Logical Methods in Computer Science
Vol. 8 (3:??) 2012, pp. 1–37
www.lmcs-online.org

Submitted Oct. 14, 2011
Published Aug. ??, 2012

TRANSFER FUNCTION SYNTHESIS WITHOUT QUANTIFIER ELIMINATION

JÖRG BRAUER [a] AND ANDY KING [b]

[a] Verified Systems International GmbH, Am Fallturm 1, 28359 Bremen, Germany and Embedded Software Laboratory, RWTH Aachen University, Ahornstr. 55, 52074 Aachen, Germany
e-mail address: brauer@verified.de

[b] Portcullis Computer Security Limited, Pinner, HA5 2EX, UK
e-mail address: a.m.king@kent.ac.uk

Abstract. Traditionally, transfer functions have been designed manually for each operation in a program, instruction by instruction. In such a setting, a transfer function describes the semantics of a single instruction, detailing how a given abstract input state is mapped to an abstract output state. The net effect of a sequence of instructions, a basic block, can then be calculated by composing the transfer functions of the constituent instructions. However, precision can be improved by applying a single transfer function that captures the semantics of the block as a whole. Since blocks are program-dependent, this approach necessitates automation. There has thus been growing interest in computing transfer functions automatically, most notably using techniques based on quantifier elimination. Although conceptually elegant, quantifier elimination inevitably induces a computational bottleneck, which limits the applicability of these methods to small blocks. This paper contributes a method for calculating transfer functions that finesses quantifier elimination altogether, and can thus be seen as a response to this problem. The practicality of the method is demonstrated by generating transfer functions for input and output states that are described by linear template constraints, which include intervals and octagons.

1. Introduction

In model checking [4] the behaviour of a program is formally specified with a model. Using the model, all paths through the program are then exhaustively checked against its requirements. The detailed nature of the requirements entails that the program is simulated in a fine-grained way, sometimes down to the level of individual bits. Because of the complexity of this reasoning there has been much interest in abstracting away from the detailed nature of states. Then, the program checker operates over classes of related states — collections of states that are equivalent in some sense — rather than individual states.

1998 ACM Subject Classification: D.2.4, F.3.1.

Key words and phrases: Abstract interpretation, static analysis, automatic abstraction, transfer functions, linear constraints.

LOGICAL METHODS IN COMPUTER SCIENCE

DOI:10.2168/LMCS-8 (3:??) 2012

1.1. **Program analysis by abstract interpretation.** Abstract interpretation [26] provides a systematic way to construct such program checkers. The key idea is to simulate the execution of each concrete operation $g : C \to C$ in a program with an abstract analogue $f : D \to D$ where C and D are domains of concrete values and descriptions, respectively. Each abstract operation f is designed to faithfully model its concrete counterpart g in the sense that if $d \in D$ describes a concrete value $c \in C$, sometimes written relationally as $d \propto c$ [55], then the result of applying g to c is described by the action of applying f to d, that is, $f(d) \propto g(c)$. Even for a fixed set of abstractions, there are typically many ways of designing the abstract operations. Ideally the abstract operations should compute abstractions that are as descriptive, that is, as accurate as possible, though there is usually interplay with accuracy and complexity, which is one reason why the literature is so rich. Normally the abstract operations are manually designed up front, prior to the analysis itself, but there are distinct advantages in synthesising the abstract operations from their concrete versions as part of the analysis itself, in a fully automatic way, which is one reason why the topic is attracting increasing attention [13, 14, 50, 57, 58, 65, 68] .

1.2. **The motivation for automatic abstraction.** One reason for automation stems from operations that arise in sequences that are known as blocks. Suppose that such a sequence is formed of n concrete operations $g_1, g_2, \ldots, g_n$, and each operation g_i has its own abstract counterpart f_i, henceforth referred to as its transfer function [46]. Suppose too that the input to the sequence $c \in C$ is described by an input abstraction $d \in D$, that is, $d \propto c$. Then the result of applying the n concrete operations to the input (one after another) is described by applying the composition of the n transfer functions to the abstract input, that is, $f_n(\ldots f_2(f_1(d))) \propto g_n(\ldots g_2(g_1(c)))$. However, a more accurate result can be obtained by deriving a single transfer function f for the block $g_n \circ \ldots \circ g_2 \circ g_1$ as a whole, designed so that $f(d) \propto g_n(\ldots g_2(g_1(c)))$. The value of this approach has been demonstrated for linear congruences [41] in the context of verifying bit-twiddling code [50].

To illustrate this interplay between block-level abstraction and precision, consider a block consisting of three instructions x := y - x; y := y - x; x := x + y that swaps the values of the variables x and y without using a third variable [86, Chap. 2.19]. To aid reasoning about the block as a whole, fresh variables are introduced, static single assignment [30] style, so as to separate different assignments to the same variable. This gives $x'' := y - x$; $y' := y - x''$; $x' := x'' + y'$ where x'' is an intermediate and x and x' (resp. y and y') represent the values of the variable x (resp. y) on entry and exit from the block. Since $x'' = y - x \wedge y' = y - x'' \wedge x' = x'' + y' \models y' = x \wedge x' = y$ it follows that cumulatively the block can be described by a pair of two variable equalities $x' = y \wedge y' = x$ which can be interpreted as transfer function for the block. From this transfer function it follows that if $x = 1$ holds on entry to the block then $y' = 1$ holds on exit. Note that equalities $x = 1$ and $y' = 1$ are considered to be two-variable since they contain no more than two variables. Now consider applying transfer functions for each of the three assignments in turn. Again, suppose that $x = 1$ holds prior to the assignment $x'' := y - x$. Since the ternary constraint $x'' = y - 1$ cannot be expressed within the two-variable equality domain then the best that can ever be inferred by any transfer function operating over this domain is $x = 1$ for the post-state. Likewise the best that can be inferred for a transfer function that simulates y' $:= y - x''$ is $x = 1$ for its post-state, and similarly for $x' := x'' + y'$. Thus by composing transfer functions over two-variable equalities one cannot show that $y' = 1$ holds on exit from the block. Therefore, the transfer function for a block can be strictly more precise

than the composition of the transfer functions for the constituent instructions. Since blocks are program-dependent, such an approach relies on automation rather than the manual provision of transfer functions for each instruction.

Another compelling reason for automation is the complexity of the concrete operations themselves; a problem that is heightened by the finite nature of machine arithmetic. For instance, even a simple concrete operation, such as increment by one, is complicated by the finite nature of computer arithmetic: if increment is applied to the largest integer that can be stored in a word, then the result is the smallest integer that is representable. As the transfer function needs to faithfully simulate concrete increment, the corner case inevitably manifests itself (if not in the transfer function itself then elsewhere [78]).

The problem of deriving transfer functions for machine instructions, such as those of the x86, is particularly acute [5] since these operations not only update registers and memory locations, but also side effect status flags [9, 75], of which there are many. When deriving a transfer functions for a sequence of machine instructions it is necessary to reason about how the status flags are used to pass state from one instruction to another. To illustrate the importance of status flags, consider double-length addition, where the operands are pairs of 32-bit words (x_1, x_0) and (y_1, y_0), the result is denoted (z_1, z_0), and the 1 subscript denotes the most significant half and 0 the least significant. Then the following block $z_0 := x_0 + y_0; c := (z_0 < x_0); z_1 := x_1 + y_1 + c$ realises 64-bit addition, providing $<$ denotes an unsigned comparison [86, Chap. 2.15]. Without considering the carry flag c, it is not clear how one can reconstruct that $(2^{32} \cdot z_1 + z_0) = (2^{32} \cdot x_1 + x_0) + (2^{32} \cdot y_1 + y_0)$ modulo 2^{64} which is the high-level abstraction of the semantics of the block without resorting to a reduced cardinal power construction [27, Theorem 10.2.0.1]. In such a construction, a domain that can express relations such as $z_0 < x_0$, henceforth called the base domain, is refined with respect to a domain which traces the value of c, an adjunct that is sometimes called the exponent domain. This refinement enables c to monitor whether $z_0 < x_0$ holds or not. Although a base domain can always been refined in this way, and the transfer functions enriched to support the extra expressiveness, an alternative approach is to derive a transfer function for a block of instructions which, in cases such as the above, better match against what can be expressed in the base domain.

As a final piece of motivation, it is worth noting that there are several ways of implementing double-length addition, and numerous ways of realising other commonly occurring operations [86], and therefore pattern matching can never yield a systematic nor a reliable way of computing transfer functions for basic blocks.

1.3. **Specifying extreme values with universal quantifiers.** Monniaux [57, 58] recently addressed the vexing question of automatic abstraction by focussing on template domains [72] which include, most notably, intervals [44] and octagons [56]. He showed that if the concrete operations are specified as piecewise linear functions, then it is possible to derive transfer functions for blocks using quantifier elimination. To illustrate the role of quantification, suppose a piecewise linear function models a block that updates three registers whose values on entry and exit are represented by bit-vectors $\boldsymbol{x}$, $\boldsymbol{y}$ and $\boldsymbol{z}$ and $\boldsymbol{x}'$, $\boldsymbol{y}'$ and $\boldsymbol{z}'$ respectively. To derive a transfer function for interval analysis, it is necessary to ascertain how the maximal value of $\boldsymbol{x}'$, denoted $\boldsymbol{x}'_u$ say, relates to the minimal and maximal values of $\boldsymbol{x}$, $\boldsymbol{y}$ and $\boldsymbol{z}$, denoted $\boldsymbol{x}_\ell$ and $\boldsymbol{x}_u$, $\boldsymbol{y}_\ell$ and $\boldsymbol{y}_u$ and $\boldsymbol{z}_\ell$ and $\boldsymbol{z}_u$ respectively. The value of $\boldsymbol{x}'_u$ can be specified in logic [57] by asserting that:

- for all values of $\boldsymbol{x}$, $\boldsymbol{y}$ and $\boldsymbol{z}$ that fall within the intervals $\boldsymbol{x} \in [\boldsymbol{x}_\ell, \boldsymbol{x}_u]$, $\boldsymbol{y} \in [\boldsymbol{y}_\ell, \boldsymbol{y}_u]$ and $\boldsymbol{z} \in [\boldsymbol{z}_\ell, \boldsymbol{z}_u]$, the value of $\boldsymbol{x}'_u$ is greater or equal to $\boldsymbol{x}'$
- for some combination of values of $\boldsymbol{x}$, $\boldsymbol{y}$ and $\boldsymbol{z}$ such that $\boldsymbol{x} \in [\boldsymbol{x}_\ell, \boldsymbol{x}_u]$, $\boldsymbol{y} \in [\boldsymbol{y}_\ell, \boldsymbol{y}_u]$ and $\boldsymbol{z} \in [\boldsymbol{z}_\ell, \boldsymbol{z}_u]$, the output $\boldsymbol{x}'$ takes the value of $\boldsymbol{x}'_u$.

The "for some" can be expressed with existential quantification, and the "for every" with universal quantification. By applying quantifier elimination, a direct relationship between $\boldsymbol{x}_\ell$, $\boldsymbol{x}_u$, $\boldsymbol{y}_\ell$, $\boldsymbol{y}_u$, $\boldsymbol{z}_\ell$, $\boldsymbol{z}_u$, and $\boldsymbol{x}'_u$ can be found, yielding a mechanism for computing $\boldsymbol{x}'_u$ in terms of $\boldsymbol{x}_\ell$, $\boldsymbol{x}_u$, $\boldsymbol{y}_\ell$, $\boldsymbol{y}_u$, $\boldsymbol{z}_\ell$, $\boldsymbol{z}_u$. This construction is ingenious but quantifier elimination is at least exponential for rational and real piecewise linear systems [20, 87], and is doubly exponential when quantifiers alternate [31]. Hence, its application requires extreme care [81].

As an alternative to operating over piecewise linear systems [13], one can instead express the semantics of a basic block with a Boolean formula; an idea that is familiar in model checking where it is colloquially referred to as bit-blasting [22]. First, bit-vector logic is used to represent the semantics of a block as a single CNF formula f_{block} (an excellent tutorial on flattening bit-vector logic into propositional logic is given in [51, Chap. 6]). Thus, each n-bit integer variable is represented as a separate vector of n propositional variables. Second, the above specification is applied to express the maximal value (or conversely the minimal value) of an output bit-vector in terms of the ranges on the input bit-vectors. This gives a propositional formula f_{spec} which is essentially f_{block} augmented with universal quantifiers and existential quantifiers. Third, the quantifiers are removed from f_{spec} to obtain f_{simp} – a simplification of f_{spec}. Of course, f_{simp} is just a Boolean formula and does not prescribe how to compute a transfer function. However, a transfer function can be extracted from f_{simp} by abstracting f_{simp} with linear affine equations [48] which directly relate the output ranges to the input ranges. This fourth step (which is analogous to that proposed for abstracting formulae with congruences [50]) is the final step in the construction.

This proposal for computing transfer functions [13] may seem attractive since computing $\forall \boldsymbol{y} : \varphi$, where φ is a system of propositional constraints and $\boldsymbol{y}$ is a vector of variables, is straightforward when the formula φ is in CNF. When φ is an arbitrary propositional system, a CNF formula ψ that is equisatisfiable, denoted $\equiv$, to φ can be found [64] by introducing fresh variables $\boldsymbol{z}$ to give $\varphi \equiv \exists \boldsymbol{z} : \psi$. However, then the transfer function synthesis problem amounts to solving $\forall \boldsymbol{y} : \exists \boldsymbol{z} : \psi$ where ψ is in CNF. To eliminate the existentially quantified variables $\boldsymbol{z}$, resolution [51, Chap. 9.2.3] can be applied, but the quadratic nature of each resolution step compromises tractability as the size of $\boldsymbol{z}$ increases. The size of $\boldsymbol{z}$ is proportional to the number of logical connectives in φ which, in turn, depends on the size of the bit-vectors and the complexity of the block under consideration. It is therefore no surprise that this approach has only been demonstrated for blocks of microcontroller code where the word-size is just 8 bits [13]. Although no polynomial-time algorithms are known for existential quantifier elimination of CNF, new algorithms are emerging [16] which will no doubt permit transfer functions to be derived for larger blocks. Nevertheless, it would be preferable if quantifier elimination was avoided altogether.

1.4. **Avoiding quantifier elimination.** This paper develops the work reported in [14] to contribute a method for deriving transfer functions which replaces quantifier elimination with successive calls to a SAT solver, where the number of calls grows linearly with the word-size rather than the size of the formula that encodes the semantics of the block.

To illustrate, consider an octagon [56] which consists of a system of inequalities of the form $\pm x \pm y \leq d$. For each of these inequalities, our approach derives the least $d \in \mathbb{Z}$ (which is uniquely determined) such that the inequality holds for all feasible values of x and y as defined by some propositional formula. As an example, consider the inequality $x + y \leq d$. The constant d is defined as $d = \min\{c \in \mathbb{Z} \mid \forall \boldsymbol{x} : \forall \boldsymbol{y} : f(\boldsymbol{x}, \boldsymbol{y}) \wedge \boldsymbol{x} + \boldsymbol{y} \leq c\}$ where $f(\boldsymbol{x}, \boldsymbol{y})$ is a propositional formula constraining the bit-vectors $\boldsymbol{x}$ and $\boldsymbol{y}$. Furthermore, given a machine with word-length w, the maximal value in an unsigned representation of x and y is $2^w - 1$, and thus we can derive an initial constraint $0 \leq d \wedge d \leq 2 \cdot (2^w - 1)$ for d, which can be expressed disjunctively as $\mu_\ell \vee \mu_u$ where:

- $\mu_\ell = 0 \leq d \wedge d \leq 2^w - 1$
- $\mu_u = 2^w \leq d \wedge d \leq 2 \cdot (2^w - 1)$

To determine which disjunct characterises d, it is sufficient to test the propositional formula $\exists \boldsymbol{x} : \exists \boldsymbol{y} : f(\boldsymbol{x}, \boldsymbol{y}) \wedge \boldsymbol{x} + \boldsymbol{y} \geq 2^w$ for *satisfiability*. If satisfiable, then μ_u is entailed by the inequality $x + y \leq d$, and μ_ℓ otherwise. We proceed by decomposing the new characterisation into a disjunction — as in dichotomic or binary search — and repeating this step w times to give d exactly. Likewise, constants d can be found for all inequalities of the form $\pm x \pm y \leq d$, which provides a mechanism for computing an octagonal abstraction that describes a given propositional formula. The force of this abstraction technique is that it provides a way of deriving octagonal guards which must hold for a block to be executed in a particular mode. For example, a block might have three modes of operation, depending on whether an operation underflows, overflows, or does neither. Which mode is applicable then depends on the values of variables on entry to the block, which motivates using guards to separate and describe the different modes of operation. Knowing that a particular mode is applicable permits a specialised transfer function to be applied for inputs that conform to that mode. It is important to note that separating modes is a crucial step in the process of applying abstract domains that operate on unbounded integers, such as affine equalities, to describe finite bit-vector semantics. As an example, consider incrementing a variable x by 1. If x and its representative x' on output are unbounded integers, the affine relation is merely $x' = x + 1$. Now suppose that x and x' are 32-bit variables. Then, if $x < 2^{32} - 1$ it follows $x' = x + 1$, and $x' = 0$ otherwise. Even though each of the two cases can be described in the affine domain, the join of these two affine relations conveys no useful information at all. Separating modes ultimately leads to a transfer function being formulated as a system of guarded updates, where the updates stipulate how the entry values are mapped to exit updates, and the guards indicate which mode holds and therefore which type of update is applicable.

This leaves the problem of how to compute the updates themselves; the input-output transformers that constitute the heart of the transfer function. We show that updates can be also computed without resorting to quantifier elimination. We demonstrate this construction not only for intervals, but for transfer functions over octagons. The method is based on computing an affine abstraction of a Boolean formula that is derived to describe the mode. For intervals, the update details how the bounds of an input interval are mapped to new bounds of an output interval. For octagons, the update maps the constants on the input octagonal inequalities to new constants on the output inequalities.

1.5. **Contributions.** Overall, the approach to computing transfer functions that is presented in this paper confers the following advantages:

- it is amenable to instructions whose semantics is presented as propositional formulae or Satisfiability Modulo Theory (SMT) [10] formulae. The force of this is that such encodings are readily available for instructions, due to the rise in popularity of SAT-based model checking;
- it avoids the computational problems associated with eliminating variables from piecewise linear systems and propositional formulae, particularly with regard to alternating quantifiers;
- it proposes the use of transfer functions that are action systems of guarded updates. These transfer functions are attractive both in terms of their expressiveness and the ease with which they can be evaluated (only one expression need be evaluated for each inequality that describes the state on exit from the block);
- it shows how the modes of a block can be found and how, for a given mode, the guards can be computed using repeated SAT solving. It is also shown how the updates for that mode can be deriving by interleaving SAT solving with affine abstraction;
- it shows how update operations, which in the case of interval analysis, compute bounds on the output intervals from bounds on the input intervals, need not be linear functions. Non-linear update operations can also be supported for transfer functions over octagons. In this context, the update operation computes the constants on the output octagonal inequalities from the constants on the input inequalities (the coefficients are fixed in both the input and output octagons hence computing a transfer function amounts to adjusting constants);
- it explains how to handle operations that underflow, overflow, or do neither and even combinations of such behaviours, providing a way to seamlessly integrate template inequalities with finite precision arithmetic.

2. Outline of the approach

Overall the paper proposes a systematic technique for inferring transfer functions that are defined as systems of guarded updates. This section illustrates the syntactic form of transfer functions, so as to provide an outline of the approach and a roadmap for the whole paper. The roadmap explains which sections of the paper are concerned with deriving which components of the transfer function.

2.1. **Modes.** Transfer functions are inferred for blocks, such as the assembly code listing in Fig. 3.2. (The approach is illustrated for blocks of 32-bit AVR UC3 assembly code [1], though the techniques are completely generic.) Each instruction is modelled by at least one, and at most four, Boolean functions according to whether it overflows or underflows, or is exact, that is, whether the instruction neither overflows nor underflows. This division into three cases reflects the ways the two's complement overflow (V) flag is set or clearer [1]. In exceptional cases this flag is used in tandem with the negative (N) flag [1] and thus it is natural to refine these three cases according to whether the negative flag is also set or clearer. However, if the instruction overflows then the result is necessarily negative whereas if it underflows then the result is non-negative, hence only the exact case needs to be further partitioned. This gives four cases in all, overflow, underflow, exact and negative, exact and non-negative, each of which can be precisely expressed with a Boolean function that describes a so-called *mode*. The different instructions that make up the block may operate in different modes, though the mode of one instruction may preclude a mode of another

being applicable. A mode combination is then chosen for each instruction, and a single Boolean formula is constructed for the block by composing a formula for each instruction in the prescribed mode. If the composed formula is unsatisfiable, then the mode combination is inconsistent. Otherwise, the mode combination is feasible and the formula describes one type of wrapping (or non-wrapping) behaviour that can be realised within the block.

