Automatic Abstraction for Intervals using Boolean Formulae

Jörg Brauer[1] and Andy King[2]

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

Abstract. Traditionally, transfer functions have been manually designed for each operation in a program. Recently, however, there has been growing interest in computing transfer functions, motivated by the desire to reason about sequences of operations that constitute basic blocks. This paper focuses on deriving transfer functions for intervals — possibly the most widely used numeric domain — and shows how they can be computed from Boolean formulae which are derived through bit-blasting. This approach is entirely automatic, avoids complicated elimination algorithms, and provides a systematic way of handling wrap-arounds (integer overflows and underflows) which arise in machine arithmetic.

1 Introduction

The key idea in abstract interpretation [6] 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. 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 $d \propto c$ [17], 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.

1.1 The drive 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. 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 [11] in the context of verifying bit-twiddling code [14]. Since blocks are program dependent, such an approach relies on automation rather than human intervention.

Another compelling reason for automation is the complexity of the concrete operations themselves. 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, then the corner case inevitably manifests itself (if not in the transfer function itself then elsewhere [29]). The problem of deriving transfer functions for low-level instructions, such as those of the x86, is particularly acute [2] since these operations not only update registers and memory locations, but also side effect status flags. Automatic abstraction offers a way to potentially tame this complexity.

1.2 Specifying extreme values with universal quantifiers

Monniaux [20] recently addressed the vexing question of automatic abstraction by focussing on template domains [28] which include, most notably, intervals [7]. He showed that if the concrete operations are specified as piecewise linear functions, then it is possible to derive transfer functions for blocks. The transfer functions relate the values of variables on entry to a block to their values on exit. To illustrate, suppose the variables x, y and z occur in a block and consider the maximum value of x on exit from the block. Monniaux shows how quantification can be used to specify the maximal output value of x in terms of the extreme values that x, y and z can take on entry to the block. The specification states that: the maximal output value of x is an upper bound on all the output values of x that are feasible for the values of x, y and z that fall within their input ranges. It also asserts that: the maximal output value of x is smaller than any other upper bound on the output value of x. These requirements are naturally formulated with universal quantification. Universal quantifier elimination is then used to find a direct linear relationship between the maximal value of x on exit and the ranges of x, y and z on entry; it is direct in that intermediate variables that occur in the specification are removed. This construction is ingenious but no polynomial elimination algorithm is known for piecewise systems, or is ever likely to exist [3]. Indeed, this computational bottleneck remains a problem [21].

1.3 Finessing universal quantifiers with Boolean formulae

This paper suggests that as an alternative to operating over piecewise linear systems one can instead express the semantics of a basic block with a Boolean

formula; an idea that is familar in model checking where it is colloquially referred to as bit-blasting. Since Boolean formulae are more expressive than piecewise linear formulae, one would expect universal quantifier elimination to be just as difficult for Boolean formulae (or even harder since they are discrete). However, this is not so. To illustrate, consider $\forall x.f$ where $f = (x \vee \neg y) \wedge (\neg x \vee y \vee \neg z)$. Then $\forall x.f = f[x \mapsto 0] \wedge f[x \mapsto 1] = (\neg y) \wedge (y \vee \neg z)$. Observe that $\forall x.f$ can be obtained directly from f by removing the x and $\neg x$ literals from all the clauses of f. This is no coincidence and holds for any formula f presented in CNF not containing a vacuous clause that includes both x and $\neg x$ [15]. This suggests the following four step method for automatically deriving transfer functions for intervals (and related domains [18, 19]): First, use bit-vector logic 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 [15, Chap. 6]). Thus each n-bit integer variable is represented as a separate vector of n propositional variables. Second, apply the specification of Monniaux [20, Sect. 3.2] 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. Third, the universal quantifiers are removed from f_{spec} to obtain f_{simp} – a simplification of f_{spec}. Thus although universal qualification is a hinderance for linear piecewise functions, it is actually helps in a propositional formulation. Of course, f_{simp} is just a 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 [13] which directly relate the output ranges to the input ranges. This fourth step (which is analogous to that proposed for abstracting formulae with congruences [14]) is the final step in the construction. Overall, this new approach to computing transfer functions confers the following advantages:

- it is amenable to instructions whose semantics is presented as Boolean formulae. The force of this is that propositional encodings are readily available for instructions, due to the rise in popularity of SAT-based model checking. Moreover, it is not obvious how to express the semantics of some (bit-level) instructions with piecewise linear functions;
- it avoids the computational problems associated with eliminating variables from piecewise linear systems;
- it distills transfer functions from Boolean formulae that are action systems of guarded updates. The guards are systems of octagonal constraints [19]. A guard tests whether a particular behaviour can arise, for example, whether an operation wraps, and the update revises the ranges accordingly. One guarded update might be applicable when an operation underflows and another when it overflows, thus a transfer function is a system of guarded updates. The updates are expressed with affine equations that specify how the extreme values of variables at the end of the block relate to their extreme values on entry into the block. The guards that are derived are optimal (for the class of octagons) as are the update operations (for the class of affine equalities). Once derived, a transfer function is evaluated using linear programming.

2 Worked Examples

The ethos of our approach is to express the semantics of a block in the computational domain of Boolean formulae. This concrete domain is rich enough to allow the extreme values (ranges) of variables to be specified in a way that is analogous to that of Monniaux [20]. However, in contrast to Monniaux, universal quantifier elimination is performed in the concrete setting, which is attractive computationally. Abstraction is then applied to synthesise guarded updates from quantifier-free formulae. Thus, the approach of Monniaux is abstraction then elimination, whereas ours is elimination then abstraction. We illustrate the power of this transposition by deriving transfer functions for some illustrative blocks of ATmega16 8-bit microcontroller instructions [1].

2.1 Deriving a transfer function for a block

Consider deriving a transfer function for the sequence of instructions `EOR R0 R1`; `EOR R1 R0`; `EOR R0 R1` that constitutes a block. An instruction `EOR R0 R1` stores the exclusive-or of registers `R0` and `R1` in `R0`. The operands are unsigned. To specify the semantics of the block, let $\boldsymbol{r0}$ and $\boldsymbol{r1}$ denote 8-bit vectors of propositional variables that will be used to represent the symbolic initial values of `R0` and `R1`, and likewise let $\boldsymbol{r0'}$ and $\boldsymbol{r1'}$ be bit-vectors of propositional variables that denote their final values. Furthermore, let $\boldsymbol{x}[i]$ denote the i^{th} element of the bit-vector $\boldsymbol{x}$ where $\boldsymbol{x}[0]$ is the low bit. By introducing a bit-vector $\boldsymbol{y}$ to denote the intermediate value of `R0` (which is akin to applying static single assigment) the semantics of the block can be stated propositionally as:

$$\begin{aligned}\varphi(\boldsymbol{y}) = {} & (\wedge_{i=0}^{7}\boldsymbol{y}[i] \leftrightarrow \boldsymbol{r0}[i] \oplus \boldsymbol{r1}[i]) \ \wedge \\ & (\wedge_{i=0}^{7}\boldsymbol{r1'}[i] \leftrightarrow \boldsymbol{y}[i] \oplus \boldsymbol{r1}[i]) \wedge (\wedge_{i=0}^{7}\boldsymbol{r0'}[i] \leftrightarrow \boldsymbol{y}[i] \oplus \boldsymbol{r1'}[i])\end{aligned}$$

where $\oplus$ denotes exclusive-or. Such formulae can be derived algorithmically by composing formulae [5, 14] – one formula for each instruction in the sequence.

The formula $\varphi(\boldsymbol{y})$ specifies the relationship between the inputs $\boldsymbol{r0}$ and $\boldsymbol{r1}$ and the outputs $\boldsymbol{r0'}$ and $\boldsymbol{r1'}$, but not a relationship between their ranges. This has to be derived. To do so, let the bit-vectors $\boldsymbol{r0}_\ell$ and $\boldsymbol{r0}_u$ (resp. $\boldsymbol{r1}_\ell$ and $\boldsymbol{r1}_u$) denote the minimal and maximal values for $\boldsymbol{r0}$ (resp. $\boldsymbol{r1}$). To express these ranges in propositional logic, define the formula:

$$\boldsymbol{x} \leq \boldsymbol{y} = (\boldsymbol{x}[7] \wedge \neg\boldsymbol{y}[7]) \vee (\vee_{j=0}^{6}(\neg\boldsymbol{x}[j] \wedge \boldsymbol{y}[j] \wedge (\wedge_{k=j+1}^{7}\boldsymbol{x}[k] \leftrightarrow \boldsymbol{y}[k])))$$

and for abbreviation let $\boldsymbol{x} \leq \boldsymbol{y} \leq \boldsymbol{z} = (\boldsymbol{x} \leq \boldsymbol{y}) \wedge (\boldsymbol{y} \leq \boldsymbol{z})$. Moreover, let $\phi = (\boldsymbol{r0}_\ell \leq \boldsymbol{r0} \leq \boldsymbol{r0}_u) \wedge (\boldsymbol{r1}_\ell \leq \boldsymbol{r1} \leq \boldsymbol{r1}_u)$ to express the requirement that $\boldsymbol{r0}$ and $\boldsymbol{r1}$ are confined to their ranges. With $\boldsymbol{r0}$ and $\boldsymbol{r1}$ in range, let $\boldsymbol{r0}^\star_\ell$ and $\boldsymbol{r0}^\star_u$ denote the resulting extreme values for $\boldsymbol{r0'}$. This amounts to requiring that, firstly, $\boldsymbol{r0}^\star_\ell$ and $\boldsymbol{r0}^\star_u$ are respectively lower and upper bounds on the range of $\boldsymbol{r0'}$. Secondly, any other lower and upper bounds on $\boldsymbol{r0'}$, say, $\boldsymbol{r0}'_\ell$ and $\boldsymbol{r0}'_u$, are respectively less or equal to and greater or equal to $\boldsymbol{r0}^\star_\ell$ and $\boldsymbol{r0}^\star_u$. Analogous

requirements hold for the extreme values $\boldsymbol{r1}^\star_\ell$ and $\boldsymbol{r1}^\star_u$ of $\boldsymbol{r1}'$. We impose the first requirement with the formula $\theta(\boldsymbol{y}) = \forall \boldsymbol{r0} : \forall \boldsymbol{r1} : \forall \boldsymbol{r0}' : \forall \boldsymbol{r1}' : \theta'(\boldsymbol{y})$ where:

$$\theta'(\boldsymbol{y}) = (\phi \wedge \varphi(\boldsymbol{y})) \Rightarrow (\boldsymbol{r0}^\star_\ell \leq \boldsymbol{r0}' \leq \boldsymbol{r0}^\star_u \wedge \boldsymbol{r1}^\star_\ell \leq \boldsymbol{r1}' \leq \boldsymbol{r1}^\star_u)$$

