(x))$
$\rightarrow (P_1(x-\delta) \vee P_2(x-\delta)))$

$notB_E$:
$\forall x.P(x) \wedge \neg(\exists j \in [\delta].\exists b \in B.P(b+j)) \rightarrow P(x) \rightarrow P(x-\delta)$
$\Longrightarrow \forall x.\neg(\exists j \in [\delta].\exists b \in B.P(b+j)) \rightarrow P(x) \rightarrow P(x-\delta)$

$iff_{-\infty\wedge}$:
$[\![\exists z_1.\forall x.x < z_1 \rightarrow (P_1(x) \leftrightarrow P_2(x));$
$\exists z_2.\forall x.x < z_2 \rightarrow (Q_1(x) \leftrightarrow Q_2(x))]\!]$
$\Longrightarrow \exists z.\forall x.x < z \rightarrow ((P_1(x) \wedge Q_1(x)) \leftrightarrow (P_2(x) \wedge Q_2(x)))$

$iff_{-\infty\vee}$:
$[\![\exists z_1.\forall x.x < z_1 \rightarrow (P_1(x) \leftrightarrow P_2(x));$
$\exists z_2.\forall x.x < z_2 \rightarrow (Q_1(x) \leftrightarrow Q_2(x))]\!]$
$\Longrightarrow \exists z.\forall x.x < z \rightarrow ((P_1(x) \vee Q_1(x)) \leftrightarrow (P_2(x) \vee Q_2(x)))$

$iff_{-\infty}$: $\exists z.\forall x.x < z \rightarrow (P(x) \leftrightarrow P(x))$

$iff_{-\infty=}$: $\exists z.\forall x.x < z \rightarrow ((0 = x + t \leftrightarrow False)$

$iff_{-\infty\neq}$: $\exists z.\forall x.x < z \rightarrow (\neg(0 = x + t) \leftrightarrow True)$

$iff_{-\infty<}$: $\exists z.\forall x.x < z \rightarrow (0 < 1{\cdot}x + t \leftrightarrow False)$

$iff_{-\infty<-}$: $\exists z.\forall x.x < z \rightarrow (0 < -1{\cdot}x + t \leftrightarrow True)$

$iff_{-\infty\mathsf{dvd}}$: $\exists z.\forall x.x < z \rightarrow ((d\ \mathsf{dvd}\ x + t) \leftrightarrow (d\ \mathsf{dvd}\ x + t))$

$iff_{-\infty\neg\mathsf{dvd}}$: $\exists z.\forall x.x < z \rightarrow (\neg(d\ \mathsf{dvd}\ x + t) \leftrightarrow \neg(d\ \mathsf{dvd}\ x + t))$

ac_{dvd}: $0 \neq k \Longrightarrow (m\ \mathsf{dvd}\ c{\cdot}n + t) \leftrightarrow (k{\cdot}m\ \mathsf{dvd}\ (k{\cdot}c){\cdot}n + k{\cdot}t)$

$ac_{<}$: $0 < k \Longrightarrow (m < c{\cdot}n + t) \leftrightarrow (k{\cdot}m < (k{\cdot}c){\cdot}n + k{\cdot}t)$

$ac_{=}$: $0 \neq k \Longrightarrow (m = c{\cdot}n + t) \leftrightarrow (k{\cdot}m = (k{\cdot}c){\cdot}n + k{\cdot}t)$

$iff_{-\delta\wedge}$: $[\![\forall x,k.P(x) \leftrightarrow P(x-k{\cdot}\delta);\ \forall x,k.Q(x) \leftrightarrow Q(x-k{\cdot}\delta)]\!]$
$\Longrightarrow \forall x,k.(P(x) \wedge Q(x)) \leftrightarrow (P(x-k{\cdot}\delta) \wedge Q(x-k{\cdot}\delta))$

$iff_{-\delta\vee}$: $[\![\forall x,k.P(x) \leftrightarrow P(x-k{\cdot}\delta);\ \forall x,k.Q(x) \leftrightarrow Q(x-k{\cdot}\delta)]\!]$
$\Longrightarrow \forall x,k.(P(x) \vee Q(x)) \leftrightarrow (P(x-k{\cdot}\delta) \vee Q(x-k{\cdot}\delta))$

$iff_{-\delta}$: $\forall x,k.\ P \leftrightarrow P$

$iff_{-\delta\mathsf{dvd}}$: $d\ \mathsf{dvd}\ \delta \Longrightarrow \forall x,k.\ (d\ \mathsf{dvd}\ x + t) \leftrightarrow (d\ \mathsf{dvd}\ x - k{\cdot}\delta + t)$

$iff_{-\delta\neg\mathsf{dvd}}$: $d\ \mathsf{dvd}\ \delta \Longrightarrow \forall x,k.\ \neg(d\ \mathsf{dvd}\ x + t) \leftrightarrow \neg(d\ \mathsf{dvd}\ x - k{\cdot}\delta + t)$

$notB_{id}$: $\forall x.Q(x) \wedge \neg(\exists j \in [\delta].\exists b \in B.Q(b+j)) \rightarrow F \rightarrow F$