2.2. **Transfer functions.** The composed formula is then used to distill a guard paired with an update; one pair is computed for each feasible mode combination. For example, the block listed in Fig. 3.2 has nine feasible mode combinations in all, yielding nine guard and update pairs of the form:

$$\left.\begin{array}{rl} 2^{31} \leq \langle\!\langle \boldsymbol{r0} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq 2^{31} & \wedge \\ 1 \leq \langle\!\langle \boldsymbol{r0} \rangle\!\rangle \leq 2^{31} - 1 & \wedge \\ 1 \leq \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq 2^{31} - 1 & \end{array}\right\} \rightarrow \left\{\begin{array}{ll} (\langle\!\langle \boldsymbol{r0}'_\ell \rangle\!\rangle = -2^{31}) & \wedge \\ (\langle\!\langle \boldsymbol{r0}'_u \rangle\!\rangle = -2^{31}) & \end{array}\right.$$

$$\left.\begin{array}{rl} 2^{31}+1 \leq \langle\!\langle \boldsymbol{r0} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq 2^{32} - 2 & \wedge \\ 0 \leq \langle\!\langle \boldsymbol{r0} \rangle\!\rangle \leq 2^{31} - 1 & \wedge \\ 0 \leq \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq 2^{31} - 1 & \end{array}\right\} \rightarrow \left\{\begin{array}{ll} (\langle\!\langle \boldsymbol{r0}'_\ell \rangle\!\rangle = 2^{32} - \langle\!\langle \boldsymbol{r0}_u \rangle\!\rangle - \langle\!\langle \boldsymbol{r1}_u \rangle\!\rangle) & \wedge \\ (\langle\!\langle \boldsymbol{r0}'_u \rangle\!\rangle = 2^{32} - \langle\!\langle \boldsymbol{r0}_\ell \rangle\!\rangle - \langle\!\langle \boldsymbol{r1}_\ell \rangle\!\rangle) & \end{array}\right.$$

$$\vdots$$

$$\left.\begin{array}{rl} -2^{31}+1 \leq \langle\!\langle \boldsymbol{r0} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq -1 & \wedge \\ 0 \leq \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq 2^{31} - 1 & \end{array}\right\} \rightarrow \left\{\begin{array}{ll} (\langle\!\langle \boldsymbol{r0}'_\ell \rangle\!\rangle = -\langle\!\langle \boldsymbol{r0}_u \rangle\!\rangle - \langle\!\langle \boldsymbol{r1}_u \rangle\!\rangle) & \wedge \\ (\langle\!\langle \boldsymbol{r0}'_u \rangle\!\rangle = -\langle\!\langle \boldsymbol{r0}_\ell \rangle\!\rangle - \langle\!\langle \boldsymbol{r1}_\ell \rangle\!\rangle) & \end{array}\right.$$

Each guard is a conjunction of linear template constraints over the inputs of the block, in this case $\langle\!\langle \boldsymbol{r0} \rangle\!\rangle$ and $\langle\!\langle \boldsymbol{r1} \rangle\!\rangle$, which denote the (signed) values of the registers `R0` and `R1` on entry to the block. The guards express properties of `R0` and `R1` which must hold for the instructions to operate in the modes that make up the feasible mode combination.

The update operations that augment the guards detail how the values of the registers are mutated for a given mode combination. For example, if the first guard is applicable, then the update asserts that the output value of `R0` takes a value in the range $[-2^{31}, -2^{31}]$ (which actually prescribes a single value); the lower and upper bounds of `R0` on exit are denoted $\langle\!\langle \boldsymbol{r0}'_\ell \rangle\!\rangle$ and $\langle\!\langle \boldsymbol{r0}'_u \rangle\!\rangle$ in the update. The second update illustrates how $\langle\!\langle \boldsymbol{r0}'_\ell \rangle\!\rangle$ and $\langle\!\langle \boldsymbol{r0}'_u \rangle\!\rangle$ can depend on the values of `R0` and `R1` on entry to the block, where the input lower and upper bounds for `R0` are denoted $\langle\!\langle \boldsymbol{r0}_\ell \rangle\!\rangle$ and $\langle\!\langle \boldsymbol{r0}_u \rangle\!\rangle$, and likewise for `R1`.

2.3. **Automatic derivation and roadmap.** The guard is constructed one inequality at a time, by applying a form of dichotomic (or binary) search. This step amounts to a series of calls to a SAT solver, as is explained in Sect. 3. Updates can be computed by inferring an affine relationship between the bound on an output symbolic constraint and the input symbolic bounds. Such a relationship can again be derived by repeated SAT solving, as detailed in Sect. 4. Replicating this construction for each of the symbolic output constants gives the update operation for a feasible mode combination. (Sect. 3 and Sect. 4 return to the example introduced given above, detailing the steps in the derivation of this transfer function.) Yet, situations can arise for which the updates cannot be expressed using affine relationships, motiving the study, in Sect. 5, of complementary classes of update which can be formed from linear template inequality constraints [21] and non-linear template equality constraints [24]. Updates that relate symbolic output constants to symbolic input constraints using equalities are complementary to those based on inequalities: both are useful when

transfer functions are evaluated. Sect. 6 focuses on this topic and explains how guards and updates are applied during fixed point computation. Evidence is presented in Sect. 7 which demonstrates that the techniques presented in the paper are capable of synthesising transfer functions for blocks, where previous approaches based on quantifier elimination were prohibitively expensive. Finally, Sect. 8 surveys the related work and Sect. 9 concludes.

3. Deriving Guards

We express the concrete semantics of a block with Boolean formulae so as to ultimately infer a set of guards that distinguish that wrapping behaviour of a block. The construction given in [13] formulates this problem using quantification, so that quantifier elimination can be applied to solve it. However, whereas universal quantifier elimination is attractive computationally, this is not so for the elimination of existentially quantified variables. We overcome this problem by reformulating the construction given in [13], and replace quantifier elimination by a series of calls to a SAT solver. This section illustrates the power of this transposition by deriving guards for some illustrative blocks of microcontroller instructions.

3.1. **Deriving interval guards by range refinement.** Consider deriving a transfer function for the operation `INC R0`, which increments the value of `R0` by one and stores the result in `R0`. For this example, we assume that the operands are unsigned. We represent the value of `R0` by a bit-vector $\boldsymbol{r0}$ and let $\langle \boldsymbol{r0} \rangle = \sum_{i=0}^{31} 2^i \cdot \boldsymbol{r0}[i]$ where $\boldsymbol{r0}[i]$ denotes the i^{th} element of $\boldsymbol{r0}$. Note that in the sequel the following notational distinction is maintained: `R0` for a register, $\boldsymbol{r0}$ for a bit-vector representing `R0` and $\langle \boldsymbol{r0} \rangle$ and $\langle\!\langle \boldsymbol{r0} \rangle\!\rangle$ for, respectively, the unsigned and signed interpretation of the bit-vector $\boldsymbol{r0}$. The instruction itself can operate in one of two modes: (1) it overflows (iff $\langle \boldsymbol{r0} \rangle = 2^{32} - 1$) or (2) it is exact (otherwise). Note that in the sequel the term exact is used to refer to a mode that is neither underflowing nor overflowing. The semantics of these two modes can be expressed as two formulae:

$$\begin{array}{lrcl} (1) & \varphi_O(\boldsymbol{X}) & = & \varphi(\boldsymbol{X}) \wedge (\bigwedge_{i=0}^{31} \boldsymbol{r0}[i]) \\ (2) & \varphi_E(\boldsymbol{X}) & = & \varphi(\boldsymbol{X}) \wedge (\bigvee_{i=0}^{31} \neg \boldsymbol{r0}[i]) \end{array}$$

where $\varphi(\boldsymbol{X})$ encodes the increment over bit-vectors $\boldsymbol{X} = \{\boldsymbol{r0}, \boldsymbol{r0'}\}$ as follows:

$$\varphi(\boldsymbol{X}) \;=\; \bigwedge_{i=0}^{31} \left(\boldsymbol{r0'}[i] \leftrightarrow \boldsymbol{r0}[i] \oplus \bigwedge_{j=0}^{i-1} \boldsymbol{r0}[j] \right)$$

Both formulae can be converted into equisatisfiable formulae in CNF by introducing fresh variables $\boldsymbol{z}$ [64, 84]. We therefore denote the resulting formulae by $\varphi_E(\boldsymbol{X}, \boldsymbol{z})$ and $\varphi_O(\boldsymbol{X}, \boldsymbol{z})$. Following our initial approach [13], the transfer function for a multi-modal block (where the internal instructions can wrap) is described as a system of guarded updates. In the one-dimensional case, octagonal guards coincide with intervals. Each guard constitutes an upper-approximation of those inputs that are compatible with the specific mode. In case of the increment, we derive guards g_O and g_E defined as:

$$\begin{array}{lrcccccc} (1) & g_O & = & 2^{32} - 1 & \leq & \langle \boldsymbol{r0} \rangle & \leq & 2^{32} - 1 \\ (2) & g_E & = & 0 & \leq & \langle \boldsymbol{r0} \rangle & \leq & 2^{32} - 2 \end{array}$$

These guards partition the inputs into two disjoint spaces: (1) a single point for the overflow case and (2) exact operation. To obtain these guards, we provide an algorithm which solves a series of SAT instances, rather than following a monolithic all-in-one approach based on quantifier elimination [13]. To illustrate our strategy, consider the computation of a least

```
1 : ADD R0 R1;    2 : MOV R2 R0;    3 : EOR R2 R1;    4 : LSL R2;
5 : SBC R2 R2;    6 : ADD R0 R2;    7 : EOR R0 R2;
```

Figure 1: Assembly listing corresponding to the assignment `R0' := isign(R0+R1,R1)`

upper bound d for $\langle \boldsymbol{r0} \rangle$ for the formula $\varphi_E(\boldsymbol{X}, \boldsymbol{z})$. Clearly, we have $0 \leq d \leq 2^{32} - 1$. We start by putting:

$$\psi_E^1(\boldsymbol{X}, \boldsymbol{z}) \quad = \quad \varphi_E(\boldsymbol{X}, \boldsymbol{z}) \wedge \langle \boldsymbol{r0} \rangle \geq 2^{31}$$

Recall that we use a binary encoding of integers in the Boolean formulae. Further, as 2^{31} is a power of two, we can finesse the need for a complicated Boolean encoding of the predicate $\langle \boldsymbol{r0} \rangle \geq 2^{31}$ by using the equivalent formula:

$$\psi_E^{\mathsf{simp},1}(\boldsymbol{X}, \boldsymbol{z}) \quad = \quad \varphi_E(\boldsymbol{X}, \boldsymbol{z}) \wedge \boldsymbol{r0}[31]$$

which is simpler both to formulate and to solve. Then, the satisfiability of $\psi_E^{\mathsf{simp},1}(\boldsymbol{X}, \boldsymbol{z})$ shows that $\boldsymbol{r0}$ takes a value in the range $2^{31} \leq \langle \boldsymbol{r0} \rangle \leq 2^{32} - 1$. Consequently, d occurs in the same range. We can thus further refine this range by testing:

$$\psi_E^2(\boldsymbol{X}, \boldsymbol{z}) \quad = \quad \varphi_E(\boldsymbol{z}) \wedge \langle \boldsymbol{r0} \rangle \geq (2^{31} + 2^{30})$$

for satisfiability, or equivalently:

$$\varphi_E^{\mathsf{simp},2}(\boldsymbol{X}, \boldsymbol{z}) \quad = \quad \varphi_E(\boldsymbol{z}) \wedge \boldsymbol{r0}[31] \wedge \boldsymbol{r0}[30]$$

As $\psi_E^{\mathsf{simp},2}(\boldsymbol{X}, \boldsymbol{z})$ is satisfiable, we infer that d satisfies $2^{30} + 2^{31} \leq d \leq 2^{32} - 1$. The method continues to refine the constraint on d into two equally sized halves. Only in the last iteration is the satisfiability check found to fail, from which we conclude that $d = \sum_{i=1}^{31} 2^i = 2^{32} - 2$. Overall, this deduction requires 32 SAT instances, but the similarity of the instances suggests that the overhead can be mitigated somewhat by incremental SAT.

3.2. **Deriving octagonal guards by range refinement.** In a second example, we show how to extend the refinement technique from intervals to octagons. To illustrate the method, consider the program fragment in Fig. 3.2. This program corresponds to an assignment `R0' := isign(R0+R1,R1)` for signed values. The function `isign` assigns `abs(R0+R1)` to `R0` if `R1` is positive, and `-abs(R0+R1)` otherwise. `R2` is used as a temporary register. The sum of `R0` and `R1` is computed by instruction (1), and instructions (2) – (7) implement `isign`. The semantics of even this simple block is not obvious due to the bounded nature of machine arithmetic. For instance, if `abs` is applied to the smallest representable integer -2^{31} then the result is 2^{31} subject to overflow, which gives -2^{31}. To derive octagons that describe such corner cases, we have to consider all combinations of over- and underflow modes of the instructions. In the above program, the instructions `ADD` (sum) and `LSL` (left-shift) can wrap in different ways, and thus are multi-modal. Neither `EOR` nor `MOV` can wrap; they are both uni-modal. Note that in general, the instruction `SBC` (subtract-with-carry) is multi-modal. However, in the case of two equal operands, the instruction can only result in 0 or -1, depending on the carry-flag. We thus ignore the wrapping of `SBC R2 R2` and consider it to be uni-modal for simplicity of presentation. Note that only overflows occurred in the previous example since the single operand was unsigned.

3.2.1. *Finding the feasible mode combinations.* In what follows, let $\mu(\boldsymbol{X})$ defined as

$$\begin{aligned}\mu(\boldsymbol{X}) = \quad & \Big(\textstyle\bigwedge_{i=0}^{31} \boldsymbol{r0'}[i] \leftrightarrow \boldsymbol{r0}[i] \oplus \boldsymbol{r1}[i] \oplus \boldsymbol{c}[i]\Big) \wedge \\ & \neg\boldsymbol{c}[0] \wedge \Big(\textstyle\bigwedge_{i=0}^{30} \boldsymbol{c}[i+1] \leftrightarrow (\boldsymbol{r0}[i] \wedge \boldsymbol{r1}[i]) \vee (\boldsymbol{r0}[i] \wedge \boldsymbol{c}[i]) \vee (\boldsymbol{r1}[i] \wedge \boldsymbol{c}[i])\Big)\end{aligned}$$

denote the Boolean encoding of the instruction `ADD R0 R1` over bit-vectors $\boldsymbol{X} = \{\boldsymbol{r0}, \boldsymbol{r1}, \dots\}$ obtained through static single assignment conversion. Here, $\boldsymbol{c}$ is a bit-vector of intermediate carry bits. The semantics of `ADD R0 R1` is to compute the sum of `R0` and `R1` and store the result in `R0`. Since we are now working with signed objects, let

$$\langle\!\langle \boldsymbol{x} \rangle\!\rangle \quad = \quad (\textstyle\sum_{i=0}^{w-2} 2^i \cdot \boldsymbol{x}[i]) - 2^{w-1} \cdot \boldsymbol{x}[w-1]$$

denote the value of a bit-vector $\boldsymbol{x}$ of length w, where $\boldsymbol{x}[w-1]$ is interpreted as the sign-bit. Then, `ADD R0 R1` has four modes of operation: overflow, underflow, exact and non-negative, exact and negative. Underflow occurs, for example, if the arithmetic sum of $\langle\!\langle \boldsymbol{r0} \rangle\!\rangle$ and $\langle\!\langle \boldsymbol{r1} \rangle\!\rangle$ is less than -2^{31}. The constraints for these modes, which are obtained directly from the instruction set specification [1, p. 127], can be expressed as four Boolean formulae:

$$\begin{aligned}\mu_O(\boldsymbol{X}) &= \neg\boldsymbol{r0}[31] \wedge \neg\boldsymbol{r1}[31] \wedge \boldsymbol{r0'}[31] \\ \mu_U(\boldsymbol{X}) &= \boldsymbol{r0}[31] \wedge \boldsymbol{r1}[31] \wedge \neg\boldsymbol{r0'}[31] \\ \mu_P(\boldsymbol{X}) &= (\boldsymbol{r0}[31] \vee \boldsymbol{r1}[31] \vee \neg\boldsymbol{r0'}[31]) \wedge (\neg\boldsymbol{r0}[31] \vee \neg\boldsymbol{r1}[31] \vee \boldsymbol{r0'}[31]) \wedge \neg\boldsymbol{r0'}[31] \\ &= (\neg\boldsymbol{r0}[31] \vee \neg\boldsymbol{r1}[31] \vee \boldsymbol{r0'}[31]) \wedge \neg\boldsymbol{r0'}[31] \\ &= (\neg\boldsymbol{r0}[31] \vee \neg\boldsymbol{r1}[31]) \wedge \neg\boldsymbol{r0'}[31] \\ \mu_N(\boldsymbol{X}) &= (\boldsymbol{r0}[31] \vee \boldsymbol{r1}[31] \vee \neg\boldsymbol{r0'}[31]) \wedge (\neg\boldsymbol{r0}[31] \vee \neg\boldsymbol{r1}[31] \vee \boldsymbol{r0'}[31]) \wedge \boldsymbol{r0'}[31] \\ &= (\boldsymbol{r0}[31] \vee \boldsymbol{r1}[31] \vee \neg\boldsymbol{r0'}[31]) \wedge \boldsymbol{r0'}[31] \\ &= (\boldsymbol{r0}[31] \vee \boldsymbol{r1}[31]) \wedge \boldsymbol{r0'}[31]\end{aligned}$$

For example, the formula $\mu(\boldsymbol{X}) \wedge \mu_O(\boldsymbol{X})$ describes the input-output relationships for `ADD R0 R1` in overflow mode. The instruction `LSL R2` shifts register `R2` to the left by one bit-position; the most-significant bit of `R2` is moved into the carry-flag. If the carry-flag is set, an overflow occurs; there is no underflow for `LSL`. Let $\nu(\boldsymbol{X}) \wedge \nu_O(\boldsymbol{X})$ and $\nu(\boldsymbol{X}) \wedge \nu_E(\boldsymbol{X})$ thus express the overflow and exact modes of `LSL R2`. In an analogous way to the first `ADD` instruction, let $\eta(\boldsymbol{X}) \wedge \eta_O(\boldsymbol{X})$, $\eta(\boldsymbol{X}) \wedge \eta_U(\boldsymbol{X})$, $\eta(\boldsymbol{X}) \wedge \eta_P(\boldsymbol{X})$ and $\eta(\boldsymbol{X}) \wedge \eta_N(\boldsymbol{X})$ express the semantics of the instruction `ADD R0 R2`. Using these encodings that satisfy a single mode, we can compose a Boolean formula for a fixed mode combination that expresses the possibility of one mode of one operation being consistent with another mode of another operation; the unsatisfiability of this formula indicates that the chosen modes are inconsistent. For example, the combination of $\mu_U(\boldsymbol{X})$, $\nu_E(\boldsymbol{X})$ and $\eta_P(\boldsymbol{X})$ is infeasible. The above block thus constitutes $4 \cdot 2 \cdot 4 = 32$ combinations of modes, but only 9 of which are satisfiable, which is depicted in Tab. 1. It is thus necessary to derive guards only for the feasible combinations.

