NON BISIMILAR TRANSFORMS

Contents

1 Normalized Loops

- A set of iteration variables, $\{i_1 \dots i_n\}$ has a normalized iteration space $\mathcal{I}$ if variables start from 0, are incremented by 1 and the upper-bounds $U_1 \dots U_n$ can be expressed as functions of previous iteration variables.

$$\mathcal{I} = \{i_1 \dots i_n \mid \forall t.\ 0 \leq i_t \leq U_t(i_1, \dots, i_{t-1})\}$$

  This was the previous formulation, which caused some problems and so was abandoned. We generalize this to work with arbitrary nonzero lower bounds and arbitrary increments later. However the rest of the points about normalization remain the same.

- We denote the set of constraints defining $\mathcal{I}$ by $C_\mathcal{I}$ which is a boolean function with the domain $(i_1 \dots i_n)$. The t^{th} constraint, $0 \leq i_t \leq U_t(i_1, \dots, i_{t-1})$, is denoted by $C_\mathcal{I}[t]$, which is a boolean function with the domain $(i_1 \dots i_t)$. Hence we get:

$$C_\mathcal{I}(i_1 \dots i_n) = \bigwedge_{t=1}^{n} C_\mathcal{I}[t](i_1 \dots i_{t-1})$$

- The ordering $\prec_\mathcal{I}$ is the lexicographic ordering on the variables $i_1 \dots i_n$. For some $\vec{i} \in \mathcal{I}$, $\text{next}(\vec{i})$ represents the next value in the iteration space according to $\prec_\mathcal{I}$.

- For loop body B we have a set of variables $\mathcal{V}$ which are assigned to inside the loop body and are alive at the end of the loop body, excluding the normalized iteration variables.

- The set of variables $\mathcal{P} \subseteq \mathcal{V}$ are the ones which can be expressed as functions of the normalized iteration variables via invariant generation, $\mathcal{P} = \{p_1 \dots p_k\} = H(\mathcal{I})$

- The set of variables which are live in the loop body and cannot be expressed as a function of $\mathcal{I}$ by using invariant generation, $\mathcal{X} = \{x_1 \dots x_m\} = \mathcal{V} \setminus \mathcal{P}$

- The transfer function of a non-normalized loop B has the type:

$$B : (\mathcal{X}, \mathcal{P}) \to \mathcal{X}, \mathcal{P}$$

  If $\vec{x}^*, \vec{p}^* = B(\vec{x}, \vec{p})$ then the following condition must be satisfied:

$$\vec{p} = H(\vec{i}) \implies \vec{p}^* = H(\text{next}(\vec{i}))$$

- For a normalized loop we modify the body to create another function, B^*, with the type:

$$B^* : (\mathcal{X}, \mathcal{I}) \to \mathcal{X}$$

  B^* can be constructed as:

  $B^*(\vec{x}, \vec{i}) \equiv$
  `let` $\vec{x}^*, \vec{p}^* = B(\vec{x}, H(\vec{i}))$
  `return` $\vec{x}^*$

- A normalized loop, where $\vec{x} \in \mathcal{X}$, can be written as:

  `For` $\vec{i} \in \mathcal{I}$ `by` $\prec_{\mathcal{I}}$ `do` $\vec{x} = B^*(\vec{x}, \vec{i})$

2 Notion of Equivalence

Instead of checking the "equivalence" of these two programs we will try to generate invariants after the loop bodies in both programs. These invariants would be a weakening of the invariants that exist before the two loops in the product transfer graph of these two programs.
Let the source and target have the following syntax:

Source:

`For` $\vec{i} \in \mathcal{I}$ `by` $\prec_{\mathcal{I}}$ `do` $\vec{x} = B_S(\vec{x}, \vec{i})$

Target:

`For` $\vec{j} \in \mathcal{J}$ `by` $\prec_{\mathcal{J}}$ `do` $\vec{y} = B_T(\vec{y}, \vec{j})$

Let $Inv(\vec{x}, \vec{y})$ be the invariants that hold before the two loop nests. The mathematical formulation is:

$$
\begin{aligned}
&\text{For some } M : (\mathcal{X} \times \mathcal{Y}) \to \{0,1\} \text{ such that} \\
&\quad Inv(\vec{x}, \vec{y}) \implies M(\vec{x}, \vec{y}) \wedge \\
&\quad \exists\, F : \mathcal{J} \mapsto \mathcal{I} \\
&\quad (\forall\, \vec{x} \in \mathcal{X},\ \vec{y} \in \mathcal{Y},\ \vec{j} \in \mathcal{J} : M(\vec{x}, \vec{y}) \implies M(B_S(\vec{x}, F(\vec{j})), B_T(\vec{y}, \vec{j}))) \wedge \\
&\quad (\forall\, \vec{j_1}, \vec{j_2} \in \mathcal{J} : F(\vec{j_1}) \prec_{\mathcal{I}} F(\vec{j_2}) \wedge \vec{j_2} \prec_{\mathcal{J}} \vec{j_1} \implies \\
&\qquad\qquad \forall\, \vec{x} \in \mathcal{X} : B_S(B_S(\vec{x}, F(\vec{j_1})), F(\vec{j_2})) = B_S(B_S(\vec{x}, F(\vec{j_2})), F(\vec{j_1})))
\end{aligned}
\tag{1}
$$

Then $M(\vec{x}, \vec{y})$ is also true after the two loops.

Here $\mapsto$ symbols is used for denoting bijection, so F must be a bijection between $\mathcal{I}$ and $\mathcal{J}$.

This is a generalization of the formula presented in PERMUTE. If we take M as the identity function and $B_T(\vec{x}, \vec{j}) =_{syn} B_S(\vec{x}, F(\vec{j}))$ (where $=_{syn}$ means syntactically equal) then the definitions become equivalent as the check for $M(\vec{x}, \vec{y}) \implies M(B_S(\vec{x}, F(\vec{i})), B_T(\vec{y}, \vec{i}))$ becomes irrelevant.

The $\forall\ \vec{x} \in \mathcal{X},\ \vec{y} \in \mathcal{Y},\ \vec{j} \in \mathcal{J} : M(\vec{x}, \vec{y}) \implies M(B_S(\vec{x}, F(\vec{j})), B_T(\vec{y}, \vec{j}))$ part essentially means that the following Hoare logic triple holds:

$$\{M(\vec{x}, \vec{y}) \wedge \vec{i} = F(\vec{j})\}[\vec{x} = B_S(\vec{x}, \vec{i}); \vec{y} = B_T(\vec{y}, \vec{j})]\{M(\vec{x}, \vec{y})\}$$

While the part

$$
\begin{aligned}
&\forall\, \vec{j_1}, \vec{j_2} \in \mathcal{J} : F(\vec{j_1}) \prec_{\mathcal{I}} F(\vec{j_2}) \ \wedge\ \vec{j_2} \prec_{\mathcal{J}} \vec{j_1} \implies \\
&\qquad \forall\, \vec{x} \in \mathcal{X} : B_S(B_S(\vec{x}, F(\vec{j_1})), F(\vec{j_2})) = B_S(B_S(\vec{x}, F(\vec{j_2})), F(\vec{j_1}))
\end{aligned}
$$

is basically the same condition as the one in the permute paper, which says that if the iteration vectors $\vec{i_1}$ and $\vec{i_2}$ get their order flipped due to F then the composition of the body

at those iteration vectors should be the same.

3 Need for Normalization

Making loop normalized has two basic conditions: the iteration space, $\mathcal{I}$, should be normalized and that we should remove the variables $\mathcal{P}$. these two steps are important in different ways, and we will talk about them individually.

3.1 Normalized Iteration Space : Outdated

In the original paper, the authors assumed that the iteration variables are already known along with the iteration space of the loops, $\mathcal{I}$ and $\prec_{\mathcal{I}}$. They assume that this information is annotated by the compiler or somehow easily obtainable by heuristics. We cannot make that assumption because of two reasons:

- In assembly code or other code with gotos it is hard to figure out which variable is acting as an iteration variable, so we would have to use complicated heuristics here.
- We have to exactly quantify what the iteration space is. This would mean using heuristics for getting the lower and upper bounds, getting the ordering $\prec_{\mathcal{I}}$ and using even more heuristics for cases where the iteration variables aren't simply incremented by 1. In the case of normalized iteration variables we can generate the upper bounds U_x by invariant generation and the iteration space gets known without any heuristics.

In the new formulation, we use invariant generation to figure out $\mathcal{I}$ and $\prec_{\mathcal{I}}$ using arbitrary iteration variables.

3.2 Removing $\mathcal{P}$

By removing variables which can be expressed as functions of the normalized iteration variables in each iteration, we increase the possibilities of equivalences that can be proven. We give an example where this is the case, consider the following loop reversal transform:

Source:

```
x1 := 0
x2 := 0
for i = 0; i < N; i ++:
    x2 += x1
    x1 ++
```

Target:

```
y1 := N
y2 := 0
for j = 0; j < N; j ++:
    y2 += y1
    y1 --
```

Here $B_S(x_1, x_2, i) = x_1 + 1, x_1 + x_2$ and $B_T(y_1, y_2, j) = y_1 - 1, y_1 + y_2$. As you can see the value of F doesn't matter here, which is why the proof will not go through.
However if we change the loops to:
Now x_1 and y_1 are no longer alive across the loop, they are not considered in $\mathcal{X}$ and $\mathcal{Y}$. Now the bodies are: $B_S(x_2, i) = x_2 + i$ and $B_T(y_2, j) = y_2 + N - j$. For us to prove equivalence we can take M as $M(x_2, y_2) \equiv y_2 = x_2$. The value of F can be taken as $i = F(j) = N - j$. Under these assumptions we can verify that:

Source:

```
x2 := 0
for i = 0; i < N; i ++:
    x1 = i
    x2 += x1
```

Target:

```
y2 := 0
for j = 0; j < N; j ++:
    y1 = N - j
    y2 += y1
```

$$M(x_2, y_2) \implies M(B_s(x_2, F(j)), B_t(y_2, j))$$
$$\therefore x_2 = y_2 \implies B_s(x_2, N - j) = B_t(y_2, j)$$
$$\therefore x_2 = y_2 \implies x_2 + N - j = y_2 + N - j$$

And also:

$$\forall i_1, i_2 : i_1 < i_2 \wedge N - i_2 < N - i_1 \implies$$
$$\forall x : B_S(B_S(x, i_1), i_2) = B_S(B_S(x, i_2), i_1)$$
$$\equiv$$
$$\forall i_1, i_2 : i_1 < i_2 \implies$$
$$\forall x : x + i_1 + i_2 = x + i_2 + i_1$$

4 Generalization for Loop Splitting : Outdated

For loop splitting generalization we can weaken the definition of F from being a bijection to a one-one function. We assume that one loop nest in the source maps to multiple loop nests in the target, so the syntax is:

Source:

For $\vec{i} \in \mathcal{I}$ by $\prec_{\mathcal{I}}$ do $\vec{x} = B_S(\vec{x}, \vec{i})$

Target:

For $\vec{j} \in \mathcal{J}_1$ by $\prec_{\mathcal{J}_1}$ do $\vec{y} = B_T^1(\vec{y}, \vec{j})$
For $\vec{j} \in \mathcal{J}_2$ by $\prec_{\mathcal{J}_3}$ do $\vec{y} = B_T^2(\vec{y}, \vec{j})$
$\vdots$
For $\vec{j} \in \mathcal{J}_k$ by $\prec_{\mathcal{J}_k}$ do $\vec{y} = B_T^k(\vec{y}, \vec{j})$

