A Formally Verified Solver for Homogeneous Linear Diophantine Equations*

Florian Meßner, Julian Parsert, Jonas Schöpf, and Christian Sternagel

University of Innsbruck, Austria
florian.g.messner@uibk.ac.at julian.parsert@uibk.ac.at
jonas.schoepf@uibk.ac.at christian.sternagel@uibk.ac.at

Abstract. In this work we are interested in minimal complete sets of solutions for homogeneous linear diophantine equations. Such equations naturally arise during AC-unification—that is, unification in the presence of associative and commutative symbols. Minimal complete sets of solutions are for example required to compute AC-critical pairs. We present a verified solver for homogeneous linear diophantine equations that we formalized in Isabelle/HOL. Our work provides the basis for formalizing AC-unification and will eventually enable the certification of automated AC-confluence and AC-completion tools.

Keywords: homogeneous linear diophantine equations, code generation, mechanized mathematics, verified code, Isabelle/HOL

1 Introduction

(Syntactic) unification of two terms s and t, is the problem of finding a substitution σ that, applied to both terms, makes them syntactically equal: $s\sigma = t\sigma$. For example, it is easily verified that $\sigma = \{x \mapsto z, y \mapsto z\}$ is a solution to the unification problem $\mathsf{f}(x, y) \approx^? \mathsf{f}(z, z)$. Several syntactic unification algorithms are known, some of which have even been formalized in proof assistants.

By throwing a set of equations E into the mix, we arrive at *equational* or *E-unification*, where we are interested in substitutions σ that make two given terms equivalent with respect to the equations in E, written $s\sigma \approx_E t\sigma$. While for syntactic unification most general solutions, called *most general unifiers*, are unique, E-unification is distinctly more complex: depending on the specific set of equations, E-unification might be undecidable, have unique solutions, have minimal complete sets of solutions, etc.

For AC-unification we instantiate E from above to a set AC of associativity and commutativity equations for certain function symbols. For example, by taking $\mathsf{AC} = \{(x \cdot y) \cdot z \approx x \cdot (y \cdot z), x \cdot y \approx y \cdot x\}$, we express that $\cdot$ (which we write infix, for convenience) is the only associative and commutative function symbol. Obviously, the substitution σ from above is also a solution to the AC-unification problem $x \cdot y \approx^?_{\mathsf{AC}} z \cdot z$ (since trivially $z \cdot z \approx_{\mathsf{AC}} z \cdot z$). You might ask: *is it the only*

* This work is supported by the Austrian Science Fund (FWF): project P27502.

$x + y = 2z$

solution	x	y	z
z_1	2	0	1
z_2	0	2	1
z_3	1	1	1

Table 1: An example HLDE and its minimal complete set of solutions

one? It turns out that it is not. More specifically, there is a minimal complete set (see Section 2 for a formal definition) consisting of the five AC-unifiers:

$$\begin{array}{lll}
\{x \mapsto z_3, & y \mapsto z_3, & z \mapsto z_3\} \\
\{x \mapsto z_1 \cdot z_1, & y \mapsto z_2 \cdot z_2, & z \mapsto z_1 \cdot z_2\} \\
\{x \mapsto z_1 \cdot z_1 \cdot z_3, & y \mapsto z_3, & z \mapsto z_1 \cdot z_3\} \\
\{x \mapsto z_3, & y \mapsto z_2 \cdot z_2 \cdot z_3, & z \mapsto z_2 \cdot z_3\} \\
\{x \mapsto z_1 \cdot z_1 \cdot z_3, & y \mapsto z_2 \cdot z_2 \cdot z_3, & z \mapsto z_1 \cdot z_2 \cdot z_3\}
\end{array}$$

But how can we compute it? The answer involves minimal complete sets of solutions for homogeneous linear diophantine equations (HLDEs for short). From the initial AC-unification problem $x \cdot y \approx^?_{\mathsf{AC}} z \cdot z$ we derive the equation in Table 1, which basically tells us that, no matter what we substitute for x, y, and z, there have to be exactly twice as many occurrences of the AC-symbol $\cdot$ in the substitutes for x and y than there are in the substitute for z.

The minimal complete set of solutions to this equation, labeled by fresh variables, is depicted in Table 1, where the numbers indicate how many occurrences of the corresponding fresh variable are contributed to the substitute for the variable in the respective column. The AC-symbol $\cdot$ is used to combine fresh variables occurring more than once. For example, the solution labeled by z_1 contributes two occurrences of z_1 to the substitute for x and one occurrence of z_1 to the substitute for z, while not touching the substitute for y at all.

Now each combination of solutions for which x, y, and z are all nonzero[1] gives rise to an independent minimal AC-unifier (in general, given n solutions, there are 2^n combinations, one for each subset of solutions). The unifiers above correspond to the combinations: $\{z_3\}, \{z_1, z_2\}, \{z_1, z_3\}, \{z_2, z_3\}, \{z_1, z_2, z_3\}$.

We refer to the literature for details on how exactly we obtain unifiers from sets of solutions to HLDEs and why this works [1,12]. Suffice it to say that minimal complete sets of solutions to HLDEs give rise to minimal complete sets of AC-unifiers.[2] The main application we have in mind, relying on minimal complete sets of AC-unifiers, is computing AC-critical pairs. This is for example useful for proving confluence of rewrite systems with and without AC-symbols [6,10,11] and required for normalized completion [8,14].

In this paper we investigate how to compute minimal complete sets of solutions of HLDEs, with our focus on formal verification using a proof assistant.

[1] The "nonzero" condition naturally arises from the fact that substitutions cannot replace variables by nothing.

[2] Actually, this only holds for elementary AC-unification problems, which are those consisting only of variables and one specific AC-symbol. However, arbitrary AC-unification problems can be reduced to sets of elementary AC-unification problems.

In other words, we are only interested in *verified* algorithms (that is, algorithms whose correctness has been machine-checked). More specifically, our contributions are as follows:

- We give an Isabelle/HOL formalization of HLDEs and their minimal complete sets of solutions (Section 3).
- We describe a simple algorithm that computes such minimal complete sets of solutions (Section 2) and discuss an easy correctness proof that we formalized in Isabelle/HOL (Section 4).
- After several rounds of program transformations, making use of standard optimization techniques and improved bounds from the literature (Section 5), we obtain a more efficient solver (Section 6)—to the best of our knowledge, the first formally verified solver for HLDEs.