3.2.2. *Incremental elimination of mode combinations.* The number of mode combinations in a single basic block is, in the worst case, exponential in the number of instructions in the block. The number of calls to a SAT solver required to determine feasibility is thus exponential too. Further, incremental SAT solving [89], which greatly affects the efficiency of modern solvers, cannot be exploited when the feasibility of the mode combinations are checked one-by-one. We therefore present a strategy for incrementally checking the feasibility

Table 1: Feasible and infeasible modes for the program in Fig. 3.2

`ADD R0 R1`	`LSL R2`	`ADD R0 R2`	feasible?	`ADD R0 R1`	`LSL R2`	`ADD R0 R2`	feasible?
O	E	O	no	P	E	O	no
O	E	U	no	P	E	U	no
O	E	P	no	P	E	P	**yes**
O	E	N	no	P	E	N	no
O	O	O	no	P	O	O	no
O	O	U	**yes**	P	O	U	no
O	O	P	no	P	O	P	**yes**
O	O	N	**yes**	P	O	N	no
U	E	O	no	N	E	O	no
U	E	U	no	N	E	U	no
U	E	P	no	N	E	P	no
U	E	N	no	N	E	N	**yes**
U	O	O	no	N	O	O	no
U	O	U	no	N	O	U	**yes**
U	O	P	**yes**	N	O	P	no
U	O	N	**yes**	N	O	N	**yes**

of mode combinations. To illustrate, let $\varphi(\boldsymbol{X})$ encode the instructions of the entire block and consider the case where `ADD R0 R1` underflows and `LSL R2` is exact. The formula

$$\varphi(\boldsymbol{X}) \wedge \mu_U(\boldsymbol{X}) \wedge \nu_E(\boldsymbol{X})$$

describes this compound mode, independent of the second `ADD`. Since this formula is unsatisfiable is follows that the mode combinations

$$\begin{gathered}\varphi(\boldsymbol{X}) \wedge \mu_U(\boldsymbol{X}) \wedge \nu_E(\boldsymbol{X}) \wedge \eta_O(\boldsymbol{X}),\\ \varphi(\boldsymbol{X}) \wedge \mu_U(\boldsymbol{X}) \wedge \nu_E(\boldsymbol{X}) \wedge \eta_U(\boldsymbol{X}),\\ \varphi(\boldsymbol{X}) \wedge \mu_U(\boldsymbol{X}) \wedge \nu_E(\boldsymbol{X}) \wedge \eta_P(\boldsymbol{X}),\\ \varphi(\boldsymbol{X}) \wedge \mu_U(\boldsymbol{X}) \wedge \nu_E(\boldsymbol{X}) \wedge \eta_N(\boldsymbol{X})\end{gathered}$$

are also infeasible. This suggests extending the formula $\varphi(\boldsymbol{X})$ with mode constraints, such as $\mu_U(\boldsymbol{X})$, in a tree-like fashion, instruction by instruction. A sub-tree, which represents a different modes of one instruction, is then created and followed iff the formula is satisfiable. This strategy is illustrated in Fig. 2. Observe that the technique may increase the overall number of SAT instances to be solved: 36 instead of 32 for the running example. In the worst case, if all leaves are reachable, the strategy requires an exponential number of SAT calls. However, the tree-like strategy integrates smoothly with incremental SAT solving [89] since the additional mode constraints can be passed as assumptions, thereby permitting an incremental SAT solver to reuse learnt clauses. Which technique outperforms the other strongly depends on the distribution of feasible modes, there is thus no clear winner.

3.2.3. *Deriving guards for the feasible mode combinations.* For all feasible mode combinations, it is still necessary to compute (abstract) guards which describe an over-approximation of those inputs that satisfy the respective mode. To illustrate the technique, consider the case where instruction (1) underflows, instruction (4) overflows and instruction (6) is exact and

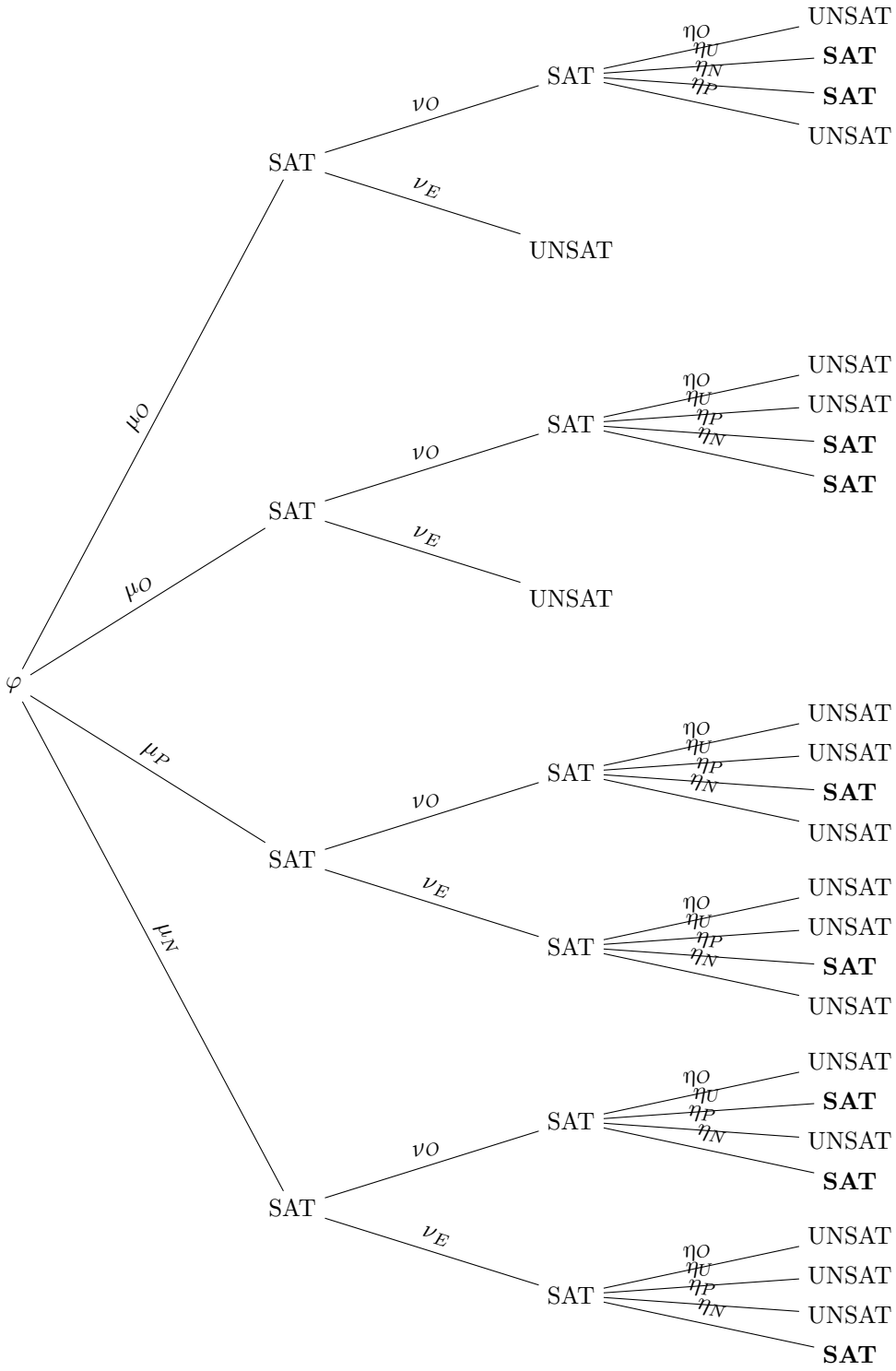

Figure 2: Incremental elimination of feasible modes

Algorithm 1 Compute the least signed value d of the k-bit vector $\boldsymbol{d} = (\boldsymbol{d}[0], \ldots, \boldsymbol{d}[k-1])$ such that the Boolean formula φ and the inequality $\sum_{i=1}^{n} c_i \cdot \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle \leq \langle\!\langle \boldsymbol{d} \rangle\!\rangle$ both hold; where the formula κ encodes $\sum_{i=1}^{n} c_i \cdot \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle = \langle\!\langle \boldsymbol{d} \rangle\!\rangle$

Input: φ, κ
1: $\phi \leftarrow \varphi \wedge \kappa$
2: {check the sign}
3: **if** $\phi \wedge \neg \boldsymbol{d}[k-1]$ is satisfiable **then**
4: $\quad d \leftarrow 0$
5: $\quad \phi \leftarrow \phi \wedge \neg \boldsymbol{d}[k-1]$
6: **else**
7: $\quad d \leftarrow -2^{k-1}$
8: $\quad \phi \leftarrow \phi \wedge \boldsymbol{d}[k-1]$
9: **end if**
10: {iterate over bits $k-2, \ldots 0$}
11: **for** $i = 1 \rightarrow k-1$ **do**
12: $\quad$ **if** $\phi \wedge \boldsymbol{d}[j]$ is satisfiable **then**
13: $\quad\quad d \leftarrow d + 2^{k-i-1}$
14: $\quad\quad \phi \leftarrow \phi \wedge \boldsymbol{d}[j]$
15: $\quad$ **else**
16: $\quad\quad \phi \leftarrow \phi \wedge \neg \boldsymbol{d}[j]$
17: $\quad$ **end if**
18: $\quad i \leftarrow i + 1$
19: **end for**
20: **return** d

non-negative. With $\varphi(\boldsymbol{X})$ encoding the instructions that constitute the block as before, the formula $\xi(\boldsymbol{X})$ which encodes this mode combination is thus defined as:

$$\xi(\boldsymbol{X}) \quad = \quad \varphi(\boldsymbol{X}) \wedge \mu_P(\boldsymbol{X}) \wedge \nu_E(\boldsymbol{X}) \wedge \eta_P(\boldsymbol{X})$$

To derive an octagonal abstraction of the inputs that satisfy $\xi(\boldsymbol{X})$, first consider the problem of computing the least upper bound d for the octagonal expression $\langle\!\langle \boldsymbol{r0} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1} \rangle\!\rangle$. To do so, let κ be a formula encoding $\langle\!\langle \boldsymbol{d} \rangle\!\rangle = \langle\!\langle \boldsymbol{r0} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1} \rangle\!\rangle$ where $\boldsymbol{d}$ is extended to 34 bits to prevent wraps in the octagonal expression (cp. [25, Sect. 3.3]). Then, check

$$\psi^1(\boldsymbol{X}) = \xi(\boldsymbol{X}) \wedge \kappa \wedge \neg \boldsymbol{d}[33]$$

for satisfiability to derive a coarse approximation of d. The satisfiability of $\psi^1(\boldsymbol{X})$ shows that $d \geq 0$. We thus proceed with testing

$$\psi^2(\boldsymbol{X}) = \xi(\boldsymbol{X}) \wedge \kappa \wedge \neg \boldsymbol{d}[33] \wedge \boldsymbol{d}[32]$$

for satisfiability. The unsatisfiability of $\psi^2(\boldsymbol{X})$ indicates $d < 2^{32}$. Next we consider

$$\psi^3(\boldsymbol{X}) = \xi(\boldsymbol{X}) \wedge \kappa \wedge \neg \boldsymbol{d}[33] \wedge \neg \boldsymbol{d}[32] \wedge \boldsymbol{d}[31]$$

The unsatisfiability of $\psi^3(\boldsymbol{X})$ shows $d < 2^{31}$. Then we test

$$\psi^4(\boldsymbol{X}) = \xi(\boldsymbol{X}) \wedge \kappa \wedge \neg \boldsymbol{d}[33] \wedge \neg \boldsymbol{d}[32] \wedge \neg \boldsymbol{d}[31] \wedge \boldsymbol{d}[30]$$

This and the ensuing formulae are all satisfiable. The exact least upper bound is thus $\langle\!\langle \boldsymbol{d} \rangle\!\rangle = 2^{30} + 2^{29} + \ldots + 2^0 = 2^{31} - 1$ hence $\langle\!\langle \boldsymbol{r0} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq 2^{31} - 1$.

Alg. 1 presents this tactic for the general case of maximising a linear expression of n variables. The algorithm relies on a propositional encoding for an affine inequality constraint $\sum_{i=0}^{n-1} c_i \cdot \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle \leq d$ where $c_1, \ldots, c_n, d \in \mathbb{Q}$. To see that such an encoding is possible assume, without loss of generality, that the inequality is integral and d is non-negative. Then rewrite the inequality as $\sum_{i=0}^{n-1} c_i^+ \cdot \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle \leq d + \sum_{i=0}^{n-1} c_i^- \cdot \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle$ where $(c_1^+, \ldots, c_n^+), (c_1^-, \ldots, c_n^-) \in \mathbb{N}^n$ and $\mathbb{N} = \{i \in \mathbb{Z} \mid 0 \leq i\}$. Let $c^+ = \sum_{i=0}^{n-1} c_i^+$ and $c^- = \sum_{i=0}^{n-1} c_i^-$. Since $\langle\!\langle \boldsymbol{v}_i \rangle\!\rangle \in [-2^{w-1}, 2^{w-1}-1]$ for each bit-vector $\boldsymbol{v}_i$, it follows that computing the sums $\sum_{i=1}^{n} c_i^+ \cdot \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle$ and $d + \sum_{i=1}^{n} c_i^- \cdot \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle$ with a signed $1 + \lceil \log_2(1 + \max(2^w \cdot c^+, b + 2^w \cdot c^-)) \rceil$ bit representation is sufficient to avoid wraps [13, Sect. 3.2]. Lines 4–9 provide special treatment for the sign. Lines 11–20 represent the core of the algorithm. Since the goal is maximisation, the algorithm instantiates each bit $\boldsymbol{d}$ with 1, starting with $\boldsymbol{d}[k-2]$, and checks satisfiability of the respective formula. If satisfiable, the bit $\boldsymbol{d}[k-i]$ is fixed at 1, and then the next highest bit is examined. If unsatisfiable, the bit $\boldsymbol{d}[k-i]$ can only take the value of 0, and the algorithm moves on to maximise the next highest bit. Variants of this algorithm have been reported elsewhere [11, 23].

Repeating this tactic for all five feasible modes, we compute the following optimal octagonal guards:

$$
\begin{aligned}
g_{O^{(1)},O^{(4)},U^{(6)}} &= \begin{cases} 2^{31} \leq \langle\!\langle \boldsymbol{r0} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq 2^{31} & \wedge \\ 1 \leq \langle\!\langle \boldsymbol{r0} \rangle\!\rangle \leq 2^{31} - 1 & \wedge \\ 1 \leq \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq 2^{31} - 1 \end{cases} \\
g_{O^{(1)},O^{(4)},N^{(6)}} &= \begin{cases} 2^{31} + 1 \leq \langle\!\langle \boldsymbol{r0} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq 2^{32} - 2 & \wedge \\ 0 \leq \langle\!\langle \boldsymbol{r0} \rangle\!\rangle \leq 2^{31} - 1 & \wedge \\ 0 \leq \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq 2^{31} - 1 \end{cases} \\
g_{U^{(1)},O^{(4)},P^{(6)}} &= \begin{cases} -2^{32} + 1 \leq \langle\!\langle \boldsymbol{r0} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq -2^{31} - 1 & \wedge \\ -2^{31} \leq \langle\!\langle \boldsymbol{r0} \rangle\!\rangle \leq -1 & \wedge \\ -2^{31} \leq \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq -1 \end{cases} \\
g_{U^{(1)},O^{(4)},N^{(6)}} &= \begin{cases} -2^{31} \leq \langle\!\langle \boldsymbol{r0} \rangle\!\rangle \leq -2^{31} & \wedge \\ -2^{31} \leq \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq -2^{31} \end{cases} \\
g_{P^{(1)},E^{(4)},P^{(6)}} &= \begin{cases} 0 \leq \langle\!\langle \boldsymbol{r0} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq 2^{31} - 1 & \wedge \\ 0 \leq \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq 2^{31} - 1 \end{cases} \\
g_{P^{(1)},O^{(4)},P^{(6)}} &= \begin{cases} 0 \leq \langle\!\langle \boldsymbol{r0} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq 2^{31} - 1 & \wedge \\ -2^{31} \leq \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq -1 \end{cases} \\
g_{N^{(1)},E^{(4)},N^{(6)}} &= \begin{cases} -2^{31} + 1 \leq \langle\!\langle \boldsymbol{r0} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq -1 & \wedge \\ -2^{31} \leq \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq -1 \end{cases} \\
g_{N^{(1)},O^{(4)},U^{(6)}} &= \begin{cases} 0 \leq \langle\!\langle \boldsymbol{r0} \rangle\!\rangle \leq 0 & \wedge \\ -2^{31} \leq \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq -2^{31} \end{cases} \\
g_{N^{(1)},O^{(4)},N^{(6)}} &= \begin{cases} -2^{31} + 1 \leq \langle\!\langle \boldsymbol{r0} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq -1 & \wedge \\ 0 \leq \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq 2^{31} - 1 \end{cases}
\end{aligned}
$$

Here, redundant inequalities, which are themselves entailed by the given guards, are omitted for clarity of presentation. Note that if the non-negative and negative sub-cases where not distinguished then the feasible modes $P^{(1)}, O^{(4)}, P^{(6)}$ and $N^{(1)}, O^{(4)}, N^{(6)}$ would be conflated into a single mode, for which the guard would be $-2^{31} + 1 \leq \langle\!\langle \boldsymbol{r0} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq 2^{31} - 1 \wedge -2^{31} \leq \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq 2^{31} - 1$ which is almost vacuous. The net effect of such a guard is that its accompanying update operation would be applied frequently, possibly unnecessarily,

inducing a loss of precision. This explains why it is attractive to separate the exact modes into two sub-cases. One can imagine enriching the modes by additionally considering, for instance, the zero flag though as yet we have not encountered an example that warrants resolving modes to this finer level of granularity.

3.2.4. *Complexity.* A total of $4 \cdot 34 + 4 \cdot 33$ SAT instances is solved for each octagonal guard. This is due to the bit-extended representation for constraints $\pm v_1 \pm v_2 \leq d$, whereas 33 bits are used for constraints $\pm v_1 \leq d$. While this may appear large, it is important to appreciate that the number of SAT instances grows linearly with the bit-width. By way of comparison with [13], adding a single propositional variable to a formula can increase the complexity of resolution quadratically [51, Sect. 9.2.3].

3.3. **Deriving template guards by range refinement.** The generality of Alg. 1 hints that the approach to deriving guards can be generalised to template inequalities where the coefficients are restricted to take a finite range of possible values. Logahedra [45] and octahedra [21] satisfy this property, the former being a class of two variable inequality where the coefficients are limited a range $\{-2^k, \ldots, -2^3, -2^2, -2^1, 0, 2^1, 2^2, 2^3, \ldots, 2^k\}$, and the latter being a class of n variable inequality where the coefficients are drawn from $\{-1, 0, 1\}$. The approach straightforwardly generalises to other finite classes of inequality, though it becomes less attractive as the number of template inequalities increase.

4. Deriving Updates with Affine Equations

Transformers over template constraints have been previously formulated using quantification [13, 57]. To avoid this, we derive affine relationships between output variables and input variables. These relations are then lifted to symbolic constraints that detail how the bounds of an input interval are mapped to the bounds of an output interval. The technique is then refined to support octagons, so as to derive linear relationships between the symbolic constants of the input octagon and the symbolic constants of the output octagon. Note that Sect. 4.2 and Sect. 4.3 are just given for pedagogical purposes; they build towards Sect. 4.4 which provides a linear symbolic update operation that is optimal (if any equality relation exists between the input and output symbolic constants then it will be found). Sect. 4.2 and Sect. 4.3 motivate Sect. 4.4 rather than provide technical background, hence the latter section can be read independently of the former sections if so desired.

