# Transfer Function Synthesis without Quantifier Elimination

Jörg Brauer[1] and Andy King[2]

[1] Embedded Software Laboratory, RWTH Aachen University, Germany
[2] Portcullis Computer Security Limited, Pinner, UK

**Abstract.** Recently it has been shown how transfer functions for linear template constraints can be derived for bit-vector programs by operating over propositional Boolean formulae. The drawback of this method is that it relies on existential quantifier elimination, which induces a computational bottleneck. The contribution of this paper is a novel method for synthesising transfer functions that does not rely on quantifier elimination. We demonstrate the practicality of the method for generating transfer functions for both intervals and octagons.

## 1 Introduction

In model checking [3] the behaviour of a system is formally specified with a 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 representing states of a program symbolically, as Boolean functions, which enables states that share commonality to be represented without duplicating their commonality.

The key idea in abstract interpretation [10] is to abstract 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. If the number of classes is small, then all the paths through the program can be examined without incurring the problems of state-space explosion. When carefully constructed, the classes of states can preserve sufficient information to prove correctness. However, sometimes so much detail is lost when working with classes that the technique cannot infer useful information. This is because it critically depends on the expressiveness of the classes and the class transformers chosen to model the instructions that arise in the program. Class transformers are traditionally known as transfer functions [16]; they express how input states are mapped to output states by an instruction. If an input state is described by a class, then the transfer function is required to simulate the execution of the instruction by computing a class which faithfully describes the output state. Constructing transfer functions is difficult, especially when the instructions are low-level and operate over finite machine arithmetic. This is

because classes are themselves expressed as high-level geometric concepts such as affine and polyhedral spaces, presenting a semantic gap that has to be bridged.

The seminal work of Reps and his colleagues [29] advocated the automatic synthesis of transfer functions, though recently the topic has attracted increasing attention [6, 19, 23] because of the desire to generate transfer functions for blocks rather than single instructions. This can improve precision when there is a close coupling between the operations that constitute a block [19], which is often the case when recovering high-level semantics from assembly code, for instance, recovering a 16-bit addition from two consecutive 8-bit additions with carry.

### 1.1 Deriving transfer functions by quantifier elimination

Boolean formulae are germane to the problem of transfer function synthesis since the semantics of blocks are naturally represented as input-output relations — a technique that is colloquially referred to as bit-blasting [7]. This formulation of the semantics of a block dovetails with the quantifier-based approach [23] to transfer function synthesis since, when a formula is presented in CNF, existential and universal quantifier elimination can be realised straightforwardly [20].

To illustrate the role of quantification, suppose a formula models a block that mutates a single register whose values on entry and exit are represented by bit-vectors $\boldsymbol{x}$ and $\boldsymbol{x}'$, respectively. To derive a transfer function for interval analysis, it is necessary to ascertain how the maximal value of $\boldsymbol{x}'$, denoted $\boldsymbol{x}'_u$, relates to the minimal and maximal values of $\boldsymbol{x}$, denoted $\boldsymbol{x}_\ell$ and $\boldsymbol{x}_u$. The value of $\boldsymbol{x}'_u$ can be specified in logic [6, 23] by asserting that: (i) for every value of $\boldsymbol{x}$ that falls in the interval $[\boldsymbol{x}_\ell, \boldsymbol{x}_u]$, the value of $\boldsymbol{x}'_u$ is greater or equal to $\boldsymbol{x}'$, and (ii) for some value of $\boldsymbol{x}$ in $[\boldsymbol{x}_\ell, \boldsymbol{x}_u]$, the output $\boldsymbol{x}'$ takes the value of $\boldsymbol{x}'_u$. The "for some" can be expressed with existential quantification, but the "for every" can only be expressed with universal quantification. By applying quantifier elimination, a direct relationship between $\boldsymbol{x}_\ell$, $\boldsymbol{x}_u$, and $\boldsymbol{x}'_u$ can be found, yielding a mechanism for computing $\boldsymbol{x}'_u$ in terms of $\boldsymbol{x}_l$ and $\boldsymbol{x}_u$.

### 1.2 The drawbacks of quantifier elimination

Transfer function synthesis thus involves eliminating quantified variables from $\forall \boldsymbol{y} : \varphi$ where $\varphi$ is a system of propositional constraints and $\boldsymbol{y}$ is a tuple of variables. When $\varphi$ is propositional, a CNF formula $\psi$ that is equisatisfiable, denoted $\equiv$, to $\varphi$ can be straightforwardly found [27] by introducing fresh variables, denoted $\boldsymbol{z}$, to give $\varphi \equiv \exists \boldsymbol{z} : \psi$. The transfer function synthesis problem then amounts to solving $\forall \boldsymbol{y} : \exists \boldsymbol{z} : \psi$ where $\psi$ is in CNF. Alternating quantifiers also arise when transfer functions are synthesised from piece-wise linear constraints [23].

