Loop Leaping with Closures

Sebastian Biallas[1], Jörg Brauer[1,2], Andy King[3,4] and Stefan Kowalewski[1]

[1] Embedded Software Laboratory, RWTH Aachen University, Germany
[2] Verified Systems International GmbH, Bremen, Germany
[3] Portcullis Computer Security, Pinner, UK
[4] School of Computing, University of Kent, UK

Abstract. Loop leaping is the colloquial name given to a form of program analysis in which summaries are derived for nested loops starting from the innermost loop and proceeding in a bottom-up fashion considering one more loop at a time. Loop leaping contrasts with classical approaches to finding loop invariants that are iterative; loop leaping is compositional requiring each stratum in the nest of loops to be considered exactly once. The approach is attractive in predicate abstraction where disjunctive domains are increasingly used that present long ascending chains. This paper proposes a simple and an efficient approach for loop leaping for these domains based on viewing loops as closure operators.

1 Introduction

Abstract interpretation [9] provides a compelling theory for modelling a program with descriptions of concrete data values. Not only does it show how domains can be defined, refined and related to their concrete counterparts, but it provides a methodology for constructing transformers that simulate the behaviour of the primitive operations that arise in a program. Best transformers can, at least in principle, always be automatically constructed for domains of finite height [30] which, notably, includes the abstract domain of conjunctions of predicates [4] that has proved so popular in verification [14]. Techniques for deriving transformers for whole blocks of code have recently emerged due, in part, to the development of robust decision procedures [6,21,25] and efficient quantifier elimination techniques [7,24,27]. The step beyond blocks is the automatic synthesis of transformers for loops.

Calculational techniques for deriving transformers for loops are colloquially referred to loop leaping [2] or loop frogging [22,23]. These evocative terms capture the central idea of jumping over the computational obstacle presented by repeatedly reaching, iterating and stabilising on each loop in a nest of loops. Instead, the whole loop nest is summarised in a straight-line block, ideally with the summary computed in a compositional fashion, starting with the innermost and ending with the outermost loop. The case for loop summarisation becomes more convincing for domains with long chains such as those admitted by Boolean formulae over large numbers of predicates [28]. Boolean formulae can be widened, even in ways that are sensitive to the underlying Boolean function rather than

merely its representation [20], yet it is our contention that the rich structure of formulae aids rather than impedes loop analysis when loop leaping is applied.

Ideally one would derive a best transformer that summarises the execution of a loop, or loop nest, to the limits of what is expressible in the abstract domain. A best transformer is exactly that: a mapping from the set of input descriptions to the set of output descriptions where the output description given by the transformer is the most precise characterisation of the set of all the output states that are reachable from all the input states described by the input description. This immediately presents a problem for Boolean formulae: the number of input descriptions. Even for the sub-class of monotonic Boolean formulae, the simplest domain that can express both conjunctive and disjunctive properties, the number of formulae grows rapidly with the number of predicates: 2, 3, 6, 20, 168, 7581, 7828354, 2414682040998, 5613043722868755790788 [37]. It is therefore not surprising that previous work has sought to curtail the representation, for instance, by bounding the number of disjuncts [28]. Without exploiting common structure in and between the input and output formulae, the only realistic prospective is to design a transformer whose representation is suitably compact and whose summary is sufficiently precise: the former can be ensured through design but the latter can only be tested empirically.

The contribution of this paper is simple. It is to show how loop transformers can be computed from maps of the form $\uparrow f : \Sigma \to \wp(\wp(\Sigma))$ where $\Sigma = \{\sigma_1, \ldots, \sigma_n\}$ is the finite set of predicates under consideration. If $\sigma_i \in \Sigma$ then $\Delta_i = \uparrow f(\sigma_i)$ is interpreted as a monotonic formula in DNF. For example $\{\{\sigma_1, \sigma_2\}, \{\sigma_1, \sigma_3\}\}$ represents the formula $\theta = (\sigma_1 \wedge \sigma_2) \vee (\sigma_1 \wedge \sigma_3) = \sigma_1 \wedge (\sigma_2 \vee \sigma_3)$. Crucially the map $\uparrow f$ is defined by just n formulae $\Delta_1 = \uparrow f(\sigma_1), \ldots, \Delta_n = \uparrow f(\sigma_n)$. The map $\uparrow f$ does not constitute a loop transformer itself since it only specifies how to map an input formula, which is one of the predicates, to an output formula. Yet $\uparrow f$ is designed so that logical combinators can be applied to $\Delta_1, \ldots, \Delta_n$ to compute an output formula for an arbitrary input formula. To illustrate, if the input formula is θ then the output formula is $\Delta_1 \wedge (\Delta_2 \vee \Delta_3)$, where here the distinction between a monotonic Boolean function and its representation is blurred. The construction rests on $\uparrow f : \Sigma \to \wp(\wp(\Sigma))$, or rather its extension $\uparrow f : \wp(\wp(\Sigma)) \to \wp(\wp(\Sigma))$, being a closure operator, that is, a map which is monotonic, idempotent and extensive (extensivity means that the operator relaxes a formula whenever it is applied). The centrality of these three concepts explains the title of the paper and the (mysterious) $\uparrow$ symbol that indicates closure. These three properties square with the way a loop transformer maps an input formula to an output formula which describes the final state of a loop. This fit leads to a loop summarisation method that is both simple and effective.

Expositionally this paper is laid out as follows: First, Sect. 2 explains the key ingredients of our method for both, unnested and nested loops, by means of an example, followed by a formalisation and correctness arguments in Sect. 3. Then, Sect. 4 presents experimental evidence which compares the precision of our techniques to related ones based on predicate abstraction. Finally the paper concludes with a survey of related work in Sect. 5 and a discussion in Sect. 6.

```
(1)  assume  i = 0;
(2)  assume  n > 0;
(3)  while  i < n  do
(4)       b := nondet();
(5)       if  b ≠ 0  then
(6)            i := i + 1;
(7)       else
(8)            skip;
(9)       endif
(10) endwhile
```

```
if  i < n  then
     b := nondet();
     if  b ≠ 0  then
          i := i + 1;
     else
          skip;
     endif
else
     skip
endif
```

```
(1)  if  i < n  then
(2)       b′ := nondet();
(3)       if  b′ ≠ 0  then
(4)            i′ := i + 1;
(5)       else
(6)            i′ := i;
(7)       endif
(8)  else
(9)       i′ := i;
(10) endif
(11) n′ := n
```

Fig. 1. Single loop example: (a) code; (b) loop block; (c) loop block in a SSA-form

2 Worked Examples

The ethos of our method is to summarise a loop with a closure operator on the domain of monotonic Boolean formulae, D, where the predicates are drawn from a given finite set of predicates, Σ, that is defined up-front. Monotonic Boolean formulae are a class of propositional functions which take the following syntactic form: if $\sigma \in \Sigma$ then $\sigma \in D$ and if $f_1, f_2 \in D$ then it follows $f_1 \wedge f_2 \in D$ and $f_1 \vee f_2 \in D$ [31]. The domain D is ordered by entailment $\models$ and with appropriate factoring (the details of which are postponed to the sequel) a finite lattice $\langle D, \models, \vee, \wedge \rangle$ can be obtained.