A quantifier-free version of $\theta(\boldsymbol{y})$ is then obtained by putting $\theta'(\boldsymbol{y})$ into CNF using standard transformations [25]. This introduces fresh variables, denoted $\boldsymbol{y}'$, and thus we write the CNF formula as $\theta''(\boldsymbol{y}, \boldsymbol{y}')$. The intermediate variables of $\boldsymbol{y}$ and $\boldsymbol{y}'$ (which are existentially quantified) are then removed by repeatedly applying resolution [15]. Those literals that involve variables in $\boldsymbol{r0}, \boldsymbol{r1}, \boldsymbol{r0}'$ and $\boldsymbol{r1}'$ are then simply struck out to obtain the desired quantifier-free model $\theta(\boldsymbol{y}, \boldsymbol{y}')$.

The second requirement is enforced by introducing other lower and upper bounds on $\boldsymbol{r0}'$ and $\boldsymbol{r1}'$, namely, $\boldsymbol{r0}'_\ell$ and $\boldsymbol{r0}'_u$, and $\boldsymbol{r1}'_\ell$ and $\boldsymbol{r1}'_u$. The requirement is formally stipulated as:

$$\psi(\boldsymbol{z}) = \forall \boldsymbol{r0}'_\ell : \forall \boldsymbol{r0}'_u : \forall \boldsymbol{r1}'_\ell : \forall \boldsymbol{r1}'_u : \forall \boldsymbol{r0} : \forall \boldsymbol{r1} : \forall \boldsymbol{r0}' : \forall \boldsymbol{r1}' : \psi'(\boldsymbol{z})$$

where:

$$\psi'(\boldsymbol{z}) = ((\phi \wedge \varphi(\boldsymbol{z})) \Rightarrow (\boldsymbol{r0}'_\ell \leq \boldsymbol{r0}' \leq \boldsymbol{r0}'_u \wedge \boldsymbol{r1}'_\ell \leq \boldsymbol{r1}' \leq \boldsymbol{r1}'_u)) \Rightarrow \kappa$$

and $\kappa = \boldsymbol{r0}'_\ell \leq \boldsymbol{r0}^\star_\ell \wedge \boldsymbol{r0}^\star_u \leq \boldsymbol{r0}'_u \wedge \boldsymbol{r1}'_\ell \leq \boldsymbol{r1}^\star_\ell \wedge \boldsymbol{r1}^\star_u \leq \boldsymbol{r1}'_u$. As before, we derive a quantifier-free version of $\psi(\boldsymbol{z})$, namely $\psi(\boldsymbol{z}, \boldsymbol{z}')$, where $\boldsymbol{z}'$ are the fresh variables introduced in CNF conversion. To avoid accidental variable coupling between $\theta(\boldsymbol{y}, \boldsymbol{y}')$ and $\psi(\boldsymbol{z}, \boldsymbol{z}')$ we apply renaming (if necessary) to ensure that $(var(\boldsymbol{y}) \cup var(\boldsymbol{y}')) \cap (var(\boldsymbol{z}) \cup var(\boldsymbol{z}')) = \emptyset$ where $var(o)$ denotes the set of propositional variables in the object o.

Finally, the relationship between the bounds $\boldsymbol{r0}_\ell$, $\boldsymbol{r0}_u$, $\boldsymbol{r1}_\ell$ and $\boldsymbol{r1}_u$ and the extrema $\boldsymbol{r0}^\star_\ell$, $\boldsymbol{r0}^\star_u$, $\boldsymbol{r1}^\star_\ell$ and $\boldsymbol{r1}^\star_u$, is specified with the conjunction:

$$f_{\text{simp}} = \theta(\boldsymbol{y}, \boldsymbol{y}') \wedge \psi(\boldsymbol{z}, \boldsymbol{z}')$$

The formula f_{simp} is free from universal quantifiers and, moreover, we can abstract it using affine equations [13] to discover linear relationships between the variables of $S = \{\boldsymbol{r0}^\star_\ell, \boldsymbol{r0}^\star_u, \boldsymbol{r1}^\star_\ell, \boldsymbol{r1}^\star_u, \boldsymbol{r0}_\ell, \boldsymbol{r0}_u, \boldsymbol{r1}_\ell, \boldsymbol{r1}_u\}$ as desired. We regard this abstraction operation, denoted $\alpha_{\text{aff}}(f_{\text{simp}}, S)$, as a blackbox and defer presentation of the details to the following section. However, to state the outcome, let $\langle \boldsymbol{x} \rangle = \sum_{i=0}^{7} 2^i \boldsymbol{x}[i]$ where $\boldsymbol{x}$ is an 8-bit vector of propositional variables. Then

$$\alpha_{\text{aff}}(f_{\text{simp}}, S) = \begin{cases} \langle \boldsymbol{r0}^\star_\ell \rangle = \langle \boldsymbol{r1}_\ell \rangle \wedge \langle \boldsymbol{r0}^\star_u \rangle = \langle \boldsymbol{r1}_u \rangle \wedge \\ \langle \boldsymbol{r1}^\star_\ell \rangle = \langle \boldsymbol{r0}_\ell \rangle \wedge \langle \boldsymbol{r1}^\star_u \rangle = \langle \boldsymbol{r0}_u \rangle \end{cases}$$

This shows that the ranges of `R0` and `R1` are swapped. Indeed, the sequence of `EOR` instructions is a common idiom for exchanging the contents of two registers without employing a third. The resulting transfer function can be realised with four updates. For this example, it is difficult to see how range analysis can be usefully performed without deriving a transfer function at this level of granularity. Furthermore, it is not clear how such a transfer function could be derived from a system of piecewise linear functions [20] since such systems cannot express exclusive-or constraints.

2.2 Deriving a transfer function for an operation with many modes

As a second example, consider computing a transfer function for the single operation `ADD R0 R1` which calculates the sum of `R0` and `R1` and stores the result in `R0`. We assume that the operands are signed and to interpret the value of such a vector let $\langle\langle \boldsymbol{x} \rangle\rangle = (\sum_{i=0}^{6} 2^i \boldsymbol{x}[i]) - 2^7 \boldsymbol{x}[7]$ where $\boldsymbol{x}[7]$ is read as the sign bit. The `ADD R0 R1` instruction is interesting because it is an exemplar of an instruction that can operate in one of three modes: it overflows (the sum exceeds 127); it underflows (the sum is strictly less than -128); or it neither overflows nor underflows (it is exact). The semantics in these respective modes, can be expressed with three Boolean formulae that are defined as follows:

$$\begin{aligned}
\varphi_O(\boldsymbol{c}) &= \varphi(\boldsymbol{c}) \wedge (\neg \boldsymbol{r0}[7] \wedge \neg \boldsymbol{r1}[7] \wedge \boldsymbol{r0'}[7]) \\
\varphi_U(\boldsymbol{c}) &= \varphi(\boldsymbol{c}) \wedge (\boldsymbol{r0}[7] \wedge \boldsymbol{r1}[7] \wedge \neg \boldsymbol{r0'}[7]) \\
\varphi_E(\boldsymbol{c}) &= \varphi(\boldsymbol{c}) \wedge (\neg \boldsymbol{r0}[7] \vee \neg \boldsymbol{r1}[7] \vee \boldsymbol{r0'}[7]) \wedge (\boldsymbol{r0}[7] \vee \boldsymbol{r1}[7] \vee \neg \boldsymbol{r0'}[7])
\end{aligned}$$

where $\boldsymbol{c}$ is a bit-vector that represents the intermediate carry bits and:

$$\begin{aligned}
\varphi(\boldsymbol{c}) = &\Big(\textstyle\bigwedge_{i=0}^{7} \boldsymbol{r0'}[i] \leftrightarrow \boldsymbol{r0}[i] \oplus \boldsymbol{r1}[i] \oplus \boldsymbol{c}[i]\Big) \wedge \\
&\neg \boldsymbol{c}[0] \wedge \Big(\textstyle\bigwedge_{i=0}^{6} \boldsymbol{c}[i+1] \leftrightarrow (\boldsymbol{r0}[i] \wedge \boldsymbol{r1}[i]) \vee (\boldsymbol{r0}[i] \wedge \boldsymbol{c}[i]) \vee (\boldsymbol{r1}[i] \wedge \boldsymbol{c}[i])\Big)
\end{aligned}$$

It is important to appreciate that the formulae that model an instruction need to be prescribed just once for each instruction. Thus, the complexity is barely more than that of bit-blasting.