Our formalization is available in the *Archive of Formal Proofs* [9] (development version, changeset `d5fabf1037f8`). Through Isabelle's code generation feature [4] a verified solver can be obtained from our formalization.

2 Main Ideas

For any formalization challenge it is a good idea to start from as simple a grounding as possible: trying to reduce the number of involved concepts to a bare minimum *and* to keep the complexity of involved proofs in check.

When formalizing an algorithm, once we have a provably correct implementation, we might still want to make it more efficient. Instead of doing all the (potentially hard) proofs again for a more efficient (and probably more involved) variant, we can often prove that the two variants are equivalent and thus carry over the correctness result from a simple implementation to an efficient one. This is also the general plan we follow for our formalized HLDE solver.

To make things simpler when computing minimal complete sets of solutions for an HLDE $a \bullet x = b \bullet y$ (where a and b are lists of coefficients and $v \bullet w$ denotes the *dot product* of two lists $v = [v_1, \ldots, v_k]$ and $w = [w_1, \ldots, w_k]$ defined by $v_1 w_1 + \cdots + v_k w_k$), we split the task into three separate phases:

- *generate* a finite search-space that covers all potentially minimal solutions
- *check* necessary criteria for minimal solutions (throwing away the rest)
- *minimize* the remaining collection of candidates

Generate. For the first phase we make use of the fact that for every minimal solution (x, y) the entries of x are bounded by the maximal coefficient in b, while the entries of y are bounded by the maximal coefficient in a (which we will prove in Section 3).

Moreover, we generate the search-space in *reverse lexicographic* order, where for arbitrary lists of numbers $u = [u_1, \ldots, u_k]$ and $v = [v_1, \ldots, v_k]$ we have $u <_{\mathsf{rlex}} v$ iff there is an $i \leq k$ such that $u_i < v_i$ and $u_j = v_j$ for all $i < j \leq k$. This allows for a simple recursive implementation and can be exploited in the minimization phase.

Assuming that x-entries of solutions are bounded by `A` and y-entries are bounded by `B`, we can implement the generate-phase by the function

```
generate A B m n = tl [(x, y). y ← gen B n, x ← gen A m]
```

where we use Haskell-like list comprehension and `tl` is the standard *tail* function on lists dropping the first element—which in this case is the trivial (and non-minimal) solution consisting only of zeroes—and `gen B n` computes all lists of natural numbers of length `n` whose entries are bounded by `B`, in reverse lexicographic order.

```
gen B 0 = [[]]
gen B (Suc n) = [x#xs. xs ← gen B n, x ← [0..B]]
```

Our initial example $x + y = 2z$ can be represented by the two lists of coefficients `[1,1]` and `[2]` and the corresponding search-space is generated by `generate 2 1 2 1`, resulting in

```
[([1,0],[0]),([2,0],[0]),([0,1],[0]),([1,1],[0]),
 ([2,1],[0]),([0,2],[0]),([1,2],[0]),([2,2],[0]),
 ([0,0],[1]),([1,0],[1]),([2,0],[1]),([0,1],[1]),
 ([1,1],[1]),([2,1],[1]),([0,2],[1]),([1,2],[1]),([2,2],[1])]
```

Check. Probably the most obvious necessary condition for (x, y) to be a minimal solution is that it is actually a solution, that is, $a \bullet x = b \bullet y$ (taking the later minimization phase into account, it is in fact also a sufficient condition). We can implement the check-phase, given two lists of coefficients `a` and `b`, by

```
check a b = filter (λ(x, y). a • x = b • y)
```

using the standard *filter* function on lists that only preserves elements satisfying the given predicate.

For our initial example `check [1,1] [2] (generate 2 1 2 1)` computes the first two phases, resulting in `[([2,0],[1]),([1,1],[1]),([0,2],[1])]`.

Minimize. It is high time that we specify in what sense minimal solutions are to be minimal. To this end, we use the *pointwise less-than-or-equal* order $\leq_{\mathsf{v}}$ on lists (whose strict part $<_{\mathsf{v}}$ is defined by $x <_{\mathsf{v}} y$ iff $x \leq_{\mathsf{v}} y$ but not $y \leq_{\mathsf{v}} x$). Now minimization can be implemented by the function

```
minimize [] = []
minimize ((x,y)#xs) =
  (x,y) # filter (λ(u,v). x@y ≮ᵥ u@v) (minimize xs)
```

where `@` is Isabelle/HOL's list concatenation. This is also where we exploit the fact that the input to `minimize` is sorted in reverse lexicographic order: then, since `(x,y)` is up front, we know that all elements of `xs` are strictly greater with respect to $<_{\mathsf{rlex}}$; moreover, $u <_{\mathsf{v}} v$ implies $u <_{\mathsf{rlex}} v$ for all u and v; and thus, `x@y` is not $<_{\mathsf{v}}$-greater than any element of `xs`, warranting that we put it in the resulting minimized list without further check.

A Simple Algorithm. Putting all three phases together we obtain a straightforward algorithm for computing all minimal solutions of an HLDE given by its lists of coefficients `a` and `b`

```
solutions a b =
  let A = max b; B = max a; m = length a; n = length b in
  minimize (check a b (generate A B m n))
```

where `length` xs—which we sometimes write $|xs|$—computes the length of a list xs. We will prove the correctness of `solutions` in Section 4.

Performance Tuning. There are several potential performance improvements over the simple algorithm from above. In a first preparatory step, we categorize solutions into *special* and *non-special* solutions (Section 5). The former are minimal by construction and can thus be excluded from the minimization phase. For the latter, several necessary conditions are known that are monotone in the sense that all prefixes and suffixes of a list satisfy them whenever the list itself does. Now merging the generate and check phases by "pushing in" these conditions as far as possible has the potential to drastically cut down the explored search-space. We will discuss the details in Section 6.