4.1. **Inferring affine equalities.** Our algorithm computes an affine abstraction of the models for a given mode-combination. To solve for affine input-output relations, let $\boldsymbol{X}$ denote the set of bit-vectors as before. Consider the Boolean formula $\xi(\boldsymbol{X})$ for the case where (1) underflows, (4) overflows and (6) is exact and non-negative. The process of deriving an affine abstraction follows the scheme first presented in [13, Sect. 3.2]. It starts with solving the formula $\xi(\boldsymbol{X})$, which produces a model $\mathbf{m}_1$. Suppose the SAT solver yields:

$$\mathbf{m}_1 = \left\{ \begin{array}{ll} \langle\!\langle \boldsymbol{r0'} \rangle\!\rangle = -2^{31} & \langle\!\langle \boldsymbol{r1'} \rangle\!\rangle = -1 \\ \langle\!\langle \boldsymbol{r0} \rangle\!\rangle = -2^{31} + 1 & \langle\!\langle \boldsymbol{r1} \rangle\!\rangle = -1 \end{array} \right\}$$

We can equivalently write $\mathbf{m}_1$ as an affine matrix, denoted $\mathbf{M}_1 \in \mathbb{Z}^{4\times 5}$. With the variable ordering $\langle \boldsymbol{r0'}, \boldsymbol{r1'}, \boldsymbol{r0}, \boldsymbol{r1}\rangle$ on columns, this gives:

$$\mathbf{M}_1 \quad = \quad \left[\begin{array}{cccc|c} 1 & 0 & 0 & 0 & -2^{31} \\ 0 & 1 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 & -2^{31}+1 \\ 0 & 0 & 0 & 1 & -1 \end{array}\right]$$

We then add a disequality constraint $\langle\!\langle \boldsymbol{r1}\rangle\!\rangle \neq -1$ to $\xi(\boldsymbol{X})$ in order to obtain a new solution that is not covered by $\mathbf{M}_1$. Denote this formula by $\xi'(\boldsymbol{X})$. Then, solving for $\xi'(\boldsymbol{X})$ produces a different model $\mathbf{m}_2$, say:

$$\mathbf{m}_2 = \left\{ \begin{array}{ll} \langle\!\langle \boldsymbol{r0'}\rangle\!\rangle = -2^{31}+2 & \langle\!\langle \boldsymbol{r1'}\rangle\!\rangle = -3 \\ \langle\!\langle \boldsymbol{r0}\rangle\!\rangle = -2^{31}+1 & \langle\!\langle \boldsymbol{r1}\rangle\!\rangle = -3 \end{array} \right\}$$

Joining $\mathbf{M}_1$ with $\mathbf{M_2}$, which is likewise obtained from $\mathbf{m}_2$, using the algorithm of Müller-OIm and Seidl [60] yields a matrix that describes that affine relations common to both models:

$$\begin{aligned} \mathbf{M}_1 \sqcup \mathbf{M}_2 \quad &= \quad \left[\begin{array}{cccc|c} 1 & 0 & 0 & 0 & -2^{31} \\ 0 & 1 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 & -2^{31}+1 \\ 0 & 0 & 0 & 1 & -1 \end{array}\right] \sqcup \left[\begin{array}{cccc|c} 1 & 0 & 0 & 0 & -2^{31}+2 \\ 0 & 1 & 0 & 0 & -3 \\ 0 & 0 & 1 & 0 & -2^{31}+1 \\ 0 & 0 & 0 & 1 & -3 \end{array}\right] \\ &= \quad \left[\begin{array}{cccc|c} 1 & 1 & 0 & 0 & -2^{31}-1 \\ 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & -2^{31}+1 \end{array}\right] \end{aligned}$$

Our algorithm now attempts to find a model that violates the constraint given through the last row, that is, $\langle\!\langle \boldsymbol{r0}\rangle\!\rangle = -2^{31}+1$. Adding a disequality constraint to $\xi'(\boldsymbol{X})$ yields a new formula $\xi''(\boldsymbol{X})$, for which a SAT solver finds a model:

$$\mathbf{m}_3 = \left\{ \begin{array}{ll} \langle\!\langle \boldsymbol{r0'}\rangle\!\rangle = -2^{31} & \langle\!\langle \boldsymbol{r1'}\rangle\!\rangle = -4 \\ \langle\!\langle \boldsymbol{r0}\rangle\!\rangle = -2^{31}+4 & \langle\!\langle \boldsymbol{r1}\rangle\!\rangle = -4 \end{array} \right\}$$

Then, we join $\mathbf{M}_1 \sqcup \mathbf{M}_2$ with $\mathbf{M}_3$ to give:

$$\begin{aligned} (\mathbf{M}_1 \sqcup \mathbf{M}_2) \sqcup \mathbf{M}_3 \quad &= \quad \left[\begin{array}{cccc|c} 1 & 1 & 0 & 0 & -2^{31}-1 \\ 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & -2^{31}+1 \end{array}\right] \sqcup \left[\begin{array}{cccc|c} 1 & 0 & 0 & 0 & -2^{31} \\ 0 & 1 & 0 & 0 & -4 \\ 0 & 0 & 1 & 0 & -2^{31}+4 \\ 0 & 0 & 0 & 1 & -4 \end{array}\right] \\ &= \quad \left[\begin{array}{cccc|c} 1 & 0 & 1 & 1 & -2^{32} \\ 0 & 1 & 0 & -1 & 0 \end{array}\right] \end{aligned}$$

Adding a disequality constraint to suppress $\langle\!\langle \boldsymbol{r1'}\rangle\!\rangle - \langle\!\langle \boldsymbol{r1}\rangle\!\rangle = 0$ yields an unsatisfiable formula, likewise for $\langle\!\langle \boldsymbol{r0'}\rangle\!\rangle + \langle\!\langle \boldsymbol{r1}\rangle\!\rangle + \langle\!\langle \boldsymbol{r0}\rangle\!\rangle = -2^{32}$. Indeed, we have

$$(\mathbf{M}_1 \sqcup \mathbf{M}_2) \sqcup \mathbf{M}_3 \quad = \quad \bigsqcup_{i\in\mathbb{N}} \mathbf{M}_i$$

where $\mathbf{M}_i$ are matrices describing different models $\boldsymbol{m}_i$ of $\xi(\boldsymbol{X})$. Indeed, an affine summary of a mode-combination is in some sense universally quantified, since its relation is satisfied by every model. Moreover $(\mathbf{M}_1 \sqcup \mathbf{M}_2) \sqcup \mathbf{M}_3$ represents the best affine abstraction of $\xi(\boldsymbol{X})$ [13, 50]. Note too that the chain-length in the affine domain is linear in the number of variables in the system [48]. Thus, the number of iterations required to compute a fixed point is bounded by the number of variables and does not depend on the bit-width.

The resulting equations, however, express relationships between variables but not between the ranges of the input and output intervals. As it turns out, we can lift $(\mathbf{M}_1 \sqcup \mathbf{M}_2) \sqcup \mathbf{M}_3$ to an equation system over intervals by applying a set of straightforward transformations. This is arguably the most natural way of deriving a transformer for intervals, though we shall see that it does not extend well to octagons.

4.2. **Lifting affine equalities to interval updates.** We explain how to transform the resulting affine system $(\mathbf{M}_1 \sqcup \mathbf{M}_2) \sqcup \mathbf{M}_3$ over variables in $\boldsymbol{X}$ into an equation system over range boundaries that prescribes an update. To do so, let $\boldsymbol{V} \subseteq \boldsymbol{X}$ denote the bit-vectors on entry of the block, and likewise let $\boldsymbol{V}' \subseteq \boldsymbol{X}$ denote the bit-vectors on exit. Further, introduce sets of fresh variables

$$\begin{array}{ll} \boldsymbol{V}_\ell = \{\boldsymbol{r0}_\ell, \boldsymbol{r1}_\ell\} & \boldsymbol{V}_u = \{\boldsymbol{r0}_u, \boldsymbol{r1}_u\} \\ \boldsymbol{V}'_\ell = \{\boldsymbol{r0}'_\ell, \boldsymbol{r1}'_\ell\} & \boldsymbol{V}'_u = \{\boldsymbol{r0}'_u, \boldsymbol{r1}'_u\} \end{array}$$

to represent symbolic boundaries of each bit-vector in $\boldsymbol{V} \cup \boldsymbol{V}'$. If necessary transform the equations such that the left-hand side consists of only one variable in $\boldsymbol{V}'$. For the above system, this transformation gives:

$$\begin{array}{rcl} \langle\!\langle \boldsymbol{r1}' \rangle\!\rangle & = & \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \\ \langle\!\langle \boldsymbol{r0}' \rangle\!\rangle & = & -\langle\!\langle \boldsymbol{r0} \rangle\!\rangle - \langle\!\langle \boldsymbol{r1} \rangle\!\rangle - 2^{32} \end{array}$$

These equations imply the following affine relations on interval boundaries:

$$\begin{array}{rclcrcl} \langle\!\langle \boldsymbol{r1}' \rangle\!\rangle_u & = & \langle\!\langle \boldsymbol{r1}' \rangle\!\rangle_u & \qquad & \langle\!\langle \boldsymbol{r0}' \rangle\!\rangle_u & = & -\langle\!\langle \boldsymbol{r1} \rangle\!\rangle_\ell - \langle\!\langle \boldsymbol{r0} \rangle\!\rangle_\ell - 2^{32} \\ \langle\!\langle \boldsymbol{r1}' \rangle\!\rangle_\ell & = & \langle\!\langle \boldsymbol{r1}' \rangle\!\rangle_\ell & & \langle\!\langle \boldsymbol{r0}' \rangle\!\rangle_\ell & = & -\langle\!\langle \boldsymbol{r1} \rangle\!\rangle_u - \langle\!\langle \boldsymbol{r0} \rangle\!\rangle_u - 2^{32} \end{array}$$

To derive such as system, transform each of the original equations into the form

$$\lambda_{\boldsymbol{v}'} \cdot \boldsymbol{v}' \quad = \quad \textstyle\sum_{\boldsymbol{v} \in \boldsymbol{V}} \lambda_{\boldsymbol{v}} \cdot \boldsymbol{v} + d$$

where $\boldsymbol{v}' \in \boldsymbol{V}'$, $\lambda_{\boldsymbol{v}'} > 0$ and $\lambda_{\boldsymbol{v}} \in \mathbb{Z}$ for all $\boldsymbol{v} \in \boldsymbol{V}$. This can always be achieved due to the variable ordering. For example, the system below on the left can be transformed into the system on the right by applying elementary row operations:

$$\left[\begin{array}{cccc|c} 1 & -1 & 0 & 0 & 1 \\ 0 & 1 & 0 & -1 & 2 \end{array}\right] \quad \rightsquigarrow \quad \left[\begin{array}{cccc|c} 1 & 0 & 0 & -1 & 3 \\ 0 & 1 & 0 & -1 & 2 \end{array}\right]$$

Note that the leading coefficients are positive. We then replace each original equation by a pair of equations as follows:

$$\begin{array}{rcl} \lambda_{\boldsymbol{v}'} \cdot \boldsymbol{v}'_u & = & \sum_{\boldsymbol{v} \in \boldsymbol{X}} \lambda_{\boldsymbol{v}} \cdot \beta(\lambda_{\boldsymbol{v}}, \boldsymbol{v}) + d \\ \lambda_{\boldsymbol{v}'} \cdot \boldsymbol{v}'_\ell & = & \sum_{\boldsymbol{v} \in \boldsymbol{X}} \lambda_{\boldsymbol{v}} \cdot \beta(-\lambda_{\boldsymbol{v}}, \boldsymbol{v}) + d \end{array}$$

where the map $\beta : \mathbb{Z} \times \boldsymbol{V} \to (\boldsymbol{V}_\ell \cup \boldsymbol{V}_u)$ is defined as:

$$\beta(\lambda, \boldsymbol{v}) \quad = \quad \begin{cases} \boldsymbol{v}_\ell & : \text{if } \lambda < 0 \\ \boldsymbol{v}_u & : \text{otherwise} \end{cases}$$

The key idea when constructing the upper bound is to replace each occurrence of a variable in the original system with its upper bound in case its coefficient is positive, and with its lower bound otherwise. This task is performed by β. An analogous technique is applied when defining the lower bound. Applying this technique to all affine systems, we obtain the

following five transfer functions over symbolic ranges, rather than concrete variables (with the identity constraints on $\boldsymbol{r1}'_\ell$ and $\boldsymbol{r1}'_u$ omitted):

$$
\begin{array}{rcl}
f_{O^{(1)},O^{(4)},U^{(6)}} & = & \left\{ \begin{array}{lcl} (\langle\!\langle \boldsymbol{r0}'_\ell \rangle\!\rangle & = & -2^{31}) \quad \wedge \\ (\langle\!\langle \boldsymbol{r0}'_u \rangle\!\rangle & = & -2^{31}) \end{array} \right. \\
f_{O^{(1)},O^{(4)},N^{(6)}} & = & \left\{ \begin{array}{lcl} (\langle\!\langle \boldsymbol{r0}'_\ell \rangle\!\rangle & = & 2^{32} - \langle\!\langle \boldsymbol{r0}_u \rangle\!\rangle - \langle\!\langle \boldsymbol{r1}_u \rangle\!\rangle) \quad \wedge \\ (\langle\!\langle \boldsymbol{r0}'_u \rangle\!\rangle & = & 2^{32} - \langle\!\langle \boldsymbol{r0}_\ell \rangle\!\rangle - \langle\!\langle \boldsymbol{r1}_\ell \rangle\!\rangle) \end{array} \right. \\
f_{U^{(1)},O^{(4)},P^{(6)}} & = & \left\{ \begin{array}{lcl} (\langle\!\langle \boldsymbol{r0}'_\ell \rangle\!\rangle & = & -2^{32} - \langle\!\langle \boldsymbol{r0}_u \rangle\!\rangle - \langle\!\langle \boldsymbol{r1}_u \rangle\!\rangle) \quad \wedge \\ (\langle\!\langle \boldsymbol{r0}'_u \rangle\!\rangle & = & -2^{32} - \langle\!\langle \boldsymbol{r0}_\ell \rangle\!\rangle - \langle\!\langle \boldsymbol{r1}_\ell \rangle\!\rangle) \end{array} \right. \\
f_{U^{(1)},O^{(4)},N^{(6)}} & = & \left\{ \begin{array}{lcl} (\langle\!\langle \boldsymbol{r0}'_\ell \rangle\!\rangle & = & 0) \quad \wedge \\ (\langle\!\langle \boldsymbol{r0}'_u \rangle\!\rangle & = & 0) \end{array} \right. \\
f_{P^{(1)},E^{(4)},P^{(6)}} & = & \left\{ \begin{array}{lcl} (\langle\!\langle \boldsymbol{r0}'_\ell \rangle\!\rangle & = & \langle\!\langle \boldsymbol{r0}_\ell \rangle\!\rangle + \langle\!\langle \boldsymbol{r1}_\ell \rangle\!\rangle) \quad \wedge \\ (\langle\!\langle \boldsymbol{r0}'_u \rangle\!\rangle & = & \langle\!\langle \boldsymbol{r0}_u \rangle\!\rangle + \langle\!\langle \boldsymbol{r1}_u \rangle\!\rangle) \end{array} \right. \\
f_{P^{(1)},O^{(4)},P^{(6)}} & = & \left\{ \begin{array}{lcl} (\langle\!\langle \boldsymbol{r0}'_\ell \rangle\!\rangle & = & -\langle\!\langle \boldsymbol{r0}_u \rangle\!\rangle - \langle\!\langle \boldsymbol{r1}_u \rangle\!\rangle) \quad \wedge \\ (\langle\!\langle \boldsymbol{r0}'_u \rangle\!\rangle & = & -\langle\!\langle \boldsymbol{r0}_\ell \rangle\!\rangle - \langle\!\langle \boldsymbol{r1}_\ell \rangle\!\rangle) \end{array} \right. \\
f_{N^{(1)},E^{(4)},N^{(6)}} & = & \left\{ \begin{array}{lcl} (\langle\!\langle \boldsymbol{r0}'_\ell \rangle\!\rangle & = & \langle\!\langle \boldsymbol{r0}_\ell \rangle\!\rangle + \langle\!\langle \boldsymbol{r1}_\ell \rangle\!\rangle) \quad \wedge \\ (\langle\!\langle \boldsymbol{r0}'_u \rangle\!\rangle & = & \langle\!\langle \boldsymbol{r0}_u \rangle\!\rangle + \langle\!\langle \boldsymbol{r1}_u \rangle\!\rangle) \end{array} \right. \\
f_{N^{(1)},O^{(4)},U^{(6)}} & = & \left\{ \begin{array}{lcl} (\langle\!\langle \boldsymbol{r0}'_\ell \rangle\!\rangle & = & -2^{31}) \quad \wedge \\ (\langle\!\langle \boldsymbol{r0}'_u \rangle\!\rangle & = & -2^{31}) \end{array} \right. \\
f_{N^{(1)},O^{(4)},N^{(6)}} & = & \left\{ \begin{array}{lcl} (\langle\!\langle \boldsymbol{r0}'_\ell \rangle\!\rangle & = & -\langle\!\langle \boldsymbol{r0}_u \rangle\!\rangle - \langle\!\langle \boldsymbol{r1}_u \rangle\!\rangle) \quad \wedge \\ (\langle\!\langle \boldsymbol{r0}'_u \rangle\!\rangle & = & -\langle\!\langle \boldsymbol{r0}_\ell \rangle\!\rangle - \langle\!\langle \boldsymbol{r1}_\ell \rangle\!\rangle) \end{array} \right.
\end{array}
$$

To illustrate the accuracy of this result, consider the application of the transfer function $f_{U^{(1)},O^{(4)},P^{(6)}}$ to the input intervals defined by:

$$\langle\!\langle \boldsymbol{r0}_\ell \rangle\!\rangle = -2^{31} + 1 \qquad \langle\!\langle \boldsymbol{r0}_u \rangle\!\rangle = -2^{31} + 4 \qquad \langle\!\langle \boldsymbol{r1}_\ell \rangle\!\rangle = -20 \qquad \langle\!\langle \boldsymbol{r1}_u \rangle\!\rangle = -10$$

Then, the above transfer function defines the output intervals by modelling the wrap that occurs in the first instruction `ADD R0 R1` to give $\langle\!\langle \boldsymbol{r0}'_\ell \rangle\!\rangle = -2^{31} + 6$ and $\langle\!\langle \boldsymbol{r0}'_u \rangle\!\rangle = -2^{31} + 19$. When multiple guards are applicable, however, a merge operation need be applied to combine the results of the different updates. This looses information. Further details of the evaluation mechanism are discussed in section 6.

It is interesting to compare this with how an interval analysis would proceed for the block which, recall, is listed in Fig. 3.2. Initially, the `R0`, `R1` and `R2` would respectively be assigned the intervals $[-2^{31}+1, -2^{31}+4]$, $[-20, -10]$ and $[-2^{31}, 2^{31}-1]$ the third interval being vacuous. The `ADD R0 R1` instruction will assign `R0` to $[2^{31}-19, 2^{31}-6]$ simulating an underflow and `MOV R2 R0` will update `R2` to $[2^{31}-19, 2^{31}-6]$. The `EOR R2 R1` instruction will then reassign `R2` to $[-2^{31}, -2^{31}+2^{29}-1]$ which is adjusted to $[0, 2^{30}-1]$ by `LSL R2`. In a carefully constructed interval analysis the transfer function for `LSR R2` will also assign the carry flag to 1. In such an analysis, the instruction `SBC R2 R2` might even assign `R2` to [-1, -1] rather than a wider interval. Under this assumption `ADD R0 R2` will update `R0` to $[2^{31}-20, 2^{31}-7]$. Then, since the sign bit is clear and following 26 high bits of `R0` are set for all values in the interval $[2^{31}-20, 2^{31}-7]$, a transfer function for `EOR R0 R2` could conceivable assign `R0` to $[-2^{31}, -2^{31}+31]$.