The transfer functions for multi-modal operations are formulated as action systems of guarded updates. Given a system of input intervals, which defines a hypercube, the guards test whether an update is applicable and, if so, the corresponding update is applied. The update maps the input intervals to the resulting output intervals. An update is applicable if interval and guard constraints are simultaneously satisfiable. The guards are formulated as octagonal constraints [19] over the inputs to the (trivial) block, namely, $S' = \{\boldsymbol{r0}, \boldsymbol{r1}\}$. Guards are derived with an abstraction operator, denoted $\alpha_{\text{oct}}(\varphi_i(\boldsymbol{c}), S')$, which discovers the octagonal inequalities that hold in the formula $\varphi_i(\boldsymbol{c})$ between the variables of S'. This abstraction can be calculated automatically, though for reasons of continuity we defer the details until the following section. Applying this abstraction to the three $\varphi_i(\boldsymbol{c})$ formulae yields the guards:

$$\begin{aligned}
\alpha_{\text{oct}}(\varphi_O(\boldsymbol{c}), S') &= \begin{cases} 128 \le \langle\langle \boldsymbol{r0} \rangle\rangle + \langle\langle \boldsymbol{r1} \rangle\rangle \le 254 \wedge \\ 1 \le \langle\langle \boldsymbol{r0} \rangle\rangle \le 127 \qquad\qquad \wedge\ 1 \le \langle\langle \boldsymbol{r1} \rangle\rangle \le 127 \end{cases} \\
\alpha_{\text{oct}}(\varphi_U(\boldsymbol{c}), S') &= \begin{cases} -256 \le \langle\langle \boldsymbol{r0} \rangle\rangle + \langle\langle \boldsymbol{r1} \rangle\rangle \le -129 \wedge \\ -128 \le \langle\langle \boldsymbol{r0} \rangle\rangle \le -1 \qquad\qquad \wedge\ -128 \le \langle\langle \boldsymbol{r1} \rangle\rangle \le -1 \end{cases} \\
\alpha_{\text{oct}}(\varphi_E(\boldsymbol{c}), S') &= \begin{cases} -128 \le \langle\langle \boldsymbol{r0} \rangle\rangle + \langle\langle \boldsymbol{r1} \rangle\rangle \le 127 \wedge \\ -128 \le \langle\langle \boldsymbol{r0} \rangle\rangle \le 127 \qquad\qquad \wedge\ -128 \le \langle\langle \boldsymbol{r1} \rangle\rangle \le 127 \end{cases}
\end{aligned}$$

Guards are computed using perfect integers (the detail of which is explained in the following section) rather than modulo 256. Hence the linear inequality

$\langle\!\langle \boldsymbol{r0} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq 254$ which follows from the positivity requirements on $\langle\!\langle \boldsymbol{r0} \rangle\!\rangle$ and $\langle\!\langle \boldsymbol{r1} \rangle\!\rangle$, namely, $1 \leq \langle\!\langle \boldsymbol{r0} \rangle\!\rangle \leq 127$ and $1 \leq \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq 127$.

An affine update is computed for each mode $\varphi_i(\boldsymbol{c})$ from the Boolean formula $f_{i,\mathrm{simp}} = \theta_i(\boldsymbol{c}, \boldsymbol{c}') \wedge \psi_i(\boldsymbol{d}, \boldsymbol{d}')$ where $\theta_i(\boldsymbol{c}, \boldsymbol{c}')$ and $\psi_i(\boldsymbol{d}, \boldsymbol{d}')$ are quantifier-free formulae derived from $\varphi_i(\boldsymbol{c})$ in an analogous way to before. As before, we desire affine relationships over $S = \{\boldsymbol{r0}^\star_\ell, \boldsymbol{r0}^\star_u, \boldsymbol{r1}^\star_\ell, \boldsymbol{r1}^\star_u, \boldsymbol{r0}_\ell, \boldsymbol{r0}_u, \boldsymbol{r1}_\ell, \boldsymbol{r1}_u\}$, hence we calculate $\alpha_{\mathrm{aff}}(f_{i,\mathrm{simp}}, S)$ which yields:

$$\alpha_{\mathrm{aff}}(f_{O,\mathrm{simp}}, S) = \begin{cases} \langle\!\langle \boldsymbol{r0}^\star_\ell \rangle\!\rangle = \langle\!\langle \boldsymbol{r0}_\ell \rangle\!\rangle + \langle\!\langle \boldsymbol{r1}_\ell \rangle\!\rangle - 256 \wedge \\ \langle\!\langle \boldsymbol{r0}^\star_u \rangle\!\rangle = \langle\!\langle \boldsymbol{r0}_u \rangle\!\rangle + \langle\!\langle \boldsymbol{r1}_u \rangle\!\rangle - 256 \wedge \\ \langle\!\langle \boldsymbol{r1}^\star_\ell \rangle\!\rangle = \langle\!\langle \boldsymbol{r1}_\ell \rangle\!\rangle \wedge \langle\!\langle \boldsymbol{r1}^\star_u \rangle\!\rangle = \langle\!\langle \boldsymbol{r1}_u \rangle\!\rangle \end{cases}$$

$$\alpha_{\mathrm{aff}}(f_{U,\mathrm{simp}}, S) = \begin{cases} \langle\!\langle \boldsymbol{r0}^\star_\ell \rangle\!\rangle = \langle\!\langle \boldsymbol{r0}_\ell \rangle\!\rangle + \langle\!\langle \boldsymbol{r1}_\ell \rangle\!\rangle + 256 \wedge \\ \langle\!\langle \boldsymbol{r0}^\star_u \rangle\!\rangle = \langle\!\langle \boldsymbol{r0}_u \rangle\!\rangle + \langle\!\langle \boldsymbol{r1}_u \rangle\!\rangle + 256 \wedge \\ \langle\!\langle \boldsymbol{r1}^\star_\ell \rangle\!\rangle = \langle\!\langle \boldsymbol{r1}_\ell \rangle\!\rangle \wedge \langle\!\langle \boldsymbol{r1}^\star_u \rangle\!\rangle = \langle\!\langle \boldsymbol{r1}_u \rangle\!\rangle \end{cases}$$

$$\alpha_{\mathrm{aff}}(f_{E,\mathrm{simp}}, S) = \begin{cases} \langle\!\langle \boldsymbol{r0}^\star_\ell \rangle\!\rangle = \langle\!\langle \boldsymbol{r0}_\ell \rangle\!\rangle + \langle\!\langle \boldsymbol{r1}_\ell \rangle\!\rangle \ \wedge \\ \langle\!\langle \boldsymbol{r0}^\star_u \rangle\!\rangle = \langle\!\langle \boldsymbol{r0}_u \rangle\!\rangle + \langle\!\langle \boldsymbol{r1}_u \rangle\!\rangle \ \wedge \\ \langle\!\langle \boldsymbol{r1}^\star_\ell \rangle\!\rangle = \langle\!\langle \boldsymbol{r1}_\ell \rangle\!\rangle \wedge \langle\!\langle \boldsymbol{r1}^\star_u \rangle\!\rangle = \langle\!\langle \boldsymbol{r1}_u \rangle\!\rangle \end{cases}$$

When coupled with the guards, this gives an action system of three guarded updates reflecting the three distinct modes of operation.

To illustrate an application of the derived transfers function, suppose $\langle\!\langle \boldsymbol{r0} \rangle\!\rangle$ and $\langle\!\langle \boldsymbol{r1} \rangle\!\rangle$ are clamped to fall within given input range. Moreover, suppose the range is expressed as the following system of inequalities:

$$r = \begin{cases} \langle\!\langle \boldsymbol{r0}_\ell \rangle\!\rangle = -2 \wedge \langle\!\langle \boldsymbol{r0}_u \rangle\!\rangle = 5 \quad \wedge \langle\!\langle \boldsymbol{r0}_\ell \rangle\!\rangle \leq \langle\!\langle \boldsymbol{r0} \rangle\!\rangle \leq \langle\!\langle \boldsymbol{r0}_u \rangle\!\rangle \\ \langle\!\langle \boldsymbol{r1}_\ell \rangle\!\rangle = 1 \quad \wedge \langle\!\langle \boldsymbol{r1}_u \rangle\!\rangle = 126 \wedge \langle\!\langle \boldsymbol{r1}_\ell \rangle\!\rangle \leq \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq \langle\!\langle \boldsymbol{r1}_u \rangle\!\rangle \end{cases}$$

In addition to bound the extreme output values let:

$$r' = \begin{cases} -128 \leq \langle\!\langle \boldsymbol{r0}^\star_\ell \rangle\!\rangle \leq 127 \wedge -128 \leq \langle\!\langle \boldsymbol{r0}^\star_u \rangle\!\rangle \leq 127 \wedge \\ -128 \leq \langle\!\langle \boldsymbol{r1}^\star_\ell \rangle\!\rangle \leq 127 \wedge -128 \leq \langle\!\langle \boldsymbol{r1}^\star_u \rangle\!\rangle \leq 127 \end{cases}$$

Now consider the overflow mode. Observe that the output value of $\langle\!\langle \boldsymbol{r0}^\star_\ell \rangle\!\rangle$ can be found by solving a linear programming problem that minimises $\langle\!\langle \boldsymbol{r0}^\star_\ell \rangle\!\rangle$ subject to the linear system $r \wedge \alpha_{\mathrm{oct}}(\varphi_O(\boldsymbol{c}), S') \wedge \alpha_{\mathrm{aff}}(f_{O,\mathrm{simp}}, S) \wedge r'$. This gives $\langle\!\langle \boldsymbol{r0}^\star_\ell \rangle\!\rangle = -128$. By solving three more linear programming problems we can likewise deduce $\langle\!\langle \boldsymbol{r0}^\star_u \rangle\!\rangle = -125$, $\langle\!\langle \boldsymbol{r1}^\star_\ell \rangle\!\rangle = 1$ and $\langle\!\langle \boldsymbol{r1}^\star_u \rangle\!\rangle = 126$. When repeating this process for the underflow mode, we find that the system $r \wedge \alpha_{\mathrm{oct}}(\varphi_U(\boldsymbol{c}), S') \wedge \alpha_{\mathrm{aff}}(f_{U,\mathrm{simp}}, S) \wedge r'$ is infeasible, hence this guarded update is not applicable for the given input range. However, the final mode is applicable (like the first) and gives the extrema: $\langle\!\langle \boldsymbol{r0}^\star_\ell \rangle\!\rangle = -1$, $\langle\!\langle \boldsymbol{r0}^\star_u \rangle\!\rangle = 127$, $\langle\!\langle \boldsymbol{r1}^\star_\ell \rangle\!\rangle = 1$ and $\langle\!\langle \boldsymbol{r1}^\star_u \rangle\!\rangle = 126$. More generally, evaluating a system of m guarded updates for a block that involves n variables will require at most $2mn$ linear programs to be solved.