To eliminate existentially quantified variables, resolution [20, Chap. 9.2.3] is applied, which may be prohibitively expensive: 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 $\varphi$ 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 [6]. Thus far, the complexity of resolution has thwarted the wider applicability of this technique, even when applied carefully, which motivates the search for alternative methods.

### 1.3 Avoiding quantifier elimination

Our contribution is to eliminate the need for existential quantifier elimination altogether and replace resolution with successive calls to a SAT solver, where the number of calls grows linearly with the word-size. To illustrate, consider an octagon [22] 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 predicate.

As an example, consider the inequality $x + y \leq d$. The constant $d$ is characterised as $d = \min\{c \in \mathbb{Z} \mid \forall \boldsymbol{x} : \forall \boldsymbol{y} : P(\boldsymbol{x}, \boldsymbol{y}) \wedge (\boldsymbol{x} + \boldsymbol{y} \leq c)\}$ where $P(\boldsymbol{x}, \boldsymbol{y})$ is a predicate constraining the values of bit-vectors $\boldsymbol{x}$ and $\boldsymbol{y}$. Further, given a machine with word-length $w$, the maximal value in an unsigned representation is given as $2^w - 1$, 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:

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

To determine which disjunct characterises $d$, it is sufficient to test the formula $\exists \boldsymbol{x} : \exists \boldsymbol{y} : P(\boldsymbol{x}, \boldsymbol{y}) \wedge (\boldsymbol{x} + \boldsymbol{y} \geq 2^w)$ for *satisfiability*. If satisfiable, then $\mu_u$ is entailed by $d$, and $\mu_\ell$ otherwise. We proceed by decomposing the new characterisation into a disjunction and repeating this step $w$ times to give $d$ exactly. When a transfer function is formulated as a system of guarded affine updates [6, Sect. 2] then this range refinement technique can be applied to synthesise guards on the input values of variables.

The second contribution is to finesse the need for quantifier elimination in the generation of the input-output transformers that constitute the updates of the transfer functions. 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. Operationally, an update is applied to those inputs that satisfy the respective guard; the update details how the bounds of an input interval are mapped to new bounds of an output interval. In the case of octagons, the update maps the constants on the input octagonal inequalities to new constants on the output inequalities. Deriving updates for octagons requires range refinement to be interleaved with affine abstraction, which represents a third contribution. As a fourth contribution, we suggest a simple way of evaluating these transfer functions.

### 1.4 Outline of the approach

Overall, the paper proposes a systematic technique for inferring transfer functions that are defined as systems of guarded updates. Transfer functions are inferred

for a block by modeling each instruction as one of (at most) three Boolean functions, according to whether it overflows, underflows or does neither (is exact). 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 describes one type of wrapping (or non-wrapping) behaviour that can be realised within the block. The formula is then used to distill a guard paired with an update; one pair is computed for each feasible mode combination.

The guard, which is the optimal octagonal abstraction of the formula, is constructed one octagonal constraint at a time, by applying a form of dichotomy search, which amounts to a series of calls a SAT solver, as is explained in Sect. 2. The update component of the pair specifies how, when the guard is satisfied, the constraints in an input octagon are mapped to constraints in the output octagon (or in the degenerative case how to adjust bounds on intervals). Computing the update amounts to inferring a relationship between the bound on an output constraint and the bounds on the input constraints. Such a relationship can again be derived by repeated SAT solving, as detailed in Sect. 3. Replicating this construction for each of the output constraints gives the update operation for the feasible mode combination.

All these techniques are illustrated for blocks of 32-bit AVR UC3 assembly code [1], though the techniques are completely generic. We present experimental evidence in Sect. 5 which shows that the techniques presented in the paper are able to synthesise transfer functions for blocks where previous approaches based on quantifier elimination were prohibitively expensive. Sect. 6 surveys the related work and Sect. 7 concludes.

## 2 Deriving Guards

We express the concrete semantics of a block with Boolean formulae. 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 [6] for the synthesis of guards.