To illustrate how a loop can be summarised using closures over D consider the program that is listed to the leftmost column of Figure 1. Observe that the loop transforms the state that the program has when the loop is first encountered into the state that is obtained by repeated applications of the loop body. A loop summary expresses this transformation. Since state is described in terms of monotonic formulae, the summary is itself a mapping from an input formula to an output formula. The input formula describes the initial state at the head of the loop: the state of the program when the loop is first encountered. The output formula describes all the states that are reachable at the head of the loop, by repeatedly applying the loop body, from any of the initial states. Since the number of monotonic formulae grows rapidly with $|\Sigma|$ [37], the challenge is to find a way to summarise a loop that is both descriptive and yet can be represented compactly and derived straightforwardly.

Observe that the while loop is equivalent to repeated applications of the block of statements in the middle column that will collectively be referred to as S. Suppose too that the set of predicates is defined as $\Sigma = \Sigma_0 \cup \Sigma_1 \cup \Sigma_2$ where:

$$\begin{aligned}\Sigma_0 &= \{\ (n < 0)\,,\ (n = 0)\,,\ (n > 0)\}\\ \Sigma_1 &= \{\ \ (i < 0)\,,\ \ (i = 0)\,,\ \ (i > 0)\}\\ \Sigma_2 &= \{\ \ (i < n),\ \ (i = n),\ \ (i > n)\}\end{aligned}$$

Although Σ is entirely natural given the predicates in the program, observe that S does not mutate n, hence S does not alter the truth or falsity of the predicates of Σ_0. We shall thus restrict our attention to summaries over the predicates $\Sigma_1 \cup \Sigma_2$; extending the summaries to Σ increases the number of cases that need to be considered without offering the reader fresh insight.

2.1 Closing the Loop over Σ

Using SMT-based reachability analysis [8,14], a function f is computed which maps input formulae, which coincide with each of the predicates $\sigma \in \Sigma_1 \cup \Sigma_2$, to their corresponding output formulae. To derive the output formulae, S is put into a form of single static assignment [10], which gives the block listed in the rightmost column, denoted S'. The three paths through S' correspond to three systems of constraints that are:

$$\begin{aligned} c_1 &= (i < n) \wedge (b' \neq 0) \wedge (i' = i + 1) \wedge (n' = n) \\ c_2 &= (i < n) \wedge \neg(b' \neq 0) \wedge (i' = i) \wedge (n' = n) \\ c_3 &= \neg(i < n) \wedge (i' = i) \wedge (n' = n) \end{aligned}$$

To illustrate, consider computing the abstract transformer $\alpha_{\Sigma_1' \cup \Sigma_2'}((i = 0) \wedge c_1)$ of $(i = 0) \in \Sigma_1$ subject to path c_1, where the abstraction map α is outlined below and Σ_1' and Σ_2' denote sets of predicates, analogous to Σ_1 and Σ_2 respectively, but defined over primed output variables. Passing $(i = 0) \wedge c_1$ to an SMT solver gives a model $\mathbf{m}_1$, e.g.:

$$\mathbf{m}_1 = \big\{\, (i = 0) \wedge (n = 2) \wedge (i' = 1) \wedge (n' = 2) \,\big\}$$

Since we can check that a concrete model $\mathbf{m}$ satisfies a given predicate $\sigma \in \Sigma$, that is, $\mathbf{m} \in \gamma_\Sigma(\sigma)$, then $\alpha_\Sigma(\mathbf{m})$ can be computed thus:

$$\alpha_\Sigma(\mathbf{m}) = \bigwedge \{\sigma \in \Sigma \mid \mathbf{m} \in \gamma_\Sigma(\sigma)\}$$

Note that α is parametric in the set Σ. By abstracting $\mathbf{m}_1$, we obtain $\alpha_{\Sigma'}(\mathbf{m}_1) = (i' > 0) \wedge (i' < n')$. In a second iteration, we add $\neg\alpha_{\Sigma'}(\mathbf{m}_1)$ to the SMT instance as a blocking clause. Then, passing $(i = 0) \wedge c_1 \wedge \neg\alpha_{\Sigma'}(\mathbf{m}_1)$ to a solver yields a different model $\mathbf{m}_2$, in which all concrete values described by $\alpha_{\Sigma'}(\mathbf{m}_1)$ are blocked. Suppose $\mathbf{m}_2$ is defined as:

$$\mathbf{m}_2 = \big\{\, (i = 0) \wedge (n = 1) \wedge (i' = 1) \wedge (n' = 1) \,\big\}$$

This model induces an output $\alpha_{\Sigma'}(\mathbf{m}_2) = (i' > 0) \wedge (i' = n')$. Then, the formula $(i = 0) \wedge c_1 \wedge \neg\alpha_{\Sigma'}(\mathbf{m}_1) \wedge \neg\alpha_{\Sigma'}(\mathbf{m}_2)$ becomes unsatisfiable, and thus $(i = 0) \wedge c_1 \models \alpha_{\Sigma'}(\mathbf{m}_1) \vee \alpha_{\Sigma'}(\mathbf{m}_2)$, which entails $f(i = 0 \wedge c_1) = (i' > 0) \wedge ((i < n') \vee (i' = n'))$. Applying this strategy to $(i = 0) \wedge c_2$ and $(i = 0) \wedge c_3$ gives:

$$\begin{aligned} f((i = 0) \wedge c_1) &= (i' > 0) \wedge ((i' = n') \vee (i' < n')) \\ f((i = 0) \wedge c_2) &= (i' = 0) \wedge (i' < n') \\ f((i = 0) \wedge c_3) &= (i' = 0) \wedge ((i' = n') \vee (i' > n')) \end{aligned}$$

Combining these three results we derive a formula which describes the effect of executing S under input that satisfies the predicate $\sigma = (i = 0)$. In what follows simplifications have been applied to make the presentation more accessible:

$$\begin{aligned} f(i = 0) &= \textstyle\bigvee_{j=1}^{3} f((i = 0) \wedge c_j) \\ &= f((i = 0) \wedge c_1) \vee f((i = 0) \wedge c_2) \vee f((i = 0) \wedge c_3) \\ &= (i' = 0) \vee ((i' > 0) \wedge ((i' < n') \vee (i' = n'))) \end{aligned}$$

Likewise, for the remaining predicates in Σ, we compute:

$$\begin{aligned} f(i < 0) &= (i' < 0) \vee ((i' = 0) \wedge ((i' < n') \vee (i' = n'))) \\ f(i > 0) &= (i' > 0) \\ f(i < n) &= (i' < n') \vee (i' = n') \\ f(i = n) &= (i' = n') \\ f(i > n) &= (i' > n') \end{aligned}$$

The map f characterises one iteration of the block S. To describe many iterations, f is relaxed to a closure, that is, an operator over D which is idempotent, monotonic and extensive. Idempotent so as to capture the effect to repeatedly applying S until the output formula does not change; monotonic since if the input formula is relaxed then so is the output formula; and extensive so as to express that the output formula is weaker than in the input formula. The last point deserves amplification: the input formula characterises the state that holds when the loop is first encountered whereas the output summarises that states that hold when the loop head is first and then subsequently encountered, hence the former entails the latter.