The overall result is obtained by merging the results from different modes, and there is no reason why power sets could not be deployed to summarise the final value of $\langle\!\langle \boldsymbol{r0} \rangle\!\rangle$ as $[-128, -125] \cup [-1, 127]$ (with the understanding that adjacent and overlapping intervals are merged for compactness).

2.3 Deriving a transfer function for a block with many modes

Consider the following non-trivial sequence of instructions:

1: `INC R0;`	2: `MOV R1, R0;`	3: `LSL R1;`
4: `SBC R1, R1;`	5: `EOR R0, R1;`	6: `SUB R0, R1;`

`INC R0` increments `R0`; `MOV R1, R0` copies the contents of `R0` into `R1`; `LSL R1` leftshifts `R1` by one bit position setting the carry to the sign; `SBC R1, R1` subtracts `R1`, summed with the carry, from `R1` and stores the result in `R1`; and `SUB R0, R1` subtracts `R1` from `R0` without considering the carry. (The net effect of instructions 3 and 4 is to set all the lower bits of `R1` to its sign bit, so that `R1` contains either 0 or -1.)

Analogous to before, the sequence is bit-blasted using additional bit-vectors $\boldsymbol{w}$, $\boldsymbol{x}$, $\boldsymbol{y}$, and $\boldsymbol{z}$ to represent intermediate values of registers: $\boldsymbol{w}$ for the value of `R0` immediately after instruction 1 (and the value of `R1` after instruction 2); $\boldsymbol{x}$ for the value of `R0` after instruction 5; $\boldsymbol{y}$ for the negation of `R1` after instruction 5 which is then used in the following subtraction; and $\boldsymbol{z}$ for the carry bits that are also used in subtract. With some simplification (needed to make the presentation accessible) the semantics of the block can be expressed as:

$$\varphi(\boldsymbol{w},\boldsymbol{x},\boldsymbol{y},\boldsymbol{z}) = \begin{cases} \left(\wedge_{i=0}^{7}\boldsymbol{w}[i] \leftrightarrow (\boldsymbol{r0}[i] \oplus \wedge_{j=0}^{i-1}\boldsymbol{r0}[j])\right) & \wedge \\ \left(\wedge_{i=0}^{7}\boldsymbol{r1}'[i] \leftrightarrow \boldsymbol{w}[7]\right) & \wedge \\ \left(\wedge_{i=0}^{7}\boldsymbol{x}[i] \leftrightarrow (\boldsymbol{w}[i] \oplus \boldsymbol{r1}'[i])\right) & \wedge \\ \left(\wedge_{i=0}^{7}\boldsymbol{y}[i] \leftrightarrow (\neg\boldsymbol{r1}'[i] \oplus \wedge_{j=0}^{i-1}\neg\boldsymbol{r1}'[j])\right) & \wedge \\ (\neg\boldsymbol{z}[0]) & \wedge \\ \left(\wedge_{i=0}^{6}\boldsymbol{z}[i+1] \leftrightarrow (\boldsymbol{x}[i] \wedge \boldsymbol{y}[i]) \vee (\boldsymbol{x}[i] \wedge \boldsymbol{z}[i]) \vee (\boldsymbol{y}[i] \wedge \boldsymbol{z}[i])\right) & \wedge \\ \left(\wedge_{i=0}^{7}\boldsymbol{r0}'[i] \leftrightarrow \boldsymbol{x}[i] \oplus \boldsymbol{y}[i] \oplus \boldsymbol{z}[i]\right) \end{cases}$$

For brevity, let vector $\boldsymbol{u}$ denote the concatenation of the vectors $\boldsymbol{w}$, $\boldsymbol{x}$, $\boldsymbol{y}$ and $\boldsymbol{z}$ so as to write $\varphi(\boldsymbol{u})$ for $\varphi(\boldsymbol{w},\boldsymbol{x},\boldsymbol{y},\boldsymbol{z})$.

For the increment, there are two modes of operation depending on whether it overflows or not. These can be expressed by: $\mu_O = (\neg\boldsymbol{r0}[7]) \wedge (\wedge_{i=0}^{6}\boldsymbol{r0}[i])$ and $\mu_E = \neg\mu_O$ respectively. For the leftshift, there are two modes depending on whether it overflows or not, as defined by the formulae $\eta_O = \boldsymbol{w}[7]$ and $\eta_E = \neg\eta_E$ respectively. For subtraction, there are again three modes according to whether it overflows or underflows or does neither. These modes can be expressed as: $\nu_U = (\neg\boldsymbol{r0}'[7] \wedge \boldsymbol{x}[7] \wedge \boldsymbol{y}[7])$, $\nu_O = (\boldsymbol{r0}'[7] \wedge \neg\boldsymbol{x}[7] \wedge \neg\boldsymbol{y}[7])$ and $\nu_E = (\neg\nu_1) \wedge (\neg\nu_2)$. All other instructions in the block are unimodal; indeed they neither overflow nor underflow [1]. This gives twelve different mode combinations overall. However, the following formulae are unsatisfiable:

$\mu_O \wedge \eta_O \wedge \nu_O \wedge \varphi(\boldsymbol{u})$	$\mu_O \wedge \eta_O \wedge \nu_E \wedge \varphi(\boldsymbol{u})$	$\mu_O \wedge \eta_E \wedge \nu_U \wedge \varphi(\boldsymbol{u})$
$\mu_O \wedge \eta_E \wedge \nu_O \wedge \varphi(\boldsymbol{u})$	$\mu_O \wedge \eta_E \wedge \nu_E \wedge \varphi(\boldsymbol{u})$	$\mu_E \wedge \eta_O \wedge \nu_U \wedge \varphi(\boldsymbol{u})$
$\mu_E \wedge \eta_O \wedge \nu_O \wedge \varphi(\boldsymbol{u})$	$\mu_E \wedge \eta_E \wedge \nu_U \wedge \varphi(\boldsymbol{u})$	$\mu_E \wedge \eta_E \wedge \nu_O \wedge \varphi(\boldsymbol{u})$

indicating these mode combinations are not feasible. Henceforth, only three combinations needed to be considered when synthesising the action system.

Hence let $\varphi_1(\boldsymbol{u}) = \mu_O \wedge \eta_O \wedge \nu_U \wedge \varphi(\boldsymbol{u})$, $\varphi_2(\boldsymbol{u}) = \mu_E \wedge \eta_O \wedge \nu_E \wedge \varphi(\boldsymbol{u})$ and $\varphi_3(\boldsymbol{u}) = \mu_E \wedge \eta_E \wedge \nu_E \wedge \varphi(\boldsymbol{u})$. The inputs to the block are $S' = \{\boldsymbol{r0}, \boldsymbol{r1}\}$ and calculating guards for these combinations gives:

$$\begin{array}{ll}
\alpha_{\mathrm{oct}}(\varphi_1(\boldsymbol{v}), S') = & 127 \leq \langle\!\langle \boldsymbol{r0} \rangle\!\rangle \leq 127 \wedge -128 \leq \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq 127 \\
\alpha_{\mathrm{oct}}(\varphi_2(\boldsymbol{v}), S') = & -128 \leq \langle\!\langle \boldsymbol{r0} \rangle\!\rangle \leq -2 \wedge -128 \leq \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq 127 \\
\alpha_{\mathrm{oct}}(\varphi_3(\boldsymbol{v}), S') = & -1 \leq \langle\!\langle \boldsymbol{r0} \rangle\!\rangle \leq 126 \wedge -128 \leq \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq 127
\end{array}$$

Note that all three guards impose vacuous constraints on $\langle\!\langle \boldsymbol{r1} \rangle\!\rangle$. This is because `R1` is written before it is read (which could be inferred prior to deriving the transfer functions though this is not strictly necessary).