### 2.1 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 $i^{\text{th}}$ element of $\boldsymbol{r0}$. 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). The semantics of these two modes can be expressed as two formulae:

$$(1) \quad \varphi_O(\boldsymbol{X}) = \varphi(\boldsymbol{X}) \wedge \left(\bigwedge_{i=0}^{31} \boldsymbol{r0}[i]\right)$$
$$(2) \quad \varphi_E(\boldsymbol{X}) = \varphi(\boldsymbol{X}) \wedge \left(\bigvee_{i=0}^{31} \neg \boldsymbol{r0}[i]\right)$$

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 CNF by introducing fresh variables $\boldsymbol{z}$. We therefore denote the resulting formulae by $\varphi_E(\boldsymbol{X}, \boldsymbol{z})$ and $\varphi_O(\boldsymbol{X}, \boldsymbol{z})$. Following our initial approach [6], 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:

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

To obtain these guards, we solve a series of SAT instances, rather than following a monolithic all-in-one approach based on quantifier elimination [6]. To illustrate, consider the computation of a least upper bound $d$ for $\langle \boldsymbol{r0} \rangle$ for the formula $\varphi_E(\boldsymbol{X}, \boldsymbol{z})$. We start by putting:

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

As $2^{31}$ is a power of two, we can finesse the need for a complicated Boolean encoding of the predicate $\geq$ by using the equivalent formula:

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

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:

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

for satisfiability, or equivalently $\varphi_E^{\mathsf{simp},2}(\boldsymbol{X}, \boldsymbol{z}) = \varphi_E(\boldsymbol{z}) \wedge \boldsymbol{r0}[31] \wedge \boldsymbol{r0}[30]$. 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.

### 2.2 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 following fragment:

```
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;
```

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. Note that only overflows occurred in the previous example since the single operand was unsigned.

**Finding the feasible mode-combinations** In what follows, let $\mu(\boldsymbol{X})$ 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. 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 $\langle\!\langle \boldsymbol{x} \rangle\!\rangle = (\sum_{i=0}^{w-2} 2^i \cdot \boldsymbol{x}[i]) - 2^{w-1} \cdot \boldsymbol{x}[w-1]$ denote the value of $\boldsymbol{x}$ where $\boldsymbol{x}[31]$ is interpreted as the sign-bit. Then, `ADD R0, R1` has three modes of operation: overflow, underflow and exact operation. 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 semantics of these modes can be expressed as three Boolean formulae:

$$\mu_O(\boldsymbol{X}) = \mu(\boldsymbol{X}) \wedge \neg \boldsymbol{r0}[31] \wedge \neg \boldsymbol{r1}[31] \wedge \boldsymbol{r0'}[31]$$
$$\mu_U(\boldsymbol{X}) = \mu(\boldsymbol{X}) \wedge \boldsymbol{r0}[31] \wedge \boldsymbol{r1}[31] \wedge \neg \boldsymbol{r0'}[31]$$
$$\mu_E(\boldsymbol{X}) = \mu(\boldsymbol{X}) \wedge (\boldsymbol{r0}[31] \vee \boldsymbol{r1}[31] \vee \neg \boldsymbol{r0'}[31]) \wedge (\neg \boldsymbol{r0}[31] \vee \neg \boldsymbol{r1}[31] \vee \boldsymbol{r0'}[31])$$

The instruction `LSL R2` shifts `R2` to the left by one bit-position, and the most-significant bit is moved into the carry-flag. If the carry-flag is set, an overflow occurs. Let $\nu_O(\boldsymbol{X})$ and $\nu_E(\boldsymbol{X})$ thus express the overflow and exact modes of `LSL R2`. In an analogous way to the first `ADD`, let $\eta_O(\boldsymbol{X})$, $\eta_U(\boldsymbol{X})$ and $\eta_E(\boldsymbol{X})$ 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 $\eta_E(\boldsymbol{X})$ is infeasible. The above block constitutes $3 \cdot 2 \cdot 3$ combinations of modes, but only 6 of which are satisfiable. We thus have to derive guards only for the feasible combinations.

**Deriving guards for the feasible mode-combinations** Consider the case where (1) underflows, (4) overflows and (6) is exact, with the corresponding

formula denoted $\xi(\boldsymbol{X})$. 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 $\kappa$ 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 signed and $\kappa$ is extended to 34 bits to prevent wraps in the octagonal expression (cp. [9, 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. Satisfiability of $\psi^2(\boldsymbol{X})$ shows that $d \geq 2^{32}$. Following this strategy, the remaining instantiated formulae are unsatisfiable, and we thus infer the exact bound $\langle\!\langle \boldsymbol{d} \rangle\!\rangle = 2^{32}$. Rearranging $-\langle\!\langle \boldsymbol{r0} \rangle\!\rangle - \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq 2^{32}$ we obtain $-2^{32} \leq \langle\!\langle \boldsymbol{r0} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1} \rangle\!\rangle$. Using the same tactic, we derive $\langle\!\langle \boldsymbol{r0} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq -2^{31} - 1$. Repeating this tactic for all five feasible mode-combinations, we obtain the following optimal octagonal guards:

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

Redundant inequalities, which are themselves entailed by the given guards, are omitted for clarity of presentation.

