Program Analysis and Verification: Final Project

Ilya Shervin 308640218
Dmitry Kuznetsov 322081183

September 2, 2019

Intro

This document accompanies the code of the final project in Program Analysis and Verification class. Following is a brief rundown of code structure and components.

All of the analysis tasks are implemented with Python 3 (3.7+) with the environment and dependencies managed by Pipenv[1] (see docs for elaborate usage examples).

Our code relies on the following external software packages:

- SLY[2] - a Python implementation of lex/yacc analogues lexers and parsers.
- pySMT[3] - a front-end for SAT and SMT solvers
- z3-solver[4] - a back-end for PySMT based on Z3
- graphviz[5] - a library for visualization of control flow graphs
- sympy[6] - a library implementing symbolic algebra, used in shape analysis node length computations
- pytest[7] - a test harness

The general method of operation for all of the analysis implementations is as following:

1. Parse the input file according to the language definition of the program (parity/-sum/shape).
2. Construct the control flow graph of the program.
3. Initialize all the nodes in control flow graph to the initial abstract state (depending on the abstract domain of specific analysis).
4. Perform chaotic iteration until no more updates happen.

[1] https://docs.pipenv.org
[2] https://github.com/dabeaz/sly
[3] https://github.com/pysmt/pysmt
[4] http://z3prover.github.io
[5] https://github.com/xflr6/graphviz
[6] https://www.sympy.org/
[7] https://docs.pytest.org

5. Iterate over assert statements and verify their validity against the abstract state of the graph node.

Finally, the results are either printed to the screen as DOT graphs or written to disk as PNG files.

Installing and Running

Development of the analysis was done on Linux (any recent Fedora/Ubuntu). To set up a Python environment, run `pipenv install` in the root directory of the project. This will initialize a new virtual environment and pull required packages. Additionally several system packages are required for execution:

- `graphviz`
- `z3solver`

Invocation of analysis is done through `pipenv run analyze`:

```
usage: analyze.py [-h] --type {parity,sum,shape} [--debug] [--no-
    debug]
                  [--output-dir OUTPUT_DIR]
                  [--sum-max-combination-size
                     SUM_MAX_COMBINATION_SIZE]
                  path

positional arguments:
  path                  Path to source

optional arguments:
  -h, --help            show this help message and exit
  --type {parity,sum,shape}
                        Type of analysis to perform
  --debug               Show debug logs
  --no-debug            Do not show debug logs
  --output-dir OUTPUT_DIR
                        If specified, analysis products will be
                           placed in this
                        directory
  --sum-max-combination-size SUM_MAX_COMBINATION_SIZE
                        Sum analysis: maximal combination size of
                           variables to
                        track
```

The project directory layout is as following:

- `analyzeframework/` - Code for basic components used in all analyses (e.g. chaotic iteration, control flow graphs, visualization).
- `analyzenumeric/` - Code specific for parity and summation analyses.
- `analyzeshape/` - Code specific for shape analysis.
- `examples/` - Code examples for all three kinds of included analysis.

- `docs/` - Code for building this file.
- `analyze.py` - Main entry-point of our program.
- `tests/` - Automatic tests for our analyses code.

Parity analysis

Overview

In the first analysis we implemented, we built an abstract state that tracks parity properties of the variables. The abstract state aims to track the following aspects of execution:

- Storing parity of variables when they are assigned a specific value
- Storing a relation of similar parity between a pair of variables when one is assigned to the other.
- Storing a relation of opposite parity when one is assigned ± 1 of another variable.

With this information stored in the state, we're able to verify `assert` predicates (details in next section).

Abstract Domain

Our abstract state is defined as a 3-tuple of the from (M, S, A). Where:

- $M : Symbols \to 2^{\{O,E\}}$ - a mapping that tracks parity of symbols in the program. Each of the symbols is mapped to one of the values:
  - $\emptyset = \bot$ - if no info about symbol parity is known.
  - $\{O\}$ - if symbol is known to be odd-valued.
  - $\{E\}$ - if symbol is known to be even-value.
  - $\{O, E\} = \top$ - if symbol can be either odd or even.
- $S : Symbols \to 2^{Symbols}$ - a mapping of symbols to symbol set of potentially similar parity.

  For example, given x, y, if we know that x is now equal to y, after the assignment we'll change the state so that we'll have $y \in S[x]$.
- $A : Symbols \to 2^{Symbols}$ - a mapping of symbols to symbol set of potentially opposing parity (i.e. for two given symbols, one is odd and other is even).

Our abstract domain is then all possible assignments to the 3-tuple.