As before, the updates are computed for each $\varphi_i(\boldsymbol{u})$ from three formulae $f_{i,\mathrm{simp}} = \theta_i(\boldsymbol{u}, \boldsymbol{u'}) \wedge \psi_i(\boldsymbol{v}, \boldsymbol{v'})$ where $\theta_i(\boldsymbol{u}, \boldsymbol{u'})$ and $\psi_i(\boldsymbol{v}, \boldsymbol{v'})$ are quantifier-free and derived from $\varphi_i(\boldsymbol{u})$ as previously. Hence we calculate affine relationships over $S = \{\boldsymbol{r0}^\star_\ell, \boldsymbol{r0}^\star_u, \boldsymbol{r1}^\star_\ell, \boldsymbol{r1}^\star_u, \boldsymbol{r0}_\ell, \boldsymbol{r0}_u, \boldsymbol{r1}_\ell, \boldsymbol{r1}_u\}$ which yields:

$$\begin{array}{lll}
\alpha_{\mathrm{aff}}(f_{1,\mathrm{simp}}, S) = & \langle\!\langle \boldsymbol{r0} \rangle\!\rangle_\ell = 127 & \wedge \langle\!\langle \boldsymbol{r0} \rangle\!\rangle_u = 127 \quad \wedge \\
& \langle\!\langle \boldsymbol{r0} \rangle\!\rangle^\star_\ell = -128 & \wedge \langle\!\langle \boldsymbol{r0} \rangle\!\rangle^\star_u = -128 \\
\alpha_{\mathrm{aff}}(f_{2,\mathrm{simp}}, S) = & \langle\!\langle \boldsymbol{r0} \rangle\!\rangle^\star_\ell = -\langle\!\langle \boldsymbol{r0} \rangle\!\rangle_u - 1 & \wedge \langle\!\langle \boldsymbol{r0} \rangle\!\rangle^\star_u = -\langle\!\langle \boldsymbol{r0} \rangle\!\rangle_\ell - 1 \\
\alpha_{\mathrm{aff}}(f_{3,\mathrm{simp}}, S) = & \langle\!\langle \boldsymbol{r0} \rangle\!\rangle^\star_\ell = \langle\!\langle \boldsymbol{r0} \rangle\!\rangle_\ell + 1 & \wedge \langle\!\langle \boldsymbol{r0} \rangle\!\rangle^\star_u = \langle\!\langle \boldsymbol{r0} \rangle\!\rangle_u + 1
\end{array}$$

Note that no affine constraints are inferred for $\boldsymbol{r1}^\star_\ell$ and $\boldsymbol{r1}^\star_u$ reflecting that `R1` is used merely to store an intermediate value (though combining affine equations with congruences [14] would preserve some information pertaining to the final value of `R1`). Note how ranges are swapped for the second mode since in this circumstance `R0` is negative. From the resulting action system, it can be seen that the block overwrites `R0` with the absolute value of (`R0` + 1) subject to wrap around ($128 = -128 \mod 256$). To the best of our knowledge, no other approach can derive a useful transfer function for a block such as this.

3 Abstracting Boolean Formulae

This section shows how to construct octagonal and affine abstractions of a given Boolean formula $\varphi(\boldsymbol{v})$ defined over a set of bit-vectors $S = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$ of size k and a single bit-vector $\boldsymbol{v}$ that identifies any intermediate variables. For presentational purposes, we focus on deriving abstractions that relate the values of $\langle\!\langle \boldsymbol{x}_1 \rangle\!\rangle, \ldots, \langle\!\langle \boldsymbol{x}_n \rangle\!\rangle$, though the construction for unsigned values is analogous.

3.1 Abstracting Boolean Formulae with Octagonal Inequalities

Let $\lambda, \mu \in \{-1, 0, 1\}$ and $1 \leq i \leq j \leq n$ be fixed and consider the derivation of an octagonal inequality $\lambda\langle\!\langle \boldsymbol{x}_i \rangle\!\rangle + \mu\langle\!\langle \boldsymbol{x}_j \rangle\!\rangle \leq c^\star$ where $c^\star \in \mathbb{Z}$. Since k is fixed it follows $-2^k \leq \lambda\langle\!\langle \boldsymbol{x}_i \rangle\!\rangle + \mu\langle\!\langle \boldsymbol{x}_j \rangle\!\rangle \leq 2^k$, hence the problem reduces to finding the least $-2^k \leq c^\star \leq 2^k$ such that if $\varphi(\boldsymbol{v})$ holds then $\lambda\langle\!\langle \boldsymbol{x}_i \rangle\!\rangle + \mu\langle\!\langle \boldsymbol{x}_j \rangle\!\rangle \leq c^\star$ also holds. To this end, let $\boldsymbol{c}^\star$ denote a bit-vector of size $k + 2$ and suppose $\phi_{\lambda,\mu,\boldsymbol{x}_i,\boldsymbol{x}_j}(\boldsymbol{u})$ is a propositional encoding of $\lambda\langle\!\langle \boldsymbol{x}_i \rangle\!\rangle + \mu\langle\!\langle \boldsymbol{x}_j \rangle\!\rangle \leq \boldsymbol{c}^\star$ where the sum is calculated

to a width of $k+2$ bits and $\leq$ is encoded as in Sect. 2.1, likewise operating on vectors of size $k+2$. In this formulation, the vector $\boldsymbol{v}$ denotes the intermediate variables required for addition and negation. Since $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ are k-bit this construction avoids wraps, hence $\phi_{\lambda,\mu,\boldsymbol{x}_i,\boldsymbol{x}_j}(\boldsymbol{v})$ holds iff $\lambda\langle\!\langle \boldsymbol{x}_i \rangle\!\rangle + \mu\langle\!\langle \boldsymbol{x}_j \rangle\!\rangle \leq c^\star$ holds. Likewise, suppose that $\phi'_{\lambda,\mu,\boldsymbol{x}_i,\boldsymbol{x}_j}(\boldsymbol{v}')$ is a propositional formula that holds iff $\lambda\langle\!\langle \boldsymbol{x}_i \rangle\!\rangle + \mu\langle\!\langle \boldsymbol{x}_j \rangle\!\rangle \leq c'$ holds where $\boldsymbol{c}'$ is $k+2$ bit-vector distinct from $\boldsymbol{c}^\star$. Finally, let κ denote a Boolean formula that holds iff $\boldsymbol{c}^\star \leq \boldsymbol{c}'$ holds.

Single inequalities With the formulae $\phi_{\lambda,\mu,\boldsymbol{x}_i,\boldsymbol{x}_j}(\boldsymbol{v})$ and $\phi'_{\lambda,\mu,\boldsymbol{x}_i,\boldsymbol{x}_j}(\boldsymbol{v}')$ thus defined, we can apply universal quantification to specify the least $c^\star$ as the unique value $\langle\!\langle \boldsymbol{c}^\star \rangle\!\rangle$ which satisfies $\theta_{\lambda,\mu,\boldsymbol{x}_i,\boldsymbol{x}_j}(\boldsymbol{u},\boldsymbol{v}) \wedge \psi_{\lambda,\mu,\boldsymbol{x}_i,\boldsymbol{x}_j}(\boldsymbol{u}',\boldsymbol{v}')$ where:

$$\theta_{\lambda,\mu,\boldsymbol{x}_i,\boldsymbol{x}_j}(\boldsymbol{u},\boldsymbol{v}) = \forall \boldsymbol{x}_i : \forall \boldsymbol{x}_j : (\varphi(\boldsymbol{u}) \Rightarrow \phi_{\lambda,\mu,\boldsymbol{x}_i,\boldsymbol{x}_j}(\boldsymbol{v}))$$
$$\psi_{\lambda,\mu,\boldsymbol{x}_i,\boldsymbol{x}_j}(\boldsymbol{u}',\boldsymbol{v}') = \forall \boldsymbol{x}_i : \forall \boldsymbol{x}_j : \forall \boldsymbol{c}' : ((\varphi(\boldsymbol{u}') \Rightarrow \phi'_{\lambda,\mu,\boldsymbol{x}_i,\boldsymbol{x}_j}(\boldsymbol{v}')) \Rightarrow \kappa)$$

and $\boldsymbol{u}$ and $\boldsymbol{u}'$ are renamed apart so as to avoid cross-coupling. More generally, the octagonal abstraction of $\varphi(\boldsymbol{v})$ over the set S is given by:

$$\alpha_{\mathrm{oct}}(\varphi(\boldsymbol{v}), S) = \bigwedge \left\{ \lambda\langle\!\langle \boldsymbol{x}_i \rangle\!\rangle + \mu\langle\!\langle \boldsymbol{x}_j \rangle\!\rangle \leq \langle\!\langle \boldsymbol{c}^\star \rangle\!\rangle \;\middle|\; \begin{array}{ll} \exists \lambda, \mu \in \{-1, 0, 1\} & \wedge \\ \exists 1 \leq i \leq j \leq n & \wedge \\ \theta_{\lambda,\mu,\boldsymbol{x}_i,\boldsymbol{x}_j}(\boldsymbol{u},\boldsymbol{v}) \wedge \psi_{\lambda,\mu,\boldsymbol{x}_i,\boldsymbol{x}_j}(\boldsymbol{u}',\boldsymbol{v}') \text{ holds} & \end{array} \right\}$$

Note that in $\lambda\langle\!\langle \boldsymbol{x}_i \rangle\!\rangle + \mu\langle\!\langle \boldsymbol{x}_j \rangle\!\rangle \leq \langle\!\langle \boldsymbol{c}^\star \rangle\!\rangle$ the symbols $\langle\!\langle \boldsymbol{x}_i \rangle\!\rangle$ and $\langle\!\langle \boldsymbol{x}_j \rangle\!\rangle$ denote variables whereas $\langle\!\langle \boldsymbol{c}^\star \rangle\!\rangle$ is a value that is fixed by the formula $\theta_{\lambda,\mu,i,j}(\boldsymbol{u},\boldsymbol{v}) \wedge \psi_{\lambda,\mu,i,j}(\boldsymbol{u}',\boldsymbol{v}')$. Of course, as previously explained, quantifier-free versions of $\theta_{\lambda,\mu,i,j}(\boldsymbol{u},\boldsymbol{v})$ and $\psi_{\lambda,\mu,i,j}(\boldsymbol{u}',\boldsymbol{v}')$ can be obtained through CNF conversion and clause simplification, hence $\alpha_{\mathrm{oct}}(\varphi(\boldsymbol{v}), S)$ can be computed as well as specified. The resulting octagonal abstraction is closed (though this is not necessary in our setting).