Table 2: Intermediate results for inferring exact affine transformers for octagons

	$\langle\!\langle \boldsymbol{d}'_1 \rangle\!\rangle$	$\langle\!\langle \boldsymbol{d}_1 \rangle\!\rangle$	$\langle\!\langle \boldsymbol{d}_2 \rangle\!\rangle$	$\langle\!\langle \boldsymbol{d}_3 \rangle\!\rangle$	$\langle\!\langle \boldsymbol{d}_4 \rangle\!\rangle$	$\langle\!\langle \boldsymbol{d}_5 \rangle\!\rangle$	$\langle\!\langle \boldsymbol{d}_6 \rangle\!\rangle$	$\langle\!\langle \boldsymbol{d}_7 \rangle\!\rangle$	$\langle\!\langle \boldsymbol{d}_8 \rangle\!\rangle$	$\max(\langle\!\langle \boldsymbol{d}' \rangle\!\rangle)$
$\boldsymbol{m}_1$	1	1	1	0	0	1	0	1	1	2
$\boldsymbol{m}_2$	8	3	3	−1	−1	5	−2	2	0	10
$\boldsymbol{m}_3$	22	8	7	0	1	13	3	4	0	26
$\boldsymbol{m}_4$	4	0	3	2	0	3	1	6	3	6

4.3. **Lifting affine equalities to octagonal updates.** Consider now the more general problem of deriving a transfer function for octagons for `ADD R0 R1; LSL R0` where `ADD` and `LSL` operate in exact non-negative modes. Computing the affine relation for this mode-combination gives $(\langle\!\langle \boldsymbol{r0}' \rangle\!\rangle = 2 \cdot \langle\!\langle \boldsymbol{r0} \rangle\!\rangle + 2 \cdot \langle\!\langle \boldsymbol{r1} \rangle\!\rangle) \wedge (\langle\!\langle \boldsymbol{r1}' \rangle\!\rangle = \langle\!\langle \boldsymbol{r1} \rangle\!\rangle)$. We aim to construct an update that maps octagonal input constraints with symbolic constants to octagonal outputs likewise with symbolic constants of the form:

$$\left\{\begin{array}{rcl} \langle\!\langle \boldsymbol{r0} \rangle\!\rangle & \leq & d_1 \\ \langle\!\langle \boldsymbol{r1} \rangle\!\rangle & \leq & d_2 \\ -\langle\!\langle \boldsymbol{r0} \rangle\!\rangle & \leq & d_3 \\ -\langle\!\langle \boldsymbol{r1} \rangle\!\rangle & \leq & d_4 \\ \hline \langle\!\langle \boldsymbol{r0} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1} \rangle\!\rangle & \leq & d_5 \\ -\langle\!\langle \boldsymbol{r0} \rangle\!\rangle - \langle\!\langle \boldsymbol{r1} \rangle\!\rangle & \leq & d_6 \\ -\langle\!\langle \boldsymbol{r0} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1} \rangle\!\rangle & \leq & d_7 \\ \langle\!\langle \boldsymbol{r0} \rangle\!\rangle - \langle\!\langle \boldsymbol{r1} \rangle\!\rangle & \leq & d_8 \end{array}\right\} \quad \rightsquigarrow \quad \left\{\begin{array}{rcl} \langle\!\langle \boldsymbol{r0}' \rangle\!\rangle & \leq & 2 \cdot (d_1 + d_2) \\ \langle\!\langle \boldsymbol{r1}' \rangle\!\rangle & \leq & d_2 \\ -\langle\!\langle \boldsymbol{r0}' \rangle\!\rangle & \leq & 2 \cdot (d_3 + d_4) \\ -\langle\!\langle \boldsymbol{r1}' \rangle\!\rangle & \leq & d_4 \\ \langle\!\langle \boldsymbol{r0}' \rangle\!\rangle + \langle\!\langle \boldsymbol{r1}' \rangle\!\rangle & \leq & 2 \cdot d_1 + 3 \cdot d_2 \\ -\langle\!\langle \boldsymbol{r0}' \rangle\!\rangle - \langle\!\langle \boldsymbol{r1}' \rangle\!\rangle & \leq & 2 \cdot d_3 + 3 \cdot d_4 \\ -\langle\!\langle \boldsymbol{r0}' \rangle\!\rangle + \langle\!\langle \boldsymbol{r1}' \rangle\!\rangle & \leq & 2 \cdot (d_3 + d_4) + d_2 \\ \langle\!\langle \boldsymbol{r0}' \rangle\!\rangle - \langle\!\langle \boldsymbol{r1}' \rangle\!\rangle & \leq & 2 \cdot (d_1 + d_2) + d_4 \end{array}\right\}$$

We start by constructing an update operation that uses the unary input constraints only, which appear above the bar separator. We modify the method presented in Sect. 4.2 so as to express output constraints in terms of symbolic variables $d_1, \ldots, d_4$ from the input constraints. We obtain the four output unary constraints by an analogous technique as before by substituting the symbolic minima and maxima for the symbolic output constants. The binary output constraints are derived by linear combinations of the unary output constraints. Since the output constraints do not use relational information from the inputs, such as $\langle\!\langle \boldsymbol{r0} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq d_5$, we obtain a sub-optimal update. To illustrate, suppose $0 \leq \langle\!\langle \boldsymbol{r0} \rangle\!\rangle \leq 4$, $0 \leq \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq 1$ and $\langle\!\langle \boldsymbol{r0} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq 4$. Then we derive:

$$0 \leq \langle\!\langle \boldsymbol{r0}' \rangle\!\rangle \leq 10 \qquad 0 \leq \langle\!\langle \boldsymbol{r1}' \rangle\!\rangle \leq 1 \qquad 0 \leq \langle\!\langle \boldsymbol{r0}' \rangle\!\rangle + \langle\!\langle \boldsymbol{r1}' \rangle\!\rangle \leq 11$$

An optimal transfer function, however, would derive $\langle\!\langle \boldsymbol{r0}' \rangle\!\rangle \leq 8$ and $\langle\!\langle \boldsymbol{r0}' \rangle\!\rangle + \langle\!\langle \boldsymbol{r1}' \rangle\!\rangle \leq 8$. Although the above method fails to propagate the effect of some inputs into the outputs, it retains the property that the update can be constructed straightforwardly by lifting the affine relations. In what follows, we will describe how to derive more precise affine relations for the outputs.

4.4. **Inferring affine inequalities for octagonal updates.** To derive more precise affine updates for octagons, let $\xi(\boldsymbol{X})$ denote the propositional encoding for `ADD R0 R1; LSL R0` where again `ADD` and `LSL` operate in exact non-negative modes. Consider inequality $\langle\!\langle \boldsymbol{r0}' \rangle\!\rangle \leq d'_1$ in the output octagon and in particular the problem of discovering a relationship between d'_1 and the symbolic constants $d_1, \ldots, d_8$ of the input octagon, as detailed previously.

We proceed by introducing signed 34-bit vectors $\boldsymbol{d}_1, \ldots, \boldsymbol{d}_8$ to represent the symbolic constants $d_1, \ldots, d_8$. Further, let κ denote a Boolean formula that holds iff the eight inequalities $\langle\!\langle \boldsymbol{r0} \rangle\!\rangle \leq \langle\!\langle \boldsymbol{d_1} \rangle\!\rangle$, $\ldots$, $\langle\!\langle \boldsymbol{r0} \rangle\!\rangle - \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq \langle\!\langle \boldsymbol{d_8} \rangle\!\rangle$ simultaneously hold. Furthermore, let η denote a formula that encodes the equality $\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle = \langle\!\langle \boldsymbol{d'_1} \rangle\!\rangle$ where $\boldsymbol{d'_1}$ is a signed bit-vector representing d'_1. Presenting the compound formula $\kappa \wedge \xi(\boldsymbol{X}) \wedge \eta$ to a SAT solver produces a model:

$$\boldsymbol{m}_1 = \left\{ \; \langle\!\langle \boldsymbol{d'_1} \rangle\!\rangle = 1, \quad \langle\!\langle \boldsymbol{d_1} \rangle\!\rangle = 1, \quad \langle\!\langle \boldsymbol{d_2} \rangle\!\rangle = 1, \quad \ldots, \langle\!\langle \boldsymbol{d_7} \rangle\!\rangle = 1, \quad \langle\!\langle \boldsymbol{d_8} \rangle\!\rangle = 1 \quad \right\}$$

which is fully detailed in Tab. 2. The assignment $\langle\!\langle \boldsymbol{d'_1} \rangle\!\rangle = 1$ does not necessarily represent the maximum value of $\langle\!\langle \boldsymbol{d'_1} \rangle\!\rangle$ for the partial assignment $\langle\!\langle \boldsymbol{d_1} \rangle\!\rangle = 1$, $\ldots$, $\langle\!\langle \boldsymbol{d_8} \rangle\!\rangle = 1$. Thus let ζ_1 denote a formula that holds iff $\langle\!\langle \boldsymbol{d_1} \rangle\!\rangle = 1$, $\ldots$, $\langle\!\langle \boldsymbol{d_8} \rangle\!\rangle = 1$ all hold. Then range refinement can be applied to find the maximal value of $\langle\!\langle \boldsymbol{d'_1} \rangle\!\rangle$ subject to $\kappa \wedge \xi(\boldsymbol{X}) \wedge \eta \wedge \zeta$. This gives $\langle\!\langle \boldsymbol{d'_1} \rangle\!\rangle = 2$ and a model:

$$\boldsymbol{m}'_1 \quad = \quad \left\{ \langle\!\langle \boldsymbol{d'_1} \rangle\!\rangle = 2, \langle\!\langle \boldsymbol{d_1} \rangle\!\rangle = 1, \ldots, \langle\!\langle \boldsymbol{d_8} \rangle\!\rangle = 1 \right\}$$

An affine summary of all such maximal models can be found by interleaving range refinement with affine join. Thus suppose the matrix $\mathbf{M}_1$ is constructed from $\boldsymbol{m}'_1$ by using the variable ordering $\langle d'_1, d_1, \ldots, d_8 \rangle$ on columns:

$$\mathbf{M}_1 = \left[\begin{array}{ccccccccc|c} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{array}\right]$$

The method proceeds in an analogous fashion to before by constructing a formula μ that holds iff $\langle\!\langle \boldsymbol{d_8} \rangle\!\rangle \neq 1$ holds. Solving the formula $\kappa \wedge \xi(\boldsymbol{X}) \wedge \eta \wedge \mu$ gives the model $\boldsymbol{m}_2$ detailed in Tab. 2. The model $\boldsymbol{m}_2$, itself, defines a formula ζ_2 that is equi-satisfiable with the conjunction of $\langle\!\langle \boldsymbol{d_1} \rangle\!\rangle = 3$, $\ldots$, $\langle\!\langle \boldsymbol{d_8} \rangle\!\rangle = 0$. Maximising $\langle\!\langle \boldsymbol{d'_1} \rangle\!\rangle$ subject to $\kappa \wedge \xi(\boldsymbol{X}) \wedge \eta \wedge \zeta_2$ gives $\langle\!\langle \boldsymbol{d'_1} \rangle\!\rangle = 10$ which defines the model

$$\boldsymbol{m}'_2 \quad = \quad \left\{ \langle\!\langle \boldsymbol{d'_1} \rangle\!\rangle = 10, \langle\!\langle \boldsymbol{d_1} \rangle\!\rangle = 3, \ldots, \langle\!\langle \boldsymbol{d_8} \rangle\!\rangle = 0 \right\}$$

and $\mathbf{M}_2$, which in turn yields the join $\mathbf{M}_1 \sqcup \mathbf{M}_2$ as follows:

$$\mathbf{M}_1 \sqcup \mathbf{M}_2 = \left[\begin{array}{ccccccccc|c} 1 & 0 & 0 & 0 & 0 & -2 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 2 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \end{array}\right]$$

Repeating this process two more times then gives:

$$\begin{array}{rcl} \boldsymbol{m}'_3 & = & \left\{ \langle\!\langle \boldsymbol{d'_1} \rangle\!\rangle = 26, \langle\!\langle \boldsymbol{d_1} \rangle\!\rangle = 8, \ldots, \langle\!\langle \boldsymbol{d_8} \rangle\!\rangle = 0 \right\} \\ \boldsymbol{m}'_4 & = & \left\{ \langle\!\langle \boldsymbol{d'_1} \rangle\!\rangle = 6, \langle\!\langle \boldsymbol{d_1} \rangle\!\rangle = 0, \ldots, \langle\!\langle \boldsymbol{d_8} \rangle\!\rangle = 3 \right\} \end{array}$$

$$\mathbf{M}_1 \sqcup \mathbf{M}_2 \sqcup \mathbf{M}_3 = \left[\begin{array}{ccccccccc|c} 1 & 0 & 0 & 0 & 0 & -2 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 1 & -1 & 0 & 0 & 0 & 0 & 0 \end{array}\right]$$

$$\mathbf{M}_1 \sqcup \mathbf{M}_2 \sqcup \mathbf{M}_3 \sqcup \mathbf{M}_4 = \left[\begin{array}{ccccccccc|c} 1 & 0 & 0 & 0 & 0 & -2 & 0 & 0 & 0 & 0 \end{array}\right]$$

The system $\mathbf{M}_1 \sqcup \mathbf{M}_2 \sqcup \mathbf{M}_3 \sqcup \mathbf{M}_4$ then expresses the relationship $\langle\!\langle \boldsymbol{d}'_1 \rangle\!\rangle = 2 \cdot \langle\!\langle \boldsymbol{d}_5 \rangle\!\rangle$. In summary, each iteration of the algorithm involves the following steps: find a model of $\kappa \wedge \xi(\boldsymbol{X}) \wedge \eta \wedge \mu$ where μ ensures that the model is not already summarised by $\sqcup_{i=1}^{\ell} \mathbf{M}_i$; apply range refinement to maximise $\langle\!\langle \boldsymbol{d}'_1 \rangle\!\rangle$ whilst keeping $\langle\!\langle \boldsymbol{d}'_1 \rangle\!\rangle$, $\langle\!\langle \boldsymbol{d}_1 \rangle\!\rangle$, ..., $\langle\!\langle \boldsymbol{d}_8 \rangle\!\rangle$ fixed; join the resulting model with $\sqcup_{i=1}^{\ell} \mathbf{M}_i$ to give $\sqcup_{i=1}^{\ell+1} \mathbf{M}_i$.

To verify that $\langle\!\langle \boldsymbol{d}'_1 \rangle\!\rangle = 2 \cdot \langle\!\langle \boldsymbol{d}_5 \rangle\!\rangle$ is a fixed point, unlike before, it not sufficient to impose the disequality $\langle\!\langle \boldsymbol{d}'_1 \rangle\!\rangle \neq 2 \cdot \langle\!\langle \boldsymbol{d}_5 \rangle\!\rangle$ and check for unsatisfiability. This is because $\langle\!\langle \boldsymbol{d}'_1 \rangle\!\rangle$ is defined through maximisation. Instead the check amounts to testing whether $\kappa \wedge \xi(\boldsymbol{X}) \wedge \eta$ is unsatisfiable when combined with a formula encoding the strict inequality $\langle\!\langle \boldsymbol{d}'_1 \rangle\!\rangle > 2 \cdot \langle\!\langle \boldsymbol{d}_5 \rangle\!\rangle$ (note that if $\langle\!\langle \boldsymbol{d}'_1 \rangle\!\rangle > 2 \cdot \langle\!\langle \boldsymbol{d}_5 \rangle\!\rangle$ holds then it follows that $\langle\!\langle \boldsymbol{d}'_1 \rangle\!\rangle \neq 2 \cdot \langle\!\langle \boldsymbol{d}_5 \rangle\!\rangle$ holds). Since the combined system is unsatisfiable, we conclude that the update for this mode-combination includes $d'_1 = 2 \cdot d_5$. The complete affine update consists of:

$$\begin{array}{lcllcl} d'_1 & = & 2 \cdot d_5 & \qquad d'_5 & = & 2 \cdot d_5 + d_2 \\ d'_2 & = & d_2 & d'_6 & = & 2 \cdot d_6 + d_4 \\ d'_3 & = & 2 \cdot d_6 & d'_7 & = & 2 \cdot d_6 + d_2 \\ d'_4 & = & d_4 & d'_8 & = & 2 \cdot d_5 + d_4 \end{array}$$

This result is superior to that computed in Sect. 4.3. To illustrate, consider again an input octagon defined by $0 \leq \langle\!\langle \boldsymbol{r0} \rangle\!\rangle \leq 4$, $0 \leq \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq 1$ and $\langle\!\langle \boldsymbol{r0} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq 4$, hence:

$$\begin{array}{ll} d_1 = 4 & d_3 = 0 \\ d_2 = 1 & d_4 = 0 \\ d_5 = 4 & \end{array}$$

Applying the computed transformer to derive d'_5 on output gives:

$$d'_5 \quad = \quad 2 \cdot 4 + 1 \quad = \quad 9$$

Hence, we have $\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1'} \rangle\!\rangle \leq 9$, whereas the previously discussed technique based on applying the β map yields $\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1'} \rangle\!\rangle \leq 11$. Indeed, these linear symbolic update operations are optimal in the sense that if a symbolic output constant d'_j is equal to a linear function of the symbolic input constants $d_1, \ldots, d_8$, then that function will be derived.

Interestingly, Miné [56, Fig. 27] also discusses the relative precision of transfer functions, though where the base semantics is polyhedral rather than Boolean. Using his classification, the transfer functions derived using the synthesis techniques presented in Sect. 4.3 and Sect. 4.4 might be described as medium and exact. The following theorem confirms this intuition. For ease of presentation, the result states the exactitude of the update on the symbolic constant d'_1; analogous results hold for updates on $d'_2, \ldots, d'_8$.

Theorem 4.1. Suppose an octagonal update is derived of the form $\mathbf{M}\langle d'_1, d_1, \ldots, d_8, -1 \rangle = 0$. Moreover suppose that

- for all values of $\langle\!\langle \boldsymbol{r0} \rangle\!\rangle, \langle\!\langle \boldsymbol{r1} \rangle\!\rangle$ such that $\langle\!\langle \boldsymbol{r0} \rangle\!\rangle \leq d_1$, ..., $\langle\!\langle \boldsymbol{r0} \rangle\!\rangle - \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq d_8$ and $\xi(\boldsymbol{X})$ hold it follows that $\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle \leq c + c_1 \cdot d_1 + \ldots + c_8 \cdot d_8$ holds
- for all values of $\langle\!\langle \boldsymbol{r0} \rangle\!\rangle, \langle\!\langle \boldsymbol{r1} \rangle\!\rangle$ there exists a value of $\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle$ such that $\langle\!\langle \boldsymbol{r0} \rangle\!\rangle \leq d_1$, ..., $\langle\!\langle \boldsymbol{r0} \rangle\!\rangle - \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq d_8$, $\xi(\boldsymbol{X})$ and $\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle = c + c_1 \cdot d_1 + \ldots + c_8 \cdot d_8$ hold

Then $\mathbf{M}\langle d'_1, d_1, \ldots, d_8, -1 \rangle = 0 \models d'_1 = c + c_1 \cdot d_1 + \ldots + c_8 \cdot d_8$

Proof. Suppose that $\mathbf{M}$ is derived by $\mathbf{M} = \mathbf{M}_1 \sqcup \mathbf{M}_2 \sqcup \ldots \sqcup \mathbf{M}_\ell$. Suppose $\mathbf{M}_1$ is constructed from the model $\mathbf{m}_1 = \{d'_1 = v'_1, d_1 = v_1, \ldots, d_8 = v_8\}$ where the value v'_1 is maximal. Yet $\mathbf{m}_1$ is derived from a formula that encodes the equality $\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle = \langle\!\langle \boldsymbol{d'_1} \rangle\!\rangle$ where $\boldsymbol{d'_1}$ is a signed bit-vector representing d'_1. Since v'_1 is maximal it follows that the value of $\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle$ is maximal hence $(d'_1 = v'_1) \wedge (d_1 = v_1) \wedge \ldots \wedge (d_8 = v_8) \models d'_1 = c + c_1 \cdot d_1 + \ldots + c_8 \cdot d_8$ by the two assumptions. Therefore $\mathbf{M}_1\langle d'_1, d_1, \ldots, d_8, -1\rangle = 0 \models d'_1 = c + c_1 \cdot d_1 + \ldots + c_8 \cdot d_8$ since $\mathbf{M}_1 = [I \mid \langle v'_1, v_1, \ldots, v_8\rangle]$. Likewise $\mathbf{M}_i\langle d'_1, d_1, \ldots, d_8, -1\rangle = 0 \models d'_1 = c + c_1 \cdot d_1 + \ldots + c_8 \cdot d_8$ for all $1 \leq i \leq \ell$. The result follows since $\mathbf{M}$ is the least upper bound of $\mathbf{M}_1$, $\mathbf{M}_2$, …, $\mathbf{M}_\ell$ whereas $d'_1 = c + c_1 \cdot d_1 + \ldots + c_8 \cdot d_8$ is an upper bound. □

5. Deriving Updates with Templates

The previous section showed how linear equalities can be used to relate a symbolic constant of an inequality in the output octagon to the symbolic constants on the inequalities of the input octagon. In this section we develop complementary techniques for updates that cannot be characterised in this way. To illustrate the problem, Sect. 5.1 introduces an example which demonstrates why it can be propitious to base updates on symbolic bounds (range) constraints. Then, Sect. 5.2 refines this observation, demonstrating the role of octagonal inequalities in constructing updates, while Sect. 5.3 shows how equality constraints can be combined with auxiliary variables [3, 24, 60], to derive non-linear relationships between an output constant and the symbolic input constants. These techniques all share the use of templates, either in the syntactic form of the linear inequalities, or the terms that arise in the non-linear equalities.