With renaming applied to eliminate the auxiliary predicates of $\Sigma'_1 \cup \Sigma'_2$, the closure of $f(i = 0)$, denoted $\uparrow f(i = 0)$, is computed so as to satisfy:

$$\uparrow f(i = 0) = \uparrow f(i = 0) \vee (\uparrow f(i > 0) \wedge \uparrow f(i < n)) \vee (\uparrow f(i > 0) \wedge \uparrow f(i = n))$$

Likewise, the closures for all predicates in Σ are required such that:

$$\begin{aligned} \uparrow f(i < 0) &= \uparrow f(i < 0) \vee (\uparrow f(i = 0) \wedge \uparrow f(i < n)) \vee (\uparrow f(i = 0) \wedge \uparrow f(i = n)) \\ \uparrow f(i > 0) &= \uparrow f(i > 0) \\ \uparrow f(i < n) &= \uparrow f(i < n) \vee \uparrow f(i = n) \\ \uparrow f(i = n) &= \uparrow f(i = n) \\ \uparrow f(i > n) &= \uparrow f(i > n) \end{aligned}$$

This recursive equation system can be solved iteratively until it stabilises, a property that is guaranteed due to monotonicity and finiteness of the domain. In fact it is straightforward to see that $\uparrow f(i > 0) = (i > 0)$, $\uparrow f(i = n) = (i = n)$, and $\uparrow f(i > n) = (i > n)$. Using substitution, we then obtain $\uparrow f(i < n) = (i < n) \vee (i = n)$. Likewise, for $(i = 0)$, we compute:

$$\uparrow f(i = 0) = (i = 0) \vee ((i > 0) \wedge (i < n)) \vee ((i > 0) \wedge (i = n))$$

Also by simplification we obtain:

$$\begin{aligned} \uparrow f(i < 0) &= (i < 0) \vee ((i > 0) \wedge ((i < n) \vee (i = n))) \vee (\uparrow f(i = 0) \wedge (i = n)) \\ &= (i < 0) \vee (i < n) \vee (i = n) \end{aligned}$$

which completes the derivation of the closure.

2.2 Applying Closures

Thus far, we have computed a function $\uparrow f$ that maps each predicate $\sigma \in \Sigma$ to a formula that represents the states reachable at the head of the loop from σ. Yet $\uparrow f$ can be interpreted as more than a loop transformer over just Σ since if $\sigma_1, \sigma_2 \in \Sigma$ it follows that:

$$\uparrow f(\sigma_1 \wedge \sigma_2) \models \uparrow f(\sigma_1) \wedge \uparrow f(\sigma_2)$$

This holds because the closure operator is monotonic. Moreover, due to the rich structure of our domain, we also have:

$$\uparrow f(\sigma_1 \vee \sigma_2) = \uparrow f(\sigma_1) \vee \uparrow f(\sigma_2)$$

This follows from the way $\sigma_1 \vee \sigma_2$ is formally interpreted as set union and the operator $\uparrow f$ is defined so as to distribute over union. The force of this is that $\uparrow f$ can be lifted to an arbitrary formula over Σ, thereby prescribing a loop transformer that is sufficiently general to handle any conceivable input formula. As an example, suppose that the loop is first reached with state described by the input formula $(i = 0) \wedge (i < n)$. Then

$$\begin{aligned}
&\uparrow f((i = 0) \wedge (i < n)) \\
&\quad \sqsubseteq \uparrow f(i = 0) \wedge \uparrow f(i < n) \\
&\quad = ((i = 0) \vee ((i > 0) \wedge (i < n)) \vee ((i > 0) \wedge (i = n))) \wedge ((i < n) \vee (i = n)) \\
&\quad = ((i = 0) \vee (i > 0)) \wedge ((i < n) \vee (i = n))
\end{aligned}$$

which, with some simplifications applied, describes all the states that are reachable at the head of the loop. The complete loop transformer then amounts to intersecting this formula with the negation of the loop-condition, that is, $(i = n) \vee (i > n)$, which gives the formula $((i = 0) \vee (i > 0)) \wedge (i = n)$ which characterises the states that hold on exit from the loop as desired. The importance of this final step cannot be overlooked.

2.3 Leaping Nested Loops

The strength of the construction is that it can be used to compositionally summarise nested loops. Given an inner loop S_I, we first compute a loop transformer $\uparrow f_I$, which is then incorporated into the body of the outer loop S_O. Our analysis thus computes loop transformers bottom-up, which is both attractive for conceptual as well computational reasons. As an example, consider the program in Fig. 2 (`nested.c` from [15]) with the sets of predicates defined as:

$$\begin{aligned}
\Sigma_1 &= \{(y < 0), (y = 0), (y > 0)\} & \Sigma_4 &= \{(t < m),\ (t = m),\ (t > m)\} \\
\Sigma_2 &= \{(t < 0),\ (t = 0),\ (t > 0)\} & \Sigma_5 &= \{(y < m), (y = m), (y > m)\} \\
\Sigma_3 &= \{(t < y),\ (t = y),\ (t > y)\}
\end{aligned}$$

```
(1)  assume  y = 0;
(2)  assume  m ≥ 0;
(3)  assume  t = 0;
(4)  while  y < m  do
(5)       y := y + 1;
(6)       t := 0;
(7)       while  t < y  do
(8)            t := t + 1;
(9)       endwhile
(10) endwhile
(11) assert  y = m
```

```
(1)  assume  y = 0;
(2)  assume  m ≥ 0;
(3)  assume  t = 0;
(4)  while  y < m  do
(5)       y′ := y + 1;
(6)       t′ := 0;
(7)       if  y′ < 0  then  assume  y″ < 0  endif
(8)       if  y′ = 0  then  assume  y″ = 0  endif
(9)       if  y′ > 0  then  assume  y″ > 0  endif
(10)      if  t′ = 0  then  assume  t″ ≥ 0  endif
(11)      if  t′ > 0  then  assume  t″ > 0  endif
(12)      if  t′ < y′  then  assume  t″ ≤ y″  endif
(13)      if  t′ = y′  then  assume  t″ = y″  endif
(14)      if  t′ > y″  then  assume  t″ > y″  endif
(15)      assume  t″ ≥ y″
(16) endwhile
(17) assert  y = m
```

Fig. 2. Bottom-up derivation of transformer for a nested loop from [15]

Applying our technique to the inner loop on predicates $\Sigma_1 \cup \Sigma_2 \cup \Sigma_3$, we compute the map f_I as follows:

$$f_I(y < 0) = (y < 0)$$
$$f_I(y = 0) = (y = 0)$$
$$f_I(y > 0) = (y > 0)$$
$$f_I(t < y) = (t < y) \vee (t = y)$$
$$f_I(t = y) = (t = y)$$
$$f_I(t > y) = (t > y)$$

$$f_I(t < 0) = (t < 0) \vee (t = 0)$$
$$f_I(t = 0) = (t = 0) \vee (t > 0)$$
$$f_I(t > 0) = (t > 0)$$

Then, as before, we compute the closure of f_I to give:

$$\uparrow f_I(y < 0) = (y < 0)$$
$$\uparrow f_I(y = 0) = (y = 0)$$
$$\uparrow f_I(y > 0) = (y > 0)$$
$$\uparrow f_I(t < 0) = \uparrow f_I(t < 0) \vee \uparrow f_I(t = 0) = (t < 0) \vee (t = 0) \vee (t > 0) = \mathsf{true}$$
$$\uparrow f_I(t = 0) = (t = 0) \vee (t > 0)$$
$$\uparrow f_I(t > 0) = (t > 0)$$
$$\uparrow f_I(t < y) = \uparrow f_I(t < y) \vee \uparrow f_I(t = y) = (t < y) \vee (t = y)$$
$$\uparrow f_I(t = y) = (t = y)$$
$$\uparrow f_I(t > y) = (t > y)$$

To abstract the outer loop in Fig. 2, we replace the inner loop, defined at lines (7)–(9) on the left, by its summary. This gives the program on the right. Here,

lines (7)–(14) encode an application of the closure, whereas line (15) models the loop exit condition of S_I. Note that lines 10 and 12 relax strict inequalities to non-strict inequalities to simultaneously express two predicates (which is merely for presentational purposes). Even though the transformed program appears to have multiple paths, it is not treated as such: lines (7)–(14) rather model auxiliary constraints imposed by the closure on a single path.