The join operation is defined as following: given $\alpha_1 = (M_1, S_1, A_1)$ and $\alpha_2 = (M_2, S_2, A_2)$ abstract states:

$$\alpha_1 \sqcup \alpha_2 = (M_1 \sqcup M_2, S_1 \sqcup S_2, A_1 \sqcup A_2)$$

$$\forall s \in Symbols :$$

$$(M_1 \sqcup M_2)[s] = M_1[s] \cup M_2[s]$$
$$(S_1 \sqcup S_2)[s] = S_1[s] \cup S_2[s]$$
$$(A_1 \sqcup A_2)[s] = A_1[s] \cup A_2[s]$$

When verifying `assert PRED` statements, our analysis computes satisfiability of the following formula: $state_\alpha \wedge \neg pred$. We'll detail how $state_\alpha$ is obtained from $\alpha = (M, S, A)$:

$$state_\alpha = basis_\alpha \wedge modulo_\alpha \wedge relations_\alpha$$

$$basis_\alpha = \bigwedge \{s_{even} \leftrightarrow \neg s_{odd} \mid s \in Symbols\}$$

In this model, atoms are of the form sym_{even} and sym_{odd}, meaning that sym is even or odd respectively. We know no variable can be both odd and even at the same time, and cannot also be neither of them. $basis_\alpha$ formula restricts those pairs of atoms so exactly one can be true.

$$\begin{aligned} modulo_\alpha &= moduloE_\alpha \wedge moduloO_\alpha \\ moduloE_\alpha &= \bigwedge \{s_{even} \mid s \in Symbols \wedge M[s] = \{E\}\} \\ moduloO_\alpha &= \bigwedge \{s_{odd} \mid s \in Symbols \wedge M[s] = \{O\}\} \end{aligned}$$

$modulo_\alpha$ further restricts assignments to the atoms, if we know for sure that a variable is odd or even (through M mapping), then we'll add clause that requires the relevant atom to be true.

$$\begin{aligned} relations_\alpha &= \bigwedge \{relationsE_{sym} \wedge relationsO_{sym} \mid sym \in Symbols\} \\ relationsE_{sym} &= \left[\left[\bigwedge \{s_{even} \mid s \in S[sym]\}\right] \wedge \left[\bigwedge \{s_{odd} \mid s \in A[sym]\}\right]\right] \rightarrow sym_{even} \\ relationsO_{sym} &= \left[\left[\bigwedge \{s_{odd} \mid s \in S[sym]\}\right] \wedge \left[\bigwedge \{s_{even} \mid s \in A[sym]\}\right]\right] \rightarrow sym_{odd} \end{aligned}$$

$relations_\alpha$ encodes the relations between variables. If concrete parity information is unknown for variable x, we can deduce that it is even, if all variables in $S[x]$ are even and all variables in $A[x]$ are odd (and the other way around for odd case).

With the above formula, and the formula derived from the `assert` statement, we use the SAT solver to check if there is an assignment that satisfies the state yet violates the assertion predicate. If such assignment exists, we decide that assertion is invalid.

Example

Given variables x, y, z and the following state:

$$\begin{aligned} &M[x] = \{O\} \quad M[y] = \top \quad M[z] = \top \\ &S[x] = \emptyset \quad S[y] = \{x\} \quad S[z] = \emptyset \\ &A[x] = \emptyset \quad A[y] = \emptyset \quad A[z] = \{y\} \end{aligned}$$

The state formula would be:

$$state = \overbrace{(x_{even} \leftrightarrow \neg x_{odd}) \wedge (y_{even} \leftrightarrow \neg y_{odd}) \wedge (z_{even} \leftrightarrow \neg z_{odd})}^{basis} \wedge \overbrace{x_{odd}}^{modulo} \wedge \overbrace{(x_{even} \rightarrow y_{odd}) \wedge (x_{odd} \rightarrow y_{odd}) \wedge (y_{odd} \rightarrow z_{even}) \wedge (y_{even} \rightarrow z_{odd})}^{relations}$$

Abstract Transformers

We'll now define the transformers we used. All nodes are initialized with the $\alpha_\perp$ state, defined as:

$$\alpha_\perp = (M_\perp, S_\perp, A_\perp)$$
$$\forall s \in Symbols : M_\perp[s] = \emptyset \quad S_\perp[s] = \emptyset \quad A_\perp[s] = \emptyset$$