5.1. **Bounds constraints.** To illustrate the problem with affine updates, consider the following code block:

$$1 : \texttt{AND R0 15;} \qquad 2 : \texttt{AND R1 15;} \qquad 3 : \texttt{XOR R0 R1;} \qquad 4 : \texttt{ADD R0 R1;}$$

The operations `AND` and `XOR` are uni-modal; `ADD` is multi-modal but it only operates in the exact non-negative mode for this block. Since the `AND` instructions truncate to contents of `R0` and `R1` to the values stored in their low bytes (an operation which is non-linear), no affine relationship exists between the symbolic constants d_i that characterise the input octagon and those d'_i that characterise the output octagon. However, observe that it is still possible to find a bound on d'_1. In fact, range refinement, as detailed in Sect. 3.2.3, can be applied to maximise $\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle$ to infer $\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle \leq 30$, hence the update $d'_1 = 30$. Repeating this tactic for remaining the symbolic output constants yields:

$$\begin{array}{llllllll} d'_1 = 30 & & d'_3 = 0 & & d'_5 = 45 & & d'_7 = 0 \\ d'_2 = 15 & & d'_4 = 0 & & d'_6 = 0 & & d'_8 = 15 \end{array}$$

5.2. **Octagonal inequality constraints.** Ranges are merely a degenerate form of octagonal inequality, which suggests using octagons to relate an output d'_i to an input d_j. To illustrate this idea, consider the following code that rounds `R0` up the next multiple of 16:

$$1 : \texttt{MOV R1 R0;} \qquad 2 : \texttt{NEG R1;} \qquad 3 : \texttt{AND R0 15;} \qquad 4 : \texttt{ADD R0 R1;}$$

The instruction `NEG R1` computes the two's complement of `R1`, updating `R1` with the result. The instructions `NEG R1` and `ADD R0, R1` are multi-modal, thus consider the feasible mode in which both instructions are exact, the former being negative and the latter non-negative. To search for a relationship between d'_1 and d_1, the expression $\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle - \langle\!\langle \boldsymbol{r0} \rangle\!\rangle$ is maximised to infer $\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle - \langle\!\langle \boldsymbol{r0} \rangle\!\rangle \leq 15$, hence $\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle \leq \langle\!\langle \boldsymbol{r0} \rangle\!\rangle + 15$ thus the update $d'_1 = d_1 + 15$. Maximising the remaining expressions

$$\begin{array}{ll} & \langle\!\langle \boldsymbol{r0'} \rangle\!\rangle - (+\langle\!\langle \boldsymbol{r1} \rangle\!\rangle + \langle\!\langle \boldsymbol{r0} \rangle\!\rangle) \\ \langle\!\langle \boldsymbol{r0'} \rangle\!\rangle - (+\langle\!\langle \boldsymbol{r1} \rangle\!\rangle) & \langle\!\langle \boldsymbol{r0'} \rangle\!\rangle - (-\langle\!\langle \boldsymbol{r1} \rangle\!\rangle - \langle\!\langle \boldsymbol{r0} \rangle\!\rangle) \\ \langle\!\langle \boldsymbol{r0'} \rangle\!\rangle - (-\langle\!\langle \boldsymbol{r0} \rangle\!\rangle) & \langle\!\langle \boldsymbol{r0'} \rangle\!\rangle - (-\langle\!\langle \boldsymbol{r1} \rangle\!\rangle + \langle\!\langle \boldsymbol{r0} \rangle\!\rangle) \\ \langle\!\langle \boldsymbol{r0'} \rangle\!\rangle - (-\langle\!\langle \boldsymbol{r1} \rangle\!\rangle) & \langle\!\langle \boldsymbol{r0'} \rangle\!\rangle - (+\langle\!\langle \boldsymbol{r1} \rangle\!\rangle - \langle\!\langle \boldsymbol{r0} \rangle\!\rangle) \end{array}$$

in general, derives invariants of the form $d'_1 \leq d_2 + c$, ..., $d'_1 \leq d_7 + c$ where c is some constant, either of which can be strengthened to an equality and interpreted as an update. However, in this case, these additional updates do not yield any further useful bounds on d'_1. Observe too that some of the above expressions involve 3 variables, whereas some of expressions that bound d'_5, d'_6, d'_7 and d'_8 involve 4 variables, even though the updates are themselves octagonal. Completing this derivation for the above example yields:

$$\begin{array}{lcllcl} d'_1 & = & d_1 + 15 & d'_5 & = & d_1 + 30 \\ d'_2 & = & 15 & d'_6 & = & d_3 \\ d'_3 & = & d_3 & d'_7 & = & d_3 + 15 \\ d'_4 & = & 0 & d'_8 & = & d_1 + 15 \end{array}$$

5.3. **Non-linear equality constraints.** Relaxing the equalities to inequalities provides one degree of freedom for generalising updates; relaxing linear equalities to non-linear ones provides another. Polynomial extensions [60, Sect. 6] have been proposed for generalising linear equality analysis, and there is no reason why this technique cannot be adapted to the problem of deriving transfer functions.

5.3.1. *Generating non-linear equality constraints.* The idea is to augment the original variables in the block with fresh variables specifically introduced to denote non-linear terms. The terms are drawn from a finite language of templates that typically includes monomials up to a fixed degree. To illustrate, consider the following basic block which computes the location an offset relative to the start location of two-dimension array where the registers `R0` and `R1` represent the row and column coordinates (which are indexed from 0). Register `R2` represents row size; all registers are signed.

$$1 : \texttt{MUL R0 R2;} \qquad 2 : \texttt{ADD R0 R1;}$$

Assume the block is described as a Boolean formula $\varphi(\boldsymbol{X})$ and all operations are exact. As before, the values of `R0`, `R1` and `R2` on input are represented using bit-vectors $\boldsymbol{r0}$, $\boldsymbol{r1}$ and

$\boldsymbol{r2}$, whereas $\boldsymbol{r0'}$ denotes the value of `R0` on output. Passing $\varphi(\boldsymbol{X})$ to a solver yields a model $\mathbf{m}_1$ as before, say:

$$\mathbf{m}_1 = \{\ \langle\!\langle \boldsymbol{r0} \rangle\!\rangle = 2, \qquad \langle\!\langle \boldsymbol{r1} \rangle\!\rangle = 4, \qquad \langle\!\langle \boldsymbol{r2} \rangle\!\rangle = 3, \qquad \langle\!\langle \boldsymbol{r0'} \rangle\!\rangle = 10\ \}$$

Instead of directly representing these values as an affine system, auxiliary variables are introduced whose sole purpose is to represent some non-linear terms drawn from a set of templates. In this case, we have a set that contains the single non-linear term $\langle\!\langle \boldsymbol{r0} \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2} \rangle\!\rangle$, hence we introduce a fresh variable s defined as $s = \langle\!\langle \boldsymbol{r0} \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2} \rangle\!\rangle$. Since $\langle\!\langle \boldsymbol{r0} \rangle\!\rangle = 2$ and $\langle\!\langle \boldsymbol{r2} \rangle\!\rangle = 3$, it follows $s = 6$. With the variable ordering $\langle \boldsymbol{r0'}, \boldsymbol{r0}, \boldsymbol{r1}, \boldsymbol{r2}, s \rangle$ on columns, we obtain the following affine system $\mathbf{M}_1 \in \mathbb{Z}^{5\times 6}$ as follows:

$$\mathbf{M}_1 \quad = \quad \left[\begin{array}{ccccc|c} 1 & 0 & 0 & 0 & 0 & 10 \\ 0 & 1 & 0 & 0 & 0 & 2 \\ 0 & 0 & 1 & 0 & 0 & 4 \\ 0 & 0 & 0 & 1 & 0 & 3 \\ 0 & 0 & 0 & 0 & 1 & 6 \end{array}\right]$$

Now the procedure proceeds much like before. The formula $\varphi(\boldsymbol{X})$ is now augmented with the constraint $\langle\!\langle \boldsymbol{r0} \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2} \rangle\!\rangle \neq 6$, the resulting formula being denoted $\varphi'(\boldsymbol{X})$. (Propositional encodings have been suggested for systems of inequality constraints over polynomial terms whose size is quadratic in the number of symbols required to define the constraints [35, Theorem 7].) Passing $\varphi'(\boldsymbol{X})$ to a SAT solver yields a model $\mathbf{m}_2$:

$$\mathbf{m}_2 = \{\ \langle\!\langle \boldsymbol{r0} \rangle\!\rangle = 3, \qquad \langle\!\langle \boldsymbol{r1} \rangle\!\rangle = 4, \qquad \langle\!\langle \boldsymbol{r2} \rangle\!\rangle = 8, \qquad \langle\!\langle \boldsymbol{r0'} \rangle\!\rangle = 28\ \}$$

which implies that $s = 24$. The merge is then computed thus:

$$\begin{aligned} \mathbf{M}_1 \sqcup \mathbf{M}_2 \quad &= \quad \left[\begin{array}{ccccc|c} 1 & 0 & 0 & 0 & 0 & 10 \\ 0 & 1 & 0 & 0 & 0 & 2 \\ 0 & 0 & 1 & 0 & 0 & 4 \\ 0 & 0 & 0 & 1 & 0 & 3 \\ 0 & 0 & 0 & 0 & 1 & 6 \end{array}\right] \sqcup \left[\begin{array}{ccccc|c} 1 & 0 & 0 & 0 & 0 & 28 \\ 0 & 1 & 0 & 0 & 0 & 3 \\ 0 & 0 & 1 & 0 & 0 & 4 \\ 0 & 0 & 0 & 1 & 0 & 8 \\ 0 & 0 & 0 & 0 & 1 & 24 \end{array}\right] \\ &= \quad \left[\begin{array}{ccccc|c} 1 & -18 & 0 & 0 & 0 & -26 \\ 0 & 5 & 0 & -1 & 0 & 7 \\ 0 & 0 & 1 & 0 & 0 & 4 \\ 0 & 0 & 0 & 18 & -5 & 24 \end{array}\right] \end{aligned}$$

and the formula further augmented thus:

$$\varphi''(\boldsymbol{X}) = \varphi'(\boldsymbol{X}) \wedge (18 \cdot \langle\!\langle \boldsymbol{r2} \rangle\!\rangle - 5 \cdot \langle\!\langle \boldsymbol{r0} \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2} \rangle\!\rangle \neq 24)$$

This formula gives another model:

$$\mathbf{m}_3 = \{\ \langle\!\langle \boldsymbol{r0} \rangle\!\rangle = 2, \qquad \langle\!\langle \boldsymbol{r1} \rangle\!\rangle = 2, \qquad \langle\!\langle \boldsymbol{r2} \rangle\!\rangle = 2, \qquad \langle\!\langle \boldsymbol{r0'} \rangle\!\rangle = 6\ \}$$

From this we deduce $s = 4$ and hence obtain:

$$\begin{aligned}\mathbf{M}_1 \sqcup \mathbf{M}_2 \sqcup \mathbf{M}_3 &= \left[\begin{array}{ccccc|c} 1 & -18 & 0 & 0 & 0 & -26 \\ 0 & 5 & 0 & -1 & 0 & 7 \\ 0 & 0 & 1 & 0 & 0 & 4 \\ 0 & 0 & 0 & 18 & -5 & 24 \end{array}\right] \sqcup \left[\begin{array}{ccccc|c} 1 & 0 & 0 & 0 & 0 & 6 \\ 0 & 1 & 0 & 0 & 0 & 2 \\ 0 & 0 & 1 & 0 & 0 & 2 \\ 0 & 0 & 0 & 1 & 0 & 2 \\ 0 & 0 & 0 & 0 & 1 & 4 \end{array}\right] \\ &= \left[\begin{array}{ccccc|c} 1 & -18 & -2 & 0 & 0 & -34 \\ 0 & 10 & 1 & -2 & 0 & 18 \\ 0 & 0 & 4 & -18 & 5 & -8 \end{array}\right]\end{aligned}$$

By passing:

$$\varphi''(\boldsymbol{X}) \wedge (4 \cdot \langle\!\langle \boldsymbol{r1} \rangle\!\rangle - 18 \cdot \langle\!\langle \boldsymbol{r2} \rangle\!\rangle + 5 \cdot \langle\!\langle \boldsymbol{r0} \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2} \rangle\!\rangle) \neq -8$$

to the solver, we generate yet another model:

$$\mathbf{m}_3 = \{\ \langle\!\langle \boldsymbol{r0} \rangle\!\rangle = 4, \qquad \langle\!\langle \boldsymbol{r1} \rangle\!\rangle = 5, \qquad \langle\!\langle \boldsymbol{r2} \rangle\!\rangle = 2, \qquad \langle\!\langle \boldsymbol{r0'} \rangle\!\rangle = 13\ \}$$

Joining $\mathbf{M}_4$ with $\mathbf{M}_1 \sqcup \mathbf{M}_2 \sqcup \mathbf{M}_3$ then produces the system:

$$\mathbf{M}_1 \sqcup \mathbf{M}_2 \sqcup \mathbf{M}_3 \sqcup \mathbf{M}_4 = \left[\begin{array}{ccccc|c} 12 & -64 & 0 & -70 & 10 & -152 \\ 0 & 64 & -12 & 70 & -23 & 152 \end{array}\right]$$

Proceeding with one more iteration, we obtain the system:

$$\mathbf{M}_1 \sqcup \mathbf{M}_2 \sqcup \mathbf{M}_3 \sqcup \mathbf{M}_4 \sqcup \mathbf{M}_5 = \left[\begin{array}{ccccc|c} 1 & 0 & -1 & 0 & -1 & 0 \end{array}\right]$$

Since adding another disequality constraint yields an unsatisfiable system, the equation:

$$\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle = \langle\!\langle \boldsymbol{r1} \rangle\!\rangle + s = \langle\!\langle \boldsymbol{r1} \rangle\!\rangle + \langle\!\langle \boldsymbol{r0} \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2} \rangle\!\rangle$$

characterises the polynomial input-output relation implemented by this block. Observe that the total number of calls to a SAT solver is still linear in the number of overall variables — program variables plus auxiliary variables — due to linear chain lengths in affine spaces. Our prototype implementation written in Java on top of the [mc]square framework [73] and Sat4J [52] computes this update in no more than 0.25s.

5.3.2. *Lifting non-linear equality constraints to intervals.* Of course, as in the Sect. 4.1, the derived constraint relates neither internal bounds nor symbolic constants on the input and output octagons. One would expect that non-linear equality constraints can be straightforwardly lifted to updates using either the technique described in Sect. 4.3 or by using the maximisation technique explained in Sect. 4.4. However, this is not so.

To illustrate, consider polynomial extension applied to interval relations, and in particular the problem of lifting the above affine system to construct an update over the symbolic bounds $\boldsymbol{r0}_\ell$, $\boldsymbol{r0}_u$, $\boldsymbol{r1}_\ell$, $\boldsymbol{r1}_u$, $\boldsymbol{r2}_\ell$, $\boldsymbol{r2}_u$, $\boldsymbol{r0}'_\ell$ and $\boldsymbol{r0}'_u$. Observe that the original equation

$$\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle = \langle\!\langle \boldsymbol{r1} \rangle\!\rangle + \langle\!\langle \boldsymbol{r0} \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2} \rangle\!\rangle$$

gives rise to two updates:

$$\begin{aligned}\langle\!\langle \boldsymbol{r0}'_\ell \rangle\!\rangle &= \langle\!\langle \boldsymbol{r1}_\ell \rangle\!\rangle + \min\{\langle\!\langle \boldsymbol{r0}_\ell \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2}_\ell \rangle\!\rangle, \langle\!\langle \boldsymbol{r0}_\ell \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2}_u \rangle\!\rangle, \langle\!\langle \boldsymbol{r0}_u \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2}_\ell \rangle\!\rangle, \langle\!\langle \boldsymbol{r0}_u \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2}_u \rangle\!\rangle\} \\ \langle\!\langle \boldsymbol{r0}'_u \rangle\!\rangle &= \langle\!\langle \boldsymbol{r1}_u \rangle\!\rangle + \max\{\langle\!\langle \boldsymbol{r0}_\ell \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2}_\ell \rangle\!\rangle, \langle\!\langle \boldsymbol{r0}_\ell \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2}_u \rangle\!\rangle, \langle\!\langle \boldsymbol{r0}_u \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2}_\ell \rangle\!\rangle, \langle\!\langle \boldsymbol{r0}_u \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2}_u \rangle\!\rangle\}\end{aligned}$$

that involve, respectively, minimisation and maximisation operations. These operations are required because it is not until the symbolic bounds are instantiated that the relative sizes of the non-linear terms can be compared. (These comparisons are redundant for linear terms because they are monotonic.)

To present this transformation formally, let $\boldsymbol{V} \cup \boldsymbol{V}'$ denote the input and output variables, and S denote a set of templates (monomials) over the variables $\boldsymbol{V}$. Thus if $s \in S$ then $s = \Pi_{i=1}^{n} v_i$ for some $v_i \in \boldsymbol{V}$. We introduce a map $\mu(s) = \{\Pi_{i=1}^{n} w_i \mid w_i = v_{i\ell} \vee w_i = v_{iu}\}$ so that, for example, if $s = \langle\!\langle \boldsymbol{r0} \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2} \rangle\!\rangle$ then:

$$\mu(\langle\!\langle \boldsymbol{r0} \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2} \rangle\!\rangle) = \{\langle\!\langle \boldsymbol{r0}_\ell \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2}_\ell \rangle\!\rangle, \langle\!\langle \boldsymbol{r0}_\ell \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2}_u \rangle\!\rangle, \langle\!\langle \boldsymbol{r0}_u \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2}_\ell \rangle\!\rangle, \langle\!\langle \boldsymbol{r0}_u \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2}_u \rangle\!\rangle\}$$

Each of the polynomially extended equations take the form:

$$\lambda_{\boldsymbol{v}'} \cdot \boldsymbol{v}' \;=\; \textstyle\sum_{\boldsymbol{v} \in \boldsymbol{V}} \lambda_{\boldsymbol{v}} \cdot \boldsymbol{v} + \sum_{s \in \mathcal{S}} \lambda_s \cdot s + d$$

where $\boldsymbol{v}' \in \boldsymbol{V}'$, $\lambda_{\boldsymbol{v}'} \in \mathbb{N}$, and $\lambda_{\boldsymbol{v}} \in \mathbb{Z}$ for all $\boldsymbol{v} \in \boldsymbol{V}$, and $\lambda_s \in \mathbb{Z}$ for all $s \in S$.

We then replace each polynomially extended equation by a pair of equations as follows:

$$\begin{array}{rcl} \lambda_{\boldsymbol{v}'} \cdot \boldsymbol{v}'_\ell & = & \sum_{\boldsymbol{v} \in \boldsymbol{V}} \lambda_{\boldsymbol{v}} \cdot \beta(-\lambda_{\boldsymbol{v}}, \boldsymbol{v}) \;+\; \sum_{s \in \mathcal{S}} \lambda_s \cdot \gamma(-\lambda_s, s) \;+\; d \\ \lambda_{\boldsymbol{v}'} \cdot \boldsymbol{v}'_u & = & \sum_{\boldsymbol{v} \in \boldsymbol{V}} \lambda_{\boldsymbol{v}} \cdot \beta(\lambda_{\boldsymbol{v}}, \boldsymbol{v}) \;+\; \sum_{s \in \mathcal{S}} \lambda_s \cdot \gamma(\lambda_s, s) \;+\; d \end{array}$$

where β is defined as before (in Sect. 4.2) and γ transforms the monomials as follows:

$$\gamma(\lambda, s) \;=\; \begin{cases} \min(\mu(s)) & : \text{ if } \lambda < 0 \\ \max(\mu(s)) & : \text{ otherwise} \end{cases}$$

Note that linear terms are transformed in the same manner as before.