3 An Isabelle/HOL Theory of HLDEs and their Solutions

In this section, after putting our understanding of HLDEs and their solutions on firmer grounds, we obtain bounds on minimal solutions that serve as a basis for the two algorithms we present in later sections.

A *homogeneous linear diophantine equation* is an equation of the form

$$a_1 x_1 + a_2 x_2 + \cdots + a_m x_m = b_1 y_1 + b_2 y_2 + \cdots + b_n y_n$$

where coefficients a_i and b_j are fixed natural numbers. Moreover, we are only interested in solutions (x, y) over the naturals.

That means that all the required information can be encoded into two lists of natural numbers $a = [a_1, \ldots, a_m]$ and $b = [b_1, \ldots, b_n]$. From now on, let a and b be fixed, which is achieved by Isabelle's locale mechanism in our formalization:[3]

locale *hlde* = **fixes** a b :: *nat list* **assumes** $0 \notin$ `set` a **and** $0 \notin$ `set` b

In the locale, we also assume that a and b do not have any zero entries (which is useful for some proofs; note that arbitrary HLDEs can be transformed into equivalent HLDEs satisfying this assumption by dropping all zero-coefficients).

Solutions of the HLDE represented by a and b are those pairs of lists (x, y) that satisfy $a \bullet x = b \bullet y$. Formally, the *set of solutions* $\mathcal{S}(a, b)$ is given by

$$\mathcal{S}(a, b) = \{(x, y) \mid a \bullet x = b \bullet y \wedge |x| = m \wedge |y| = n\}$$

[3] For technical reasons (regarding *code generation*) we actually have the two locales *hlde-ops* and *hlde* in our formalization.

A solution is *(pointwise) minimal* iff there is no nonzero solution that is pointwise strictly smaller. The *set of (pointwise) minimal* solutions is given by

$$\mathcal{M}(a,b) = \{(x,y) \in \mathcal{S}(a,b) \mid x \neq 0 \wedge \nexists (u,v) \in \mathcal{S}(a,b).\ u \neq 0 \wedge u \mathbin{@} v <_{\mathsf{v}} x \mathbin{@} y\}$$

where we use the notation $v \neq 0$ to state that a list v is *nonzero*, that is, does not exclusively consist of zeroes. While the above definition might look asymmetric, since we only require x and u to be nonzero, we actually also have that y and v are nonzero, because (x,y) and (u,v) are both solutions and a and b do not contain any zeroes.

Huet [5, Lemma 1] has shown that, given a minimal solution (x,y), the entries of x and y are bounded by $\mathtt{max}\ b$ and $\mathtt{max}\ a$, respectively. In preparation for the proof of this result, we prove the following auxiliary fact.

Lemma 1. *If x is a list of natural numbers of length n, then either*

(1) $x_i \equiv 0 \pmod{n}$ *for some* $1 \leq i \leq n$, *or*
(2) $x_i \equiv x_j \pmod{n}$ *for some* $1 \leq i < j \leq n$.

Proof. Let X be the set of elements of x and $M = \{y \bmod n \mid y \in X\}$. If $|M| < |X|$ then property (2) follows by the pigeonhole principle. Otherwise, $|M| = |X|$ and either x contains already duplicates and we are done (again by establishing property (2)), or the elements of x are pairwise disjoint. In the latter case, we know that $|M| = n$. Since all elements of M are less than n by construction, we obtain $M = \{0, \ldots, n-1\}$. This, in turn, means that property (1) is satisfied. □

Now we are in a position to prove a variant of Huet's Lemma 1 for improved bounds (which were, to the best of our knowledge, first mentioned by Clausen and Fortenbacher [2]), where, given two lists u and v of same length, we use $\mathtt{max}_v^{\neq 0}(u)$ to denote $\mathtt{max}(\{0\} \cup \{u_i \mid 1 \leq i \leq |v| \wedge v_i \neq 0\})$, that is, the maximum of those u-elements whose corresponding v-elements are nonzero.

Lemma 2. *Let (x,y) be a minimal solution. Then we have $x_i \leq \mathtt{max}_y^{\neq 0}(b)$ for all $1 \leq i \leq m$ and $y_j \leq \mathtt{max}_x^{\neq 0}(a)$ for all $1 \leq j \leq n$.*

Proof. Since the two statements above are symmetric, we concentrate on the first one. Let $M = \mathtt{max}_y^{\neq 0}(b)$ and assume that there is $x_k > M$ with $1 \leq k \leq m$. We will show that this contradicts the minimality of (x,y). We have

$$M \cdot \sum_{j=1}^{n} y_j \geq b \bullet y = a \bullet x \geq a_k x_k > a_k \cdot M$$

and thus $\sum_{j=1}^{n} y_j > a_k$.

At this point we give an explicit construction for a corresponding existential statement in Huet's original proof. The goal is to construct a pointwise increasing sequence of lists $\boldsymbol{u} = \boldsymbol{u}^1, \ldots, \boldsymbol{u}^{a_k}$ such that for all $v \in \mathtt{set}\ \boldsymbol{u}$ we have (1) $v \leq_{\mathsf{v}} y$ and also (2) $0 < \sum_{i=1}^{n} v_i \leq a_k$. This is achieved by taking $\boldsymbol{u}^i = (\mathtt{inc}\ y\ 0)^i\ 0_{|y|}$ where 0_n denotes a list of n zeroes and we employ the auxiliary function

```
inc y i v =
  if i < length y then
    if v ! i < y ! i then v[i := v ! i + 1]
    else inc y (Suc i) v
  else v
```

that, given two lists y and v, increments v at the smallest position $j \geq i$ such that $v_j < y_j$ (if this is not possible, the result is v). Here $x\,!\,i$ denotes the ith element of list x and $x[i := v]$ a variant of list x, where the ith element is v.

As long as there is "enough space" (as guaranteed by $\sum_{j=1}^{n} y_j > a_k$), $\boldsymbol{u}^i$ is pointwise smaller than y and the sum of its elements is i for all $1 \leq i \leq a_k$, thereby satisfying both of the above properties.