For any statement we'll denote the tagged mappings as mappings after the transformation:

$$[\![\texttt{stmt}]\!]^\sharp((M, S, A)) = (M', S', A')$$

For `skip` statement, the state is not modified:

$$M' = M, \quad S' = S, \quad A' = A$$

For `i := ?` statement, all information regarding i is removed. Both concrete parity info of the variable, and it's parity with regard to other variables:

$$M'[x] = \begin{Bmatrix} M[x], & x \neq i \\ \top, & x = i \end{Bmatrix}$$
$$S'[x] = \begin{Bmatrix} S[x] \setminus \{i\}, & x \neq i \\ \emptyset, & x = i \end{Bmatrix}$$
$$A'[x] = \begin{Bmatrix} A[x] \setminus \{i\}, & x \neq i \\ \emptyset, & x = i \end{Bmatrix}$$

For `i := K` statement, concrete parity of i is stored, and it's relation to other variables is removed:

$$M'[x] = \begin{Bmatrix} M[x], & x \neq i \\ \{E\}, & x = i \wedge K \text{ is even} \\ \{O\}, & x = i \wedge K \text{ is odd} \end{Bmatrix}$$
$$S'[x] = \begin{Bmatrix} S[x] \setminus \{i\}, & x \neq i \\ \emptyset, & x = i \end{Bmatrix}$$
$$A'[x] = \begin{Bmatrix} A[x] \setminus \{i\}, & x \neq i \\ \emptyset, & x = i \end{Bmatrix}$$

For `i := i` the state is not transformed.

For `i := j` statement, parity info is copied over from j to i, and a relation that i is of same parity as j is added to the state:

$$M'[x] = \begin{Bmatrix} M[x], & x \neq i \\ M[j], & x = i \end{Bmatrix}$$
$$S'[x] = \begin{Bmatrix} S[x] \setminus \{i\}, & x \neq i \\ \{j\}, & x = i \end{Bmatrix}$$
$$A'[x] = \begin{Bmatrix} A[x] \setminus \{i\}, & x \neq i \\ \emptyset, & x = i \end{Bmatrix}$$

For `i := i + 1` and `i := i - 1` statements, concrete parity of i is flipped if known, plus, i's relations to other variables is reversed:

$$M'[x] = \begin{Bmatrix} M[x], & x \neq i \\ \{E, O\} \setminus M[i], & x = i \wedge M[i] \notin \{\bot, \top\} \\ M[i], & \text{otherwise} \end{Bmatrix}$$

$$S'[x] = \begin{Bmatrix} S[x], & x \neq i \wedge i \notin A[x] \\ S[x] \cup \{i\}, & x \neq i \wedge i \in A[x] \\ A[x], & x = i \end{Bmatrix}$$

$$A'[x] = \begin{Bmatrix} A[x], & x \neq i \wedge i \notin S[x] \\ A[x] \cup \{i\}, & x \neq i \wedge i \in S[x] \\ S[i], & x = i \end{Bmatrix}$$

For `i := j + 1` and `i := j - 1` statements, state is transformed in a similar way to straight assignment, except the opposite parity is used.

$$M'[x] = \begin{Bmatrix} M[x], & x \neq i \\ \{E, O\} \setminus M[j], & x = i \wedge M[j] \in \{E, O\} \\ M[i], & \text{otherwise} \end{Bmatrix}$$

$$S'[x] = \begin{Bmatrix} S[x] \setminus \{i\}, & x \neq i \\ \emptyset, & x = i \end{Bmatrix}$$

$$A'[x] = \begin{Bmatrix} A[x] \setminus \{i\}, & x \neq i \\ \{j\}, & x = i \end{Bmatrix}$$

For `assume TRUE` statements the state is not altered, for `assume FALSE` statements, state is transformed into bottom state ($\sqcup$ - neutral) so chaotic iteration will treat it as a dead end.

For `assume i = j` statements, if i and j are known to be of opposite parities, statement is treated as `assume FALSE`, otherwise, state is transformed to create relation of similar parity between the variables, and both are set to most concrete parity known about either of them:

$$M'[x] = \begin{Bmatrix} M[i] \cap M[j], & x \in \{i, j\} \\ M[x], & \text{otherwise} \end{Bmatrix}$$

$$S'[x] = \begin{Bmatrix} S[x] \cup \{j\}, & x = i \\ S[x] \cup \{i\}, & x = j \\ S[x], & \text{otherwise} \end{Bmatrix}$$

$$A' = A$$

For `assume i != j` and `assume i != K` statements state is not transformed.