$notB_<$: $0 < \delta \Longrightarrow \forall x.Q(x) \wedge \neg(\exists j \in [\delta].\exists b \in B.Q(b+j)) \rightarrow$
$(0 < -x + a) \rightarrow (0 < -(x - \delta) + a)$

$notB_>$: $-a \in B \Longrightarrow \forall x.Q(x) \wedge \neg(\exists j \in [\delta].\exists b \in B.Q(b+j))$
$\rightarrow (0 < x + a) \rightarrow (0 < (x - \delta) + a)$

$notB_=$: $[\![0 < \delta; -a - 1 \in B]\!] \Longrightarrow \forall x.Q(x) \wedge \neg(\exists j \in [\delta].\exists b \in B.Q(b+j)) \rightarrow$
$(0 = x + a) \rightarrow (0 = (x - \delta) + a)$

$notB_\neq$: $[\![0 < \delta; -a \in B]\!] \Longrightarrow \forall x.Q(x) \wedge \neg(\exists j \in [\delta].\exists b \in B.Q(b+j)) \rightarrow$
$\neg(0 = x + a) \rightarrow \neg(0 = (x - \delta) + a)$

$notB_{\mathsf{dvd}}$: d dvd $\delta \Longrightarrow \forall x.Q(x) \wedge \neg(\exists j \in [\delta].\exists b \in B.Q(b+j)) \rightarrow d$ dvd
$x + a \rightarrow d$ dvd $(x - \delta) + a$

$notB_{\neg\mathsf{dvd}}$: d dvd $\delta \Longrightarrow \forall x.Q(x) \wedge \neg(\exists j \in [\delta].\exists b \in B.Q(b+j)) \rightarrow \neg(d$ dvd
$x + a) \rightarrow \neg(d$ dvd $(x - \delta) + a)$

$cong_\wedge$: $[\![P_1 \leftrightarrow P_2; Q_1 \leftrightarrow Q_2]\!] \Longrightarrow (P_1 \wedge Q_1) \leftrightarrow (P_2 \wedge Q_2)$

$cong_\vee$: $[\![P_1 \leftrightarrow P_2; Q_1 \leftrightarrow Q_2]\!] \Longrightarrow (P_1 \vee Q_1) \leftrightarrow (P_2 \vee Q_2)$

$cong_\rightarrow$: $[\![P_1 \leftrightarrow P_2; Q_1 \leftrightarrow Q_2]\!] \Longrightarrow (P_1 \rightarrow Q_1) \leftrightarrow (P_2 \rightarrow Q_2)$

$cong_\leftrightarrow$: $[\![P_1 \leftrightarrow P_2; Q_1 \leftrightarrow Q_2]\!] \Longrightarrow (P_1 \leftrightarrow Q_1) \leftrightarrow (P_2 \leftrightarrow Q_2)$

$cong_\neg$: $P \leftrightarrow Q \Longrightarrow \neg P \leftrightarrow \neg Q$

$cong_\exists$: $\forall x.\ P(x) \leftrightarrow Q(x) \Longrightarrow (\exists x.P(x)) \leftrightarrow (\exists x.Q(x))$

$qe_\forall$: $(\exists x.\neg P(x)) \leftrightarrow R \Longrightarrow (\forall x.P(x)) \leftrightarrow \neg R$

$nnf_\rightarrow$: $[\![\neg P \leftrightarrow P_1; Q \leftrightarrow Q_1]\!] \Longrightarrow ((P \rightarrow Q) \leftrightarrow (P_1 \vee Q_1))$

$nnf_\leftrightarrow$: $[\![(P \wedge Q) \leftrightarrow (P_1 \wedge Q_1); (\neg P \wedge \neg Q) \leftrightarrow (P_2 \wedge Q_2)]\!]$
$\Longrightarrow (P \leftrightarrow Q) \leftrightarrow (P_1 \wedge Q_1) \vee (P_2 \wedge Q_2)$

$nnf_{\neg\neg}$: $P \leftrightarrow Q \Longrightarrow \neg\neg P \leftrightarrow Q$

$nnf_{\neg\wedge}$: $[\![\neg P \leftrightarrow P_1; \neg Q \leftrightarrow Q_1]\!] \Longrightarrow \neg(P \wedge Q) \leftrightarrow (P_1 \vee Q_1)$

$nnf_{\neg\vee}$: $[\![\neg P \leftrightarrow P_1; \neg Q \leftrightarrow Q_1]\!] \Longrightarrow \neg(P \vee Q) \leftrightarrow (P_1 \wedge Q_1)$

$nnf_{\neg\rightarrow}$: $[\![P \leftrightarrow P_1; \neg Q \leftrightarrow Q_1]\!] \Longrightarrow \neg(P \rightarrow Q) \leftrightarrow (P_1 \wedge Q_1)$

$nnf_{\neg\leftrightarrow}$: $[\![(P \wedge \neg Q) \leftrightarrow (P_1 \wedge Q_1); (\neg P \wedge Q) \leftrightarrow (P_2 \wedge Q_2)]\!]$
$\Longrightarrow \neg(P \leftrightarrow Q) \leftrightarrow ((P_1 \wedge Q_1) \vee (P_2 \wedge Q_2))$