Next a predicate transformer f_O for the outer loop S_O is computed which amounts, like before, to reachability analysis over the predicates $\bigcup_{i=1}^{5} \Sigma_i$. We obtain a map $f_O : \Sigma \to \wp(\wp(\Sigma))$ defined as:

$$
\begin{aligned}
f_O(y<0) &= ((y<0) \vee (y=0)) \wedge ((t=0) \vee (t>0)) \wedge ((t=y) \vee (t>y)) \\
f_O(y=0) &= (y>0) \wedge (t>0) \wedge (t=y) \\
f_O(y>0) &= (y>0) \wedge ((t=0) \vee (t>0)) \wedge ((t=y) \vee (t>y)) \\
f_O(t<0) &= ((t=0) \vee (t>0)) \wedge ((t=y) \vee (t>y)) \\
f_O(t=0) &= ((t=0) \vee (t>0)) \wedge ((t=y) \vee (t>y)) \\
f_O(t>0) &= ((t=0) \vee (t>0)) \wedge ((t=y) \vee (t>y)) \\
f_O(t<y) &= (t=y) \\
f_O(t=y) &= (t=y) \\
f_O(t>y) &= (t>y) \\
f_O(t<m) &= (t=y) \vee (t>y) \\
f_O(t=m) &= (t=y) \vee (t>y) \\
f_O(t>m) &= (t=y) \vee (t>y) \\
f_O(y<m) &= ((y<m) \vee (y=m)) \wedge ((t=0) \vee (t>0)) \wedge ((t=y) \vee (t>y)) \\
f_O(y=m) &= (y=m) \\
f_O(y>m) &= (y>m)
\end{aligned}
$$

Analogous to before, closure computation amounts to substituting the predicates in the image of f_O. In case of the predicate $(y=0) \in \Sigma_1$, for example, computing the closure of $f_O(y=0) = (y>0) \wedge (t>0) \wedge (t=y)$ amounts to substituting $(y>0)$, $(t>0)$ and $(t=y)$ by $\uparrow f_O(y>0)$, $\uparrow f(t>0)$ and $\uparrow f(t=y)$, respectively. By repeated substitution (with entailment checking), we obtain the following closures for $(y=0) \in \Sigma_1$, $(t=0) \in \Sigma_3$ and $(y<m) \in \Sigma_5$:

$$
\begin{aligned}
\uparrow f_O(y=0) &= (y>0) \wedge (t>0) \wedge (t=y) \\
\uparrow f_O(t=0) &= ((t=0) \vee (t>0)) \wedge ((t=y) \vee (t>y)) \\
\uparrow f_O(y<m) &= ((y<m) \vee (y=m)) \wedge ((t=0) \vee (t>0)) \wedge ((t=y) \vee (t>y))
\end{aligned}
$$

Likewise, we close f for the remaining predicates.

To illustrate the precision of this type of transformer for nested loops, suppose $(y=0) \wedge (y<m) \wedge (t=0)$ holds on enter into the outer loop. The loop transformer for $(y=0) \wedge (y<m) \wedge (t=0)$ is computed as $\uparrow f_O(y=0) \wedge \uparrow f_O(y<m) \wedge \uparrow f_O(t=0)$, which simplifies to give:

$$
\begin{aligned}
&\uparrow f_O(y=0) \wedge \uparrow f_O(y<m) \wedge \uparrow f_O(t=0) \\
&\qquad = \begin{cases} (y>0) \wedge (t>0) \wedge (t=y) & \wedge \\ ((t=0) \vee (t>0)) \wedge ((t=y) \vee (t>y)) & \wedge \\ ((y<m) \vee (y=m)) \wedge ((t=0) \vee (t>0)) \wedge ((t=y) \vee (t>y)) \end{cases} \\
&\qquad = (y>0) \wedge (t>0) \wedge (t=y) \wedge ((y<m) \vee (y=m))
\end{aligned}
$$

By conjoining this output of the outer loop with the exit-condition $(y \geq m)$, we obtain the post-state of the program after the loop:

$$\begin{aligned}\uparrow f_O((y = 0) \wedge (y < m) \wedge (t = 0))) \wedge (y \geq m) \\ = (y > 0) \wedge (t > 0) \wedge (t = y) \wedge ((y < m) \vee (y = m)) \wedge (y \geq m) \\ = (y > 0) \wedge (t > 0) \wedge (t = y) \wedge (y = m)\end{aligned}$$

Clearly, the assertion in line (11) of Fig. 2 follows, as is required.

3 Semantics

In this section we formalise our approach to predicate abstraction and demonstrate its correctness. The starting is a (countable) finite concrete domain B that is interpreted as the set of possible program states, for instance, $B = [-2^{31}, 2^{31}-1]^2$ for a program with just two 32-bit signed integer variables. For generality the definition of B is left open. To illustrate the compositional nature of our analysis, the formal study focuses on a language $\mathcal{L}$ of structured statements S defined by

$$S ::= \mathsf{skip} \mid \mathsf{assume}(\rho) \mid \mathsf{transform}(\tau) \mid S; S \mid \mathsf{if}\ \rho\ \mathsf{then}\ S\ \mathsf{else}\ S \mid \mathsf{while}\ \rho\ \mathsf{do}\ S$$

where $\tau \subseteq B \times B$ is a relation between assignments and $\rho \subseteq B$ is a predicate. Since τ is a binary relation, rather than a function, the statement $\mathsf{transform}(\tau)$ can express non-determinism. If $\tau = \{\langle x, y\rangle \times \langle x', y'\rangle \in ([-2^{31}, 2^{31}-1]^2)^2 \mid x' = x\}$, for instance, then the statement $\mathsf{transform}(\tau)$ preserves the value of x but assigns y to an arbitrary 32-bit value. For brevity of presentation, we define the composition of a unary relation $\rho \subseteq B$ with a binary relation $\tau \subseteq B \times B$ which is defined thus $\rho \circ \tau = \{b' \in B \mid b \in \rho \wedge \langle b, b'\rangle \in \tau\}$ (and should not be confused with function composition whose operands are sometimes written in the reverse order). We also define $\neg\rho = B \setminus \rho$ for $\rho \subseteq B$.