5.3.3. *Non-linear equality constraints and octagons.* The minimisation and maximisation terms that arise in interval updates suggest a tactic for inferring updates for octagons in the presence of non-linear terms. To illustrate with the above example, the construction proceeds by introducing fresh variables s_1 and s_2 defined such that:

$$s_1 = \max(d_1 \cdot d_3, d_1 \cdot d_6, d_5 \cdot d_3, d_5 \cdot d_6) \qquad s_2 = \min(d_1 \cdot d_3, d_1 \cdot d_6, d_5 \cdot d_3, d_5 \cdot d_6)$$

Then maximisation is interleaved with affine join, as detailed in Sect. 4.4, so as to derive updates between a d'_i variable, the d_j and the auxiliary s_1 and s_2 variables. By applying this technique the following transfer function is derived:

$$\left\{\begin{array}{rcl} \langle\!\langle \boldsymbol{r0} \rangle\!\rangle & \leq & d_1 \\ \langle\!\langle \boldsymbol{r1} \rangle\!\rangle & \leq & d_2 \\ \langle\!\langle \boldsymbol{r2} \rangle\!\rangle & \leq & d_3 \\ -\langle\!\langle \boldsymbol{r0} \rangle\!\rangle & \leq & d_4 \\ -\langle\!\langle \boldsymbol{r1} \rangle\!\rangle & \leq & d_5 \\ -\langle\!\langle \boldsymbol{r2} \rangle\!\rangle & \leq & d_6 \\ \hline \langle\!\langle \boldsymbol{r0} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1} \rangle\!\rangle & \leq & d_7 \\ -\langle\!\langle \boldsymbol{r0} \rangle\!\rangle - \langle\!\langle \boldsymbol{r1} \rangle\!\rangle & \leq & d_8 \\ -\langle\!\langle \boldsymbol{r0} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1} \rangle\!\rangle & \leq & d_9 \\ \langle\!\langle \boldsymbol{r0} \rangle\!\rangle - \langle\!\langle \boldsymbol{r1} \rangle\!\rangle & \leq & d_{10} \\ \hline \langle\!\langle \boldsymbol{r0} \rangle\!\rangle + \langle\!\langle \boldsymbol{r2} \rangle\!\rangle & \leq & d_{11} \\ -\langle\!\langle \boldsymbol{r0} \rangle\!\rangle - \langle\!\langle \boldsymbol{r2} \rangle\!\rangle & \leq & d_{12} \\ -\langle\!\langle \boldsymbol{r0} \rangle\!\rangle + \langle\!\langle \boldsymbol{r2} \rangle\!\rangle & \leq & d_{13} \\ \langle\!\langle \boldsymbol{r0} \rangle\!\rangle - \langle\!\langle \boldsymbol{r2} \rangle\!\rangle & \leq & d_{14} \\ \hline \langle\!\langle \boldsymbol{r1} \rangle\!\rangle + \langle\!\langle \boldsymbol{r2} \rangle\!\rangle & \leq & d_{15} \\ -\langle\!\langle \boldsymbol{r1} \rangle\!\rangle - \langle\!\langle \boldsymbol{r2} \rangle\!\rangle & \leq & d_{16} \\ -\langle\!\langle \boldsymbol{r1} \rangle\!\rangle + \langle\!\langle \boldsymbol{r2} \rangle\!\rangle & \leq & d_{17} \\ \langle\!\langle \boldsymbol{r1} \rangle\!\rangle - \langle\!\langle \boldsymbol{r2} \rangle\!\rangle & \leq & d_{18} \end{array}\right\} \quad \rightsquigarrow \quad \left\{\begin{array}{rcl} \langle\!\langle \boldsymbol{r0'} \rangle\!\rangle & \leq & d_2 + s_1 \\ \langle\!\langle \boldsymbol{r1'} \rangle\!\rangle & \leq & d_2 \\ \langle\!\langle \boldsymbol{r2'} \rangle\!\rangle & \leq & d_3 \\ -\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle & \leq & d_5 + s_2 \\ -\langle\!\langle \boldsymbol{r1'} \rangle\!\rangle & \leq & d_5 \\ -\langle\!\langle \boldsymbol{r2'} \rangle\!\rangle & \leq & d_6 \\ \hline \langle\!\langle \boldsymbol{r0'} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1'} \rangle\!\rangle & \leq & d_2 + s_1 + d_2 \\ -\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle - \langle\!\langle \boldsymbol{r1'} \rangle\!\rangle & \leq & d_5 + s_2 + d_5 \\ -\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1'} \rangle\!\rangle & \leq & d_5 + s_2 + d_2 \\ \langle\!\langle \boldsymbol{r0'} \rangle\!\rangle - \langle\!\langle \boldsymbol{r1'} \rangle\!\rangle & \leq & d_2 + s_1 + d_5 \\ \hline \langle\!\langle \boldsymbol{r0'} \rangle\!\rangle + \langle\!\langle \boldsymbol{r2'} \rangle\!\rangle & \leq & d_2 + s_1 + d_3 \\ -\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle - \langle\!\langle \boldsymbol{r2'} \rangle\!\rangle & \leq & d_5 + s_2 + d_6 \\ -\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle + \langle\!\langle \boldsymbol{r2'} \rangle\!\rangle & \leq & d_5 + s_2 + d_3 \\ \langle\!\langle \boldsymbol{r0'} \rangle\!\rangle - \langle\!\langle \boldsymbol{r2'} \rangle\!\rangle & \leq & d_2 + s_1 + d_6 \\ \hline \langle\!\langle \boldsymbol{r1'} \rangle\!\rangle + \langle\!\langle \boldsymbol{r2'} \rangle\!\rangle & \leq & d_{15} \\ -\langle\!\langle \boldsymbol{r1'} \rangle\!\rangle - \langle\!\langle \boldsymbol{r2'} \rangle\!\rangle & \leq & d_{16} \\ -\langle\!\langle \boldsymbol{r1'} \rangle\!\rangle + \langle\!\langle \boldsymbol{r2'} \rangle\!\rangle & \leq & d_{17} \\ \langle\!\langle \boldsymbol{r1'} \rangle\!\rangle - \langle\!\langle \boldsymbol{r2'} \rangle\!\rangle & \leq & d_{18} \end{array}\right\}$$

Thus, for example, if the octagonal describes a cube that is offset from the origin, namely $d_1 = d_2 = d_3 = 3$ and $d_4 = d_5 = d_6 = -2$, then bound on $\langle\langle \boldsymbol{r0'} \rangle\rangle$, denoted d'_1, is calculated by $d'_1 = d_2 + s_1 = 3 + \max(3 \cdot 3, 3 \cdot -2, -2 \cdot 3, -2 \cdot -2) = 12$.

6. Evaluating Transfer Functions

Thus far, we have described how to derive transfer functions for intervals and octagons where the functions are systems of guards paired with affine updates, without reference to how they are evaluated. In our previous work [13], the application of a transfer function amounted to solving a series of integer linear programs (ILPs). To illustrate, suppose a transfer function consists of a single guard g and update u pair and let c denote a system of octagonal constraints on the input variables. A single output inequality in the output system, c', such as $r0' + r1' \leq d'_5$, can be derived by maximising $r0' + r1'$ subject to the linear system $c \wedge g \wedge u$. To construct c' in its entirety requires the solution of $O(n^2)$ ILPs where n is the number of registers (or variables) in the block. Although steady progress has been made on deriving safe bounds for integer programs [63], a more attractive solution computationally would avoid ILPs altogether.

6.1. **A single guard and update pair.** Affine updates, as derived in Sect. 4.4, relate symbolic constants on the inequalities in the input octagon to those of the output octagon. These updates confer a different, simpler, evaluation model. To compute $r0' + r1' \leq d'_5$ in c' it is sufficient to compute $c \sqcap g$ [56] which is the octagon that describes the conjoined system $c \wedge g$. This can be computed in quadratic-time when g is a single inequality and in cubic-time otherwise [56]. The meet $c \sqcap g$ then defines values for the symbolic constants d_i, though these values may include $-\infty$ and ∞. The value of d'_5 is defined by its affine update, that is, as a weighted sum of the d_i values. If there is no affine update for d'_5, then its value defaults to ∞. If bounds have been inferred for output octagons, then the d'_i can possibly be refined with a tighter bound. This evaluation mechanism thus replaces ILP with arithmetic that is both conceptually simple and computationally efficient. This is significant since transfer functions are themselves computed many times during fixed point evaluation.

6.2. **A system of guard and update pairs.** The above evaluation procedure needs to be applied for each guard g and update u pair for which $c \sqcap g$ is satisfiable. Thus several output octagons may be derived for a single block. We do not prescribe how these octagons should be combined, for example, a disjunctive representation is one possibility [36]. However, the simplest tactic is undoubtedly to apply the merge operation for octagons [56] (though this entails closing the output octagons).

6.3. **A system of updates for template inequality constraints.** Evaluating an octagonal update represented as an affine equality as discussed in Sect. 6.1 is straightforward since each symbolic bound d'_i on output is characterised by exactly one linear equation. This is not necessarily the case if template inequality constraints have been applied to derive updates, as discussed in Sect. 5.2. Recall, for example, that inequalities of the form $d'_1 \leq d_2 + c, \ldots,$ $d'_1 \leq d_7 + c$ can arise, all of which potentially induce non-trivial bounds on d'_1. In general, a

symbolic constant d'_i in the output octagon might be related to the input symbolic constants $d_1, \ldots, d_n$ through a system of m inequalities:

$$\bigwedge_{j=1}^{m} \left(c_j d'_i \leq \sum_{k=1}^{n} c_{j,k} \cdot d_k \right)$$

where $c_j > 0$ otherwise the inequality does not bound d'_i from above and can thus be discarded. Although any of these inequality can be strengthened to an equality and interpreted as an update, it is more precise to compute:

$$d'_i = \min \left\{ \left\lfloor \left(\sum_{k=1}^{n} c_{j,k} \cdot d_k \right) / c_j \right\rfloor \middle| 1 \leq j \leq m \right\}$$

Therefore, in general, transfer function evaluation can involve the evaluation of several linear expressions for each symbolic constant in the output octagon.

7. Experiments

We have implemented the techniques described in this paper in Java using the Sat4J solver [52], so as to integrate with our analysis framework for machine code [73], called [mc]square, which is also coded in Java. All experiments were performed on a MacBook Pro equipped with a 2.6 GHz dual-core processor and 4 GB of RAM, but only a single core was used in our experiments.

To evaluate transfer function synthesis without quantifier elimination, Tab. 3 compares the results for intervals for different blocks of assembly code to those obtained using the technique described in [13]. This corresponds to the techniques presented in Sect. 3 and Sect. 4. Column *#instr* contains the number of instructions, whereas column *#bits* gives the bit-width. (The 8-bit and 32-bit versions of the AVR instruction sets are analogous.) Then, *#affine* presents the number of affine relations for each block. The columns *runtime* contain the runtime and the number of SAT instances. The overall runtime of the elimination-based algorithm [13] is given in column *old* (∞ is used for timeout, which is set to 30s). Transfer function synthesis for blocks of up to 10 instruction is evaluated, which is a typical size for microcontroller code. For these size blocks, we have never observed more than 10 feasible mode combinations.

7.1. **Comparison.** Using quantifier elimination, all instances could be solved in a reasonable amount of time for 8-bit instructions. However, only the small instances could be solved for 32 bits (and only then because the Boolean encodings for the instructions were minimised prior to the synthesis of the transfer functions). It is also important to appreciate that none of the timeouts was caused by the SAT solver; it was resolution that failed to produce results in reasonable time. By way of comparison, synthesising guards for different overflow modes requires most runtime in our new approach, caused by the fact that the number of SAT instances to be solved grows linearly with the number of bits and quadratically with the number of variables (the number of octagonal inequalities is quadratic in the number of variables). Computing the affine updates consumes only a fraction of the overall time. In terms of precision, the results coincide with those previously generated [13].

The block for `swap` is interesting since it consists of three consecutive exclusive-or instructions, for which there is no coupling between different bits of the same register. The

Table 3: Experimental results for synthesis of transfer functions

block	#instr	#affine	#bits	runtime			
				guards / #SAT	affine / #SAT	overall	old
`inc`	1	2	8	0.2s / 32	0.1s / 5	0.3s	0.2s
			32	0.5s / 128	0.2s / 5	1.0s	23.0s
`inc+shift`	2	3	8	0.3s / 48	0.1s / 8	0.4s	0.3s
			32	0.8s / 192	0.2s / 8	1.0s	∞
`swap`	3	1	8	—	0.1s / 3	0.1s	0.1s
			32	—	0.1s / 3	0.1s	0.2s
`inc+flip`	4	2	8	0.2s / 32	0.2s / 5	0.4s	0.5s
			32	0.9s / 128	0.3s / 5	1.2s	∞
`abs`	5	3	8	2.5s / 216	0.3s / 8	2.8s	0.8s
			32	6.5s / 792	0.3s / 8	6.8s	∞
`inc+abs`	6	3	8	2.6s / 216	0.3s / 8	2.9s	1.4s
			32	6.7s / 792	0.3s / 8	7.0s	∞
`sum+isign`	7	9	8	5.9s / 648	0.2s / 24	4.3s	4.5s
			32	19.7s / 2376	0.4s / 24	11.1s	∞
`exchange+ abs`	10	3	8	2.8s / 216	0.3s / 8	3.1s	9.5s
			32	7.2s / 792	0.3s / 8	7.5s	∞

block is also unusual in that it is uni-modal with vacuous guards. These properties make it ideal for resolution. Even in this situation, the new technique scales better. In fact, the Boolean formulae that we present to the solver are almost trivial by modern standards, the main overhead coming from repeated SAT solving rather than solving a single large instance. SAT4J does reuse clauses learnt in an earlier SAT instances, though it does not permit clauses to be incrementally added and rescinded which is useful when solving maximisation problems [13]. Thus the timings given above are very conservative; indeed SAT4J was chosen to maintain the portability of [MC]SQUARE rather than for raw performance. Nevertheless, these timings very favourably compare with those required to compute transfer functions for intervals using BDDs [65], where in excess of 24 hours is required for single 8-bit instructions. Our experiences [11, 66] with native solvers such as MINISAT, however, indicate that a tenfold speed-up can be achieved by replacing SAT4J.

7.2. **Deriving octagonal transfer functions.** The process of deriving octagonal transfer functions by lifting (Sect. 4.3) requires an imperceivable overhead compared to computing affine relations themselves, indeed it is merely syntactic rewriting. The runtimes required for inferring affine updates by alternating range refinement and affine join (Sect. 4.4), however, is typically 3 or 4 times slower than those of computing the guards; the number of symbolic constants on the output inequalities corresponds exactly to the number of input guards. Since the octagon on input consists of 8 guards, and so does the octagon on output, the worst case requires $16 + 1$ iterations of affine abstraction and refinement; a single iteration of refinement is no more expensive as in the cases given in Tab. 3, and the affine join has imperceivable impact. We have observed the full number of iterations is only needed for programs for which there is no affine relation between octagons on input and output. We

refrain from giving exact times for the affine updates since they were computed with Z3 [32] rather than SAT4J and thus are not directly comparable.

7.3. **Further optimisations.** Since transfer functions are program dependent, one could first use a simple form of range analysis [11, 23, 66] to over-approximate the ranges a register can assume. These ranges can be encoded in the formulae, thereby pruning out some mode-combinations. For example, it is rarely the case that the absolute value function is actually applied to the smallest representable integer.

8. Related Work

The problem of designing transfer functions for numeric domains is as old as the field of abstract interpretation itself [26], and even the technique of using primed and unprimed variables to capture and abstract the semantics of instructions and functions dates back to the thesis work of Halbwachs [43]. However, even for a fixed abstract domain, there are typically many ways of designing and implementing transfer functions. Cousot and Halbwachs [29, Sect. 4.2.1], for example, discussed several ways to realise a transfer function for assignments such as $x = y \times z$ in the polyhedral domain while abstracting integer division $x = y/z$ is an interesting study within itself [77].

The problem of handcrafting best transformers is particularly challenging and Granger [40] lamented the difficulty of devising precise transfer functions for linear congruences. However, it took more than a decade after Granger's work before it was observed that best transformers could automatically be constructed for domains of finite height [68]. Nevertheless, automatic abstraction (or the automatic synthesis of abstractions) has only recently become a practical proposition, due to emergence of robust decision procedures [13, 50, 57] and efficient quantifier elimination techniques [16, 51, 59].

8.1. **Generation of symbolic best transformers.** Transfer functions can always be found for domains of finite height using the method of Reps et al. [68], provided one is prepared to pay the cost of repeatedly calling a decision procedure or a theorem prover, possibly many times on each application of a transformer. This motivates applying a decision procedure in order to compute a best transformer offline, prior to the actual analysis [13, 50], so as to both simplify and speedup their application.

Our previous work [13] shows how bit-blasting and quantifier elimination can be applied to synthesise transformers for bit-vector programs. This work was inspired by that of Monniaux [57, 59] on synthesising transfer functions for piecewise linear programs. Although his approach extends beyond octagons [80], it is unclear how to express some instructions (such as bit-wise exclusive-or) in terms of linear constraints. Universal quantification, as used in both approaches, also appears in work on inferring linear template constraints [42]. There, Gulwani and his co-authors apply Farkas' lemma in order to transform universal quantification into existential quantification, albeit at the cost of completeness since Farkas' lemma prevents integral reasoning. However, crucially, neither Monniaux nor Gulwani et al. provide a way to model integer overflow and underflow. Our work explains how to systematically handle wrap-around arithmetic in the transfer function itself (without having to the revise the notion of abstraction [78]) whilst sidestepping quantifier elimination too.

Transfer functions for low-level code have been synthesised for intervals using BDDs [18] by applying interval subdivision where the extrema representing the interval are themselves

represented as bit-vectors [65]. If $g : [0, 2^8 - 1] \to [0, 2^8 - 1]$ is a unary operation on an unsigned byte, then its abstract transformer $f : D \to D$ on $D = \{\emptyset\} \cup \{[\ell, u] \mid 0 \leq \ell \leq u < 2^8\}$ can be defined recursively. If $\ell = u$ then $f([\ell, u]) = g(\ell)$ whereas if $\ell < u$ then $f([\ell, u]) = f([\ell, m - 1]) \sqcup f([m, u])$ where $m = \lfloor u/2^n \rfloor 2^n$ and $n = \lfloor \log_2(u - \ell + 1) \rfloor$. Binary operations can likewise be decomposed by repeatedly dividing squares into their quadrants. The 8-bit inputs, ℓ and u, can be represented as 8-bit vectors, as can the 8-bit outputs, so as to represent f with a BDD. This permits caching to be applied when f is computed, which reduces the time needed to compute a best transformer to approximately 24 hours for each 8-bit operation. It is difficult to see how this approach can be extended to blocks that involve many variables without a step-change in BDD performance.

The question of how to construct a best abstract transformer has also been considered in the context of Markov decision processes (MDPs) for which the first abstract interpretation framework has recently been developed [85]. The framework affords the calculation of both lower and upper bounds on reachability probabilities, which is novel. The work focuses on predicate abstraction [39], that have had some success with large MDPs, and seeks to answer the question of, for given set of predicates, what is the most precise abstract program that still is a correct abstraction. More generally, the work illustrates that the question of how to compute the best abstract transformer is pertinent even in a probabilistic setting.

8.2. **Modular Arithmetic.** The classical approach to handling overflows is to follow the application of a transfer function with overflow and underflow checks; program variables are considered to be unbounded for the purposes of applying the transfer function but then their sizes are considered and range tests and, if necessary, range adjustments are applied to model any wrapping. This approach has been implemented in the Astree analyzer [12, 28]. However, for convex polyhedra, it is also possible to revise the concretisation map to reflect truncation so as to remove the range tests from most abstract operations [19, 78]. Another choice is to deploy congruence relations [40, 41] where the modulus is a power of two so as to reflect the wrapping in the abstract domain itself [61]. This approach can be applied to find both relationships between different words [61] and the bits that constitute words [15, 49, 50] (the relative precision of these two approaches has recently been compared [33]). Bit-level models have been combined with range inference [11, 23], though neither of these works address relational abstraction nor transfer function synthesis.

Modular arithmetic can be modelled with case splitting by introducing a propositional variable that acts as a witness to an overflow. To illustrate, consider the 8-bit comparison $x + 100 \leq 10$ [51, Sect. 6.4]. To model overflow a witness $p \Leftrightarrow (x + 100 \leq 255)$ is defined, which is used to control case selection. Case selection is realised through two constraints defined by $p \Rightarrow (x + 100) \leq 10$ and $(\neg p) \Rightarrow ((x + 100) - 256) \leq 10$. Case-based axiomisations can even be used to model underflows and rounding-to-zero in IEEE-745 floating-point arithmetic as shown in [57, Sect. 4.5]. These ideas are similar in spirit to those given in this paper for decomposing a block into its modes which are selected by guards.

8.3. **Polynomial Relations.** The last decade has seen increasing interest in the derivation of polynomial invariants, with techniques broadly falling into two classes: methods that use algebraic techniques to operate directly over polynomials and methods that model polynomial invariants in a linear setting. The work of Colón [24] is representative of the latter, for he shows how polynomial relations of bounded degree can be derived using program

transformation. To illustrate, suppose a variable a is updated using the assignment $a = a+1$. A variable s is introduced to represent the non-linear term a^2 and the program is extended by replacing the assignment $a = a+1$ with the parallel assignment $\langle a, s\rangle = \langle a+1, s+2a+1\rangle$ so as to reflect the update on a to s. Linear invariants between a, s and the other variables in the transformed program then are reinterpreted as polynomial invariants. The idea of using nonlinear terms as additional independent variables also arises in the work of Bagnara et al. [3] who use convex polyhedra to represent polynomial cones of bounded degree and thereby derive polynomial inequalities. They reduce the loss of precision induced through linearisation by additional linear inequalities, which are included in the polyhedra to express redundant non-linear constraints. The idea of extending a vector of variables with non-linear terms also arises in the work of Müller-Olm and Seidl [60] who consider the complexity of inferring polynomial equalities up to a fixed degree. They represent an affine relation with a set of vectors that generate the space through linear combination. Extending this idea to variables that represent non-linear terms naturally leads to the notion of polynomial hull which is not dissimilar to the closure algorithm that is used in this paper for computing non-linear update functions.

Quantifier elimination has been proposed as a technique for inferring polynomial inequalities directly [47] in which the invariants are templates of polynomial inequalities with undetermined coefficients. Deriving coefficients for the templates amounts to applying quantifier elimination which can be computed using a parametric (or comprehensive) Gröbner basis construction [88]. This approach resonates with the technique proposed by Monniaux for inferring loop invariants [58]. Gröbner bases also arise in techniques for calculating invariants that are based on fixed point calculation [69, 70], the main advantage of this approach being that it does not assume any a priori bound on the degree of a polynomial as an invariant. Polynomial analysis has also been applied in the field of SAT-based termination analysis [35] using term rewriting [37, 83]. Their work provides techniques for encoding polynomial equality and inequality constraints in propositional Boolean logic.