Now the formula becomes:

$$
\begin{aligned}
&\text{For some } M : (\mathcal{X} \times \mathcal{Y}) \to \{0,1\} \text{ such that} \\
&\quad Inv(\vec{x},\vec{y}) \implies M(\vec{x},\vec{y}) \wedge \\
&\quad \exists\ F_1 \ldots F_k : \text{ (where } F_t \text{ has type } \mathcal{J}_t \to \mathcal{I}) \\
&\qquad (\forall\ t \in [k] : \\
&\qquad\quad (\forall\ \vec{x} \in \mathcal{X},\ \vec{y} \in \mathcal{Y},\ \vec{j} \in \mathcal{J}_t : M(\vec{x},\vec{y}) \implies M(B_S(\vec{x}, F_t(\vec{j})) = B_T^t(\vec{y},\vec{j}))) \wedge \\
&\qquad\quad (\forall\ \vec{j_1},\vec{j_2} \in \mathcal{J}_t : \\
&\qquad\qquad (F_t(\vec{j_1}) = F_t(\vec{j_2}) \implies \vec{j_1} = \vec{j_2}) \wedge \\
&\qquad\qquad (\vec{j_1} \prec_{\mathcal{J}_t} \vec{j_2} \wedge F_t(\vec{j_2}) \prec_{\mathcal{I}} F_t(\vec{j_1}) \implies \\
&\qquad\qquad\qquad \forall\ \vec{x} \in \mathcal{X} : B_S(B_S(\vec{x}, F_t(\vec{j_1})), F_t(\vec{j_2})) = B_S(B_S(\vec{x}, F_t(\vec{j_2})), F_t(\vec{j_1})))) \wedge \\
&\qquad\quad (\forall\ s \in [t+1 : k],\ \vec{j_1} \in \mathcal{J}_t,\ \vec{j_2} \in \mathcal{J}_s : \\
&\qquad\qquad (F_t(\vec{j_1}) \neq F_s(\vec{j_2})) \wedge \\
&\qquad\qquad (F_s(\vec{j_2}) \prec_{\mathcal{I}} F_t(\vec{j_1}) \implies \\
&\qquad\qquad\qquad \forall\ \vec{x} \in \mathcal{X} : B_S(B_S(\vec{x}, F_t(\vec{j_1})), F_s(\vec{j_2})) = B_S(B_S(\vec{x}, F_s(\vec{j_2})), F_t(\vec{j_1})))) \wedge \\
&\qquad \forall\ \vec{i} \in \mathcal{I} : \exists\ t \in [k],\ \vec{j} \in \mathcal{J}_t : \vec{i} = F_t(\vec{j})
\end{aligned}
\tag{2}
$$

The condition $F_t(\vec{j_1}) = F_t(\vec{j_2}) \implies \vec{j_1} = \vec{j_2}$ is the condition for F_t being injective. The condition $F_t(\vec{j_1}) \neq F_s(\vec{j_2})$ is added because we don't want two different loops in the target map to the same iteration in the source. Finally the condition $\forall\ \vec{i} \in \mathcal{I} : \exists\ t \in [k],\ \vec{j} \in \mathcal{J}_t : \vec{i} = F_t(\vec{j})$ just checks if every iteration in the source is mapped to at least one iteration in the target, so combined all of $F_1 \ldots F_k$ are onto. The rest of the conditions are the same as the equation defined above.

The generalization to loop splitting in the new formulation is now done implicitly by the algorithm as we treat each loop as a "space" and we can correlate multiple spaces with each other.

5 Unsound Reduction of Search Space

Searching for functions F and M which directly satisfy the above definition is very hard as we do not have a good way to iterate over the search space of all functions. Hence making some reasonable assumptions about these functions is essential.

5.1 Affine F and Code Specialization

Almost all loop transforms performed by a compiler, like skewing, tiling, reversal, interchange etc, work with an F which can be expressed as multiplication with a constant matrix. Another assumption we need to make here is that code optimizations specialize more often than they de-specialize. This means that loop-unrolling and loop-tiling are more common than loop-rerolling and loop-detiling. We will assume that de-specialization doesn't happen, which means that the function F can be broken down into n affine functions $F^1 \ldots F^n$ such that $\forall t.\ i_t = F^t(\vec{j})$ holds. Note that this is not true in the reverse direction if we consider

loop tiling transform, where you have to do $j_1, j_2 = i/c, i\%c$, which isn't affine.

5.2 Backtracking Approach to finding F

Being able to divide F into $F^1, \ldots, F^n$ allows for a backtracking solution to getting the correct F. There are some checks we can run when we have constructed a partial solution for k functions $F^1, \ldots, F^k$, $k \leq n$, to confirm that they are part of an bijection between $\mathcal{I}$ and $\mathcal{J}$. For example we can check:

$$\forall\ \vec{j} \in \mathcal{J} : C_{\mathcal{J}}(\vec{j}) \implies \bigwedge_{t=1}^{k} C_{\mathcal{I}}[t](F^1(\vec{j}) \ldots F^t(\vec{j}))$$

This gives us the ability to prune the search space for F.

5.3 Heuristics for Candidate F values

One way we can get values of F is by relating the equations generated in the computation of $\mathcal{P} = H(\mathcal{I})$. Suppose in the source we have $\mathcal{P} = \{p_1, \ldots, p_k\} = H_S(\mathcal{I})$ and in the target we have $\mathcal{Q} = \{q_1, \ldots, q_l\} = H_T(\mathcal{J})$. Considering $p_t = q_s$ as a candidate relation between $\mathcal{I}$ and $\mathcal{J}$ for some t and s makes sense. Equating the temporaries in the bodies which have affine relations with the loop iteration variables is also a good idea.

5.4 Affine or Permutation M

Similar to F, we can assume M to only contain affine equality constraints. If we want to restrict it further then we can consider M to be a permutation instead of just affine. There are rarely any cases where a compiler will make a loop transform that makes M not be a permutation, an example could be the target code computing $2x$ instead of x and then dividing it by 2 after the loop ends. Hence at the very least it makes sense for us to give priority to simple permutation-like functions for M when searching for a solution.

5.5 Partial M

Computing the solution for a value of M which relates all of the variables at the same time might be difficult as we would have to guess the entire permutation. Even if one equation is wrong the solution would fail and we wouldn't be able to find any value of F. So it makes sense to try and check for the equality of one variable at a time, and we can also do this in parallel. In some cases choosing a weaker M, counterintuitively, enables us to prove more as different M values can have different F bijections in their proof. For example in the following code:

Source:

```
x1 := 0
for i = 0; i < N; i ++:
    x1 += i
```

Target:

```
y1 := 0
y2 := 0
for j = 0; j < N; j ++:
    y1 += N - j
    y2 += j
```

One candidate M can be $M_1(x_1, y_1, y_2) \equiv x_1 = y_1$ and another candidate M can be $M_2(x_1, y_1, y_2) \equiv x_1 = y_2$. If we take M to be $M_3(x_1, y_1, y_2) \equiv M_1 \wedge M_2 \equiv x_1 = y_1 \wedge x_1 = y_2$ then the proof wouldn't go through for any F, however for both M_1 and M_2 there exists a value of F which completes the proof.

However there are also cases where a stronger M is required to prove a claim than just individual affine relations. Consider the following transform:

Source:

```
x1, x2, x3 := 0
for i = 0; i < N; i ++:
    x1 += i
    x2 += N - i + 1
    x3 += x1 + x2
```

Target:

```
y1, y2, y3 := 0
for j = 0; j < N; j ++:
    y1 += N - j
    y2 += j + 1
    y3 += y1 + y2
```

Here if we take $M(x_1, x_2, x_3, y_1, y_2, y_3) \equiv x_3 = y_3$ then the proof wouldn't work, however $M(x_1, x_2, x_3, y_1, y_2, y_3) \equiv x_1 = y_1 \wedge x_2 = y_2 \wedge x_3 = y_3$ works for $i = F(j) = N - j$.

6 Hoare Logic Optimizations : Outdated

Instead of choosing an M and choosing an F individually, we can get some information about the nature of F by looking at the body and M. Suppose we choose a value for M. Using this we can compute the weakest precondition in Hoare logic as:

$$\{\mathcal{WP}(\vec{x}, \vec{y}, \vec{i}, \vec{j})\}[\vec{x} = B_S(\vec{x}, \vec{i}); \vec{y} = B_T(\vec{y}, \vec{j})]\{M(\vec{x}, \vec{y})\}$$

And we also know that any value of F must satisfy:

$$M(\vec{x}, \vec{y}) \wedge \vec{i} = F(\vec{j}) \implies \mathcal{WP}(\vec{x}, \vec{y}, \vec{i}, \vec{j})$$

We can rewrite this as:

$$\vec{i} = F(\vec{j}) \implies \forall\ x \in \mathcal{X},\ y \in \mathcal{Y} : (M(\vec{x}, \vec{y}) \implies \mathcal{WP}(\vec{x}, \vec{y}, \vec{i}, \vec{j}))$$

So we use the shorthand:

$$G(\vec{i}, \vec{j}) \equiv \forall\ x \in \mathcal{X},\ y \in \mathcal{Y} : (M(\vec{x}, \vec{y}) \implies \mathcal{WP}(\vec{x}, \vec{y}, \vec{i}, \vec{j}))$$

If we can use the weakest precondition to find equations that are implied by G and then we know that they have to be correct in any chosen value of F. We expect most reasonable loop optimizations to have the property that most affine relations between the iteration variables can be figured out by finding G.

6.1 Example : Outdated

Lets take loop skewing as an example:
For simplicity, we aren't considering the loop iteration space to be normalized as it makes the example easier to read. For $M(x, y) \equiv \forall k_1, k_2 : x[k_1, k_2] = y[k_1, k_2]$ the F function we

Source:

for $i_1 = 0\,;\ \ i_1 \leq N\,;\ \ i_1{+}{+}$:
  for $i_2 = 0\,;\ \ i_2 \leq m\,;\ \ i_2{+}{+}$:
    $\mathrm{x}\,[\,i_1, i_2\,]\ :=\ \mathrm{x}\,[\,i_1 - 1, i_2 - 1\,]$

Target:

for $j_1 = -M\,;\ \ j_1 \leq N\,;\ \ j_1{+}{+}$:
  for $j_2 = \max(0, j_1)\,;$
    $j_2 \leq \min(N, j_1 + M)\,;\ \ j_2{+}{+}$:
    $\mathrm{y}\,[\,j_2, j_2 - j_1\,]\ :=\ \mathrm{y}\,[$
      $j_2 - 1, j_2 - j_1 - 1\,]$

want is: $i_1, i_2 = F(j_1, j_2) = j_2, j_2 - j_1$. Now the command in the Hoare triple is:

$$x[i_1, i_2] := x[i_1 - 1, i_2 - 1]; y[j_2, j_2 - j_1] := y[j_2 - 1, j_2 - j_1 - 1]$$

The postcondition is simply $\forall k_1, k_2 : x[k_1, k_2] = y[k_1, k_2]$. The weakest precondition comes out to be:

If $i_1, i_2 = j_2, j_2 - j_1$ then:
  $\forall\ k_1, k_2 : (k_1, k_2 \neq i_1, i_2) \implies x[k_1, k_2] = y[k_1, k_2]$
Else:
  $y[i_1, i_2] = x[i_1 - 1, i_2 - 1] \wedge$
  $x[j_2, j_2 - j_1] = y[j_2 - 1, j_2 - j_1 - 1] \wedge$
  $\forall\ k_1, k_2 : (k_1, k_2 \neq i_1, i_2) \wedge (k_1, k_2 \neq j_2, j_2 - j_1) \implies x[k_1, k_2] = y[k_1, k_2]$

If we compute $\forall x, y : \forall k_1, k_2 : x[k_1, k_2] = y[k_1, k_2] \implies \mathcal{WP}$ then the `Else` condition is rejected because it puts additional constraints on x and y. Hence:

$$\forall\ x \in \mathcal{X},\ y \in \mathcal{Y} : (M(\vec{x}, \vec{y}) \implies \mathcal{WP}(\vec{x}, \vec{y}, \vec{i}, \vec{j})) \implies (i_1 = j_2 \wedge i_2 = j_2 - j_1)$$

This means that for this case G is equal to F.