**Complexity** A total of $4 \cdot 34 + 4 \cdot 33$ SAT instances is solved for each 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 [6], adding a single propositional variable to a formula can increase the complexity of resolution quadratically.

## 3 Deriving Updates

Transformers over template constraints have been previously formulated using quantification [6, 23]. 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. Note that Sect. 3.2 and Sect. 3.3 are just given for pedagogical purposes; only Sect. 3.4 provides a linear symbolic update operation that is optimal.

### 3.1 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. The process of deriving an affine abstraction follows the scheme given in [6, Sect. 3.2]. It starts with solving the formula $\xi(\boldsymbol{X})$, which produces a model $\mathbf{m}_1$ where:

$$\mathbf{m}_1 = \left\{ \langle\!\langle \boldsymbol{r0'} \rangle\!\rangle = -2^{31} \,, \quad \langle\!\langle \boldsymbol{r1'} \rangle\!\rangle = -1 \,, \quad \langle\!\langle \boldsymbol{r0} \rangle\!\rangle = -2^{31} + 1 \,, \quad \langle\!\langle \boldsymbol{r1} \rangle\!\rangle = -1 \right\}$$

We can equivalently write $\mathbf{m}_1$ as a matrix, denoted $\mathbf{M}_1$. With variable ordering $\langle \boldsymbol{r0'}, \boldsymbol{r1'}, \boldsymbol{r0}, \boldsymbol{r1} \rangle$ on columns, this gives:

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

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:

$$\mathbf{m}_2 = \left\{ \langle\!\langle \boldsymbol{r0'} \rangle\!\rangle = -2^{31} + 2 \,, \quad \langle\!\langle \boldsymbol{r1'} \rangle\!\rangle = -3 \,, \quad \langle\!\langle \boldsymbol{r0} \rangle\!\rangle = -2^{31} + 1 \,, \quad \langle\!\langle \boldsymbol{r1} \rangle\!\rangle = -3 \right\}$$

Joining $\mathbf{M}_1$ with $\mathbf{M_2}$, which is likewise obtained from $\mathbf{m}_2$, yields a matrix that describes that affine relations common to both models:

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

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:

$$\mathbf{m}_3 = \left\{ \langle\!\langle \boldsymbol{r0'} \rangle\!\rangle = -2^{31} \,, \quad \langle\!\langle \boldsymbol{r1'} \rangle\!\rangle = -4 \,, \quad \langle\!\langle \boldsymbol{r0} \rangle\!\rangle = -2^{31} + 4 \,, \quad \langle\!\langle \boldsymbol{r1} \rangle\!\rangle = -4 \right\}$$

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

$$\left[\begin{array}{cccc|c} 1 & 1 & 0 & 0 & -2^{31}-1 \\ 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & -2^{31}+1 \end{array}\right] \sqcup \left[\begin{array}{cccc|c} 1 & 0 & 0 & 0 & -2^{31} \\ 0 & 1 & 0 & 0 & -4 \\ 0 & 0 & 1 & 0 & -2^{31}+4 \\ 0 & 0 & 0 & 1 & -4 \end{array}\right] = \left[\begin{array}{cccc|c} 1 & 0 & 1 & 1 & -2^{32} \\ 0 & 1 & 0 & -1 & 0 \end{array}\right]$$

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

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

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 $\xi(\boldsymbol{X})$ [6, 19]. The resulting equations, however, express relationships between variables but not between symbolic 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.

**Complexity** Note that the chain-length in the affine domain is linear in the number of variables in the system [18]. 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.

### 3.2 Lifting affine equalities to interval updates

We explain how to transform $(\mathbf{M}_1 \sqcup \mathbf{M}_2) \sqcup \mathbf{M}_3$ over variables in $\boldsymbol{X}$ into an equation system over range boundaries. To do so, let $\boldsymbol{V} \subseteq \boldsymbol{X}$ denote the bit-vectors on entry of the block, and let $\boldsymbol{V}' \subseteq \boldsymbol{X}$ denote the bit-vectors on exit. Further, introduce fresh variables

$$\boldsymbol{V}_\ell = \{\boldsymbol{r0}_\ell, \boldsymbol{r1}_\ell\} \qquad \boldsymbol{V}_u = \{\boldsymbol{r0}_u, \boldsymbol{r1}_u\} \qquad \boldsymbol{V}'_\ell = \{\boldsymbol{r0}'_\ell, \boldsymbol{r1}'_\ell\} \qquad \boldsymbol{V}'_u = \{\boldsymbol{r0}'_u, \boldsymbol{r1}'_u\}$$

and if necessary transform the equations such that the left-hand side consists of only one variable in $\boldsymbol{V}'$. For the above equations, this gives:

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

These equations imply the following affine relations on interval boundaries:

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

To derive such as system, transform each of the original equations into the form $\lambda_{\boldsymbol{v}'} \cdot \boldsymbol{v}' = \sum_{\boldsymbol{v} \in \boldsymbol{V}} \lambda_{\boldsymbol{v}} \cdot \boldsymbol{v} + d$ 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:

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

The map $\beta : \mathbb{Z} \times \boldsymbol{V} \to (\boldsymbol{V}_\ell \cup \boldsymbol{V}_u)$ is defined as $\beta(\lambda, \boldsymbol{v}) = \boldsymbol{v}_\ell$ if $\lambda < 0$ and $\beta(\lambda, \boldsymbol{v}) = \boldsymbol{v}_u$ otherwise. 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 $\beta$. An analogous technique is applied when defining the lower bound. Applying this technique to all affine systems, we obtain the following five transfer functions (with the identity constraints on $\boldsymbol{r1}'_\ell$ and $\boldsymbol{r1}'_u$ omitted):

$$f_{O^{(1)},O^{(4)},U^{(6)}} = \begin{cases} (\langle\!\langle \boldsymbol{r0}' \rangle\!\rangle_\ell = -2^{31}) \wedge \\ (\langle\!\langle \boldsymbol{r0}' \rangle\!\rangle_u = -2^{31}) \end{cases}$$

$$f_{E^{(1)},E^{(4)},E^{(6)}} = \begin{cases} (\langle\!\langle \boldsymbol{r0}' \rangle\!\rangle_\ell = \langle\!\langle \boldsymbol{r0}_\ell \rangle\!\rangle + \langle\!\langle \boldsymbol{r1}_\ell \rangle\!\rangle) \ \wedge \\ (\langle\!\langle \boldsymbol{r0}' \rangle\!\rangle_u = \langle\!\langle \boldsymbol{r0}_u \rangle\!\rangle + \langle\!\langle \boldsymbol{r1}_u \rangle\!\rangle) \end{cases}$$

$$f_{U^{(1)},O^{(4)},E^{(6)}} = \begin{cases} (\langle\!\langle \boldsymbol{r0}' \rangle\!\rangle_\ell = -2^{32} - \langle\!\langle \boldsymbol{r0}_u \rangle\!\rangle - \langle\!\langle \boldsymbol{r1}_u \rangle\!\rangle) \wedge \\ (\langle\!\langle \boldsymbol{r0}' \rangle\!\rangle_u = -2^{32} - \langle\!\langle \boldsymbol{r0}_\ell \rangle\!\rangle - \langle\!\langle \boldsymbol{r1}_\ell \rangle\!\rangle) \end{cases}$$

$$f_{E^{(1)},O^{(4)},E^{(6)}} = \begin{cases} (\langle\!\langle \boldsymbol{r0}' \rangle\!\rangle_\ell = -\langle\!\langle \boldsymbol{r0}_u \rangle\!\rangle - \langle\!\langle \boldsymbol{r1}_u \rangle\!\rangle) \wedge \\ (\langle\!\langle \boldsymbol{r0}' \rangle\!\rangle_u = -\langle\!\langle \boldsymbol{r0}_\ell \rangle\!\rangle - \langle\!\langle \boldsymbol{r1}_\ell \rangle\!\rangle) \end{cases}$$

$$f_{O^{(1)},O^{(4)},E^{(6)}} = \begin{cases} (\langle\!\langle \boldsymbol{r0}' \rangle\!\rangle_\ell = 2^{32} - \langle\!\langle \boldsymbol{r0}_u \rangle\!\rangle - \langle\!\langle \boldsymbol{r1}_u \rangle\!\rangle) \wedge \\ (\langle\!\langle \boldsymbol{r0}' \rangle\!\rangle_u = 2^{32} - \langle\!\langle \boldsymbol{r0}_\ell \rangle\!\rangle - \langle\!\langle \boldsymbol{r1}_\ell \rangle\!\rangle) \end{cases}$$

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

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

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 $\langle\!\langle \boldsymbol{r0}'_u \rangle\!\rangle = -2^{31} + 20$.