3.1 Concrete Semantics

Because of the non-deterministic nature of $\mathsf{transform}(\tau)$ the semantics that is used as the basis for abstraction operates on sets of values drawn from B. The semantics is denotational in nature, associating with each statement in a program with a mapping $\wp(B) \to \wp(B)$ that expresses its behaviour. The function space $\wp(B) \to \wp(B)$ is ordered pointwise by $f_1 \sqsubseteq f_2$ iff $f_1(\rho) \subseteq f_2(\rho)$ for all $\rho \subseteq B$. In fact $\langle \wp(B) \to \wp(B), \sqcap, \sqcup, \lambda\rho.\emptyset, \lambda\rho.B\rangle$ is a complete lattice where $f_1 \sqcap f_2 = \lambda\rho.f_1(\rho) \cap f_2(\rho)$ and likewise $f_1 \sqcup f_2 = \lambda\rho.f_1(\rho) \cup f_2(\rho)$. The complete lattice $\mathcal{L} \to \wp(B) \to \wp(B)$ is defined analogously. With this structure in place a semantics for statements can be defined:

Definition 1. The mapping $[\![\cdot]\!]_C : \mathcal{L} \to \wp(B) \to \wp(B)$ is the least solution to:

$$\begin{aligned}[\![\mathsf{skip}]\!]_C &= \lambda\sigma.\sigma \\ [\![\mathsf{assume}(\rho)]\!]_C &= \lambda\sigma.\sigma \cap \rho \\ [\![\mathsf{transform}(\tau)]\!]_C &= \lambda\sigma.\sigma \circ \tau \\ [\![S_1; S_2]\!]_C &= \lambda\sigma.[\![S_2]\!]_C([\![S_1]\!]_C(\sigma)) \\ [\![\mathsf{if}\ \rho\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2]\!]_C &= \lambda\sigma.([\![S_1]\!]_C(\sigma \cap \rho)) \cup ([\![S_2]\!]_C(\sigma \cap \neg\rho)) \\ [\![\mathsf{while}\ \rho\ \mathsf{do}\ S]\!]_C &= \lambda\sigma.([\![\mathsf{while}\ \rho\ \mathsf{do}\ S]\!]_C([\![S]\!]_C(\sigma \cap \rho))) \cup (\sigma \cap \neg\rho)\end{aligned}$$

3.2 Abstract Semantics

The correctness of the bottom-up analysis, the so-called closure semantics, is argued relative a top-down analysis, the abstract semantics, which, in turn, is proved correct relative to the concrete semantics. The abstract semantics is parametric in terms of a finite set of predicates $\Sigma = \{\sigma_1, \ldots, \sigma_n\}$ where σ_1, $\ldots$, $\sigma_n \subseteq B$ with $\sigma_i \neq \sigma_j$ if $i \neq j$ are distinct predicates. A set of predicates $\delta \in \Delta \subseteq \wp(\Sigma)$ is interpreted by the following:

Definition 2. The concretisation map $\gamma : \wp(\wp(\Sigma)) \to \wp(B)$ is defined:

$$\gamma(\Delta) = \bigcup_{\delta \in \Delta} \gamma(\delta) \quad \text{where} \quad \gamma(\delta) = \bigcap_{\sigma \in \delta} \sigma$$

Example 1. Suppose $\delta_0 = \emptyset$, $\delta_1 = \{\sigma_1\}$ and $\delta_2 = \{\sigma_1, \sigma_2\}$. Then $\gamma(\delta_0) = B$, $\gamma(\delta_1) = \sigma_1$ and $\gamma(\delta_2) = \sigma_1 \cap \sigma_2$.

The concretisation map γ induces an quasi-ordering on $\wp(\wp(\Sigma))$ by $\Delta_1 \sqsubseteq \Delta_2$ iff $\gamma(\Delta_1) \subseteq \gamma(\Delta_2)$. To obtain a poset an operator $\downarrow$ is introduced to derive a canonical representation for an arbitrary $\Delta \subseteq \wp(\Sigma)$ by forming its down-set. The down-set is defined $\downarrow\Delta = \{\delta' \subseteq \Sigma \mid \exists \delta \in \Delta . \gamma(\delta') \subseteq \gamma(\delta)\}$ from which we construct $D = \{\downarrow\Delta \mid \Delta \subseteq \wp(\Sigma)\}$. Observe that if $\Delta_1, \Delta_2 \in D$ then $\Delta_1 \cap \Delta_2 \in D$. To see that $\Delta_1 \cup \Delta_2 \in D$ let $\delta \in \Delta_1 \cup \Delta_2$ and suppose $\delta \in \Delta_i$. Then if $\gamma(\delta') \subseteq \gamma(\delta)$ it follows that $\delta' \in \Delta_i \subseteq \Delta_1 \cup \Delta_2$. Moreover $\langle D, \subseteq, \cup, \cap, \emptyset, \wp(\Sigma)\rangle$ is a complete lattice where $\cap$ is meet and $\cup$ is join.

Proposition 1. The maps $\alpha : \wp(B) \to D$ and $\gamma : D \to \wp(B)$ form a Galois connection between $\langle \wp(B), \subseteq\rangle$ and $\langle D, \subseteq\rangle$ where $\alpha(\sigma) = \cap\{\Delta \in D \mid \sigma \subseteq \gamma(\Delta)\}$

Example 2. Suppose $\Sigma = \{\sigma_1, \sigma_2\}$ where $\sigma_1 = (0 \leq i \leq 1)$ and $\sigma_2 = (1 \leq i \leq 2)$. Let $\Delta_1 = \{\{\sigma_1\}, \{\sigma_1, \sigma_2\}\}$ and $\Delta_2 = \{\{\sigma_2\}, \{\sigma_1, \sigma_2\}\}$. Note that $\downarrow\Delta_1 = \Delta_1$ and $\downarrow\Delta_2 = \Delta_2$ thus $\Delta_1, \Delta_2 \in D$. However $\{\{\sigma_1\}\} \notin D$ and $\{\{\sigma_2\}\} \notin D$. Observe $\gamma(\Delta_1) = \sigma_1$ and $\gamma(\Delta_2) = \sigma_2$. Moreover $\Delta_1 \cap \Delta_2 \in D$ and $\Delta_1 \cup \Delta_2 \in D$ with $\gamma(\Delta_1 \cap \Delta_2) = \sigma_1 \cap \sigma_2 = (i = 1)$ and $\gamma(\Delta_1 \cup \Delta_2) = \sigma_1 \cup \sigma_2 = (0 \leq i \leq 2)$. Furthermore $\alpha(i = 1) = \{\{\sigma_1, \sigma_2\}\}$ and $\alpha(0 \leq i \leq 2) = \{\{\sigma_1\}, \{\sigma_2\}, \{\sigma_1, \sigma_2\}\}$.

Example 3. Observe that if $\Delta = \emptyset$ then $\Delta \in D$ and $\gamma(\Delta) = \emptyset$. But if $\delta = \emptyset$, $\delta \in \Delta$ and $\Delta \in D$ then $\Delta = \wp(\Sigma)$ since $\gamma(\delta') \subseteq B = \gamma(\delta)$ for all $\delta' \subseteq \Sigma$.

Proposition 2. If $\sigma \in \Sigma$ then $\alpha(\sigma) = \downarrow\{\{\sigma\}\}$.

Both for brevity and for continuity of the exposition, the proofs are relegated to a technical report [5].