6.2 Relations in G are easier to find : Outdated

It is much easier to get affine equality relations implied by G as compared to directly constructing F. This is because we can guess single affine relations on just one variable and test if it is implied by $G(\vec{i}, \vec{j})$. After testing several such equalities, we can AND all of them together and get a significant portion of F. In comparison if we were working on getting F directly then only after we have constructed the whole F can we test if it implies $G(\vec{i}, \vec{j})$, which would increase the search time.

6.3 CEGAR implementation : Outdated

One way to utilize the Hoare logic weakest-precondition equation is to compute candidate affine equality relations using counter-example guided abstraction refinement. We use the following notation to describe the algorithm:

- $upper'$ and $lower'$ denote a set of affine relations on $\vec{i}, \vec{j}$ such that $lower' \implies G(\vec{i}, \vec{j}) \implies upper'$.
- `AbstractConseqence` is a function which takes $lower'$ and $upper'$ and returns p' such that $lower' \implies p'$ but $p' \not\Longrightarrow upper'$.

- $A \sqcup B$ returns the affine equality relation which is implied by both A and B. So for example:
$$[i = 10, j = 12] \sqcup [i = 3, j = 5] = [i = j - 2]$$
Similarly $A \sqcap B$ returns the affine equality relation which implies by both A and B.

- `Model` is the SMT solver which can return `TimeOut`, `UNSAT` or a model `M` which satisfies the constraints given to it.

The algorithm is as follows:

$upper' \leftarrow \top$
$lower' \leftarrow \bot$
while $lower' \neq upper' \wedge$ ResourcesLeft do:
    $p' \leftarrow$ AbstractConseqence($lower', upper'$)
    M $\leftarrow$ Model($\exists\ \vec{i}, \vec{j} : G(\vec{i}, \vec{j}) \wedge \neg p'$)
    if M **is** TimeOut **then**
        break
    else if M **is** UNSAT **then**
        $upper' \leftarrow upper' \sqcap p'$
    else
        $lower' \leftarrow lower' \sqcup$ M
return $upper'$

The naive approach of finding G would involve randomly guessing affine relations and checking if they are implied by $G(\vec{i}, \vec{j})$, however this generates a lot of SMT queries. Instead we can use the SMT solver to get concrete examples, or models, $\vec{i_1}, \vec{j_1}$ such that $G(\vec{i_1}, \vec{j_1})$ is true. Then we can guess an affine relation which is satisfied by this example.

CEGAR is an extension of this idea. If we have two models $\vec{i_1}, \vec{j_1}$ and $\vec{i_2}, \vec{j_2}$ which work with G then we can find some affine relation which is satisfied in both, which is called p' in the algorithm. Then we can ask the SMT solver to generate another model $\vec{i_3}, \vec{j_3}$ in which p' isn't true. If we are unable to find it then p' is implied by G, otherwise we can add the model to our set of models, and find another p' which is satisfied by all models.

Lets take the example used above, where $G(i_1, i_2, j_1, j_2) \equiv i_1 = j_2 \wedge i_2 = j_2 - j_1$. An example run of the algorithm could look like:

$upper' := \top$
$lower' := \bot$
$p' :=$ AbstractConsequence($\bot, \top$)
$= \bot$
$M :=$Model($\exists\ \vec{i}, \vec{j} : G(\vec{i}, \vec{j}) \wedge \neg\bot$)
$= [i_1 = 2, i_2 = 1, j_2 = 2, j_1 = 1]$
$lower' := \bot \sqcup [i_1 = 2, i_2 = 1, j_2 = 2, j_1 = 1]$
$= [i_1 = 2, i_2 = 1, j_2 = 2, j_1 = 1]$
$p' :=$ AbstractConsequence($[i_1 = 2, i_2 = 1, j_2 = 2, j_1 = 1], \top$)
$= (i_1 = 2 \wedge i_2 = 1)$
$M :=$Model($\exists\ \vec{i}, \vec{j} : G(\vec{i}, \vec{j}) \wedge \neg(i_1 = 2 \wedge i_2 = 1)$)

$= [i_1 = 2, i_2 = 0, j_2 = 2, j_1 = 2]$

$lower' := [i_1 = 2, i_2 = 1, j_2 = 2, j_1 = 1] \sqcup [i_1 = 2, i_2 = 0, j_2 = 2, j_1 = 2]$

$= [i_1 = 2, i_2 = j_2 - j_1, j_2 = 2]$

$p' :=$ AbstractConsequence$([i_1 = 2, i_2 = j_2 - j_1, j_2 = 2], \top)$

$= (i_1 = 2)$

$M :=$Model$(\exists\ \vec{i}, \vec{j} : G(\vec{i}, \vec{j}) \wedge \neg(i_1 = 2))$

$= [i_1 = 3, i_2 = 0, j_2 = 3, j_1 = 3]$

$lower' := [i_1 = 2, i_2 = j_2 - j_1, j_2 = 2] \sqcup [i_1 = 3, i_2 = 0, j_2 = 3, j_1 = 3]$

$= [i_1 = j_2, i_2 = j_2 - j_1]$

$p' :=$ AbstractConsequence$([i_1 = j_2, i_2 = j_2 - j_1], \top)$

$= (i_1 = j_2)$

$M :=$Model$(\exists\ \vec{i}, \vec{j} : G(\vec{i}, \vec{j}) \wedge \neg(i_1 = j_2))$

$=$ UNSAT

$upper' = \top \sqcap (i_1 = j_2)$

$= [i_1 = j_2]$

$p' :=$ AbstractConsequence$([i_1 = j_2, i_2 = j_2 - j_1], [i_1 = j_2])$

$= (i_2 = j_2 - j_1)$

$M :=$Model$(\exists\ \vec{i}, \vec{j} : G(\vec{i}, \vec{j}) \wedge \neg(i_2 = j_2 - j_1))$

$=$ UNSAT

$upper' = [i_1 = j_2] \sqcap (i_2 = j_2 - j_1)$

$= [i_1 = j_2, i_2 = j_2 - j_1]$

$upper' = lower' :$ Break

Hence by using CEGAR we can mostly avoid searching for F or G by a searchspace.

6.4 Removing Quantified Queries

In the CEGAR algorithm above we use quantified queries to the solver which have a universal forall quantifier over x and y. However if we have a value of $\vec{i}, \vec{j}$ which works for a random choice of $\vec{x}, \vec{y}$ which satisfies $M(\vec{x}, \vec{y})$ then it is quite likely that it will work with all values of $\vec{x}$ and $\vec{y}$. This is because most of the times there are only corner-case values which give wrong $\vec{i}, \vec{j}$ values. Lets take the following program as an example:

Source:

```
x = 1
for i = 0; i < N; i ++
    x *= i
```

Target:

```
y = 1
for j = 0; j < N; j ++
    y *= N - j
```

Here the weakest-precondition for $M(x, y) \equiv x = y$ is $\mathcal{WP}(x, y, i, j) = x \times (N - j) = y \times i$. Now consider the SMT query

$$M(x, y) \wedge \mathcal{WP}(x, y, i, j)$$

$$\equiv x = y \wedge x \times (N - j) = y \times i$$

If we try this then the only example where $i \neq N - j$ that we can get is when $x = y = 0$, which is a rare corner-case if x and y are sampled uniformly at random. Now we can use

fuzzing to replace the quantified SMT query. We replace the SMT query

$$\forall x \in \mathcal{X}, y \in \mathcal{Y} : (M(\vec{x}, \vec{y}) \implies \mathcal{WP}(\vec{x}, \vec{y}, \vec{i}, \vec{j}))$$

with the following procedure:

$\vec{x_1}, \vec{y_1} \leftarrow$ Model($\exists\ \vec{x}, \vec{y} : M(\vec{x}, \vec{y})$)
$\vec{i_1}, \vec{j_1} \leftarrow$ Model($\exists\ \vec{i}, \vec{j} : \mathcal{WP}(\vec{x_1}, \vec{y_1}, \vec{i}, \vec{j})$)
check $\leftarrow$ Model($\exists\ \vec{x}, \vec{y} : M(\vec{x}, \vec{y}) \wedge \neg\mathcal{WP}(\vec{x}, \vec{y}, \vec{i_1}, \vec{j_1})$)
if check **is** UNSAT:
    return $\vec{i_1}, \vec{j_1}$

For most examples this should work fine considering that the SMT model gives randomized enough results. This is however quite often not true, and SMT solvers tend to return quite a lot of 0s. To solve this we can randomize the value of $\vec{x}$ and calculate $\vec{y}$ from that, which would very likely give us a correct value of $\vec{i_1}$ and $\vec{j_1}$.

Another way to solve this is to modify the procedure to work with multiple $\vec{x}, \vec{y}$ models. We can get these models in a counter-example guided way. So the new procedure is:

$S = \{\}$ (empty **set** with **type** $\mathcal{X} \times \mathcal{Y}$)
$M \leftarrow$ Model($\exists\ \vec{x}, \vec{y} : M(\vec{x}, \vec{y})$)
$S \leftarrow S \cup M$
while True:
    $\vec{i_1}, \vec{j_1} \leftarrow$ Model($\exists\ \vec{i}, \vec{j} : \bigwedge_{\vec{x_t}, \vec{y_t} \in S} \mathcal{WP}(\vec{x_t}, \vec{y_t}, \vec{i}, \vec{j})$)
    $M \leftarrow$ Model($\exists\ \vec{x}, \vec{y} : M(\vec{x}, \vec{y}) \wedge \neg\mathcal{WP}(\vec{x}, \vec{y}, \vec{i_1}, \vec{j_1})$)
    if M **is** UNSAT:
        return $\vec{i_1}, \vec{j_1}$
    else:
        $S \leftarrow S \cup M$

7 Loop Unrolling

Loop unrolling is a loop transform that is already handled in Counter, however it involves trying to unroll every loop individually by a λ factor. Instead of unrolling each loop, we can try making copies of the bodies by a λ factor and then figuring out which loops are unrolled using that. We can replace the command in the Hoare logic triple from $\vec{x} = B_s(\vec{x}, \vec{i}); \vec{y} = B_T(\vec{y}, \vec{j})$ to:

$$\vec{x} = B_s(\vec{x}, \vec{i_1}); \vec{x} = B_s(\vec{x}, \vec{i_2}); \ldots; \vec{x} = B_s(\vec{x}, \vec{i_\lambda}); \vec{y} = B_T(\vec{y}, \vec{j})$$

Here $\vec{i_1} \ldots \vec{i_\lambda} \in \mathcal{I}$ are all different variables, and we can denote them as $\vec{i_t} = \{i_{t1}, i_{t2}, \ldots, i_{tn}\}$. The postcondition remains the same, that is $M(\vec{x}, \vec{y})$. Now using the weakest-precondition method, if we get:

$$\forall\ \vec{x} \in \mathcal{X},\ \vec{y} \in \mathcal{Y} : (M(\vec{x}, \vec{y}) \implies \mathcal{WP}(\vec{x}, \vec{y}, \vec{i_1}, \ldots, \vec{i_\lambda}, \vec{j}))$$
$$\implies i_{1k} = i_{2k} = \cdots = i_{\lambda k}$$

Then we know that the loop hasn't been unrolled at the k^{th} loop index. If we add further conditions of $i_{1k} \leq i_{2k} \leq i_{3k} \leq \cdots \leq i_{\lambda k}$ for some k we might be able to get a constraint like $i_{1k} = i_{2k} - 1 = i_{3k} - 2 = \cdots = i_{\lambda k} - \lambda + 1$ which would tell us which specific loop index was unrolled.