Many inequalities Interestingly, many inequalities can be derived in a single call to the solver. Let $\{(\boldsymbol{y}_1, \boldsymbol{z}_1), \ldots, (\boldsymbol{y}_m, \boldsymbol{z}_m)\} \subseteq S^2$ and $\{(\lambda_1, \mu_1), \ldots, (\lambda_m, \mu_m)\} \subseteq \{-1, 0, 1\}^2$, and consider the problem of finding the set $\{c_1^\star, \ldots, c_m^\star\} \subseteq [-2^k, 2^k - 1]$ of least values such that if $\varphi(\boldsymbol{v})$ holds then $\lambda_i\langle\!\langle \boldsymbol{y}_i \rangle\!\rangle + \mu_i\langle\!\langle \boldsymbol{z}_i \rangle\!\rangle \leq c_i^\star$ holds. This problem can be formulated in an analogous way to before using bit-vectors $\boldsymbol{c}_1^\star, \ldots, \boldsymbol{c}_m^\star$ and $\boldsymbol{c}'_1, \ldots, \boldsymbol{c}'_m$ all of size $k+2$. Furthermore, let κ_i be a propositional formula that holds iff $\boldsymbol{c}_i^\star \leq \boldsymbol{c}'_i$ holds. Then the problem of simultaneously finding all the $c_i^\star$ amounts to solving $\theta(\boldsymbol{u}, \boldsymbol{v}_1, \ldots, \boldsymbol{v}_m) \wedge \psi(\boldsymbol{u}', \boldsymbol{v}'_1, \ldots, \boldsymbol{v}'_m)$ where:

$$\theta(\boldsymbol{u}, \boldsymbol{v}_1, \ldots, \boldsymbol{v}_m) = \forall \boldsymbol{y}_1 : \forall \boldsymbol{z}_1 : \ldots : \forall \boldsymbol{y}_k : \forall \boldsymbol{z}_k : (\varphi(\boldsymbol{u}) \Rightarrow \textstyle\bigwedge_{k=1}^m \phi_{\lambda_k,\mu_k,\boldsymbol{y}_k,\boldsymbol{z}_k}(\boldsymbol{v}_k))$$
$$\psi(\boldsymbol{u}, \boldsymbol{v}'_1, \ldots, \boldsymbol{v}'_m) = \forall \boldsymbol{y}_1 : \forall \boldsymbol{z}_1 : \ldots : \forall \boldsymbol{y}_k : \forall \boldsymbol{z}_k : \\ \forall \boldsymbol{c}'_1 : \ldots : \forall \boldsymbol{c}'_m : ((\varphi(\boldsymbol{u}') \Rightarrow \textstyle\bigwedge_{k=1}^m \phi'_{\lambda_k,\mu_k,\boldsymbol{y}_k,\boldsymbol{z}_k}(\boldsymbol{v}'_k)) \Rightarrow \bigwedge_{k=1}^m \kappa_k)$$

Notice that $\theta(\boldsymbol{u}, \boldsymbol{v}_1, \ldots, \boldsymbol{v}_m) \wedge \psi(\boldsymbol{u}', \boldsymbol{v}'_1, \ldots, \boldsymbol{v}'_m)$ involves one copy of $\varphi(\boldsymbol{u})$ and another of $\varphi(\boldsymbol{u}')$ rather than m copies of both.

Timings The time required to bit-blast the blocks in Sect. 2 and then eliminate the universal quantifiers were essentially non-measurable. The time required to prove unsatisfiability or compute the guards for the various mode combinations varied between 0.1s and 0.6s. Octagons were derived one inequality at a time (rather than several together) using the SAT4J solver [16] on 2.6GHz MacBook Pro. Incremental SAT solving would speedup the generation of octagons, as the intermediate results used to derive one inequality could be used to infer another.

3.2 Abstracting Boolean Formulae with Affine Equalities

Affine equations [13, 22] are related to congruences [11, 23]; indeed the former is a special case of the latter where the modulo is 0. This suggests adapting an abstraction technique for formulae that discovers congruence relationships between the propositional variables of a given formula [14]. In our setting, the problem is different. It is that of computing an affine abstraction of a formula $\varphi(\boldsymbol{v})$ defined over a set of bit-vectors S. As before, we do not aspire to derive relationships that involve intermediate variables $\boldsymbol{v}$ and we assume each $\boldsymbol{x}_i$ is signed.

Algorithm Figure 1 gives an algorithm for computing $\alpha_{\text{aff}}(\varphi(\boldsymbol{v}), S)$. In what follows, the n-ary vector $\boldsymbol{x}$ is defined as $\boldsymbol{x} = (\langle\!\langle \boldsymbol{x}_1 \rangle\!\rangle, \ldots, \langle\!\langle \boldsymbol{x}_n \rangle\!\rangle)$. Affine equations over S are represented with an augmented rational matrix $[A \mid \boldsymbol{b}]$ that we interpret as defining the set $\{\boldsymbol{x} \in [-2^{k-1}, 2^{k-1}]^n \mid A\boldsymbol{x} = \boldsymbol{b}\}$. The algorithm relies on a propositional encoding for an affine disequality constraint $(c_1, \ldots, c_n) \cdot \boldsymbol{x} \neq b$ where $c_1, \ldots, c_n, b \in \mathbb{Q}$. To see that such an encoding is possible assume, without loss of generality, that the disequality is integral and b is non-negative. Then rewrite the disequality as $(c_1^+, \ldots, c_n^+) \cdot \boldsymbol{x} \neq b + (c_1^-, \ldots, c_n^-) \cdot \boldsymbol{x}$ 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 $c^+ = \sum_{i=1}^n c_i^+$ and $c^- = \sum_{i=1}^n c_i^-$. Since each $\langle\!\langle \boldsymbol{x}_j \rangle\!\rangle \in [-2^{k-1}, 2^{k-1} - 1]$ it follows that computing the sums $(c_1^+, \ldots, c_n^+) \cdot \boldsymbol{x}$ and $b + (c_1^-, \ldots, c_n^-) \cdot \boldsymbol{x}$ with a signed $1 + \lceil \log_2(1 + \max(2^k c^+, b + 2^k c^-)) \rceil$ bit representation is sufficient to avoid wraps, allowing the disequality to be modelled exactly as a formula. In the algorithm, this formula is denoted $\phi(\boldsymbol{w})$ where $\boldsymbol{w}$ is a vector of temporary variables used for carry bits and intermediate sums.

Apart from $\phi(\boldsymbol{w})$, the abstraction algorithm is essentially the same as that proposed for congruences [14] (with a proof of correctness carrying over too). The algorithm starts with an unsatisfiable constraint $\boldsymbol{0} \cdot \boldsymbol{x} = 1$ which is successively relaxed by merging it with a series of affine systems that are derived by SAT (or SMT) solving. The truth assignment θ is considered to be a mapping $\theta : var(\varphi(\boldsymbol{v}) \wedge \phi(\boldsymbol{w})) \to \{0, 1\}$ which, when applied to a k-bit vector of variables such as $\boldsymbol{x}_j$, yields a binary vector. Such a binary vector can then be interpreted as a signed number to give a value in the range $[-2^{k-1}, 2^{k-1} - 1]$. This construction is applied in lines 5-8 to find a vector $(\langle\!\langle \theta(\boldsymbol{x}_1) \rangle\!\rangle, \ldots, \langle\!\langle \theta(\boldsymbol{x}_n) \rangle\!\rangle) \in [-2^{k-1}, 2^{k-1} - 1]^n$ which satisfies both the disequality $(a_1, \ldots, a_n) \cdot \boldsymbol{x} \neq b$ and the formula $\varphi(\boldsymbol{v})$.

The algorithm is formulated in terms of some auxiliary functions: row(M, i) extracts row i from the matrix M where the first row is taken to be row 1 (rather

```
(1)   function affine(φ(v), {x_1, ..., x_n})
(2)       [A | b] := [0, ..., 0 | 1];
(3)       i := 0; r := 1;
(4)       while i < r do
(5)           (a_1, ..., a_n, b) := row([A | b], r − i);
(6)           let φ(w) hold iff (a_1, ..., a_n) · x ≠ b holds;
(7)           if φ(v) ∧ φ(w) has a satisfying truth assignment θ
(8)               [A' | b'] := [A | b] ⊔_aff [Id | (⟨⟨θ(x_1)⟩⟩, ..., ⟨⟨θ(x_n)⟩⟩)^T];
(9)               [A | b] := triangular([A' | b']);
(10)              r := number_of_rows([A | b]);
(11)          else i := i + 1;
(12)      endwhile
(13)  return [A | b]
```

Fig. 1. Calculating the affine closure of the Boolean formula $\varphi(\boldsymbol{v})$

than 0); triangular(M) puts M into an upper triangular form using Gaussian elimination; and number_of_rows(M) returns the number of rows in M.

The rows of $[A \mid \boldsymbol{b}]$ are considered in reverse order. Each iteration of the loop tests whether there exists a truth assignment of $\varphi(\boldsymbol{v})$ that also satisfies the $\phi(\boldsymbol{w})$ formula constructed from row $r - i$. If not, then every model of $\varphi(\boldsymbol{v})$ satisfies the affine equality $(a_1, \ldots, a_n) \cdot \boldsymbol{x} = b$ represented by row $r - i$. Hence the equality constitutes a description of the formula. The counter i is then incremented to examine a row which, thus far, has not been considered. Conversely, if the instance is satisfiable, then the solution is represented as a matrix $[\mathsf{Id} \mid (\langle\langle\theta(\boldsymbol{x}_1)\rangle\rangle, \ldots, \langle\langle\theta(\boldsymbol{x}_n)\rangle\rangle)^T]$ which is merged with $[A|\boldsymbol{b}]$. Merge is an $O(n^3)$ operation [13] that yields a new summary $[A'|\boldsymbol{b}']$ that enlarges $[A|\boldsymbol{b}]$ with the freshly found solution. The next iteration of the loop will either relax $[A|\boldsymbol{b}]$ by finding another solution, or verify that the row now describes $\varphi(\boldsymbol{v})$. Triangular form ensures that all rows beneath the one under consideration will never be effected by the merge. At most n iterations are required since the affine systems enumerated by the algorithm constitute an ascending chain over n variables [13].