Now we obtain a list u that in addition to (1) and (2) also satisfies (3) $b \bullet u \equiv 0 \pmod{a_k}$. This is achieved by applying Lemma 1 to the list of natural numbers `map` $(\lambda x.\ b \bullet x)$ $\boldsymbol{u}$, and analyzing the resulting cases. Either such a list is already in $\boldsymbol{u}$ and we are done, or $\boldsymbol{u}$ contains two lists $\boldsymbol{u}^i$ and $\boldsymbol{u}^j$ with $i < j$, for which $b \bullet \boldsymbol{u}^i \equiv b \bullet \boldsymbol{u}^j \pmod{a_k}$ holds. In the latter case, the pointwise subtraction $\boldsymbol{u}^j -_{\mathsf{v}} \boldsymbol{u}^i$ satisfies properties (1) to (3).

Remember that $x_k > M$. Together with properties (1) and (2) we know

$$b \bullet u \leq M \cdot \sum_{j=1}^{n} u_j \leq M \cdot a_k < a_k x_k$$

By (3), we further have $b \bullet u = a_k c$ for some $0 < c < x_k$, showing that (x, y) is strictly greater than the nonzero solution $(0_{|m|}[k := c], u)$. Finally, a contradiction to the minimality of (x, y). □

As a corollary, we obtain Huet's result, namely that all x_i are bounded by `max` b and all y_j are bounded by `max` a, since $\texttt{max}_v^{\neq 0}(c) \leq \texttt{max}\, c$ for all lists v and c.

4 Certified Minimal Complete Sets of Solutions

Before we prove our algorithm from Section 2 correct, let us have a look at a characterization of the elements of `minimize` that we require in the process (where $<_{\mathsf{rlex}}$ as well as $<_{\mathsf{v}}$ are extended to pairs of lists by taking their concatenation).

Lemma 3. `set` (`minimize` xs) $= \{x \in \texttt{set}\ xs \mid \nexists y \in \texttt{set}\ xs.\ y <_{\mathsf{v}} x\}$ *whenever* xs *is sorted with respect to* $<_{\mathsf{rlex}}$.

Proof. An easy induction over xs shows the direction from right to left. For the other direction, let x be an arbitrary but fixed element of `minimize` xs. Another easy induction over xs shows that then x is also in xs. Thus it remains to show that there is no y in xs which is $<_{\mathsf{v}}$-smaller than x. Assume that there is such a y for the sake of a contradiction and proceed by induction over xs. If $xs =$ `[]` we are trivially done. Otherwise, $xs = z$ `#` zs and when x is in `minimize` zs and y is in zs, the result follows by IH. In the remaining cases either $z = x$ or $z = y$,

but not both (since this would yield $z <_{\mathsf{v}} z$). For the former we have $x \leq_{\mathsf{rlex}} y$ by sortedness and for the latter we obtain $y \not<_{\mathsf{v}} x$ by the definition of `minimize` (since x is in `minimize` zs), both contradicting $y <_{\mathsf{v}} x$. □

In the remainder of this section, we will prove completeness (all minimal solutions are generated) and soundness (only minimal solutions are generated) of `solutions`.

Lemma 4 (Completeness). $\mathcal{M}(a, b) \subseteq$ `set` (`solutions` a b)

Proof. Let (x, y) be a minimal solution. We use the abbreviations $A =$ `max` b, $B =$ `max` a, and $C =$ `set` (`check` a b (`generate` A B m n)). Then, by Lemma 3, we have `set` (`solutions` a b) $= \{x \in C \mid \nexists y \in C.\, y <_{\mathsf{v}} x\}$. Note that (x, y) is in C (which contains all solutions within the bounds provided by A and B, by construction) due to Lemma 2. Moreover, $y \not<_{\mathsf{v}} x$ `@` y for all $y \in C$ follows from the minimality of (x, y), since C is clearly a subset of $\mathcal{S}(a, b)$. Together, the previous two statements conclude the proof. □

Lemma 5 (Soundness). `set` (`solutions` a b) $\subseteq \mathcal{M}(a, b)$

Proof. Let (x, y) be in `solutions` a b. According to the definition of $\mathcal{M}(a, b)$ we have to show that (x, y) is in $\mathcal{S}(a, b)$ (which is trivial), x is nonzero, and that there is no $<_{\mathsf{v}}$-smaller solution (u, v) with nonzero u. Incidentally, the last part can be narrowed down to: there is no $<_{\mathsf{v}}$-smaller *minimal* solution (u, v) (since for every solution we can find a $\leq_{\mathsf{v}}$-smaller minimal solution by well-foundedness of $<_{\mathsf{v}}$, and the left component of minimal solutions is nonzero by definition).

We start by showing that x is nonzero. Since there are no zeroes in a and b, and (x, y) is a solution, x can only be a zero-list if also y is. However, the elements of `solutions` a b are sorted in strictly increasing order with respect to $<_{\mathsf{rlex}}$ and the first one is already not the pair of zero-lists, by construction.

Now, for the sake of a contradiction, assume that there is a minimal solution $(u, v) <_{\mathsf{v}} (x, y)$. By Lemma 4, we obtain that (u, v) is also in `solutions` a b. But then, due to its minimality, (u, v) is also in C (the same set we already used in the proof of Lemma 4). Moreover, (x, y) is in C by construction. Together with Lemma 3 and $(u, v) <_{\mathsf{v}} (x, y)$, this results in the desired contradiction. □

As a corollary of the previous two results, we obtain that `solutions` computes exactly all minimal solutions, that is `set` (`solutions` a b) $= \mathcal{M}(a, b)$.

5 Special and Non-Special Solutions

For each pair of variable positions i and j, there is exactly one minimal solution such that only the x-entry at position i and the y-entry at position j are nonzero. Since all other entries are 0, the equation collapses to $a_i x_i = b_j y_j$. Taking the minimal solutions (by employing the least common multiple) of this equation, we solve for x_i and then for y_j and obtain the nonzero x-entry $d_{ij} =$ `lcm`$(a_i, b_j)/a_i$ and the nonzero y-entry $e_{ij} =$ `lcm`$(a_i, b_j)/b_j$, respectively. Given

i and j, we obtain the *special solution* (x, y) where x is $[0, \ldots, d_{ij}, \ldots, 0]$ and y is $[0, \ldots, e_{ij}, \ldots, 0]$.