8 New Formula

This formula is a weakening of the previous formula, so it increases the number of transforms that we can prove. We discussed an informal proof of this, but haven't written it down yet.

$$
\begin{aligned}
&\text{For some } M : (\mathcal{X} \times \mathcal{Y}) \to \{0,1\} \text{ such that} \\
&\qquad Inv(\vec{x},\vec{y}) \implies M(\vec{x},\vec{y}) \wedge \\
&\quad \exists\, F : \mathcal{J} \mapsto \mathcal{I} \\
&\quad (\forall\, \vec{x} \in \mathcal{X},\ \vec{y} \in \mathcal{Y},\ \vec{j} \in \mathcal{J} : M(\vec{x},\vec{y}) \implies M(B_S(\vec{x}, F(\vec{j})), B_T(\vec{y},\vec{j}))) \wedge \\
&\quad (\forall\, \vec{j_1}, \vec{j_2} \in \mathcal{J},\ \vec{x} \in \mathcal{X},\ \vec{y} \in \mathcal{Y} : F(\vec{j_1}) \prec_{\mathcal{I}} F(\vec{j_2}) \wedge \vec{j_2} \prec_{\mathcal{J}} \vec{j_1} \implies \\
&\qquad\qquad (M(\vec{x},\vec{y}) \implies M(B_S(B_S(\vec{x}, F(\vec{j_1})), F(\vec{j_2})), B_T(B_T(\vec{y},\vec{j_2}),\vec{j_1}))))
\end{aligned}
\tag{3}
$$

A good example of where this works and the previous definition doesn't is a program where in the source we have some redundant calculation of a value which gets eliminated in the target because it isn't alive. The value calculated might not satisfy the exchange condition according to F, so the proof wouldn't go through.

Source:

```
x = 0, z = 0
for i = 0; i < N; i ++
    x += i
    z %= i
```

Target:

```
y = 0
for j = 0; j < N; j ++
    y += N - j
```

Here $M(x,z,y) \equiv x = y$ wouldn't have worked before, but works now.

9 Piecewise Affine F

There are examples of commonly occurring code that cannot be proven with an affine F but are still correct. An example can be something like:

Source:

```
for i = 0; i < N; i ++
    if i == 0:
        x = 0
    else:
        x += i
```

Target:

```
for j = 0; j < N; j ++
    if j == 0:
        y = 0
    else:
        y += N - i
```

For these cases we can consider a piecewise affine F. $F(0) = 0 \wedge F(i > 0) = N - i$ works in this case.

10 Problems with normalized Iteration variables

There are certain cases where normalized iteration variables cause the iteration space to be not affine, and also certain cases where finding the bounds on the affine iteration variables is considerably harder than arbitrary iteration variable. So it would be beneficial to extend our approach to arbitrary iteration variables which might not be normalized.

10.1 Normalized iterations not affine

Consider the following loop nest:

for $i_1 = -6\,;\ \ i_1 \leq 6\,;\ \ i_1$++:
  for $i_2 = \max(0, i_1)\,;\ \ i_2 \leq 10\,;\ \ i_2$++:
    for $i_3 = 0\,;\ \ i_3 \leq \min(i_2 - i_1, 10)\,;\ \ i_3$++:
      ....

Normalizing the outermost loop gets us:

for $i'_1 = 0\,;\ \ i'_1 \leq 12\,;\ \ i'_1$++:
  for $i_2 = \max(0, i'_1 - 6)\,;\ \ i_2 \leq 10\,;\ \ i_2$++:
    for $i_3 = 0\,;\ \ i_3 \leq \min(i_2 - i'_1 + 6, 10)\,;\ \ i_3$++:
      ....

Normalizing the second loop gets us:

for $i'_1 = 0\,;\ \ i'_1 \leq 12\,;\ \ i'_1$++:
  for $i'_2 = 0\,;\ \ i'_2 \leq \min(10, 16 - i'_1)\,;\ \ i_2$++:
    for $i_3 = 0\,;\ \ i_3 \leq \min(i'_2 + \max(0, -i'_1 + 6), 10)\,;\ \ i_3$++:
      ....

This loopnest isn't affine! So we have a problem now that we need to somehow compute the upper bound which looks like $\min(i'_2 + \max(0, -i'_1 + 6), 10)$. This isn't really an easy thing to do.

10.2 Lower bounds and upper bounds after normalization

A general 2 depth nested loop, assuming that the variables are incremented by 1, would look something like:

for $i_1 = C_1\,;\ \ i_1 \leq C_2\,;\ \ i_1$++:
  for $i_2 = \max(L_1(i_1), L_2(i_1) \ldots L_n(i_1))\,;\ \ i_2 \leq \min(U_1(i_1), U_2(i_1) \ldots U_m(i_1))\,;\ \ i_2$++:
    ....

When we normalize this we get:

for $i_1 = C_1\,;\ \ i_1 \leq C_2\,;\ \ i_1$++:
  for $i_2 = 0\,;\ \ i_2 \leq \min(U_1(i_1), U_2(i_1) \ldots U_m(i_1)) - \max(L_1(i_1), L_2(i_1) \ldots L_n(i_1))\,;\ \ i_2$++:
    ....

$\min(U_1(i_1), U_2(i_1) \ldots U_m(i_1)) - \max(L_1(i_1), L_2(i_1) \ldots L_n(i_1))$ can be written as $\min(U_1(i_1), U_2(i_1) \ldots U_m(i_1)) + \min(-L_1(i_1), -L_2(i_1) \cdots - L_n(i_1))$, which can then be written as $\min_{1 \leq p \leq n, 1 \leq q \leq m} (U_p(i_1) -$

$L_q(i_1)$). This means that a 2 depth iteration space is always affine even after normalization.

Now suppose we are given the upper bounds of the normalized second loop as $H_1(i_1), H_2(i_1) \ldots H_n(i_1)$. To get to the un-normalized iteration loop we can try to "break up" these upper bounds into two sets of functions $L_q(i_1)$ and $U_p(i_1)$ such that each $H_k(i_1)$ can be written as $U_p(i_1) - L_q(i_1)$. For example $\min(i_1, N, M, N + M - i_1)$ from the look skewing example can be written as $\min(N, i_1) - \max(0, i_1 - M)$.

One way to do this which would take advantage of our generated max invariants would be to try checking whether they can be subtracted from the H functions to get some common U functions. If a max term is found in the upper-bound of the third nested loop then we can test that as our $\max(L_1(i_1) \ldots L_q(i_1)$ lower bound as well.

10.3 Matrices which imply mod conditions

Consider the loop transform implied by the matrix $\left[\begin{smallmatrix} 2 & 1 \\ 0 & 1 \end{smallmatrix}\right]$. Suppose the original source loop looks like:

for $i_1 = 0\,;\ i_1 \leq N\,;\ i_1\ {+}{+}$:
  for $i_2 = 0\,;\ i_2 \leq M\,;\ i_2\ {+}{+}$:
    ....

The target loop after the transformation would look like:

for $j_1 = 0\,;\ j_1 \leq 2N + M\,;\ j_1\ {+}{+}$:
  for $j_2 = \max(j_1 \% 2, j_1 - 2N)\,;\ j_2 \leq \min(j1, M)\,;\ j_2\ {+}{=}\ 2$:
    ....

Normalizing this would get us:

for $j_1 = 0\,;\ j_1 \leq 2N + M\,;\ j_1\ {+}{+}$:
  for $j_2' = 0\,;\ j_2' \leq \min(N, \frac{M+2N-j_1}{2}, \frac{j_1}{2}, \frac{M-j_1\%2}{2})\,;\ j_2'\ {+}{+}$:
    ....

The upper bound in this case has a %2 term which is hard to determine and also non-affine. Also the divided by 2 expressions involve integer division, so we would probably need to generate invariants like $2j_2' \leq M + 2N - j_1$.

11 General Non-normalized iteration variables

Restricting ourselves to only work with the ghost iteration variables that we added is more trouble than it's worth. We can define our iteration space with any possible variable that is availible in the program for which we can completely determine the invariants. A general loop iteration variable can be defined as a variable which is incremented by a constant (or a loop invariant function) in each iteration, and for which we can get the lower and upper bounds.

11.1 Calculating iteration space

11.1.1 Oracles

There are a few oracles that would help us get the iteration variables. We will also talk about the potential implementations of these oracles and if they can be done in a way which doesn't rely on any syntactic assumptions. In the

- **Increment oracle**: Given some loop-and-function-free code this oracle can find all variables which are incremented by a constant amount in the code, along with the increment value.
- **Max oracle**: Given some loop-and-function-free code this oracle can find all invariants of the form $\frac{x}{c} = \max(\frac{f_1(\vec{v})}{c}, \frac{f_2(\vec{v})}{c}, \ldots, \frac{f_n(\vec{v})}{c})$ which holds after the code is executed, where v is the set of live variables, $f_1 \ldots f_n$ are arbitrary functions (assumed linear) and c is a specified constant. This can also be written as $\max(f_1(\vec{v}), f_2(\vec{v}), \ldots, f_n(\vec{v})) \leq x < \max(f_1(\vec{v}), f_2(\vec{v}), \ldots, f_n(\vec{v})) + c$.
- **Modulo oracle**: Given some loop-and-function-free code this oracle can find all invariants of the form $(x \mod c) = (f(\vec{v}) \mod c)$ which holds after the code is executed, where v is the set of live variables, f is an arbitrary function (assumed linear) and c is a specified constant.
- **Trip-count oracle**: Given a loop body with exit conditions this oracle generates all invariants of the form $x \leq f(\vec{v})$ which hold inside the loop, where v is the set of live variables, f is an arbitrary function (assumed linear).

11.1.2 Iteration space from oracles

Using the oracles defined above we can get an iteration space of the form $\max(L_1, \ldots L_n) \leq i \leq \min(U_1, \ldots U_m) \wedge i\%c = H$ where $L_1 \ldots L_n$ are lower bound functions, $U_1 \ldots U_m$ are upper bound functions and H is the modulo function. The procedure to do this is as follows:

1. From the loop body find all variables which are incremented by a constant amount using the increment oracle. Let one such variable be i with increment c.
2. Using the Max oracle find the max invariant of $\frac{i}{c} = \max(\frac{L_1(\vec{v})}{c}, \frac{L_2(\vec{v})}{c}, \ldots, \frac{L_n(\vec{v})}{c})$ just before the start of the loop.
3. Using the Modulo oracle find the modulo invariant just before the start of the loop, $i \mod c = h(\vec{v}) \mod c$
4. Using the trip-count oracle find all invariants of the form $i \leq U_j(\vec{v})$ which hold inside the loop.
5. At the end of the loop assert $\frac{i}{c} = \min(\frac{U_1(\vec{v})}{c}, \frac{U_2(\vec{v})}{c}, \ldots, \frac{U_m(\vec{v})}{c})$ to confirm that you have found all of the upper bounds using the trip-count oracle.

11.2 Implementation of oracles

Apart from the trip-count oracles, the other three can be implemented in a way which doesn't rely on the syntax of the code. Loop trip-count is a well studied problem which is considered pretty hard, however I will also try to provide a basic implementation of it based on some reasonable syntax assumptions specific to our problem.