As before, the abstract semantics is denotational associating each statement with a mapping $D \to D$. The function space $D \to D$ is ordered point-wise by $f_1 \sqsubseteq f_2$ iff $f_1(\Delta) \subseteq f_2(\Delta)$ for all $\Delta \in D$. Also like before $\langle D \to D, \sqcap, \sqcup, \lambda\Delta.\emptyset, \lambda\Delta.\wp(\Sigma)\rangle$ is a complete lattice where $f_1 \sqcap f_2 = \lambda\Delta.f_1(\Delta) \sqcap f_2(\Delta)$ and likewise $f_1 \sqcup f_2 = \lambda\Delta.f_1(\Delta) \sqcup f_2(\Delta)$. Moreover, the point-wise ordering on $D \to D$ lifts to define a point-wise ordering on $\mathcal{L} \to D \to D$ in an analogous manner. Since $\mathcal{L} \to D \to D$ is a complete lattice the following is well-defined:

Definition 3. The mapping $[\![\cdot]\!]_A : \mathcal{L} \to D \to D$ is the least solution to:

$$\begin{aligned}
[\![\mathsf{skip}]\!]_A &= \lambda\Delta.\Delta \\
[\![\mathsf{assume}(\rho)]\!]_A &= \lambda\Delta.\Delta \cap \alpha(\rho) \\
[\![\mathsf{transform}(\tau)]\!]_A &= \lambda\Delta.\alpha(\gamma(\Delta) \circ \tau) \\
[\![S_1; S_2]\!]_A &= \lambda\Delta.[\![S_2]\!]_A([\![S_1]\!]_A(\Delta)) \\
[\![\mathsf{if}\ \rho\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2]\!]_A &= \lambda\Delta.([\![S_1]\!]_A(\Delta \cap \alpha(\rho))) \cup ([\![S_2]\!]_A(\Delta \cap \alpha(\neg\rho))) \\
[\![\mathsf{while}\ \rho\ \mathsf{do}\ S]\!]_A &= \lambda\Delta.([\![\mathsf{while}\ \rho\ \mathsf{do}\ S]\!]_A([\![S]\!]_A(\Delta \cap \alpha(\rho))) \cup (\Delta \cap \alpha(\neg\rho))
\end{aligned}$$

Proposition 3. Let $S \in \mathcal{L}$. If $\rho \in \gamma(\Delta)$ then $[\![S]\!]_C(\rho) \subseteq \gamma([\![S]\!]_A(\Delta))$.

3.3 Closure Semantics

At the heart of the closure semantics are functions with signature $\Sigma \to D$. Join and meet lift point-wise to the function space $\Sigma \to D$ since if $f_1 : \Sigma \to D$ and $f_2 : \Sigma \to D$ then $f_1 \sqcup f_2 = \lambda\sigma.f_1(\sigma) \cup f_2(\sigma)$ and $f_1 \sqcap f_2 = \lambda\sigma.f_1(\sigma) \cap f_2(\sigma)$. The key idea is to construct a mapping $f : \Sigma \to D$ whose extension to $f : D \to D$ is a closure, that is, an operation which is monotonic, extensive and idempotent. A map $f : \Sigma \to D$ lifts to $f : \wp(\Sigma) \to D$ and then further lifts to $f : \wp(\wp(\Sigma)) \to D$ by $f(\delta) = \cap\{f(\sigma) \mid \sigma \in \delta\}$ and $f(\Delta) = \cup\{f(\delta) \mid \delta \in \Delta\}$ respectively. Observe that a lifting $f : D \to D$ is monotonic, irrespective of f, since if $\Delta_1 \subseteq \Delta_2$ then $f(\Delta_1) \subseteq f(\Delta_2)$. It also distributes over union, that is, $f(\Delta_1) \cup f(\Delta_2) = f(\Delta_1 \cup \Delta_2)$. We introduce $\uparrow f : \Sigma \to D$ to denote the idempotent relaxation of $f : \Sigma \to D$ which is defined thus:

Definition 4. If $f : \Sigma \to D$ then

$$\uparrow f = \sqcap\{f' : \Sigma \to D \mid f \sqsubseteq f' \wedge \forall\sigma \in \Sigma.f'(\sigma) = f'(f'(\sigma))\}$$

Note the use of loading within the expression $f'(f'(\sigma))$: the inner f' has type $f' : \Sigma \to D$ whereas the outer f' has type $f' : \Sigma \to D$. Observe too that $\uparrow f : \Sigma \to D$ is extensive if $f : \Sigma \to D$ is extensive. Although the above definition is not constructive, the idempotent relaxation can be computed in an iterative fashion using the following result:

Proposition 4. $\uparrow f = \sqcup_{i=0} f_i$ where $f_0 = f$ and $f_{i+1} = f_i \sqcup \lambda\sigma.f_i(f_i(\sigma))$

With $\uparrow f$ both defined and computable (by virtue of the finiteness of Σ), an analysis based on closures can be formulated thus:

Definition 5. The mapping $[\![\cdot]\!]_L : \mathcal{L} \to D \to D$ is the least solution to:

$$\begin{aligned}
[\![\mathsf{skip}]\!]_L &= \lambda\Delta.\Delta \\
[\![\mathsf{assume}(\rho)]\!]_L &= \lambda\Delta.\Delta \cap \alpha(\rho) \\
[\![\mathsf{transform}(\tau)]\!]_L &= \lambda\Delta.\alpha(\gamma(\Delta) \circ \tau) \\
[\![S_1; S_2]\!]_L &= \lambda\Delta.[\![S_2]\!]_L([\![S_1]\!]_L(\Delta)) \\
[\![\mathsf{if}\ \rho\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2]\!]_L &= \lambda\Delta.([\![S_1]\!]_L(\Delta \cap \alpha(\rho))) \cup ([\![S_2]\!]_L(\Delta \cap \alpha(\neg\rho))) \\
[\![\mathsf{while}\ \rho\ \mathsf{do}\ S]\!]_L &= \lambda\Delta.\uparrow f(\Delta) \cap \alpha(\neg\rho) \text{ where} \\
f &= \lambda\sigma.\downarrow\{\{\sigma\}\} \cup [\![S]\!]_L(\downarrow\{\{\sigma\}\} \cap \alpha(\rho))
\end{aligned}$$

Table 1. Experimental results

Program	$\lvert\Sigma\rvert$	Time	Input	Result
`counter.c`	12	0.1 s	$\mathtt{x} = 0 \wedge \mathtt{n} \geq 0$	$\mathtt{n} \geq 0 \wedge \mathtt{x} = \mathtt{n}$
`ex1a.c`	12	0.1 s	$0 \leq \mathtt{x} \leq 2 \wedge 0 \leq \mathtt{y} \leq 2$	$\mathtt{x} \geq 0 \wedge \mathtt{x} \leq 2$
`ex1b.c`	20	0.1 s	$\mathtt{m} = 0 \wedge \mathtt{x} = 0$	$\mathtt{m} \geq 0 \wedge \mathtt{n} > \mathtt{m} \wedge \mathtt{x} = \mathtt{n} \wedge \mathtt{x} > 0$
`ex3.c`	25	0.6 s	$\mathtt{x} \leq \mathtt{y} \wedge \mathtt{x} = 0 \wedge \mathtt{y} = \mathtt{m}$	$\mathtt{x} \leq \mathtt{m} \wedge \mathtt{x} = \mathtt{n} \wedge \mathtt{x} = \mathtt{y} \wedge \mathtt{y} \geq \mathtt{m}$
`lockstep.c`	12	0.1 s	$\mathtt{x} \leq \mathtt{y} \wedge \mathtt{x} \geq \mathtt{y}$	$\mathtt{x} = \mathtt{y} \wedge \mathtt{x} = \mathtt{n}$
`nested.c`	15	1.0 s	$\mathtt{t} = 0 \wedge \mathtt{y} = 0 \wedge \mathtt{m} \geq 0$	$\mathtt{t} > 0 \wedge \mathtt{t} = \mathtt{y} \wedge \mathtt{y} = \mathtt{m} \wedge \mathtt{y} > 0$
`two-loop.c`	20	0.2 s	$\mathtt{x} = 0 \wedge \mathtt{y} = 0$	$\mathtt{x} = \mathtt{n} \wedge \mathtt{y} = \mathtt{n}$

Note that $\uparrow f$ is a closure since f is extensive by construction. Observe too that $[\![\mathsf{while}\ \rho\ \mathsf{do}\ S]\!]_L$ is defined with a single call to $[\![S]\!]_L$ whereas $[\![\mathsf{while}\ \rho\ \mathsf{do}\ S]\!]_A$ is defined in terms of possibly many calls to $[\![S]\!]_A$. Thus the closure semantics can be realised without auxiliary structures such as memo tables that are needed to intercept repeated calls.