Example Consider $\varphi(\boldsymbol{w}, \boldsymbol{x})$ which is an encoding of $\langle\langle\boldsymbol{z}\rangle\rangle = 2(\langle\langle\boldsymbol{v}\rangle\rangle + 1) + \langle\langle\boldsymbol{y}\rangle\rangle$ subject to the additional constraints that $-32 \leq \langle\langle\boldsymbol{v}\rangle\rangle \leq 31$ and $-32 \leq \langle\langle\boldsymbol{y}\rangle\rangle \leq 31$:

$$\varphi(\boldsymbol{w},\boldsymbol{x}) = \begin{cases} (\neg\boldsymbol{w}[0]) \wedge \left(\wedge_{i=0}^{6}\boldsymbol{w}[i+1] \leftrightarrow (\boldsymbol{v}[i] \oplus \wedge_{j=0}^{i-1}\boldsymbol{v}[j])\right) & \wedge \\ (\neg\boldsymbol{x}[0]) & \wedge \\ \left(\wedge_{i=0}^{6}\boldsymbol{x}[i+1] \leftrightarrow (\boldsymbol{w}[i] \wedge \boldsymbol{x}[i]) \vee (\boldsymbol{w}[i] \wedge \boldsymbol{y}[i]) \vee (\boldsymbol{x}[i] \wedge \boldsymbol{y}[i])\right) & \wedge \\ \left(\wedge_{i=0}^{7}\boldsymbol{z}[i] \leftrightarrow \boldsymbol{w}[i] \oplus \boldsymbol{x}[i] \oplus \boldsymbol{y}[i]\right) & \wedge \\ ((\boldsymbol{v}[7] \leftrightarrow \boldsymbol{v}[6]) \wedge (\boldsymbol{v}[6] \leftrightarrow \boldsymbol{v}[5])) \wedge ((\boldsymbol{y}[7] \leftrightarrow \boldsymbol{y}[6]) \wedge (\boldsymbol{y}[6] \leftrightarrow \boldsymbol{y}[5])) \end{cases}$$

Suppose $\boldsymbol{x}_1 = \boldsymbol{v}$, $\boldsymbol{x}_2 = \boldsymbol{y}$ and $\boldsymbol{x}_3 = \boldsymbol{z}$. The solutions that are found in each iteration are given in the left hand column. The $[A|\boldsymbol{b}]$ and $[A'|\boldsymbol{b}]$ are immediately left and right of the equality. The arrow indicates the row under consideration. The unsatisfiable case is first encountered in the final iteration. The algorithm returns $[2, 1, -1 \mid -2]$ and thus recovers $2\langle\langle\boldsymbol{v}\rangle\rangle + \langle\langle\boldsymbol{y}\rangle\rangle - \langle\langle\boldsymbol{z}\rangle\rangle = -2$ from $\varphi(\boldsymbol{v})$.

$$
\begin{array}{lll}
(0,0,2) & \left[\begin{array}{ccc|c} 0 & 0 & 0 & 1 \end{array}\right] \sqcup_{\text{aff}} \left[\begin{array}{ccc|c} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 2 \end{array}\right] & = \left[\begin{array}{ccc|c} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 2 \leftarrow \end{array}\right] \\
(-1,0,0) & \left[\begin{array}{ccc|c} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 2 \end{array}\right] \sqcup_{\text{aff}} \left[\begin{array}{ccc|c} 1 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{array}\right] & = \left[\begin{array}{ccc|c} 2 & 0 & -1 & -2 \\ 0 & 1 & 0 & 0 \leftarrow \end{array}\right] \\
(0,1,3) & \left[\begin{array}{ccc|c} 2 & 0 & -1 & -2 \\ 0 & 1 & 0 & 0 \end{array}\right] \sqcup_{\text{aff}} \left[\begin{array}{ccc|c} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 3 \end{array}\right] & = \left[\begin{array}{ccc|c} 2 & 1 & -1 & -2 \leftarrow \end{array}\right]
\end{array}
$$

Timings Computing affine abstractions for the feasible mode combinations in the examples of the previous section took no longer than 0.4s per mode. Each mode required no more than 5 SAT instances to be solved (which is considerably fewer than that required for inferring bit-level congruences [14]) with the join taking less than 5% of the overall runtime.

4 Applying Action Systems of Guarded Updates

Thus far we have derived transfer functions that are action systems of guarded updates $T = \{(g_1, u_1), \ldots, (g_m, u_m)\}$ where each guard g_i is a system of octagonal constraints over a set of bit-vectors $S' = \{\boldsymbol{x}_i \mid i \in [1,n]\}$ and each update u_i is a system of affine constraints over $S = \{\boldsymbol{x}_{i,\ell}, \boldsymbol{x}_{i,u}, \boldsymbol{x}'_{i,\ell}, \boldsymbol{x}'_{i,u} \mid i \in [1,n]\}$. In this section we show that the application of such an action system can be reduced to linear programming. To do so, we continue working with the assumption that S and S' are all k-bit and represent signed objects.

Thus consider applying T to a system of interval constraints over S' of the form $c = \bigwedge_{i=1}^{n} \ell_i \leq \langle\!\langle \boldsymbol{x}_i \rangle\!\rangle \leq u_i$ where $\{\ell_1, u_1, \ldots, \ell_n, u_n\} \subseteq [-2^{k-1}, 2^{k-1} - 1]$. In particular, consider the application of the guarded update (g_k, u_k). Suppose $g_k = \bigwedge_{i=1}^{p} \boldsymbol{\lambda}_i \cdot \boldsymbol{x} \leq d_i$ where $\boldsymbol{x}$ is the n-ary vector $\boldsymbol{x} = (\langle\!\langle \boldsymbol{x}_1 \rangle\!\rangle, \ldots, \langle\!\langle \boldsymbol{x}_n \rangle\!\rangle)$ and each n-ary coefficient vector $\boldsymbol{\lambda}_i \in \{-1, 0, 1\}^n$ has no more than two non-zero elements and $d_i \in \mathbb{Z}$. Furthermore, denote

$$
\begin{array}{ll}
\boldsymbol{x}_\ell = (\langle\!\langle \boldsymbol{x}_{1,\ell} \rangle\!\rangle, \ldots, \langle\!\langle \boldsymbol{x}_{n,\ell} \rangle\!\rangle) & \boldsymbol{x}_u = (\langle\!\langle \boldsymbol{x}_{1,u} \rangle\!\rangle, \ldots, \langle\!\langle \boldsymbol{x}_{n,u} \rangle\!\rangle) \\
\boldsymbol{x}'_\ell = (\langle\!\langle \boldsymbol{x}'_{1,\ell} \rangle\!\rangle, \ldots, \langle\!\langle \boldsymbol{x}'_{n,\ell} \rangle\!\rangle) & \boldsymbol{x}'_u = (\langle\!\langle \boldsymbol{x}'_{1,u} \rangle\!\rangle, \ldots, \langle\!\langle \boldsymbol{x}'_{n,u} \rangle\!\rangle)
\end{array}
$$

Then u_k can be written as $u_k = \bigwedge_{i=1}^{q} \boldsymbol{\mu}_{i,\ell} \cdot \boldsymbol{x}_\ell + \boldsymbol{\mu}_{i,u} \cdot \boldsymbol{x}_u + \boldsymbol{\mu}'_{i,\ell} \cdot \boldsymbol{x}'_\ell + \boldsymbol{\mu}'_{i,u} \cdot \boldsymbol{x}'_u = f_i$ where each n-ary vector $\boldsymbol{\mu}_{i,\ell}, \boldsymbol{\mu}_{i,u}, \boldsymbol{\mu}'_{i,\ell}, \boldsymbol{\mu}'_{i,u} \in \mathbb{Z}^n$ and $f_i \in \mathbb{Z}$. To specify a linear program, let $\boldsymbol{e}_j$ denote the n-ary elementary vector $(0, \ldots, 0, 1, 0, \ldots, 0)$ where $j-1$ zeros precede the one and $n-j$ zeros follow it. Then the value of $\langle\!\langle \boldsymbol{x}'_{j,\ell} \rangle\!\rangle$ for any $j \in [1,n]$ can be found by minimising $\boldsymbol{e}_j \cdot \boldsymbol{x}'_\ell$ subject to:

$$
P = \left\{ \begin{array}{llll}
\bigwedge_{i=1}^{q} \boldsymbol{\mu}_{i,\ell} \cdot \boldsymbol{x}_\ell + \boldsymbol{\mu}_{i,u} \cdot \boldsymbol{x}_u + \boldsymbol{\mu}'_{i,\ell} \cdot \boldsymbol{x}'_\ell + \boldsymbol{\mu}'_{i,u} \cdot \boldsymbol{x}'_u = f_i & & & \wedge \\
\bigwedge_{i=1}^{n} \boldsymbol{e}_i \cdot \boldsymbol{x}_\ell = \ell_i & \wedge & \bigwedge_{i=1}^{n} \boldsymbol{e}_i \cdot \boldsymbol{x}_u = u_i & \wedge \\
\bigwedge_{i=1}^{n} \boldsymbol{e}_i \cdot \boldsymbol{x} - \boldsymbol{e}_i \cdot \boldsymbol{x}_u \leq 0 & \wedge & \bigwedge_{i=1}^{n} \boldsymbol{e}_i \cdot \boldsymbol{x}_\ell - \boldsymbol{e}_i \cdot \boldsymbol{x} \leq 0 & \wedge \\
\bigwedge_{i=1}^{n} \boldsymbol{e}_i \cdot \boldsymbol{x}'_u \leq 2^{k-1} - 1 & \wedge & \bigwedge_{i=1}^{n} -\boldsymbol{e}_i \cdot \boldsymbol{x}'_\ell \leq 2^{k-1} & \wedge \\
\bigwedge_{i=1}^{p} \boldsymbol{\lambda}_i \cdot \boldsymbol{x} \leq d_i & & &
\end{array} \right.
$$

We let ℓ'_j denote this minima. Conversely let u'_j denote the value found by maximising $\boldsymbol{e}_j \cdot \boldsymbol{x}'_u$ subject to P. Note that although P is bounded, it is not necessarily feasible, which will be detected when solving the linear program (the first stage of two-phase simplex amounts of deciding feasibility by solving the so-called auxiliary problem [4]).