11.2.1 Increment Oracle

Let m is the number of (relevant) loop invariant variables and n is the current number of counter examples generated. The matrix $A \in \mathcal{Z}^{n\times(m+1)}$ represents the values of these variables in the CEs stored in a matrix form, with the last column being 1. The vector $b \in \mathcal{Z}^n$ represents the values in the CEs of the loop iteration variable at the begining of the loop subtracted from the value of the loop iteration variable at the end of the loop. If we can find a vector $x \in \mathcal{Z}^{m+1}$ such that $Ax = b$ then we have found a proposed increment constant as a linear function of the loop invariant variables, represented by x. We can assert if this invariant always holds and if so then we are done. Otherwise we can append the generated counter example to A and continue our search. If at some point we cannot find an x satisfying $Ax = b$ then we can stop and say that the variable isn't a loop iteration variable.

11.2.2 Modulo Oracle

The modulo oracle is implemented in pretty much the same way, except instead of the matrices and vectors being from the field of integers, $\mathcal{Z}$, they would be from the field of integers modulo c, $\mathcal{Z}_c$. The relevant variables would be the previous loop iteration variables and the loop invariant variables.

11.2.3 Maximum Feasible Subset

The maximum feasiable subset problem (or MaxFS) is defined as:
Given a matrix A of size $n \times m$, where $n > m$, and a vector $b \in \mathcal{Z}^n$, find a vector $x \in \mathcal{Z}^m$ such that $(Ax)_i = (b)_i$ is satisfied at the most number of indices i. This can also be thought of as minimizing the l_0 norm of $Ax - b$, or as finding a m-dimensional hyperplane which passes through the most number of points in a n-dimensional space. Other variations of this problem replace the equality relation with $\geq$ or $>$, and there can also be multiple vectors b for which different inequalities/equalities hold. All of these problems are NP hard.
We are interested in the specific problem $b \leq Ax < b + \vec{c}$, where $\vec{c}$ is a constant vector with all entries as c. We also don't need an exact solution, a good enough solution will do. Furthermore we have a promised case where we know that there exists an x such that at least $\frac{n}{k}$ indices are satisfied for some small constant k. One way to solve this would be to encode this as an ILP problem and send that to an optimizer.

11.2.4 Max Oracle

We can construct the matrix A which would contain the values in the CEs for the relevant variables (previous loop iteration variables and loop invariant variables), and the vector b can be constructed as the values of i rounded down to the nearest multiple of c, so $c \times \lfloor \frac{i}{c} \rfloor$.

Let the lower bound be $\max(L_1(\vec{v}) \dots L_k(\vec{v}))$. We know that every CE would satisfy at least one lower bound from L_1 to L_k. This means that we have a promise that for any MaxFS call atleast $\frac{n}{k}$ indices can be satisfied.
Using the oracle for the MaxFS problem defined above we can get a candidate x^* and then we can check whether the invariant $\frac{i}{c} = \frac{\vec{v} \times x^*}{c}$ holds. If not then we would get some CEs which don't work with x^* and then hopefully the caididate x^* wouldn't be generated again. If the invariant holds then we can remove all of the CEs which satisfy $\frac{i}{c} = \frac{\vec{v} \times x^*}{c}$ from A and add the condition $\frac{i}{c} \neq \frac{\vec{v} \times x^*}{c}$ to all invariants to be generated in the future. Once we stop getting any CEs due to the SMT queries becoming SAT we would know that we have generated all the lower bounds.

11.2.5 Trip-count Oracle

Let the exit condition by $E(i, \vec{v}) \to \{0, 1\}$, so we exit the loop if it returns 1. If we assume that E flips atmost k number of times for some small constant k then we can use a similar method as above to generate the upper bounds. More formally, for all fixed $\vec{v} = v^*$ the set $\{i : i \mod c = H(v^*) \mod c \wedge E(i, v^*) = 1 \wedge E(i - c, v^*) = 0\}$ should have the size $\leq k$. If this condition is satisfied then we can generate CEs which satisfy $i \mod c = H(v^*) \mod c \wedge E(i, v^*) = 1 \wedge E(i - c, v^*) = 0$, which would give us CEs which satisfy $\frac{i}{c} \leq \min(\frac{U_1(\vec{v})}{c} \dots \frac{U_k(\vec{v})}{c})$ and then we can use the same method as the max oracle.

11.3 Asserting invariants

Let us define $\texttt{UB} = \min(U_1(\vec{v}) \dots U_m(\vec{v}))$ and $\texttt{LB} = \max(L_1(\vec{v}) \dots L_n(\vec{v}))$.
If all of the following invariants hold then we can know that the iteration variable defined is correct:

Invariant	Location
$i \leq \texttt{UB}$	In loop body
$i \geq \texttt{LB}$	In loop body
$(i \mod \texttt{c}) = (\texttt{H} \mod \texttt{c})$	In loop body, At loop head
$i = i_{\text{prev}} + \texttt{c}$	In loop body, At loop head
$(\texttt{c} > 0) \implies \texttt{LB} \leq i_{init} < \texttt{LB} + \texttt{c}$	In loop body, At loop head
$(\texttt{c} < 0) \implies \texttt{UB} + \texttt{c} < i_{init} \leq \texttt{UB}$	In loop body, At loop head
$(\texttt{c} > 0) \implies i \geq \texttt{LB}$	At loop head
$(\texttt{c} < 0) \implies i \leq \texttt{UB}$	At loop head
$(\texttt{c} > 0) \implies \texttt{LB} \leq \texttt{UB} \implies i \leq \texttt{UB} + \texttt{c}$	At loop head, After loop
$(\texttt{c} < 0) \implies \texttt{LB} \leq \texttt{UB} \implies i \geq \texttt{LB} + \texttt{c}$	At loop head, After loop
$(\texttt{c} > 0) \implies i > \texttt{UB}$	After loop
$(\texttt{c} < 0) \implies i < \texttt{LB}$	After loop

These invariants also form a set where each can be proven based on the structure of the program and the rest of the invariants defined. This has been tested in the code.

12 Piecewise Algorithm

12.1 Non-affine transform on affine iteration spaces

It is possible that the iteration spaces we have found are not related in an affine manner, so there doesn't exist a matrix F which you can multiply with one iteration space to get the other, but it is still true that the underlying transform is still affine. As an example consider the following transform:

Source	Target

for $i_1 = 0\,;\ i_1 \leq N\,;\ i_1{+}{+}$:
  for $i_2 = 0\,;\ i_2 \leq M\,;\ i_2{+}{+}$:
  $x[i_1, i_2] := x[i_1 - 1, i_2 - 1]$
	for $j_1 = 0\,;\ j_1 \leq N + M\,;\ j_1{+}{+}$:
	  for $j_2 = \max(0, j_1 - M)\,;\ j_2 \leq \min(N, j_1)\,;\ j_2{+}{+}$:
	  $y[j_2, j_2 - j_1 + M] := y[j_2 - 1, j_2 - j_1 + M - 1]$

The transform here is affine and if we get i_1, i_2, j_1 and j_2 as our iteration variables we can find the matrix which transforms one to the other, however we can also write the dst code as:

for $j_1 = 0\,;\ j_1 \leq N + M\,;\ j_1{+}{+}$:
  for $j_2 = 0\,;\ j_2 \leq \min(j_1, N, M, N + M - j_1)\,;\ j_2{+}{+}$:
    $k := j_2 + \max(0, j_1 - M)$
    $y[k, k - j_1 + M] := y[k - 1, k - j_1 + M - 1]$

This iteration space isn't affinely related to the source, so there doesn't exist a matrix we can find.

12.2 Non-affine transforms

Many reasonable transforms can also be non-affine, but instead be piecewise affine. For example we can have:

Source	Target

for $i_1 = 0\,;\ i_1 < N\,;\ i_1{+}{+}$:
  if $i_1 == 0$:
  $x = 0$
  else:
  $x{+}{=} i$
	for $j_1 = 0\,;\ j_1 < N\,;\ j_1{+}{+}$:
	  if $j_1 == 0$:
	  $x = 0$
	  else:
	  $x{+}{=} N - j_1$

In this code the first iteration in the source maps to the first iteration in the dst, while other iterations are reversed in order.

12.3 Canonical Form of Iteration spaces

We can write an iteration space in its canonical form by representing each lower/upper bound as a vector which represents a hyperplane. The iteration space is the n-dimensional space of the iteration variables. For now let us work with `inc` $= 1$ to make things simpler. Then the iteration space is defined by:

$$\bigwedge_{x=1}^{n}\left(\left(\bigwedge_{y=1}^{p_x} i_x \geq L_y^x(\vec{i_{x-1}})\right) \wedge \left(\bigwedge_{y=1}^{q_x} i_x \leq U_y^x(\vec{i_{x-1}})\right)\right)$$

Assuming all L and U are affine we can write each one of them as a multiplication with a matrix, so we get:

$$\bigwedge_{x=1}^{n}\left(\left(\bigwedge_{y=1}^{p_x} \langle \vec{lb_y^x}, \vec{i_x}\rangle \geq l_y^x\right) \wedge \left(\bigwedge_{y=1}^{q_x} \langle \vec{ub_y^x}, \vec{i_x}\rangle \leq u_y^x\right)\right)$$

This means all of these affine relations can be written as a vector dot-product with $\vec{i}$ (the vector of all iteration variables, $\vec{i_n}$, with 1 appended at the end). Now the iteration space can just be written as:

$$\bigwedge_{\vec{a}\in A} \langle \vec{a}, \vec{i}\rangle \leq 0$$

Where A is a set of vectors which define the iteration space.

12.3.1 Example

Consider this loop skewing example code:

for $i_1 = 0\,;\ \ i_1 \leq N + M\,;\ \ i_1{+}{+}$:
  for $i_2 = \max(0, i_1 - M)\,;\ \ i_2 \leq \min(N, i_1)\,;\ \ i_2{+}{+}$:
    $y[i_2, i_2 - i_1 + M] := y[i_2 - 1, i_2 - i_1 + M - 1]$

The relavant inequalities are:

$0 \leq i_1 \leq N + M$
$\max(0, i_1 - M) \leq i_2 \leq \min(N, i_1)$

After canonicalisation these become:

$-i_1 \leq 0$
$i_1 - N - M \leq 0$
$-i_2 \leq 0$
$i_1 - i_2 - M \leq 0$
$i_2 - N \leq 0$
$i_2 - i_1 \leq 0$

And for the normalized version:

for $i_1 = 0\,;\ \ i_1 \leq N + M\,;\ \ i_1{+}{+}$:
  for $i_2 = 0\,;\ \ i_2 \leq \min(i_1, N, M, N + M - i_1)\,;\ \ i_2{+}{+}$:
    $k := i_2 + \max(0, i_1 - M)$
    $y[k, k - i_1 + M] := y[k - 1, k - i_1 + M - 1]$

The relavant inequalities are:

$0 \leq i_1 \leq N + M$
$0 \leq i_2 \leq \min(i_1, N, M, N + M - i_1)$

After canonicalisation these become:

$-i_1 \leq 0$
$i_1 - N - M \leq 0$
$-i_2 \leq 0$
$i_2 - N \leq 0$
$i_2 - M \leq -$
$i_2 - i_1 \leq 0$
$i_2 + i_1 - N - M \leq 0$

12.3.2 Removing redundancies

Some of the inequalities given above are actually redundant. For example $0 \leq j_2 \leq N$ and $-M \leq j_2 - j_1 \leq 0$ together imply $0 \leq j_1 \leq N + M$. Similarly $0 \leq j_1 \leq N + M$ is implied by $0 \leq j_2 \leq \min(N, M)$, $j_2 - j_1 \leq 0$ and $j_2 + j_1 \leq N + M$. We can remove these redundant inequalities to get a smaller representation.