### 3.3 Lifting affine equalities to octagonal updates

Consider deriving a transfer function for octagons for `ADD R0 R1; LSL R0` where `ADD` and `LSL` operate in exact 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:

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

We start by constructing an update operation that uses the unary input constraints only, as indicated by the bar separator. We modify the method presented

**Table 1.** 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 |

in Sect. 2.4 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:

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

The optimal octagonal abstraction, however, confers the constraints $\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle \leq 8$ 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 in linear time by lifting the affine relations. In what follows, we will describe how to derive more precise affine relations for the outputs.

### 3.4 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 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 $\kappa$ 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 $\eta$ 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:

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

which is fully detailed in Tab. 1. 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 $\zeta_1$ 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:

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

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 $\mu$ 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. 1. The model $\boldsymbol{m}_2$, itself, defines a formula $\zeta_2$ 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

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

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:

$$\boldsymbol{m}'_3 = \big\{ \langle\!\langle \boldsymbol{d'_1} \rangle\!\rangle = 26, \langle\!\langle \boldsymbol{d_1} \rangle\!\rangle = 8, \ldots, \langle\!\langle \boldsymbol{d_8} \rangle\!\rangle = 0 \big\}$$
$$\boldsymbol{m}'_4 = \big\{ \langle\!\langle \boldsymbol{d'_1} \rangle\!\rangle = 6, \langle\!\langle \boldsymbol{d_1} \rangle\!\rangle = 0, \ldots, \langle\!\langle \boldsymbol{d_8} \rangle\!\rangle = 3 \big\}$$

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

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

The system $\mathbf{M}_1 \sqcup \mathbf{M}_2 \sqcup \mathbf{M}_3 \sqcup \mathbf{M}_4$ then expresses the relationship $\langle\!\langle \boldsymbol{d'_1} \rangle\!\rangle = 2 \cdot \langle\!\langle \boldsymbol{d_5} \rangle\!\rangle$.

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 $\langle\!\langle \boldsymbol{d}'_1 \rangle\!\rangle > 2 \cdot \langle\!\langle \boldsymbol{d}_5 \rangle\!\rangle$. 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:

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

Observe that these linear symbolic update operations are optimal.

**Reflections on octagonal transfer functions** Interestingly, Miné [22, 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. 3.3 and 3.4 might be described as medium and exact.

### 3.5 Inferring bounds for octagons

For a final example, consider the following code block:

$1:$ `AND R0, 15;` $\quad 2:$ `AND R1, 15;` $\quad 3:$ `XOR R0, R1;` $\quad 4:$ `ADD R0, R1;`

The operations `AND` and `XOR` are uni-modal; `ADD` is multi-modal but it only operates in exact mode for this block. For this single mode 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, even in such cases, it can be still possible to find a $d'_1$ such that $\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle \leq d'_1$ by applying range refinement. This gives $d'_1 = 30$. Repeating this tactic for remaining the symbolic output constants yields:

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

## 4 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 [6], 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 [26], a more attractive solution computationally would avoid ILPs altogether.

### 4.1 A single guard and update pair

Affine updates, as derived in Sect. 3.4, 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 $c \sqcap g$ [22] 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 [22]. The meet $c \sqcap g$ then defines values for the symbolic constants $d_i$, though these values may include $-\infty$ and $\infty$. 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 $\infty$. If bounds have been inferred for output octagons (Sect 3.5), 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 fixpoint evaluation.

### 4.2 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 [13]. However, the simplest tactic is undoubtedly to apply the merge operation for octagons [22] (though this entails closing the output octagons).

## 5 Experiments

We have implemented the techniques described in this paper in Java using the Sat4J solver [21], so as to integrate with our analysis framework for machine code [30], 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. 2 compares the results for intervals for different blocks of assembly code to those obtained using the technique described in [6]. 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 [6] is given in column *old* ($\infty$ 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.