Conceptually the closure semantics simulates the top-down flow of the abstract semantics from which it is derived, until a loop is encountered at which point the loop body is entered. The loop body is then evaluated, again top-down, for each of the predicates. The closure is then calculated, applied to the formula that holds on entry to the loop, and the result composed with the negation of the loop condition, to infer the formula that holds on exit from the loop. Yet because of the structured nature of the domain, the loop transformer can be represented as a straight-line block of conditional assumptions. Thus the transformer has the dual attributes of: closely mimicking the top-down abstract semantics, which aids in constructing a convincing correctness argument, whilst being fully compositional which is the key attribute in the bottom-up approach to loop summarisation.

Proposition 5. Let $S \in \mathcal{L}$ and $\Delta \in D$. Then $[\![S]\!]_A(\Delta) \subseteq [\![S]\!]_L(\Delta)$.

By composing propositions 3 and 5 the main correctness result is obtained:

Corollary 1. Let $S \in \mathcal{L}$. If $\rho \in \gamma(\Delta)$ then $[\![S]\!]_C(\rho) \subseteq \gamma([\![S]\!]_L(\Delta))$.

4 Experiments

A prototype analyser had been implemented in Ruby [13], with the express aim of evaluating the precision of our technique on some loops used elsewhere for benchmarking. The analyser faithfully realises the closure semantics as set out in Def. 5. In addition to the examples outlined in Sect. 2, we applied our prototype to the programs evaluated in [18] which are available from [15]. These sample programs test and mutate integers with loop structures are either single loops, nested loops, or sequences of loops.

The results our experiments are presented in Tab. 1. The column $|\Sigma|$ denotes the number of predicates used, followed by *Time* which indicates the runtime required to evaluate the whole program. The column *Input* gives the formula that input to the program (actually an assumption that was given in the benchmark). Likewise for Σ we chose those predicates which are listed in a comment in the benchmark itself. The *Result* column documents the formula obtained by running the program on this input (in a cleaned format as is explained below). The runtime for all tests where less than a second on a 2.6 GHz MacBook Pro equipped with 4 GiB RAM.

Interestingly, our implementation seems to outperform the invariant generation technique presented in [18] for speed in all except one benchmark (`nested.c`). This result is rather surprising as our prototype has been implemented naïvely in Ruby, more as a sanity check on the design rather than a tool for assessing performance. Considering that Ruby is interpreted, the runtimes of our proof-of-concept implementation are encouraging. It should be noted, however, that we generate the transformers for blocks off-line, prior to applying the analysis, rather than using a SMT solver to compute block transformers on-the-fly. Nevertheless the dominating time is the closure calculation since it needs to repeatedly combine formulae; pruning intermediate formulae should improve this.

In terms of precision, most output formulae are actually disjunctive, but the table gives conjunctive simplifications to make the presentation accessible. In case of `counter.c`, for instance, we write $\mathtt{n} \geq 0 \wedge \mathtt{x} = \mathtt{n}$ instead of the disjunctive formula $(\mathtt{n} = 0 \wedge \mathtt{x} = \mathtt{n}) \vee (\mathtt{n} > 0 \wedge \mathtt{x} = \mathtt{n})$. Manually we checked that each of the component cubes (conjunctions) were genuinely reachable on program exit. (It may not be feasible to infer invariants by hand but if Σ is small it is possible to manually verify that a cube is irredundant with a high degree of confidence.) We conclude that these invariants appear to be optimal even though the closure semantics can, in principle, lead to a sub-optimal transformer for loops.

5 Related Work

The key idea in predicate abstraction [3,12,14] is to describe a large, possibly infinite, set of states with a finite set of predicates. If the two predicates ρ_i and ρ_j describe, respectively, the sets of states $\gamma(\rho_i)$ and $\gamma(\rho_j)$, then all the transitions between a state in $\gamma(\rho_i)$ and a state in $\gamma(\rho_j)$ are described with a single abstract transition from ρ_i to ρ_j. The existence of a transition between $\gamma(\rho_i)$ and $\gamma(\rho_j)$, and hence an abstract one between ρ_i and ρ_j, can be determined by querying a SAT/SMT solver [8] or a theorem prover [14]. The domain of conjuncts of predicates is related to the domain of sets of states by a Galois connection [4], allowing the framework of abstract interpretation [9], as well as domain refinements such as disjunctive completion [4], to be applied to systematically derive loop invariants using iterative fixpoint computation.

5.1 Loop Summarisation

Motived by the desire to improve efficiency, a thread of work has emerged on compositional bottom-up analysis that strives to reorganise iterative fixed-point computation by applying loop summarisation [34]. The idea is to substitute a loop with a conservative abstraction of its behaviour, constructing abstract transformers for nested loops starting from the inner-most loop [2,22]. Various approaches have been proposed for loop summarisation, such as taking cues from the control structure to suggest candidate invariants that are subsequently checked for soundness [22, Sect. 3.3]. Inference rules have also been proposed for deriving summaries based on control structures [33]. Increasingly loop summarisation is finding application in termination analysis [2,36].

5.2 Quantifier Elimination

Existential quantification has also been applied to characterise inductive loop invariants. Kapur [19] uses a parameterised first-order formula as a template and specifies constraints on these parameters using quantification. Quantifiers are then eliminated to derive the loop invariants [19, Sect. 3] which, though attractive conceptually, inevitably presents a computational bottleneck [11]. Likewise Monniaux (see [25, Sect. 3.4] and [26, Sect. 3.4]) uses quantification to specify inductive loop invariants for linear templates [32].

5.3 Disjunctive Invariants

Gulwani et al. [18] derive loop invariants in bounded DNF using SAT by specifying constraints that model state on entry and exit of a loop as well as inductive relations. Monniaux and Bodin [28] apply predicate abstraction to compute automata (with a number of states that is bounded a priori) which represent the semantics of reactive nodes using predicates and an abstract transition relation. Rather than computing abstractions as arbitrary formulae over predicates, they consider disjunctions of a fixed number of cubes. The specification of loop invariants itself is not dissimilar to that in [25, Sect. 3.4]. However, bounding the problem allows for the application of incremental techniques to improve performance [28, Sect. 2.4]. Similar in spirit, though based on classical abstract interpretation rather than SMT-based predicate abstraction, is the work of Balakrishnan et al. [1] on control-structure refinement for loops in Lustre.

Disjunctive loop invariants have also been studied in other contexts, for instance, Gulwani et al. [16,17] apply auxiliary variables in the complexity analysis of multi-path loops, where disjunctive invariants describe the complexities over counter variables. Recent work by Sharma et al. [35] focusses on the structure of loops in general. The authors observed that loops, which require disjunctive invariants, often depend on a single phase-transition. They provide a technique that soundly detects whether a loop relies on such, and if so, rewrite the program so that conjunctive techniques can be applied. Such invariants are easier to handle than disjunctive ones. By way of contrast, Popeea and Chin [29] compute

disjunctions of convex polyhedra using abstract interpretation. To determine whether a pair of two polyhedra shall be merged, they apply distance metrics so to balance expressiveness against computational cost.