12.4 Hyperplanes, Corners and Edges

- A vector $\vec{a} \in A$ can also be interpreted as a hyperplane, which takes the equation $\langle \vec{a}, \vec{i} \rangle = 0$.
- A corner is defined taking the conjunction of a set of n independent hyperplanes. We call this set A_C for corner C.
- A corner can be infeasible if it is outside the iteration space. This is checked by $(\bigwedge_{\vec{a} \in A} \langle \vec{a}, \vec{i} \rangle \leq 0) \wedge (\bigwedge_{\vec{a} \in A_C} \langle \vec{a}, \vec{i} \rangle = 0)$ being UNSAT.
- An edge is a set of $n - 1$ independent hyper planes. We call this set A_E for edge E.
- An edge is adjacent to a corner if $A_E \subset A_C$.
- Two corners are adjacent if they share an edge, so they differ only by one hyperplane.

12.4.1 Corners and Conditions

There are some cases where a corner falls outside the iteration space. Consider the space defined by $0 \leq j_2 \leq 6$, $j_2 - j_1 \leq 0$ and $j_2 + j_1 \leq 16$. If we take the corner between $j_2 - j_1 = 0$ and $j_2 + j_1 = 16$ then we get $j_2 = 8$ and $j_1 = 8$. However this lies outside the space as it doesn' satisfy $j_2 \leq 6$, so it isn't an actual corner.
When dealing with iteration spaces which are dependent on loop invariant variables then the condition for a corner lying inside the iteration space implies conditions on the loop invariant variables. Consider the space $0 \leq j_2 \leq \min(N, M)$, $j_2 - j_1 \leq 0$ and $j_2 + j_1 \leq N + M$. If we take the corner defined by $j_2 - j_1 \leq 0$ and $j_2 + j_1 \leq N + M$ then we get the condition as $N = M$.
For every set of conditions we can enumerate all of the corners which exist under those conditions.

12.5 The Algorithm

12.5.1 High Level Idea

The first step of the algorithm is to map a corner or the source to a corner of the target. Let the source corner be C_S and the target corner be C_T. We repeatedly follow the following steps

1. Take an adjacent corner of C_T or C_S and try and map it to a point on one of the adjacent edges of the other corner.
2. When you have found a mapping on all adjacent edges of C_T and C_S, take the convex hull of these points.
3. The mapping should imply an affine transform in the terms of a matrix. Check if that transform works for all points in the convex hull.
4. Remove the convex hull from the source and iteration spaces. The new C_T and C_S are then set to one of the previously mapped points.
5. Repeat till the either one of the spaces is empty.

12.5.2 Oracle

We assume that we have an oracle which takes the following values:

- Constraints on the source and target iteration vectors, $\vec{i}$ and $\vec{j}$, $H(\vec{i}, \vec{j})$
- An invariant $M(\vec{x}, \vec{y})$ which relates variables in the source and target $\vec{x}$ and $\vec{y}$.
- The source and target bodies, $\vec{x} := B_S(\vec{i}, \vec{x})$ and $\vec{y} := B_T(\vec{j}, \vec{y})$, which take the iteration vectors as input and variables as input.

The oracle query $\mathcal{O}(B_S, B_T, M, H)$ is given as follows:

$$\neg(\exists \vec{i}\vec{j}.\ \forall \vec{x}\vec{y}.\ H(\vec{i}, \vec{j}) \implies M(\vec{x}, \vec{y}) \implies M(B_S(\vec{i}, \vec{x}), B_T(\vec{j}, \vec{y})))$$

If the oracle is UNSAT and gives us the counter-example $\vec{i'}$ and $\vec{j'}$ then that means we have found two iterations in the source and target which match with each other.
This query is quantified however we can turn it into a non-quantified query by using CEGAR, which on a high level is just done by sampling $\vec{x}, \vec{y}$ which satisfy $M(\vec{x}, \vec{y})$ multiple times.

12.5.3 Notations

We define the following notations:

- $\texttt{Adj}(C, A)$ is the set of adjacent corners to C in the space A which are feasible.
- If $C' \in \texttt{Adj}(C, A)$ and E is the edge between them then $A_E = A_{C'} \cap A_C$. We denote this edge by $E = Edge(C', C)$
- For any corner C or edge E we use the shorthand $C(\vec{i}) \equiv A_C(\vec{i})$ and $E(\vec{i}) \equiv A_E(\vec{i})$, where $A_C, A_E \subset A$. These can be interpreted as $\vec{i}$ being the corner C and $\vec{i}$ lying on the the edge E respectively.

- $\texttt{HP}(P_1, \ldots, P_n)$ is the hyperplane which passes through the $n-1$ points $\{P_1, \ldots, P_n\}$.
- If a is a hyperplane then $-a$ is the same hyperplane but it denotes the different side in the iteration space.
- $\texttt{Transform}([P_1^S \ldots P_{n+1}^S], [P_1^T \ldots P_{n+1}^T])$ denotes the matrix which maps the points P_i^S with P_i^T.
- This can just be calculated by $M_T M_S^{-1}$, where M_S is the matrix which has $P_1^S \ldots P_n^S$ as columns and M_T is the matrix with $P_1^T \ldots P_N^T$ as columns, with the last row of both having all 1s.

12.5.4 Pseudocode

The function `findMatchingAdjacent` takes the inputs as two corners C_1^S and C_1^T such that they map to each other, i.e. $\mathcal{O}(B_S, B_T, M, \lambda\vec{ij}.C_1^S(\vec{i}) \wedge C_1^T(\vec{j}))$ returns `SAT`.
The output is a set of source and target points $P_1^S \ldots P_{n+1}^S$ and $P_1^T \ldots P_{n+1}^T$ which satisfy the following conditions:

- All source and target points map with each other in order, i.e.

$$\forall k.\mathcal{O}(B_S, B_T, M, \lambda\vec{ij}.P_k^S(\vec{i}) \wedge P_k^T(\vec{j})) = \texttt{SAT}$$

- At least one of the source and target point pairs is an adjacent corner to C_1^S or C_1^T respectively. i.e.

$$\forall k.P_k^S \in \texttt{Adj}(C_1^S) \vee P_k^T \in \texttt{Adj}(C_1^T)$$

- All points lie on edges adjacent to C_1^S or C_1^T respectively. i.e.

$$\forall k.\exists A_E \subset A_{C_1}.|A_E| = n-1 \wedge \forall_{\vec{a} \in A_E} \langle \vec{a}, \vec{P_k} \rangle = 0$$

```
def findMatchingAdjacent(C_1^S, C_1^T, A^S, A^T):
    Points^S := [C_1^S]; Points^T := [C_1^T]
    for C_2^S ∈ Adj(C_1^S, A^S):
        if C_2^S ∈ Points^S:
            continue
        for C_2^T ∈ Adj(C_1^T, A^T):
            if C_2^T ∈ Points^T:
                continue
            E^S := Edge(C_1^S, C_2^S); E^T := Edge(C_1^T, C_2^T);
            H_1 := λij. (E^S(i) ∧ C_2^T(j)); H_2 := λij. (C_2^S(i) ∧ E^T(j))
            if ((P^S, P^T) := O(B_S, B_T, M, H_1)) or
               ((P^S, P^T) := O(B_S, B_T, M, H_2)):
                Points^S += P^S; Points^T += P^T
    return (Points^S, Points^T)
```

The recursive function `breakupSpace` takes in two points C_1^S and C_1^T which map with each other and also two iteration spaces A^S and A^T.

```
def breakupSpace(C_1^S, C_1^T, A^S, A^T):
    if A^S = φ or A^T = φ:
        return F_Found
    Points^S, Points^T := findMatchingAdjacent(C_1^S, C_1^T, A^S, A^T)
    if |Points^S| ≠ n + 1 or
       |Points^T| ≠ n + 1:
        return F_Not_Found
    T := Transform(Points^S, Points^T)
    H_1 := λ⃗i⃗j. T × i⃗ = j⃗
    a^S := HP(Points^S \ C_1^S); a^T := HP(Points^T \ C_1^T)
    A_1^S = A^S ∪ {a^S}; A_2^S = A^S ∪ {−a^S}
    A_1^T = A^T ∪ {a^T}; A_2^T = A^T ∪ {−a^T}
    H_2 := λ⃗i⃗j. A_2^S(i⃗) ∧ A_2^T(j⃗)
    if O(B_S, B_T, M, H_1 ∧ H_2) is UNSAT:
        return F_Not_Found
    C_2^S ← (Points^S \ C_1^S); C_2^T ← (Points^T \ C_1^T)
    return breakupSpace(C_2^S, C_2^T, A_1^S, A_1^T)
```

12.6 Example Run

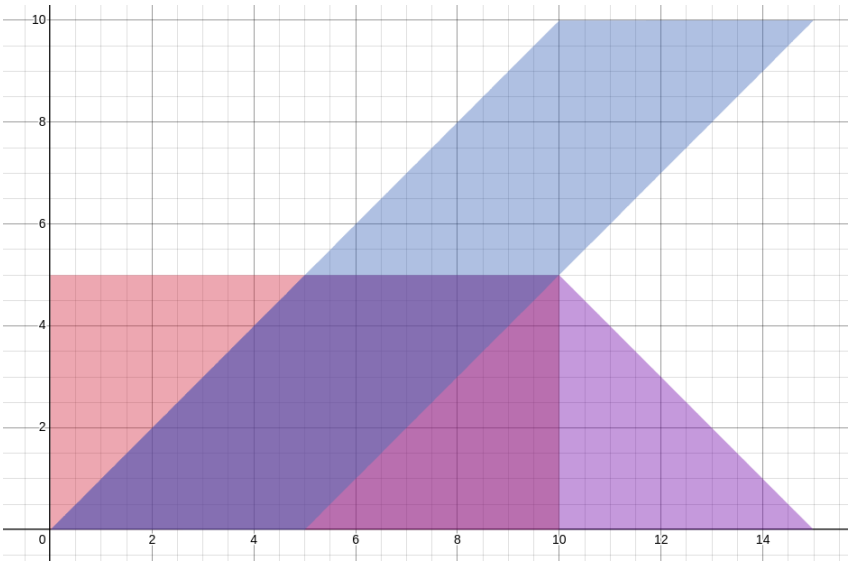
For the example run I'll work with the following loop skew transform:

Source	Target
for $i_1 = 0;\ i_1 \leq N;\ i_1$++:	**for** $j_1 = 0;\ j_1 \leq N + M;\ j_1$++:
for $i_2 = 0;\ i_2 \leq M;\ i_2$++:	**for** $j_2 = \max(0, j_1 - M);$
$x[i_1, i_2] := x[i_1 - 1, i_2 - 1]$	$j_2 \leq \min(N, j_1);\ j_2$++:
	$k := j_2 - j_1 + M$
	$y[j_2, k] := y[j_2 - 1, k - 1]$

Which then get normalized to:

```
for j_1 = 0; j_1 ≤ N + M; j_1++:
    for j_2 = 0; j_2 ≤ min(j_1, N, M, N + M − j_1)−; j_2++:
        k := j_2 + max(0, j_1 − M)
        y[k, k − j_1 + M] := y[k − 1, k − j_1 + M − 1]
```

The source iteration space is given by the red graph. The target iteration space is given by the blue graph and the normalized target iteration space is given by the violet graph.