**Table 2.** Experimental results for synthesis of transfer functions

| block | #instr | #affine | #bits | runtime | | | |
|---|---|---|---|---|---|---|---|
| | | | | guards / #SAT | affine / #SAT | overall | old |
| `inc` | 1 | 2 | 8 | 0.2s / 40 | 0.1s / 5 | 0.3s | 0.2s |
| | | | 32 | 0.5s / 136 | 0.2s / 5 | 1.0s | 23.0s |
| `inc+shift` | 2 | 3 | 8 | 0.3s / 60 | 0.1s / 8 | 0.4s | 0.3s |
| | | | 32 | 0.8s / 216 | 0.2s / 8 | 1.0s | $\infty$ |
| `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 / 40 | 0.2s / 5 | 0.4s | 0.5s |
| | | | 32 | 0.9s / 216 | 0.3s / 5 | 1.2s | $\infty$ |
| `abs` | 5 | 3 | 8 | 2.5s / 216 | 0.3s / 8 | 2.8s | 0.8s |
| | | | 32 | 6.5s / 792 | 0.3s / 8 | 6.8s | $\infty$ |
| `inc+abs` | 6 | 3 | 8 | 2.6s / 216 | 0.3s / 8 | 2.9s | 1.4s |
| | | | 32 | 6.7s / 792 | 0.3s / 8 | 7.0s | $\infty$ |
| `sum+isign` | 7 | 5 | 8 | 4.1s / 360 | 0.2s / 18 | 4.3s | 4.5s |
| | | | 32 | 10.7s / 1320 | 0.4s / 18 | 11.1s | $\infty$ |
| `exchange+abs` | 10 | 3 | 8 | 2.8s / 216 | 0.3s / 8 | 3.1s | 9.5s |
| | | | 32 | 7.2s / 792 | 0.3s / 8 | 7.5s | $\infty$ |

**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 [6].

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 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 [6]. 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 [28], where in excess of 24 hours is required for single 8-bit instructions.

**Deriving octagonal transfer functions** The process of deriving octagonal transfer functions by lifting (Sect. 3.3) requires an imperceivable overhead compared to computing affine relations themselves, indeed it is merely syntactic rewriting. The runtimes required for inferring affine inequalities by alternating range refinement and affine join (Sect. 3.4), 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. (We refrain from giving exact times since this component has not been tuned.)

**Further optimisations** Since transfer functions are program dependent, one could first use a simple form of range analysis [5] 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.

## 6 Related Work

Although the problem of constructing transfer functions has been recognised for over twenty years, for example, by Cousot and Halbwachs [12] for the polyhedral domain and by Granger [14] for linear congruences, automatic synthesis has only recently become a practical proposition due to emergence of robust decision procedures [19, 29] and quantifier elimination techniques [20, 23, 24].

Transfer functions [29] can always be found for domains that satisfy the finite ascending chain condition, provided one is prepared to pay the cost of calling a decision procedure repeatedly on each application of a transformer. This motivates applying a decision procedure in order to compute optimal transfer functions offline, prior to the actual analysis [6, 19].

Our previous work [6] 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 [23, 24] on synthesising transfer functions for piecewise linear programs. Although his approach extends beyond octagons [32], it is unclear how to express some instructions (such as exclusive-or) in terms of linear constraints. Universal quantification, as used in both approaches, also appears in work on inferring linear template constraints by Gulwani et al. [15]. There, the 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. By way of contrast, our approach explains how to systematically handle wrap-around arithmetic in the transfer function itself whilst sidestepping quantifier elimination.

Transfer functions have been automatically synthesised for intervals using BDDs by applying interval subdivision [28]. If $g : [0, 2^8 - 1] \to [0, 2^8 - 1]$ is

a unary operation on an unsigned byte, then its 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. The 8-bit inputs, $\ell$ 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 one day for each 8-bit operation.

The classical approach to handling overflow is to verify that they do not occur using unbounded domains as implemented in the Astree tool [11]. However, for the domain of polyhedra, it is also possible to revise the concretisation map to reflect the effect of truncation [31]. Another choice is to deploy congruence relations [14] where the modulus is a power of two [19, 25]. Finally, bit-blasting has been combined with range inference elsewhere [5, 8], though neither of these papers address relational abstraction nor transfer function synthesis.

## 7 Concluding Discussion

**Synopsis** This paper revisits the problem of synthesising transfer functions for programs whose semantics is defined over finite bit-vectors. The irony is that although Boolean formula initially appear attractive for synthesis because of the simplicity of universal projection [6], their real strength is the fact that they are discrete. This permits octagonal inequalities to be inferred by repeated satisfiability testing, avoiding the need for quantifier elimination, and in particular the complexity of resolution. The force of this observation is that it extends transfer function synthesis to architectures whose word size exceeds 8 bits, strengthening the case for low-level code verification [4, 30].

**Future work** The problem of synthesising transfer functions is not dissimilar to that of inferring ranking functions for bit-vector programs [9]. The existence of a ranking function on a path $\pi$ with a transition relation $r_\pi(\boldsymbol{x}, \boldsymbol{x}')$ 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. 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 $\exists \boldsymbol{y} : \nu \equiv r_\pi(\boldsymbol{x}, \boldsymbol{x}') \to (p(\boldsymbol{c}, \boldsymbol{x}) < p(\boldsymbol{c}, \boldsymbol{x}'))$. This formula is similar to those solved in [6] by 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 [17, 23]. Existing approaches for the specification of (least inductive) loop invariants rely on existential quantification, and the natural question is whether the techniques proposed in this paper can annul this complexity. It is also interesting to note that octagons derived using our approach are tightly closed [22]. Intuitively, this means that all hyperplanes defined through inequalities actually touch the enclosed volume.