6 Conclusions

This paper advocates a technique for leaping loops in predicate abstraction where the abstract domain is not merely a conjunction of predicates that simultaneously hold but rather a (possibly disjunctive) monotonic formula over the set of predicates. Each loop is summarised with a closure that enables each loop to be treated as if it were a straight-line block. Because the number of monotonic formulae grows rapidly with the number of predicates, the method, by design, does not compute a best transformer. Instead closures are derived solely for the atomic predicates and, as a result, each closure can be represented by just n monotonic formulae where n is the number of predicates. Applying the loop transformer then amounts to computing logical combinations of these n formulae. The compact nature of the loop transformers, their conceptual simplicity, as well as their accuracy which is demonstrated empirically, suggests that this notion of closure is a sweet-point in the design space for loop leaping on this domain. Future work will investigate adapting these loop leaping techniques to other abstract domains.

Acknowledgements This work was supported, in part, by the DFG research training group 1298 *Algorithmic Synthesis of Reactive and Discrete-Continuous Systems* and by the DFG *Cluster of Excellence on Ultra-high Speed Information and Communication*, German Research Foundation grant DFG EXC 89. This cooperation was funded, in part, by a Royal Society Industrial Fellowship and the Royal Society Joint Project grant JP101405.

References

1. G. Balakrishnan, S. Sankaranarayanan, F. Ivancic, and A. Gupta. Refining the Control Structure of Loops using Static Analysis. In *EMSOFT*, pages 49–58. ACM Press, 2009.
2. T. Ball, O. Kupferman, and M. Sagiv. Leaping Loops in the Presence of Abstraction. In *CAV*, volume 4590 of *LNCS*, pages 491–503. Springer, 2007.
3. T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic Predicate Abstraction of C Programs. In *PLDI*, pages 203–213, 2001.
4. T. Ball, A. Podelski, and S. K. Rajamani. Boolean and Cartesian Abstraction for Model Checking C Programs. In *TACAS*, volume 2031 of *LNCS*, pages 268–283. Springer, 2001.
5. S. Biallas, J. Brauer, A. King, and S. Kowalewski. Proof Appendix for Loop Leaping with Closures. Technical Report 12-3, University of Kent, Canterbury, CT1 7NF, UK, June 2012. `http://www.cs.kent.ac.uk/people/staff/amk/pubs.html`.
6. J. Brauer and A. King. Transfer Function Synthesis without Quantifer Elimination. *Logical Methods in Computer Science*, 2012.

7. J. Brauer, A. King, and J. Kriener. Existential Quantification as Incremental SAT. In *CAV*, volume 6806 of *LNCS*, pages 191–207. Springer, 2011.
8. E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav. SATABS: SAT-Based Predicate Abstraction for ANSI-C. In *TACAS*, volume 3440 of *LNCS*, pages 570–574. Springer, 2005.
9. 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.
10. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Effciently computing static single assignment form and the control dependence graph. *ACM Transaction on Programming Languages and Systems*, pages 451–590, 1991.
11. J. Davenport and J. Heintz. Real Quantifier Elimination is Doubly Exponential. *Journal of Symbolic Computation*, 5(1):29–35, 1988.
12. C. Flanagan and S. Qadeer. Predicate Abstraction for Software Verification. In *POPL*, pages 191–202, 2002.
13. D. Flanagan and Y. Matsumoto. *The Ruby Programming Language*. O'Reilly, 2008.
14. S. Graf and H. Saïdi. Construction of Abstract State Graphs with PVS. In *CAV*, volume 1254 of *LNCS*, pages 72–83. Springer, 1997.
15. S. Gulwani. Source Files and Invariants Generated, 2009. `http://research.microsoft.com/en-us/um/people/sumitg/benchmarks/pa.html`.
16. S. Gulwani. SPEED: Symbolic Complexity Bound Analysis. In *CAV*, volume 5643 of *LNCS*, pages 51–62. Springer, 2009.
17. S. Gulwani, K. K. Mehra, and T. M. Chilimbi. SPEED: precise and efficient static estimation of program computational complexity. In *POPL*, pages 127–139, 2009.
18. S. Gulwani, S. Srivastava, and R. Venkatesan. Constraint-based invariant inference over predicate abstraction. In *VMCAI*, volume 5403 of *LNCS*, pages 120–135. Springer, 2009.
19. D. Kapur. Automatically Generating Loop Invariants Using Quantifier Elimination. In *Deduction and Applications*, volume 05431. IBFI, 2005.
20. N. Kettle, A. King, and T. Strzemecki. Widening ROBBDs with Prime Implicants. In *TACAS*, volume 3920 of *LNCS*, pages 105–119. Springer, 2006.
21. A. King and H. Søndergaard. Automatic Abstraction for Congruences. In *VMCAI*, volume 5944 of *LNCS*, pages 197–213. Springer, 2010.
22. D. Kroening, N. Sharygina, S. Tonetta, A. Tsitovich, and C. M. Wintersteiger. Loop Summarization Using Abstract Transformers. In *ATVA*, volume 5311 of *LNCS*, pages 111–125. Springer, 2008.
23. D. Kroening, N. Sharygina, S. Tonetta, A. Tsitovich, and C. M. Wintersteiger. Loopfrog: A Static Analyzer for ANSI-C Programs. In *ASE*, pages 668–670. IEEE Computer Society, 2009.
24. D. Kroening and O. Strichman. *Decision Procedures*. Springer, 2008.
25. D. Monniaux. Automatic Modular Abstractions for Linear Constraints. In *POPL*, pages 140–151. ACM Press, 2009.
26. D. Monniaux. Automatic Modular Abstractions for Template Numerical Constraints. *Logical Methods in Computer Science*, 6(3), 2010.
27. D. Monniaux. Quantifier Elimination by Lazy Model Enumeration. In *CAV*, volume 6174 of *LNCS*, pages 585–599. Springer, 2010.
28. D. Monniaux and M. Bodin. Modular Abstractions of Reactive Nodes Using Disjunctive Invariants. In *APLAS*, volume 6996 of *LNCS*, pages 19–33, 2011.
29. C. Popeea and W.-N. Chin. Inferring Disjunctive Postconditions. In *ASIAN*, volume 4435 of *LNCS*, pages 331–345. Springer, 2006.

30. T. Reps, M. Sagiv, and G. Yorsh. Symbolic Implementation of the Best Transformer. In *VMCAI*, volume 2937 of *LNCS*, pages 252–266. Springer, 2004.
31. S. Rudeanu. *Boolean Functions and Equations*. North-Holland, 1974.
32. S. Sankaranarayanan, H. Sipma, and Z. Manna. Constraint based linear relations analysis. In *SAS*, volume 3148 of *LNCS*, pages 53–68. Springer, 2004.
33. M. N. Seghir. A Lightweight Approach for Loop Summarization. In *ATVA*, volume 6996 of *LNCS*, pages 351–365, 2011.
34. M. Sharir and A. Pnueli. Two Approaches to Interprocedural Data Flow Analysis. In *Program Flow Analysis: Theory and Applications*, pages 189–234. Prentice Hall, 1981.
35. R. Sharma, I. Dillig, T. Dillig, and A. Aiken. Simplifying Loop Invariant Generation Using Splitter Predicates. In *CAV*, volume 6806 of *LNCS*, pages 703–719. Springer, 2011.
36. A. Tsitovich, N. Sharygina, C. M. Wintersteiger, and D. Kroening. Loop Summarization and Termination Analysis. In *TACAS*, volume 6605 of *LNCS*, pages 81–95. Springer, 2011.
37. D. Wiedemann. A computation of the eighth Dedekind number. *Order*, 1(8):5–6, 1991.