The color-coded mapping of the points is:

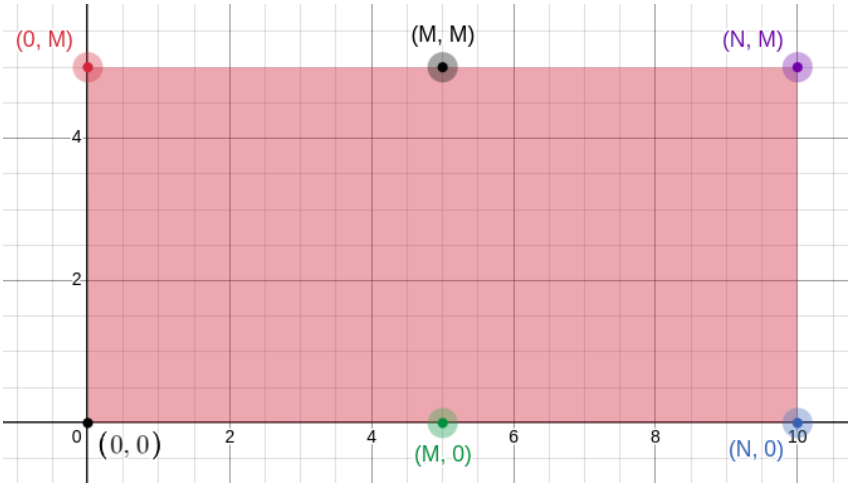


Figure 1: Source Iteration Space

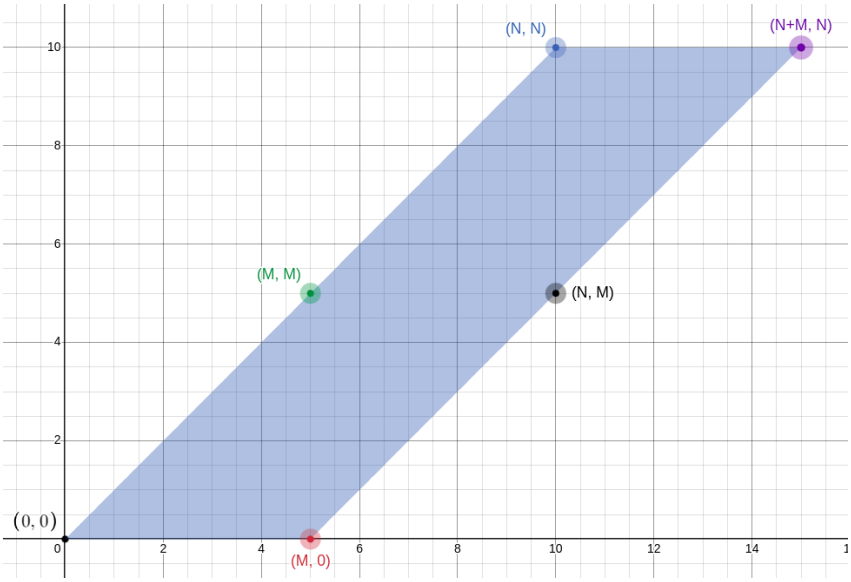


Figure 2: Target Iteration Space

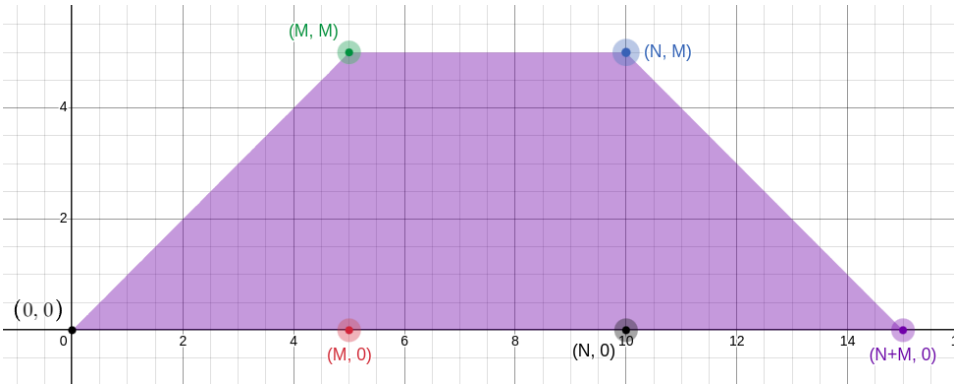
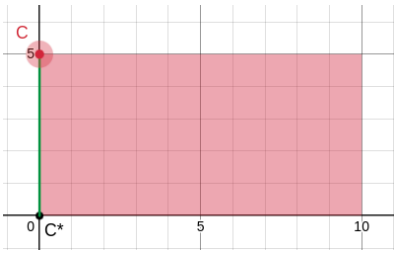
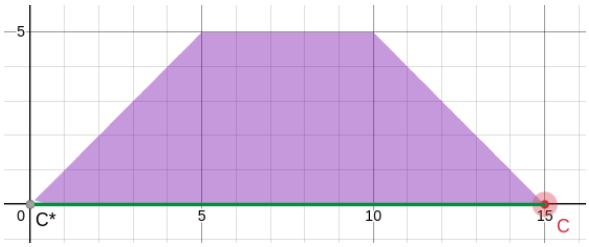


Figure 3: Normalized Target Iteration Space

We will consider one step of the function `breakupSpace` which compares the source with the normalized target space. We start from the initially mapped points $(0, 0)$ and $(0, 0)$. So in the function `findMatchingAdjacent` we get $C_1^S = (0, 0)$, $C_1^T = (0, 0)$.
Inside the for loop let $C_2^S = (0, M)$ and $C_2^T = (N + M, 0)$ (denoted by the red points).
And we get A_E^S, A_E^T as the lines connecting C_1^S, C_1^T with C_2^S, C_2^T respectivly. (denoted by the green lines).

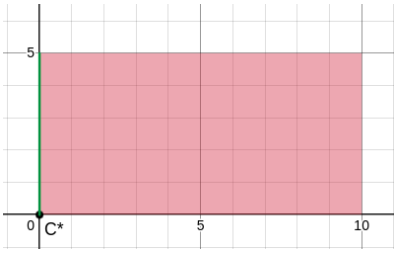


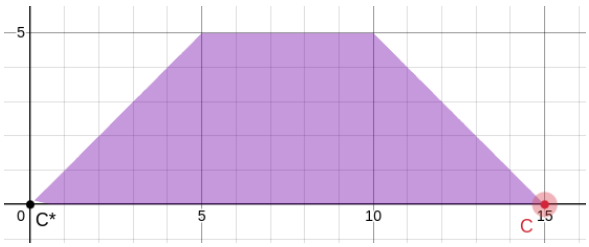


The code:

$H_1 \;:=\; \lambda\vec{ij}.\ (A_E^S(\vec{i}) \wedge C_2^T(\vec{j}))$
$(P^S, P^T) \;:=\; \mathcal{O}(B_S, B_T, M, H_1)$

Asks whether there is a point on the source green line, A_E^S, which gets mapped to the target red point, C_2^T. The oracle will return `SAT`, i.e. there exists no such point.

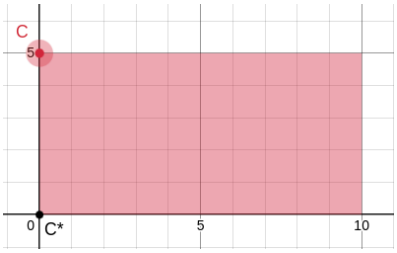


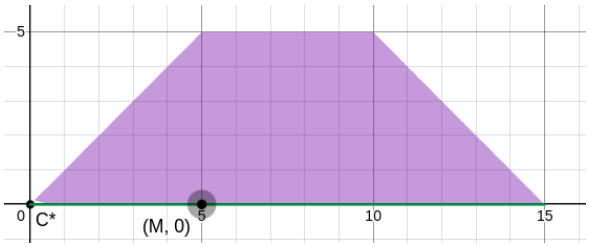


The code:

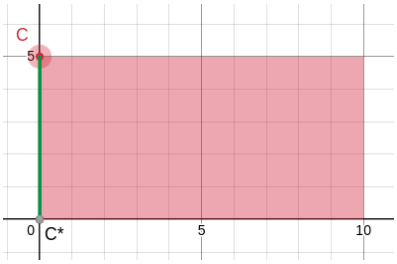
$H_2 \;:=\; \lambda\vec{ij}.\ (C_2^S(\vec{i}) \wedge A_E^T(\vec{j}))$
$(P^S, P^T) \;:=\; \mathcal{O}(B_S, B_T, M, H_2)$

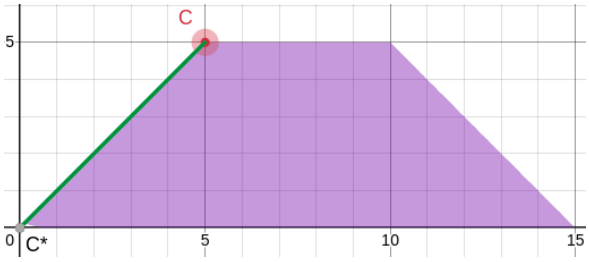
Asks whether there is a point on the target green line, A_E^T, which gets mapped to the source red point, C_2^S. The oracle will return `UNSAT`, i.e. there such a point at $(M, 0)$. Hence $P^S = C_2^S = (0, M)$ and $P^T = (M, 0)$.



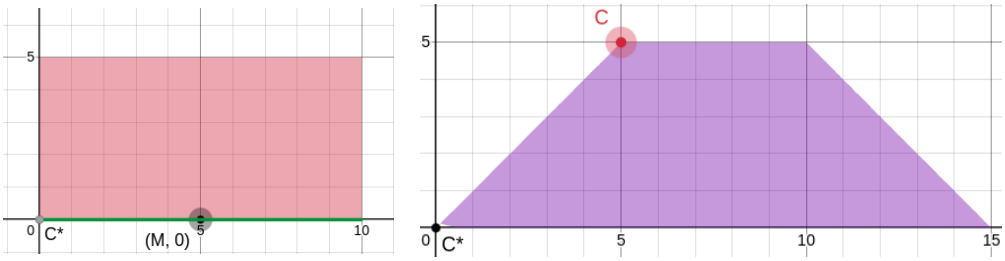


When we try and map the other corner/edge of the target with the same one of the source then we get no mappings.

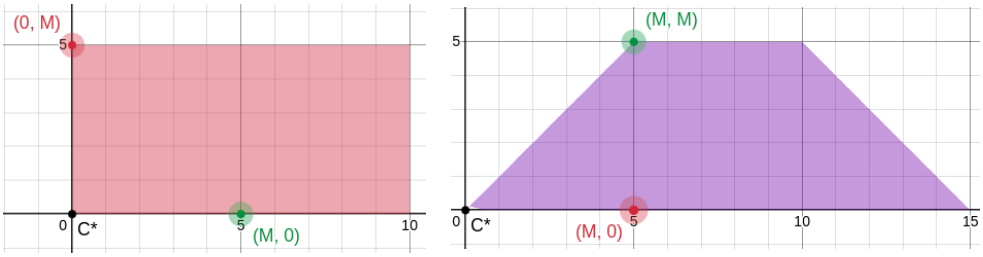




But when we also change the source corner/edge then we get the mapping of $P^S = (M, 0)$ and $P^T = (M, M)$.



Now the array $\texttt{Points}^S$ and $\texttt{Points}^T$ are $[(0,0),(0,M),(M,0)]$ and $[(0,0),(M,0),(M,M)]$ respectively.