8.4. **Procedure summaries.** Abstracting the effect of a procedure in a summary is a key problem in inter-procedural analysis [76] since it enables the effect of a call on abstract state to be determined without repeatedly tracing the call. The challenge posed by summaries is how they can be densely represented whilst supporting the function composition and function application. Gen/kill bit-vector problems [67] are amenable to efficient representation, though for other problems, such as that of tracking two variable equalities [62], it is better not to tabular the effect of a call directly. This is because if a transformer is distributive, then the lower adjoint of a transformer uniquely determines the transformer and, perhaps surprisingly, the lower adjoint can sometimes be represented more succinctly than the transformer itself.

Acceleration [38, 53, 54, 74] is attracting increasing interest as an alternative way of computing a summary of a procedure, or more exactly the loops that it contains. The idea is to track how program state changes on each loop iteration so as to compute the trajectory of these changes (in a computation is that akin to transitive closure) and hence derive, in a single step, a loop invariant that holds on all iterations of the loop.

Symbolic bounds, which are key to our transfer functions, also arise in a form of symbolic bounds analysis [71] that aspires to infer ranges on pointer and array index variables in terms of the parameters of a procedure. Lower and upper bounds on each program variable at each program point are formulated as linear functions of the parameters of the function where the coefficients are themselves parametric. The problem then amounts to inferring values for

these parametric coefficients. By assuming variables to be non-negative, inequalities between the symbolic bounds can be reduced to inequalities between the parametric coefficients, thereby reducing the problem to linear programming.

9. Concluding Discussion

9.1. **Synopsis.** This article discusses the problem of automatically computing transfer functions for programs whose semantics is defined over finite bit-vectors. The key aspect that distinguishes our work from existing techniques [13, 57, 59] is that it does not depend on quantifier elimination techniques at all. Although Boolean formulae presented in CNF initially appear attractive for this task because of the simplicity of universal quantifier elimination [13, Sect. 1.3], their real strength is the fact that they are discrete. This permits linear equalities and inequalities to be inferred by repeated (incremental) satisfiability testing, avoiding the need for quantifier elimination in the abstraction process entirely. Most notably, this technique sidesteps the complexity of binary resolution. The force of this observation is that it extends transfer function synthesis to architectures whose word size exceeds 8 bits, thereby strengthening the case for low-level code verification [6, 7, 8, 9, 17, 34, 73, 82].

9.2. **Future work.** The problem of synthesising transfer functions is not dissimilar to that of inferring ranking functions for bit-vector relations [25]. Given a path π with a transition relation $r_\pi(\boldsymbol{x}, \boldsymbol{x}')$, proving the existence of a ranking function amounts to solving the formula $\exists \boldsymbol{c} : \forall \boldsymbol{x} : \forall \boldsymbol{x}' : r_\pi(\boldsymbol{x}, \boldsymbol{x}') \to (p(\boldsymbol{c}, \boldsymbol{x}) < p(\boldsymbol{c}, \boldsymbol{x}'))$ where $p(\boldsymbol{c}, \boldsymbol{x})$ is a polynomial over the bit-vector $\boldsymbol{x}$ and $\boldsymbol{c}$ is a bit-vector of coefficients [25, Thm. 2]. However, if intermediate variables $\boldsymbol{y}$ are needed to express $r_\pi(\boldsymbol{x}, \boldsymbol{x}')$, $p(\boldsymbol{c}, \boldsymbol{x})$, $p(\boldsymbol{c}, \boldsymbol{x}')$ or $<$, then the formula actually takes the form $\exists \boldsymbol{c} : \forall \boldsymbol{x} : \forall \boldsymbol{x}' : \exists \boldsymbol{y} : \nu$ where $\exists \boldsymbol{y} : \nu$ is equisatisfiable to $r_\pi(\boldsymbol{x}, \boldsymbol{x}') \to (p(\boldsymbol{c}, \boldsymbol{x}) < p(\boldsymbol{c}, \boldsymbol{x}'))$. This formula is structurally similar to those solved in [13] by quantifier elimination, which begs the question of whether this problem — like that of transfer function synthesis — can be recast to avoid elimination altogether. We will also investigate whether transfer functions can be found, not only for sequences of instructions, but also for entire loops [47, 57]. Existing approaches for the specification of (least inductive) loop invariants rely on existential quantification [57, Sect. 3.4], and the natural question is thus whether a variation of the techniques proposed in this paper can annul this complexity.

An interesting open question is whether the techniques discussed in this paper can be further generalised to linear template constraints with variable coefficients. As discussed in Sect. 3.3, the dichotomic search can be applied to any template constraint of the form $\sum_{i=1}^{n} c_i \cdot v_i \leq d$, where $c_1, \ldots, c_n, d \in \mathbb{Z}$ are constants and $v_1, \ldots, v_n$ are variables. However, some interesting abstract domains used in program analysis — such as two variables per inequality [79, 80] — do not fall into this class. It is still unclear if and how such relations can be derived using binary search. It is also interesting to note that octagons derived using our approach are tightly closed [56, Def. 3]. Intuitively, this means that all hyperplanes defined through inequalities actually touch the enclosed volume. However, the octagons may contain redundant inequalities, which may negatively affect performance [2, Sect. 3.2]. It will therefore be interesting to evaluate if simplification is worthwhile [2, Sect. 6.1] and, if so, whether non-redundant octagons can be directly derived using SAT.

Acknowledgements. This collaboration was supported by a Royal Society International Joint Project grant, reference GP101405, and by a Royal Society travel grant, reference TG092357. The first author was supported, in part, by the DFG research training group 1298 Algorithmic Synthesis of Reactive and Discrete-Continuous Systems and the by the DFG Cluster of Excellence on Ultra-high Speed Information and Communication, German Research Foundation grant DFG EXC 89. The second author was funded, in part, by a Royal Society Industrial Fellowship, reference IF081178. We gratefully acknowledge the comments provided by the reviewers of the ESOP paper [14] on which this work is largely based, as well as the feedback provided by the reviewers of the SAS paper [13], and its predecessor the VMCAI paper [50], since it was the reviewers' critique that inspired this work. Finally, we thank Sebastian Biallas, Stefan Kowalewski, David Monniaux, Axel Simon and Harald Søndergaard for stimulating discussions.

References

[1] Atmel Products. AVR32 Architecture Manual, 2007. `http://www.atmel.com/`.
[2] R. Bagnara, P. M. Hill, and E. Zaffanella. Weakly-relational shapes for numeric abstractions: improved algorithms and proofs of correctness. *Formal Methods in System Design*, 35(3):279–323, 2009.
[3] R. Bagnara, E. Rodríguez-Carbonell, and E. Zaffanella. Generation of Basic Semi-algebraic Invariants Using Convex Polyhedra. In *SAS*, volume 3672 of *LNCS*, pages 19–34. Springer, 2005.
[4] C. Baier and J.-P. Katoen. *Principles of Model Checking*. The MIT Press, 2008.
[5] G. Balakrishnan. *WYSINWYX: What You See Is Not What You eXecute*. PhD thesis, Computer Sciences Department, University of Wisconsin, Madison, Wisconsin, USA, August 2007.
[6] G. Balakrishnan and T. Reps. WYSINWYX: What You See Is Not What You eXecute. *ACM Trans. Program. Lang. Syst.*, 32(6), 2010.
[7] S. Bardin and P. Herrmann. Structural Testing of Executables. In *ICST*, pages 22–31. IEEE Computer Society, 2008.
[8] S. Bardin and P. Herrmann. OSMOSE: Automatic Structural Testing of Executables. *Softw. Test., Verif. Reliab.*, 21(1):29–54, 2011.
[9] S. Bardin, P. Herrmann, J. Leroux, O. Ly, R. Tabary, and A. Vincent. The BINCOA Framework for Binary Code Analysis. In *CAV*, volume 6806 of *LNCS*, pages 165–170. Springer, 2011.
[10] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. *Handbook of Satisfiability*, chapter Satisfiability Modulo Theories, pages 737–797. IOS Press, 2009.
[11] E. Barrett and A. King. Range and Set Abstraction Using SAT. *Electronic Notes in Theoretical Computer Science*, 267(1):17–27, 2010.
[12] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *PLDI*, pages 196–207. ACM, 2003.
[13] J. Brauer and A. King. Automatic Abstraction for Intervals using Boolean Formulae. In *SAS*, volume 6337 of *LNCS*, pages 167–183. Springer, 2010.
[14] J. Brauer and A. King. Transfer Function Synthesis without Quantifier Elimination. In *ESOP*, volume 6602 of *LNCS*, pages 97–115. Springer, 2011.
[15] J. Brauer, A. King, and S. Kowalewski. Range Analysis of Microcontroller Code using Bit-Level Congruences. In *FMICS*, volume 6371 of *LNCS*, pages 82–98. Springer, 2010.
[16] J. Brauer, A. King, and J. Kriener. Existential Quantification as Incremental SAT. In *CAV*, volume 6806 of *LNCS*, pages 191–207. Springer, 2011.
[17] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. BAP: A Binary Analysis Platform. In *CAV*, volume 6806 of *LNCS*, pages 463–469. Springer, 2011.
[18] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.
[19] S. Bygde, B. Lisper, and N. Holsti. Fully Bounded Polyhedral Analysis of Integers with Wrapping. In *Third International Workshop on Numerical and Symbolic Abstract Domains*, 2011. To appear in Electronic Notes in Theoretical Computer Science.

[20] V. Chandru and J.-L. Lassez. Qualitative Theorem Proving in Linear Constraints. In *Verification: Theory and Practice*, volume 2772 of *LNCS*, pages 395–406. Springer, 2003.
[21] R. Clarisó and J. Cortadella. The octahedron abstract domain. In *SAS*, volume 3148 of *LNCS*, pages 312–327. Springer, 2004.
[22] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *TACAS*, volume 2988 of *LNCS*, pages 168–176. Springer, 2004.
[23] M. Codish, V. Lagoon, and P. J. Stuckey. Logic programming with satisfiability. *Theory and Practice of Logic Programming*, 8(1):121–128, 2008.
[24] M. Colón. Approximating the Algebraic Relational Semantics of Imperative Programs. In *SAS*, volume 3148 of *LNCS*, pages 296–311. Springer, 2004.
[25] B. Cook, D. Kroening, P. Rümmer, and C. Wintersteiger. Ranking Function Synthesis for Bit-Vector Relations. In *TACAS*, volume 6015 of *LNCS*, pages 236–250. Springer, 2010.
[26] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*, pages 238–252. ACM Press, 1977.
[27] P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *POPL*, pages 269–282. ACM Press, 1979.
[28] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Mine, D. Monniaux, and X. Rival. The Astrée analyser. In *ESOP*, volume 3444 of *LNCS*, pages 21–30. Springer, 2005.
[29] P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In *POPL*, pages 84–97. ACM Press, 1978.
[30] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Effciently computing static single assignment form and the control dependence graph. *ACM Transaction on Programming Languages and Systems*, pages 451–590, 1991.
[31] J. Davenport and J. Heintz. Real Quantifier Elimination is Doubly Exponential. *Journal of Symbolic Computation*, 5(1):29–35, 1988.
[32] L. M. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
[33] M. Elder, J. Lim, T. Sharma, T. Andersen, and T. W. Reps. Abstract Domains of Affine Relations. In *SAS*, volume 6887 of *LNCS*, pages 198–215. Springer, 2011.
[34] A. Flexeder, M. Petter, and H. Seidl. Side-Effect Analysis of Assembly Code. In *SAS*, volume 6887 of *LNCS*, pages 77–94. Springer, 2011.
[35] C. Fuhs, J. Giesl, A. Middeldorp, P. Schneider-Kamp, R. Thiemann, and H. Zankl. SAT Solving for Termination Analysis with Polynomial Interpretations. In *SAT*, volume 4501 of *LNCS*, pages 340–354. Springer, 2007.
[36] R. Giacobazzi and F. Ranzato. Optimal domains for disjunctive abstract interpretation. *Sci. Comput. Program.*, 32(1–3):177–210, 1998.
[37] J. Giesl, P. Schneider-Kamp, and R. Thiemann. Automatic Termination Proofs in the Dependency Pair Framework. In *IJCAR*, volume 4130 of *LNCS*, pages 281–286. Springer, 2006.
[38] L. Gonnord and N. Halbwachs. Combining Widening and Acceleration in Linear Relation Analysis. In *SAS*, volume 4134 of *LNCS*, pages 144–160. Springer, 2006.
[39] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV*, volume 1254 of *LNCS*, pages 72–83. Springer, 1997.
[40] P. Granger. Static Analysis of Arithmetical Congruences. *International Journal of Computer Mathematics*, 30(13):165–190, 1989.
[41] P. Granger. Static Analyses of Congruence Properties on Rational Numbers. In *SAS*, volume 1302 of *LNCS*, pages 278–292, 1997.
[42] S. Gulwani, S. Srivastava, and R. Venkatesan. Program Analysis as Constraint Solving. In *PLDI*, pages 281–292. ACM Press, 2008.
[43] N. Halbwachs. *Détermination Automatique de Relations Linéaires Vérifiées par les Variables d'un Programme*. PhD thesis, Universit'e Scientifique et Médicale de Grenoble, 1979. `http://www-verimag.imag.fr/~halbwach/bib.html`.
[44] W. H. Harrison. Compiler Analysis of the Value Ranges of Variables. *IEEE Transactions on Software Engineering*, SE-3(3):243–250, 1977.
[45] J. M. Howe and A. King. Logahedra: A New Weakly Relational Domain. In *ATVA*, volume 5799 of *LNCS*, pages 306–320. Springer, 2009.

[46] J. B. Kam and J. D. Ullman. Monotone Data Flow Analysis Frameworks. *Acta Informatica*, 7:305–317, 1997.
[47] D. Kapur. Automatically Generating Loop Invariants Using Quantifier Elimination. In *Deduction and Applications*, volume 05431. IBFI, 2005.
[48] M. Karr. Affine Relationships among Variables of a Program. *Acta Informatica*, 6:133–151, 1976.
[49] A. King and H. Søndergaard. Inferring Congruence Equations using SAT. In *CAV*, volume 5123 of *LNCS*, pages 281–293. Springer, 2008.
[50] A. King and H. Søndergaard. Automatic Abstraction for Congruences. In *VMCAI*, volume 5944 of *LNCS*, pages 197–213. Springer, 2010.
[51] D. Kroening and O. Strichman. *Decision Procedures*. Springer, 2008.
[52] D. Le Berre and A. Parrain. The Sat4j library. *Journal of Satisfiability, Boolean Modeling and Computation*, 7:59–64, 2010.
[53] J. Leroux and G. Sutre. Accelerated Data-Flow Analysis. In *SAS*, volume 4634 of *LNCS*, pages 184–199. Springer, 2007.
[54] J. Leroux and G. Sutre. Acceleration in Convex Data-Flow Analysis. In *FSTTCS*, volume 4855 of *LNCS*, pages 520–531. Springer, 2007.
[55] K. Marriott. Frameworks for Abstract Interpretation. *Acta Informatica*, 30(2):103–129, 1993.
[56] A. Miné. The Octagon Abstract Domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.
[57] D. Monniaux. Automatic Modular Abstractions for Linear Constraints. In *POPL*, pages 140–151. ACM Press, 2009.
[58] D. Monniaux. Automatic Modular Abstractions for Template Numerical Constraints. *Logical Methods in Computer Science*, 6(3), 2010.
[59] D. Monniaux. Quantifier Elimination by Lazy Model Enumeration. In *CAV*, volume 6174 of *LNCS*, pages 585–599. Springer, 2010.
[60] M. Müller-Olm and H. Seidl. A note on Karr's algorithm. In *ICALP*, volume 3142 of *LNCS*, pages 1016–1028. Springer, 2004.
[61] M. Müller-Olm and H. Seidl. Analysis of Modular Arithmetic. *ACM Trans. Program. Lang. Syst.*, 29(5), August 2007.
[62] M. Müller-Olm and H. Seidl. Upper Adjoints for Fast Inter-procedural Variable Equalities. In *ESOP*, volume 4960 of *LNCS*, pages 178–192. Springer, 2008.
[63] A. Neumaier and O. Shcherbina. Safe Bounds in Linear and Mixed-Integer Linear Programming. *Math. Program.*, 99(2):283–296, 2004.
[64] D. A. Plaisted and S. Greenbaum. A Structure-Preserving Clause Form Translation. *Journal of Symbolic Computation*, 2(3):293–304, September 1986.
[65] J. Regehr and A. Reid. HOIST: A System for Automatically Deriving Static Analyzers for Embedded Systems. *ACM SIGOPS Operating Systems Review*, 38(5):133–143, 2004.
[66] T. Reinbacher and J. Brauer. Precise Control Flow Reconstruction Using Boolean Logic. In *International Conference on Embedded Software (EMSOFT)*, 2011. `http://www.emsoft.org/`.
[67] T. Reps, S. Horwitz, and M. Sagiv. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Principles of Programming Languages*, pages 49–61. ACM, 1995.
[68] T. Reps, M. Sagiv, and G. Yorsh. Symbolic Implementation of the Best Transformer. In *VMCAI*, volume 2937 of *LNCS*, pages 252–266. Springer, 2004.
[69] E. Rodríguez-Carbonell and D. Kapur. An Abstract Interpretation Approach for Automatic Generation of Polynomial Invariants. In *SAS*, volume 3148 of *LNCS*, pages 280–295. Springer, 2004.
[70] E. Rodríguez-Carbonell and D. Kapur. Automatic Generation of Polynomial Invariants of Bounded Degree using Abstract Interpretation. *Sci. Comput. Program.*, 64(1):54–75, 2007.
[71] R. Rugina and M. C. Rinard. Symbolic Bounds Analysis of Pointers, Array Indices, and Accessed Memory Regions. *ACM Trans. Program. Lang. Syst.*, 27(2):185–235, 2005.
[72] S. Sankaranarayanan, H. Sipma, and Z. Manna. Constraint based linear relations analysis. In *SAS*, volume 3148 of *LNCS*, pages 53–68. Springer, 2004.
[73] B. Schlich. Model Checking of Software for Microcontrollers. *ACM Trans. Embed. Comput. Syst.*, 9(4):1–27, 2010.
[74] P. Schrammel and B. Jeannet. Logico-Numerical Abstract Acceleration and Application to the Verification of Data-Flow Programs. In *SAS*, volume 6887 of *LNCS*, pages 233–248. Springer, 2011.

[75] A. Sepp, B. Mihaila, and A. Simon. Precise Static Analysis of Binaries by Extracting Relational Information. In *WCRE*. IEEE Digital Library, 2011. To appear.
[76] M. Sharir and A. Pnueli. Two Approaches to Interprocedural Data Flow Analysis. In *Program Flow Analysis: Theory and Applications*, pages 189–234. Prentice Hall, 1981.
[77] A. Simon. *Value-Range Analysis of C Programs*. Springer, August 2008.
[78] A. Simon and A. King. Taming the Wrapping of Integer Arithmetic. In *SAS*, volume 4634 of *LNCS*, pages 121–136. Springer, 2007.
[79] A. Simon, A. King, and J. M. Howe. Two Variables per Linear Inequality as an Abstract Domain. In *LOPSTR*, volume 2664 of *LNCS*, pages 71–89. Springer, 2002.
[80] A. Simon, A. King, and J. M. Howe. The Two Variable Per Inequality Abstract Domain. *Higher-Order and Symbolic Computation*, 23(1):87–143, 2010.
[81] T. Sturm and A. Tiwari. Verification and Synthesis using Real Quantifier Elimination. In *International Symposium on Symbolic and Algebraic Computation*, pages 329–336. ACM Press, 2011.
[82] A. V. Thakur, J. Lim, A. Lal, A. Burton, E. Driscoll, M. Elder, T. Andersen, and T. W. Reps. Directed Proof Generation for Machine Code. In *CAV*, volume 6174 of *LNCS*. Springer, 2010.
[83] R. Thiemann and J. Giesl. Size-Change Termination for Term Rewriting. In *RTA*, volume 2706 of *LNCS*, pages 264–278. Springer, 2003.
[84] G. S. Tseitin. On the complexity of derivation in the propositional calculus. In A. O. Slisenko, editor, *Studies in Constructive Mathematics and Mathematical Logic*, volume Part II, pages 115–125, 1968.
[85] B. Wachter and L. Zhang. Best Probabilistic Transformers. In *VMCAI*, volume 5944 of *LNCS*, pages 362–379. Springer, 2010.
[86] H. S. Warren. *Hacker's Delight*. Addison-Wesley, 2003.
[87] V. Weispfenning. The Complexity of Linear Problems in Fields. *Journal of Symbolic Computation*, 5(1–2):3–27, 1988.
[88] V. Weispfenning. Comprehensive Gröbner Bases. *Journal of Symbolic Computation*, 14(1):1–30, 1992.
[89] J. Whittemore, J. Kim, and K. A. Sakallah. SATIRE: A new incremental satisfiability engine. In *DAC*, pages 542–545. ACM, 2001.