If P is feasible, the system of intervals generated for (g_k, u_k) is given by $\bigwedge_{j=1}^{n} \ell'_j \leq \langle\!\langle \boldsymbol{x}_j \rangle\!\rangle \leq u'_j$. If infeasible, the unsatisfiable constraint $\bot$ is output. Merging the systems generated by all the guarded updates (using power sets of intervals if desired) then gives the final output system of interval constraints.

Rationale The rationale for evaluating transfer functions with linear programming is the same as that which motivated deriving transfer functions with SAT: efficient solvers are readily (even freely) available for both. Moreover, although linear solvers have suffered problems relating to floats, steady progress has been made on both speeding up exact solvers [9] and deriving safe bounds on optima [24]. Thus soundness is no longer an insurmountable problem. We do not offer timings for evaluating transfer functions, partly because the focus of this paper (reflecting that of others [14, 20, 26]) is on deriving them; and partly because the linear programs that arise from the examples are trivial by industrial standards.

Linear programming versus closure Since the guards are octagonal one might wonder whether linear programming could be replaced with a closure calculation on octagons [19] that combines the interval constraints (which constitute a degenerate form of octagon) with the guard, thereby refining the interval bounds. These improved bounds could then be used to calculate the updates, without using linear programming. To demonstrate why this approach is sub-optimal, let $c = (0 \leq \langle\!\langle \boldsymbol{r0} \rangle\!\rangle \leq 1) \wedge (0 \leq \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq 1)$ and suppose the guard is the single constraint $g = \langle\!\langle \boldsymbol{r0} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq 1$. The closure of $c \wedge g$ would not refine the upper bounds on $\langle\!\langle \boldsymbol{r0} \rangle\!\rangle$ and $\langle\!\langle \boldsymbol{r1} \rangle\!\rangle$. Thus if these values were substituted into the update $\langle\!\langle \boldsymbol{r0}^\star_u \rangle\!\rangle = \langle\!\langle \boldsymbol{r0}_u \rangle\!\rangle + \langle\!\langle \boldsymbol{r1}_u \rangle\!\rangle$ then a maximal value of 2 would be derived for $\langle\!\langle \boldsymbol{r0}^\star_u \rangle\!\rangle$. This is safe but observe that maximising $\langle\!\langle \boldsymbol{r0}^\star_u \rangle\!\rangle$ subject to $c \wedge g$ yields the improved bound of 1, which illustrates why linear programming is preferable.

5 Related Work

The problem of computing transfer functions for numeric domains is as old as the field itself, and the seminal paper on polyhedral analysis discusses different ways to realise a transfer function for $x := x \times y$ [8, Sect. 4.2.1]. Granger [10] lamented the difficulty of handcrafting best transformers for congruences, but it took more than a decade before it was noticed that they can always be constructed for domains that satisfy the ascending chain condition [27]. The idea is to reformulate each application of a best transformer as a series of calls to a decision procedure such as a theorem prover. This differs from our work which aspires to evaluate a transfer function without a complicated decision procedure.

Contemporaneously it was observed that best transformers can be computed for intervals using BDDs [26]. The authors observe that if $g : [0, 2^8 - 1] \rightarrow$

$[0, 2^8 - 1]$ is a unary operation on an unsigned byte, then its best transformer $f : D \to D$ on $D = \{\emptyset\} \cup \{[\ell, u] \mid 0 \leq \ell \leq u < 2^8\}$ can be defined through interval subdivision. 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, 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 one day for each 8-bit operation. The approach does not scale to wider words nor to blocks.

Our work builds on that of Monniaux [20] who showed how transfer functions can be derived for operations over real-valued variables. His approach relies on universal quantifier elimination algorithm which is problematic for piecewise linear functions. Universal quantifier elimination also arises in work on inferring template constraints [12]. There the authors employ Farkas' lemma to transform universal quantification to existential quantification, albeit at the cost of compromising completeness (Farkas' lemma prevents integral reasoning).

The problem of handling limited precision arithmetic is discussed in [29]. Existing approaches are to: verify that no overflows arise using perfect numbers; revise the concretisation map to reflect truncation [29]; or deploy congruences [14, 23] where the modulo is a power of two. Our work suggests handling wraps in the generation of the transfer functions, which we think is natural.

6 Concluding Discussion

This paper advocates deriving transfer functions from Boolean formulae since the elimination of universal quantifiers is trivial in this domain. Boolean formulae are natural candidates for expressing arithmetic, logical and bitwise operations, allowing transfer functions to be derived for blocks of code that would otherwise only be amenable to the coarsest of approximation. The paper shows how to distill transfer functions that are action systems of guarded updates, where the guards are octagonal inequalities and the updates are linear affine equalities. This formulation enables the application of a transfer function for a basic block to be reduced to a series of linear programming problems. Although we have illustrated the approach using octagons, there is no reason why richer classes of template constraints [28] could not be deployed to express the guards. Moreover, linear affine equalities could be substituted with polynomial equalities of degree at most d [22], say, and correspondingly linear programming replaced with non-linear programming. Thus the ideas presented in the paper generalise quite naturally. Finally, the approach will only become more attractive as linear solvers and SAT solvers continue to improve both in terms of efficiency and scalability.

Acknowledgements We particularly thank David Monniaux for discussions at VMCAI 2010 in Madrid that motivated writing up the ideas in this paper. We thank Professor Stefan Kowalewski for his financial support that was necessary to initiate our collaboration. This work was also supported, in part, by a Royal Society industrial secondment and a Royal Society travel grant.

References

1. Atmel Corporation. The Atmel 8-bit AVR Microcontroller with 16K Bytes of In-system Programmable Flash, 2009. www.atmel.com/atmel/acrobat/doc2466.pdf.
2. G. Balakrishnan. *WYSINWYX: What You See Is Not What You eXecute*. PhD thesis, Computer Sciences Department, University of Wisconsin, Madison, Wisconsin, USA, August 2007.
3. 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.
4. V. Chvátal. *Linear Programming*. W. H. Freeman and Company, 1983.
5. 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.
6. 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.
7. P. Cousot and R. Cousot. Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. In *PLILP*, volume 631 of *LNCS*, pages 269–295. Springer, 1992.
8. P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In *POPL*, pages 84–97. ACM Press, 1978.
9. J. Edmonds and J.-F. Manrras. Note sur les Q-matrices d'Edmonds. *Recherche Opérationnella*, 32(2):203–209, 1997.
10. P. Granger. Static Analysis of Arithmetical Congruences. *International Journal of Computer Mathematics*, 30(13):165–190, 1989.
11. P. Granger. Static Analyses of Congruence Properties on Rational Numbers. In *SAS*, volume 1302 of *LNCS*, pages 278–292, 1997.
12. S. Gulwani, S. Srivastava, and R. Venkatesan. Program Analysis as Constraint Solving. In *PLDI*, pages 281–292. ACM Press, 2008.
13. M. Karr. Affine Relationships among Variables of a Program. *Acta Informatica*, 6:133–151, 1976.
14. A. King and H. Søndergaard. Automatic Abstraction for Congruences. In *VMCAI*, volume 5944 of *LNCS*, pages 197–213. Springer, 2010.
15. D. Kroening and O. Strichman. *Decision Procedures*. Springer, 2008.
16. D. Le Berre. SAT4J: Bringing the power of SAT technology to the Java platform, 2010. http://www.sat4j.org/.
17. K. Marriott. Frameworks for Abstract Interpretation. *Acta Informatica*, 30(2):103–129, 1993.
18. A. Miné. A New Numerical Abstract Domain Based on Difference-Bound Matrices. In *PADO*, volume 2053 of *LNCS*, pages 155–172. Springer, 2001.
19. A. Miné. The Octagon Abstract Domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.
20. D. Monniaux. Automatic Modular Abstractions for Linear Constraints. In *POPL*, pages 140–151. ACM Press, 2009.
21. D. Monniaux. Personal communication with Monniaux at VMCAI, January 2010.
22. M. Müller-Olm and H. Seidl. A Note on Karr's Algorithm. In *ICALP*, volume 3142 of *LNCS*, pages 1016–1028. Springer, 2004.
23. M. Müller-Olm and H. Seidl. Analysis of Modular Arithmetic. *ACM Trans. Program. Lang. Syst.*, 29(5), August 2007.

24. A. Neumaier and O. Shcherbina. Safe Bounds in Linear and Mixed-Integer Linear Programming. *Math. Program.*, 99(2):283–296, 2004.
25. D. A. Plaisted and S. Greenbaum. A Structure-Preserving Clause Form Translation. *Journal of Symbolic Computation*, 2(3):293–304, September 1986.
26. 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.
27. T. Reps, M. Sagiv, and G. Yorsh. Symbolic Implementation of the Best Transformer. In *VMCAI*, volume 2937 of *LNCS*, pages 252–266. Springer, 2004.
28. S. Sankaranarayanan, H. Sipma, and Z. Manna. Constraint based linear relations analysis. In *SAS*, volume 3148 of *LNCS*, pages 53–68. Springer, 2004.
29. A. Simon and A. King. Taming the Wrapping of Integer Arithmetic. In *SAS*, volume 4634 of *LNCS*, pages 121–136. Springer, 2007.