So $\texttt{Transform}(\texttt{Points}^S, \texttt{Points}^T)$ calculates:

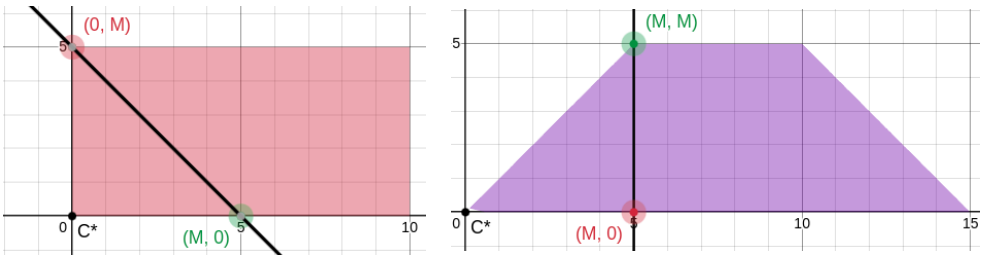
$$\begin{bmatrix} 0 & M & M \\ 0 & 0 & M \\ 1 & 1 & 1 \end{bmatrix} \times \begin{bmatrix} 0 & 0 & M \\ 0 & M & 0 \\ 1 & 1 & 1 \end{bmatrix}^{-1} = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

From the submatrix $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$ we get $j_1 = i_1 + i_2$ and $j_2 = i_1$.

Now the code:

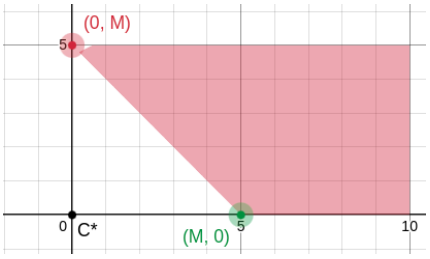
$a^S := \texttt{HP}(\texttt{Points}^S \setminus C_1^S)$
$a^T := \texttt{HP}(\texttt{Points}^T \setminus C_1^T)$

Gives us the following lines (in black):

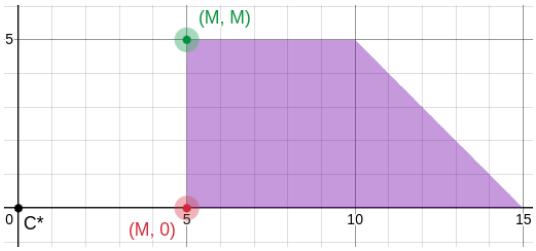


Now we can break up the space along this line by:

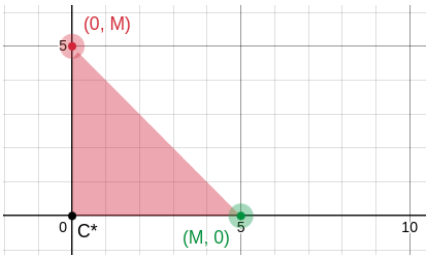
$A_1^S = A^S \cup \{a^S\}$; $A_2^S = A^S \cup \{-a^S\}$
$A_1^T = A^T \cup \{a^T\}$; $A_2^T = A^T \cup \{-a^T\}$



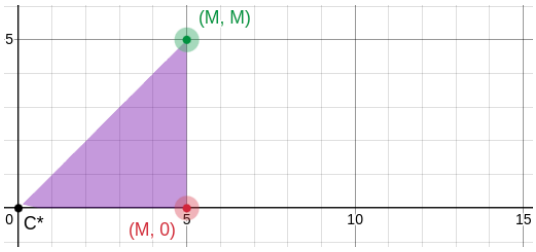
A_1^S



A_1^T



A_2^S



A_2^T

Finally when we call the oracle by $\mathcal{O}(B_S, B_T, M, H_1 \wedge H_2)$ we can check that all points in the the small triangular spaces (A_2^S and A_2^T) are mapped by the equation $j_1 = i_1 + i_2$ and $j_2 = i_1$.
After this is done we can then recursively call the function on the remaining spaces A_1^S and A_1^T.

12.7 Generalization to more spaces

The algorithm above doesn't necessarily have to work with just one source and one target space, we can have multiple of each. This means we can work with transforms like loop splitting and loop peeling trivially. Furthermore we can split an iteration space into two parts if there is something like an if condition in between of which we can determine the condition. So for example in the following code:

```
for i = 0; i ≤ N; i ++:
    if i ≤ N/2:
        x = x + i
    else:
        x = x + N − i
```

The if statements can actually be seen as just for loops, so one way to look at this code is:

```
for i = 0; i < N; i ++:
    for j = i; j ≤ min(i, N/2); j++
        x = x + j
    for j = max(N/2 + 1, i); j ≤ i; j++
        x = x + N − j
```

So we can divide the iteration space into two parts, one for $i \leq \frac{N}{2}$ and the other for $i > \frac{N}{2}$.

12.7.1 Multiple loop bodies at different levels

Consider the following code:

```
for i1 = 0; i1 < N; i1 ++:
    for i2 = 0; i2 < M; i2 ++:
        y = y + i2
    x = x + y
    for i3 = i1; i3 < L; i3 ++:
        z = z + x - i3
```

One way to approach this would be to consider this as a set of three loops and then trying to match these three loops to corresponding three loops in other code. However a better way would be to just consider this entire thing as a union of three spaces and run the algorithm on them.

12.7.2 Generalization to loop fission and fusion

Loop fission is also a natural extension to this idea. If we have a way to somehow determine which statements in the source/target match with which statements in the target/source then we can copy a loop iteration space and make it into two parts, but with different bodies. An example with deals with this idea is loop reindexing.

Source	Target
	if $(N \geq 0)$:
	$x[0] = y[-1]$
for $i = 0;\ i \leq N;\ i$++	**for** $j = 0;\ j \leq N-1;\ j$++
$y[i] = z[i]$	$y[j] = z[j]$
$x[i] = y[i-1]$	$x[j+1] = y[j]$
	if $(N \geq 0)$:
	$y[N] = z[N]$

The source loop and the target loop can be split into two parts with the same space but with different bodies. After this the source will have two spaces and the target would have 4 (two "spaces" would just be points, or 0-dimensional spaces). Then we run the algorithm on these sets of spaces and get an answer.

13 Implementation Details

13.1 Structural Changes

These changes were originally committed to the branch **non-bisimilar**. A summary of the new classes introduced in this branch is given below.

- `graph_correl_step_t<T_PC, T_E>`: This is a template interface which acts as a single step of a tfg which can be correlated with another step (of the other tfg) in the correlation graph. Thus this class replaces the class `graph_full_pathset_t` in many places in the code. In turn `graph_full_pathset_t` and a new class `graph_loopnest_t`

inherit graph_correl_step_t as they can both be taken as a single step to correlate. A tfg_correl_step_t is the instantiation of this class with <pc, tfg_edge>.

- graph_loopnest_t<T_PC, T_N, T_E>: This class inherits graph_correl_step_t<T_PC, T_E> and is yet to be completed. It is stores the region of a loopnest for now. An instantiation of this class with <pc, tfg_node, tfg_edge> is tfg_loopnest_t.

- lockstep_correl_t: This is another interface which abstracts the class path_correl_t. It has function to return a source and target tfg_correl_step_t. Apart from path_correl_t, it is also inherited by loopnest_correl_t.

- loopnest_correl_t: This class inherits lockstep_correl_t and contains one shared_ptr of tfg_loopnest_t each for src and dst.

- correl_entry_t: This was changed slightly to make it work with both kinds of correlations (bisimilar and non-bisimilar). Some data and functions were moved to derived classes.

- bisimilar_correl_entry_t: Inherits correl_entry_t and implements what the previous version of correl_entry_t implemented. Similarly enum_bisimilar_correl_entry_t and CE_propagated_bisimilar_correl_entry_t also implement their previous non-bisimilar counterparts. loopnest_correl_entry_t inherits correl_entry_t and contains a loopnest_correl_t.

- corr_graph_edge: This class was changed to being abstract, just like correl_entry_t. This now gets inherited by corr_graph_bisimilar_edge and corr_graph_loopnest_edge. A bunch of functions were made virtual and need implementation in corr_graph_loopnest_edge, and there are also certain places where we dynamically cast a corr_graph_edge to corr_graph_bisimilar_edge assuming that all edges of the correlation graph are bisimilar, so these places need more implementation.

Along with these classes the rest of the code changes are mainly just to replace classes and their function calls with their new abstract counter-parts. There is also the function tfg_get_loopnests_from_pc added in tfg.cpp which constructs returns the loopnest which starts from a specific pc, and if no such loopnest exists then it returns a nullptr. This function is then used in get_dst_unrolled_paths_from_pc (in cg_with_outgoing_dst_paths.cpp) and in get_next_potential_correlations (in correlate.cpp) to make correlations with loopnests instead of just paths.

13.2 TFG invariant generation

This part is mostly done and doesn't need any more fixes. On a high level the work done here is:

- Created a graph_with_eqclasses class which now contains a lot of the code common to tfg and corr graph invariant generation.

- Moved a bunch of eqclass functions like compute_ineq_eqclass or compute_bv_bool_eqclasses from corr_graph to graph_with_eqclass.

- The eqclass functions work using some virtual functions of the class `graph_with_eqclasses` like `get_src_pc`, `get_dst_pc` etc which can be implemented in the derived classes (corr_graph or tfg).
- Added a function `tfg_generate_invariants` which runs invariant generation on a copy of the tfg. It also changes the copy to add the alloc well-formedness conditions as assumes on edges.

13.3 Itervars and invariants

Currently the iteration variables are not found out directly from the code but given as correlation hints to eq. The implementation details for this are:

- The class `iter_var` holds the following 5 things:
  1. The expression for the iteration variable
  2. The expression for the increment
  3. The expression for the mod invariant that is satisfied by the iteration variable
  4. A list of expressions for the lower bounds
  5. A list of expression for the upper bounds
- We store the `iter_var` class as a map from `pc` to `iter_var`.
- During invariant generation we add invariants corresponding to the section 11.3 in the tfg for each `iter_var`. These invariants are added as stability invariants which means that tfg invariant generation will fail if they aren't proven. This is done in the `compute_invariants_for_iter_var` function.
- The function `add_prev_iter_var` adds the $i_{\texttt{prev}}$ variable by adding another variable in the to state which takes the value of i in the previous iteration.
- The function `get_minimal_anchor_pcs_after_loop` returns the anchor pcs which are first reached after the exit of the loop, we assert the "After loop" invariants at these pcs. There was previously a bug with this implementation where the invariants did not hold at the "After loop" pcs because some variables got their values changed before we reached that point, so to fix that we add some extra variables which take the value of the variables just at the exit of the loop. These are stored in the `var_change_mapping` map.
- The function `add_loop_taken_var_for_loop_head` adds a boolean variable in the tfg for a specific loop which at any pc helps with knowing if the loop had been taken or not. This is achieved by assuming the `loop_taken` boolean variable in the edges exiting the loop and assuming the not of the `loop_taken` variable in some edges which take a path which doesn't go through the loop. This then helps us assert the "After loop" invariants as we only want to assert them if the loop gets taken.

13.4 Next steps for implementation

Most recently, I was working on implementing `recursive_graph_loopnest_t` which would hold a tree-like structure of the loopnest, with each edge having an associated loop and iteration variable and the leafs containing loop-free code stored as a `graph_edge_composition_ref`. A `recursive_graph_loopnest_t` is constructed recursively from an input loop region and the top-most node of this tree is stored in the `graph_loopnest_t`.
After the implementation for this is completed we should work on the implementation of `correl_entry_apply` of the `loopnest_correl_entry_t`, which currently is implemented as a no-op. In this function we have to run the whole algorithm for correlation of two loopnests.