All special solutions can be computed in advance and outside of our minimization phase, since special solutions are minimal (the only entries where a special solution could decrease are d_{ij} and e_{ij}, but those are minimal due to the properties of least common multiples). We compute all special solutions by the following function

```
special_solutions a b =
  [sij a b i j. i ← [1..length a], j ← [1..length b]]
```

where

```
sij a b i j = ((replicate (length a) 0)[i := dij a b i j],
               (replicate (length b) 0)[j := eij a b i j])
dij a b i j = lcm (a ! i) (b ! j) div (a ! i)
eij a b i j = lcm (a ! i) (b ! j) div (b ! j)
```

We have already seen a relatively crude bound on minimal solutions in Section 3. A further bound, this time for minimal non-special solutions, follows.

Lemma 6. *Let (x, y) be a non-special solution such that $x_i \geq d_{ij}$ and $y_j \geq e_{ij}$ for some $1 \leq i \leq m$ and $1 \leq j \leq n$. Then (x, y) is not minimal.*

Proof. Assume that (x, y) is a minimal solution and consider the special solution $(u, v) = ([0, \ldots, d_{ij}, \ldots, 0], [0, \ldots, e_{ij}, \ldots, 0])$. Due to $x_i \geq d_{ij}$ and $y_j \geq e_{ij}$ we obviously have $u \mathbin{@} v \leq_{\mathsf{v}} x \mathbin{@} y$. Since (x, y) is not special itself, we further obtain $u \mathbin{@} v <_{\mathsf{v}} x \mathbin{@} y$, contradicting the supposed minimality of (x, y). □

This result allows us to avoid all candidates that are pointwise greater than or equal to some special solution during our generation phase, which is the motivation for the following functions for bounding the elements of non-special minimal solutions. The function `max_y`, bounding entries of y, is directly taken from Huet [5]. Moreover, `max_x` is our counterpart to `max_y` bounding entries of x. As `max_x` is symmetric to `max_y`, we only give details for the latter, which is

```
max_y x j =
  if j < n ∧ E_j x ≠ ∅ then min (E_j x)
  else max a
```

where E_j is defined by

$$E_j\ x = \{e_{ij} - 1 \mid i < |x| \wedge x_i \geq d_{ij}\}$$

from which we can show that all minimal solutions satisfy the following bounds

$$\texttt{boundr}\ x\ y \longleftrightarrow (\forall 1 \leq j \leq n.\ y_j \leq \texttt{max\_y}\ x\ j)$$

$$\texttt{subdprodl}\ x\ y \longleftrightarrow (\forall k \leq m.\ [a]^k \bullet [x]^k \leq b \bullet y)$$

$$\texttt{subdprodr}\ y \longleftrightarrow (\forall l \leq n.\ [b]^l \bullet [y]^l \leq a \bullet \texttt{map}\ (\texttt{max\_x}\ [y]^l)\ [1..m])$$

where `boundr`, `subdprodl`, and `subdprodr` are mnemonic for *bound on entries of right component, bound on sub dot product of left component,* and *bound on sub dot product of right component,* respectively.

Lemma 7. *Let* $(x, y) \in \mathcal{M}(a, b)$ *be a non-special minimal solution. Then, all of the following hold:*

(1) `boundr` x y*,*
(2) `subdprodl` x y*, and*
(3) `subdprodr` x y*.*

Proof. Property (1) directly corresponds to condition (c) of Huet. Thus, we refer to our formalization for details but note that this is where Lemma 6 is employed (apart from motivating the definitions of `max_x` and `max_y` in the first place).

Property (2), which is based on Huet's condition (d), follows from (x, y) being a solution and the fact that the dot product cannot get larger by dropping (same length) suffixes from both operands.

The last property (3) is based on condition (b) from Huet's paper. Again, we refer to our formalization for details. □

Given a bound B and a list of coefficients as, the function `alls` computes all pairs whose first component is a list xs of length $|as|$ with entries at most B and whose second component is $as \bullet xs$. Note that the resulting list is sorted in reverse lexicographic order with respect to first components of pairs.[4]

```
alls B []     = [([], 0)]
alls B (a#as) = [(x # xs, s + a * x).
                  (xs, s) ← alls B as, x ← [0..B]]
```

Example 1. For a = `[1,1]` (corresponding to the left-hand side coefficients of our initial example) and `B` = 2 the list computed by `alls B` a is

```
[([0,0],0),([1,0],1),([2,0],2),([0,1],1),([1,1],2),([2,1],3),
 ([0,2],2),([1,2],3),([2,2],4)]
```

Since for a potential solution (x, y) elements of x and of y have different bounds, we employ

```
generate A B a b =
  tl (map (λ(x, y). (fst x, fst y)) (alls2 A B a b))
```

where

```
alls2 A B a b = [(xs, ys). ys ← alls B b, xs ← alls A a]
```

[4] Also, in case you are wondering, the second component of the pairs will only play a role in Section 6, where it will avoid unnecessary recomputations of sub dot products. However, including these components already for `alls` serves the purpose of enabling later proofs of program transformations (or *code equations* as they are called in Isabelle).

Note that the result of `generate` is sorted with respect to $<_{\text{rlex}}$. If we use `max` b and `max` a as bounds for x and y, respectively, then `generate` takes care of the new generate phase.

The static bounds on individual candidate solutions we obtain from Lemma 2 can be checked by the predicate

$$\texttt{static\_bounds}\ x\ y \longleftrightarrow (\forall 1 \le i \le m.\ x_i \le \texttt{max}_y^{\neq 0}(b)) \wedge (\forall 1 \le j \le n.\ y_j \le \texttt{max}_x^{\neq 0}(a))$$

The new check phase is based on the following predicate, which is a combination of these static bounds, the fact that we are only interested in solutions, and the three further bounds from Lemma 7

```
check_cond (x, y) = static_bounds x y ∧ a • x = b • y ∧
  boundr x y ∧ subdprodl x y ∧ subdprodr y)
```

and implemented by `check' = filter check_cond`.