However, the octagons may contain redundant inequalities, and therefore it will be interesting to see if simplification is worthwhile [2] and, if so, whether non-redundant octagons can be directly derived using SAT.

*Acknowledgements* 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 travel grant, reference TG092357, and a Royal Society Industrial Fellowship, reference IF081178. We thank David Monniaux and Stefan Kowalewski for interesting technical discussions, as well as moral support, in this line of scientific enquiry.

## 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. C. Baier and J.-P. Katoen. *Principles of Model Checking*. The MIT Press, 2008.
4. G. Balakrishnan and T. Reps. WYSINWYX: What You See Is Not What You eXecute. *ACM Trans. Program. Lang. Syst.*, 32(6), 2010.
5. E. Barrett and A. King. Range and Set Abstraction Using SAT. *Electronic Notes in Theoretical Computer Science*, 267(1):17–27, 2010.
6. J. Brauer and A. King. Automatic Abstraction for Intervals using Boolean Formulae. In *SAS*, volume 6337 of *LNCS*, pages 167–183. Springer, 2010.
7. 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.
8. M. Codish, V. Lagoon, and P. J. Stuckey. Logic programming with satisfiability. *Theory and Practice of Logic Programming*, 8(1):121–128, 2008.
9. 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.
10. 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.
11. 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.
12. P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In *POPL*, pages 84–97. ACM Press, 1978.
13. R. Giacobazzi and F. Ranzato. Optimal domains for disjunctive abstract interpretation. *Sci. Comput. Program.*, 32(1–3):177–210, 1998.
14. P. Granger. Static Analysis of Arithmetical Congruences. *International Journal of Computer Mathematics*, 30(13):165–190, 1989.
15. S. Gulwani, S. Srivastava, and R. Venkatesan. Program Analysis as Constraint Solving. In *PLDI*, pages 281–292. ACM Press, 2008.
16. J. B. Kam and J. D. Ullman. Monotone Data Flow Analysis Frameworks. *Acta Informatica*, 7:305–317, 1997.

17. D. Kapur. Automatically Generating Loop Invariants Using Quantifier Elimination. In *Deduction and Applications*, volume 05431. IBFI, 2005.
18. M. Karr. Affine Relationships among Variables of a Program. *Acta Informatica*, 6:133–151, 1976.
19. A. King and H. Søndergaard. Automatic Abstraction for Congruences. In *VMCAI*, volume 5944 of *LNCS*, pages 197–213. Springer, 2010.
20. D. Kroening and O. Strichman. *Decision Procedures*. Springer, 2008.
21. D. Le Berre. SAT4J: Bringing the power of SAT technology to the Java platform, 2010. http://www.sat4j.org/.
22. A. Miné. The Octagon Abstract Domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.
23. D. Monniaux. Automatic Modular Abstractions for Linear Constraints. In *POPL*, pages 140–151. ACM Press, 2009.
24. D. Monniaux. Quantifier Elimination by Lazy Model Enumeration. In *CAV*, volume 6174 of *LNCS*, pages 585–599. Springer, 2010.
25. M. Müller-Olm and H. Seidl. Analysis of Modular Arithmetic. *ACM Trans. Program. Lang. Syst.*, 29(5), August 2007.
26. A. Neumaier and O. Shcherbina. Safe Bounds in Linear and Mixed-Integer Linear Programming. *Math. Program.*, 99(2):283–296, 2004.
27. D. A. Plaisted and S. Greenbaum. A Structure-Preserving Clause Form Translation. *Journal of Symbolic Computation*, 2(3):293–304, September 1986.
28. 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.
29. T. Reps, M. Sagiv, and G. Yorsh. Symbolic Implementation of the Best Transformer. In *VMCAI*, volume 2937 of *LNCS*, pages 252–266. Springer, 2004.
30. B. Schlich. Model Checking of Software for Microcontrollers. *ACM Trans. Embed. Comput. Syst.*, 9(4):1–27, 2010.
31. A. Simon and A. King. Taming the Wrapping of Integer Arithmetic. In *SAS*, volume 4634 of *LNCS*, pages 121–136. Springer, 2007.
32. A. Simon, A. King, and J. M. Howe. The Two Variable Per Inequality Abstract Domain. *Higher-Order and Symbolic Computation*. To appear.