The new minimization phase finally, is still implemented by `minimize`, only that this time its input will often be a shorter list.

Combining all three phases, non-special solutions are computed by

```
non_special_solutions =
  let A = max b; B = max a in
  minimize (check' (generate' A B a b))
```

By including all special solutions we arrive at the intermediate algorithm `solve`, which already separates special from non-special solutions, but still requires further optimization:

```
solve a b = special_solutions a b @ non_special_solutions a b
```

The proof that `solve` $a\ b$ correctly computes the set of minimal solutions, that is `set` (`solve` $a\ b$) $= \mathcal{M}(a, b)$, is somewhat complicated by the additional bounds, but structurally similar enough to the corresponding proof of `solutions` that we refer the interested reader to our formalization.

Having covered the correctness of our algorithm, it is high time to turn towards performance issues.

6 A More Efficient Algorithm for Code Generation

While the list of non-special solutions computed in Section 5 lends itself to formalization (due to its separation of concerns regarding the generate and check phases), it may waste a lot of time on generating lists that will not pass the later checks.

Example 2. Recall our initial example with coefficients $a =$ `[1,1]` and $b =$ `[2]`. Let `A` $=$ `max` $b = 2$ and `B` $=$ `max` $a = 1$. Then, the list generated by `alls B` b contains for example a y-entry `([0],0)`. This is combined with all nine elements of `alls A` a (listed in Example 1) before filtering takes place, even though only a single x-entry, namely `([0,0],0)`, will survive the check phase (since all others exceed the bound $\texttt{max}_{[0]}^{\neq 0}(b) =$ `0` for some entry).

We now proceed to a more efficient variant of `non_special_solutions` which computes the same results (alas, we cannot hope for better asymptotic behavior, since computing minimal complete sets of solutions of HLDEs is NP-complete).

While all of the following has been formalized, we will not give any proofs here, due to their rather technical nature and a lack of further insights. We start with the locale

locale *bounded_gen_check* =
  fixes C **and** B
  assumes $C\ (x\ \#\ xs)\ s = \mathit{False}$ **if** $x > B$
  and $C\ (x'\ \#\ xs)\ s'$ **if** $C\ (x\ \#\ xs)\ s,\ x' \leq x,\ s' \leq s$

which takes a condition C, a bound B, and defines a function `gen_check` that combines (to a certain extent) `generate'` and `check'` from the previous section.

```
gen_check [] = [([], 0)]
gen_check (a # as) = concat (map (incs a 0) (gen_check as))
```

Here, the auxiliary function `incs` is defined by (note that termination of this function relies on the fact that there is an upper bound—namely B, as ensured by the first assumption of the locale—on the entries of the generated lists):

```
incs a x (xs, s) =
  let t = s + a*x in
  if C (x#xs) t then (x#xs, t) # incs a (x+1) (xs, s) else []
```

The idea of `gen_check` is to length-incrementally (starting with rightmost elements) generates all lists whose elements are bounded by B, such that only intermediate results that satisfy C are computed.

For us, the crucial property of `gen_check` is its connection to `alls`, which is covered by the following result (for which we need the second locale assumption).

Lemma 8. `gen_check` $a =$ `filter` (`suffs` $C\ a$) (`alls` $B\ a$)

Where `suffs` $C\ a\ (x, s)$ ensures that $|x| = |a|$, $s = a \bullet x$ and all non-empty suffixes of the list x (including x itself) satisfy condition C.

Now we can define `generate_check` in terms of two instantiations of the locale *bounded_gen_check* (meaning that each time the locale parameters C and B are replaced by terms for which all assumptions of the locale are satisfied), using appropriate conditions C_1, C_2 and bounds B_1, B_2, respectively. This results in the two instances $\texttt{gen\_check}_1$ and $\texttt{gen\_check}_2$ of `gen_check`, where $\texttt{gen\_check}_1$ receives a further parameter y, which stands for a fixed y-entry against which we are trying to generate x-entries.

To be more precise, we use the following instantiations

$$B_1 = \lambda b.\ \texttt{max}\, b$$
$$B_2 = \texttt{max}\, a$$
$$C_1\ b\ y\ x\ s \longleftrightarrow x = [\,] \vee s \leq b \bullet y \wedge x \leq \texttt{max}_y^{\neq 0}(b)$$
$$C_2\ y\ s \longleftrightarrow y = [\,] \vee (y \leq \texttt{max}\, a\ \wedge s \leq a \bullet \texttt{map}\ (\texttt{max\_x}\ y)\ [1..|a|])$$

Combining $\texttt{gen\_check}_1$ and $\texttt{gen\_check}_2$ we obtain a function that computes candidate solutions as follows:

$$\texttt{generate\_check}\ a\ b = [(x, y) \mid y \leftarrow \texttt{gen\_check}_2\ b, x \leftarrow \texttt{gen\_check}_1\ y\ a]$$

Using Lemma 8 it can be shown that `generate_check` behaves exactly the same way as first generating candidates using `alls2` and then filtering them according to conditions C_1 and C_2.

$$\texttt{generate\_check}\ a\ b = \\ [(x, y) \leftarrow \texttt{alls2}\ (B_1\ b)\ B_2\ a\ b.\ \texttt{suffs}\ (C_1\ b\ (\texttt{fst}\ y))\ a\ x \wedge \texttt{suffs}\ C_2\ b\ y]$$

We further filter this list of candidate solutions in order to get rid of superfluous entries, resulting in the function `fast_filter` defined by

$$\texttt{filter}\ P\ (\texttt{map}\ (\lambda(x, y).\ (\texttt{fst}\ x, \texttt{fst}\ y))\ (\texttt{tl}\ (\texttt{generate\_check}\ a\ b)))$$

where $P\ (x, y) = \texttt{static\_bounds}\ x\ y \wedge a \bullet x = b \bullet y \wedge \texttt{boundr}\ x\ y$.

Extensionally `fast_filter` is equivalent to what `non_special_solutions` of our intermediate algorithm above does before minimization.

Lemma 9. *Let* $A = \texttt{max}\, b$ *and* $B = \texttt{max}\, a$. *Then*

$$\texttt{fast\_filter}\ a\ b = \texttt{check'}\ a\ b\ (\texttt{generate'}\ A\ B\ a\ b)$$

This finally allows us to use the following more efficient definition of `solve` for code generation (of course all results on `solve` carry over, since extensionally the two versions of `solve` are the same, as shown by Lemma 9).

$$\texttt{solve}\ a\ b = \texttt{special\_solutions}\ a\ b\ \texttt{@}\ \texttt{minimize}\ (\texttt{fast\_filter}\ a\ b)$$

Generating the Solver. At this point we generate Haskell code for `solve` (and also for the library functions `integer_of_nat` and `nat_of_integer`, which will be used in our main file) by

export-code `solve integer_of_nat nat_of_integer` **in** *Haskell* **module-name** *HLDE* **file** *"generated/"*

(For this step a working Isabelle installation is required.)

The only missing part is the (hand written) main entry point to our program in `Main.hs` (it takes an HLDE as command line argument in Haskell syntax, makes sure that the coefficients are all nonzero, hands the input over to `solve`, and prints the result):

```
main = getArgs >>= parse

parse [s] = start s
parse _   = do
  hPutStrLn stderr usage
  exitWith (ExitFailure 1)
```

HLDE with coefficients			verified algorithms			
			S	I	E	G
a	b	#sols	time (s)	time (s)	time (s)	time (s)
[1,1]	[2]	3	0.001	0.001	0.001	n/a
[1,1]	[3]	4	0.001	0.001	0.001	n/a
[1,1,1]	[3]	10	0.001	0.002	0.001	n/a
[1,1,1]	[3,3,2]	26	0.002	0.003	0.002	n/a
[1,2,5]	[1,2,3,4]	39	0.2	0.3	0.07	0.012
[1,1,1,2,3]	[1,1,2,2]	44	0.2	0.01	0.01	0.006
[2,5,9]	[1,2,3,7,8]	119	188.00	212.00	21.00	0.081
[2,2,2,3,3,3]	[2,2,2,3,3,3]	138	262.00	49.00	0.07	0.012
[1,4,4,8,12]	[3,6,9,12,20]	232	-	-	221.00	0.180

Table 2: Comparing runtimes of verified algorithms and fastest known algorithm

```
start input = do
  let (a, b) = read input :: ([Integer], [Integer])
  if 0 `elem` a || 0 `elem` b then do
    hPutStrLn stderr "0-coefficients are not allowed"
    exitWith (ExitFailure 2)
  else if null a || null b then do
    hPutStrLn stderr "empty lists coefficients are not allowed"
    exitWith (ExitFailure 3)
  else
    mapM_ (putStrLn . show . (\(x, y) ->
      (map integer_of_nat x, map integer_of_nat y))) (
      solve (map nat_of_integer a) (map nat_of_integer b))

usage = {- ... -}
```

A corresponding binary `hlde` can be compiled using the command (provided of course that our AFP entry and a Haskell compiler are both installed):

```
isabelle afp_build HLDE
```

We conclude this section by an example run (joining output lines to save space):

```
$ ./hlde "([2,1],[1,1,2])"
([1,0],[2,0,0]) ([1,0],[0,2,0]) ([1,0],[0,0,1]) ([0,1],[1,0,0])
([0,1],[0,1,0]) ([0,2],[0,0,1]) ([1,0],[1,1,0])
```

7 Evaluation

We compare our verified algorithms—the *simple* algorithm (S) of Section 4, the *intermediate* algorithm of Section 5 (I), and the *efficient* algorithm of Section 6

(E)—with the fastest unverified implementation we are aware of: a *graph* algorithm (G) due to Clausen and Fortenbacher [2].

In Table 2 we give the resulting runtimes (in seconds) for computing minimal complete sets of solutions of a small set of benchmark HLDEs (in increasing order of number of solutions; column #sols): the first four lines cover our initial example and three slight modifications, while the remaining examples are taken from Clausen and Fortenbacher).

However, there are two caveats: on the one hand, the runtimes for G are direct transcriptions from Clausen and Fortenbacher (hence also the missing entries for the first four examples), that is, they where generated on hardware from more than two decades ago; on the other hand, G uses improved bounds for the search-space of potential solutions, which are not formalized and thus out of reach for our verified implementations.

Anyway, our initial motivation was to certify minimal complete sets of AC-unifiers. Which is, why we want to stress the following: already for the first four examples of Table 2 the number of AC-unifiers goes from five, over 13, then 981, up to 65 926 605. For the remaining examples we were not even able to compute the number of minimal AC-unifiers (running out of memory on 20 GB of RAM); remember that in the worst case for an elementary unification problem whose corresponding HLDE has n minimal solutions, the number of minimal AC unifiers is in the order of 2^n. Thus, applications that rely on minimal complete sets of AC-unifiers will most likely not succeed on examples that are much bigger than the one in line three of Table 2, rendering certification moot.

On the upside, we expect HLDEs arising from realistic examples involving AC-unification to be quite small, since the nesting level of AC-symbols restricts the length of a and b and the multiplicity of variables restricts individual entries.

8 Related Work

In the literature, there are basically three approaches for solving HLDEs: lexicographic algorithms, completion procedures, and graph theory based algorithms.

Already in the 1970s Huet devised *an algorithm to generate the basis of solutions to homogeneous linear diophantine equations* in a paper of the same title [5], the first instance of a lexicographic algorithm. Our formalization of HLDEs and bounds on minimal solutions is inspired by Huet's elegant and short proofs. We also took up the idea of separating special and non-special solutions from Huet's work. Moreover, the structure of our algorithm mostly corresponds to Huet's informal description of his lexicographic algorithm: a striking difference is that we use a reverse lexicographic order. This facilities a construction relying on recursive list functions without the need of accumulating parameters. Compared to the beginning of our work, where we tried to stay with the standard lexicographic order, this turned out to lead to greatly simplified proofs.

In 1989, Lankford [7] proposed the first completion procedure solving HLDEs.

Fortenbacher and Clausen [2] give an accessible survey of these earlier approaches and in addition present the first graph theory based algorithm. They

conclude that any of the existing algorithms is suitable for AC-unification: on the one hand there are huge performance differences for some big HLDEs; on the other hand AC-unification typically requires only relatively small instances; moreover, if the involved HLDEs grow too big the number of minimal AC-unifiers explodes massively, dwarfing the resource requirements for solving those HLDEs.

Later, Contejean and Devie [3] gave the first algorithm that was able to solve *systems* of linear diophantine equations (and is inspired by a geometric interpretation of the algorithm due to Fortenbacher and Clausen).

In contrast to our purely functional algorithm, all of the above approaches have a distinctively imperative flavor, and to the best of our knowledge, none of them have been formalized using a proof assistant.

9 Conclusions and Further Work

We had two main reasons for choosing a lexicographic algorithm (also keeping in mind that the problem being NP-complete, all approaches are asymptotically equivalent): (1) our ultimate goal is AC-unification and as Fortenbacher and Clausen [2] put it "*How important are efficient algorithms which solve [HLDEs] for [AC-unification]? [...] any of the algorithms presented [...] might be chosen [...],*" and (2) Huet's lexicographic algorithm facilitates a simple purely functional implementation that is amenable to formalization.

Structure and Statistics. Our formalization comprises 3353 lines of code. These include 73 definitions and functions as well as 281 lemmas and theorems, most of which are proven using Isabelle's *Intelligible Semi-Automated Reasoning* language Isar [13]. The formalization is structured into the following theory files:

`List_Vector` covering facts (about dot products, pointwise subtraction, several orderings, etc.) concerning vectors represented as lists of natural numbers.

`Linear_Diophantine_Equations` covering the abstract results on HLDEs discussed in Section 3.

`Sorted_Wrt`, `Minimize_Wrt` covering some facts about sortedness and minimization with respect to a given binary predicate.

`Simple_Algorithm` containing the simple algorithm of Section 2 and its correctness proof (Section 4).

`Algorithm` containing an intermediate algorithm (Section 5) that separates special from non-special solutions, as well as a more efficient variant (Section 6).

`Solver_Code` issuing a single command to generate Haskell code for `solve` and compiling it into a program `hlde`.

Future Work. Our ultimate goal is of course to reuse the verified algorithm in an Isabelle/HOL formalization of AC-unification.

Another direction for future work is to further improve our algorithm. For example, the improved bounds $\sum_{i=1}^{m} x_i \leq \texttt{max}\, b$ and $\sum_{j=1}^{n} y_j \leq \texttt{max}\, a$ are discussed by Clausen and Fortenbacher [2]. Moreover, already Huet [5] mentions the optimization of explicitly computing x_1 after $([x_2, \ldots, x_m], y)$ is fixed (which potentially divides the number of generated lists by the maximum value in b).

References

1. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press, New York, NY, USA (1998)
2. Clausen, M., Fortenbacher, A.: Efficient solution of linear diophantine equations. Journal of Symbolic Computation 8(1), 201–216 (1989), doi:10.1016/S0747-7171(89)80025-2
3. Évelyn Contejean, Devie, H.: An efficient incremental algorithm for solving systems of linear diophantine equations. Information and Computation 113(1), 143–172 (1994), doi:10.1006/inco.1994.1067
4. Haftmann, F., Nipkow, T.: Code generation via higher-order rewrite systems. In: Proceedings of the 10th International Symposium on Functional and Logic Programming (FLOPS). Lecture Notes in Computer Science, vol. 6009, pp. 103–117. Springer (2010), doi:10.1007/978-3-642-12251-4_9
5. Huet, G.: An algorithm to generate the basis of solutions to homogeneous linear diophantine equations. Information Processing Letters 7(3), 144–147 (1978), doi:10.1016/0020-0190(78)90078-9
6. Klein, D., Hirokawa, N.: Confluence of non-left-linear TRSs via relative termination. In: Proceedings of the 18th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning. Lecture Notes in Computer Science, vol. 7180, pp. 258–273. Springer (2012), doi:10.1007/978-3-642-28717-6_21
7. Lankford, D.: Non-negative integer basis algorithms for linear equations with integer coefficients. Journal of Automated Reasoning 5(1), 25–35 (1989), doi:10.1007/BF00245019
8. Marché, C.: Normalized rewriting: An alternative to rewriting modulo a set of equations. Journal of Symbolic Computation 21(3), 253–288 (1996), doi:10.1006/jsco.1996.0011
9. Meßner, F., Parsert, J., Schöpf, J., Sternagel, C.: Homogeneous Linear Diophantine Equations. The Archive of Formal Proofs (Oct 2017), https://devel.isa-afp.org/entries/Diophantine_Eqns_Lin_Hom.shtml, Formal proof development
10. Nagele, J., Felgenhauer, B., Middeldorp, A.: CSI: New Evidence – A progress report. In: Proceedings of the 26th International Conference on Automated Deduction (CADE). Lecture Notes in Computer Science, vol. 10395, pp. 385–397. Springer (2017), doi:10.1007/978-3-319-63046-5_24
11. Shintani, K., Hirokawa, N.: CoLL: A confluence tool for left-linear term rewrite systems. In: Proceedings of the 25th International Conference on Automated Deduction (CADE). Lecture Notes in Computer Science, vol. 9195, pp. 127–136. Springer (2015), doi:10.1007/978-3-319-21401-6_8
12. Stickel, M.: A unification algorithm for associative-commutative functions. Journal of the ACM 28(3), 423–434 (1981), doi:10.1145/322261.322262
13. Wenzel, M.: Isabelle/Isar - A Versatile Environment for Human-Readable Formal Proof Documents. Ph.D. thesis, Technische Universität München, Institut für Informatik (2002)
14. Winkler, S., Middeldorp, A.: Normalized completion revisited. In: Proceedings of the 24th International Conference on Rewriting Techniques and Applications (RTA). Leibniz International Proceedings in Informatics, vol. 21, pp. 319–334. Schloss Dagstuhl (2013), doi:10.4230/LIPIcs.RTA.2013.319