For `assume i = K` statements, state is checked for conflict against parity of i. If there's a conflict, statement is treated as `assume FALSE`, otherwise parity information is augmented with constant's parity:

$$M'[x] = \begin{Bmatrix} \{E\}, & x = i \wedge K \equiv_2 0 \\ \{O\}, & x = i \wedge K \equiv_2 1 \\ M[x], & \text{otherwise} \end{Bmatrix}$$

$$S' = S \qquad A' = A$$

Finally, for `assert PRED` statements, transformer is similar to `skip`, state is not transformed.

In addition to the aforementioned transformations, after each such transformation and join operation, the abstract state is checked for conflicts and if those appear, they are eliminated. Specifically, if a symbol appears to be of same parity and opposite parity in regards to another variable at the same time, the relation between this pair of variables is removed from the state.

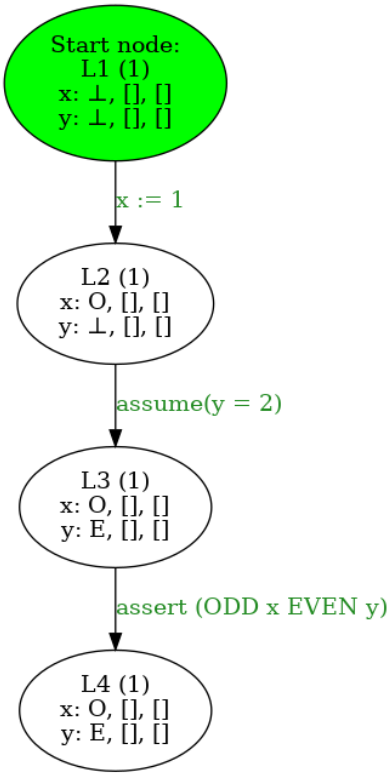
Test programs

Example 1 - Deducing parity from constants

First example of the parity analysis demonstrates the ability to deduce parity of variables based on constant assignment/assumption. The code below demonstrates how parity of x and y is deduced from constants that appear in the code.

The below control flow graph details the nodes of the program, inside them, the abstract state of the node. Format of the state is: $var : M, S, A$, where M is the modulo state of the variable (e.g. E for even), S details all the variables var is possibly in the same parity with, and A details the same with opposite parity.

Additionally, violating statements are displayed in red (relevant for asserts).



(a) Control flow graph

```
x y
L1 x := 1 L2
L2 assume (y = 2) L3
L3 assert (ODD x EVEN y) L4
```

(b) Code

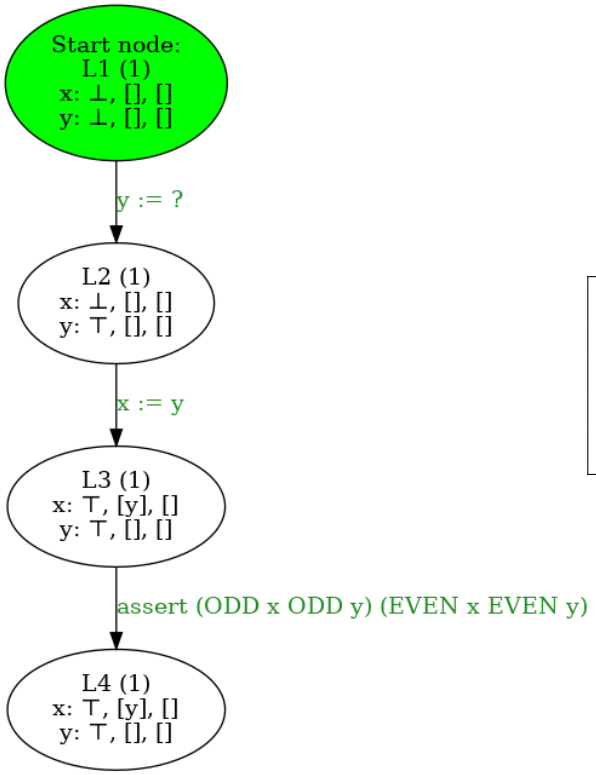
```
pipenv run analyze --type parity docs/parity/examples/example1.code
```

Example 2 - Deducing similar parity from assignments and equality

This example demonstrates that analysis is able to deduce parity of variables in relation to other variables.

Here, after $x := y$ assignment analysis knows that variables are of same parity, without knowing the concrete parity of either.

This can be seen in the state of $L3$ where x's line contains y in the second element (symbols of similar parity).



(a) Control flow graph

```
x y
L1 y := ? L2
L2 x := y L3
L3 assert (ODD x ODD y) (EVEN x
   EVEN y) L4
```

(b) Code

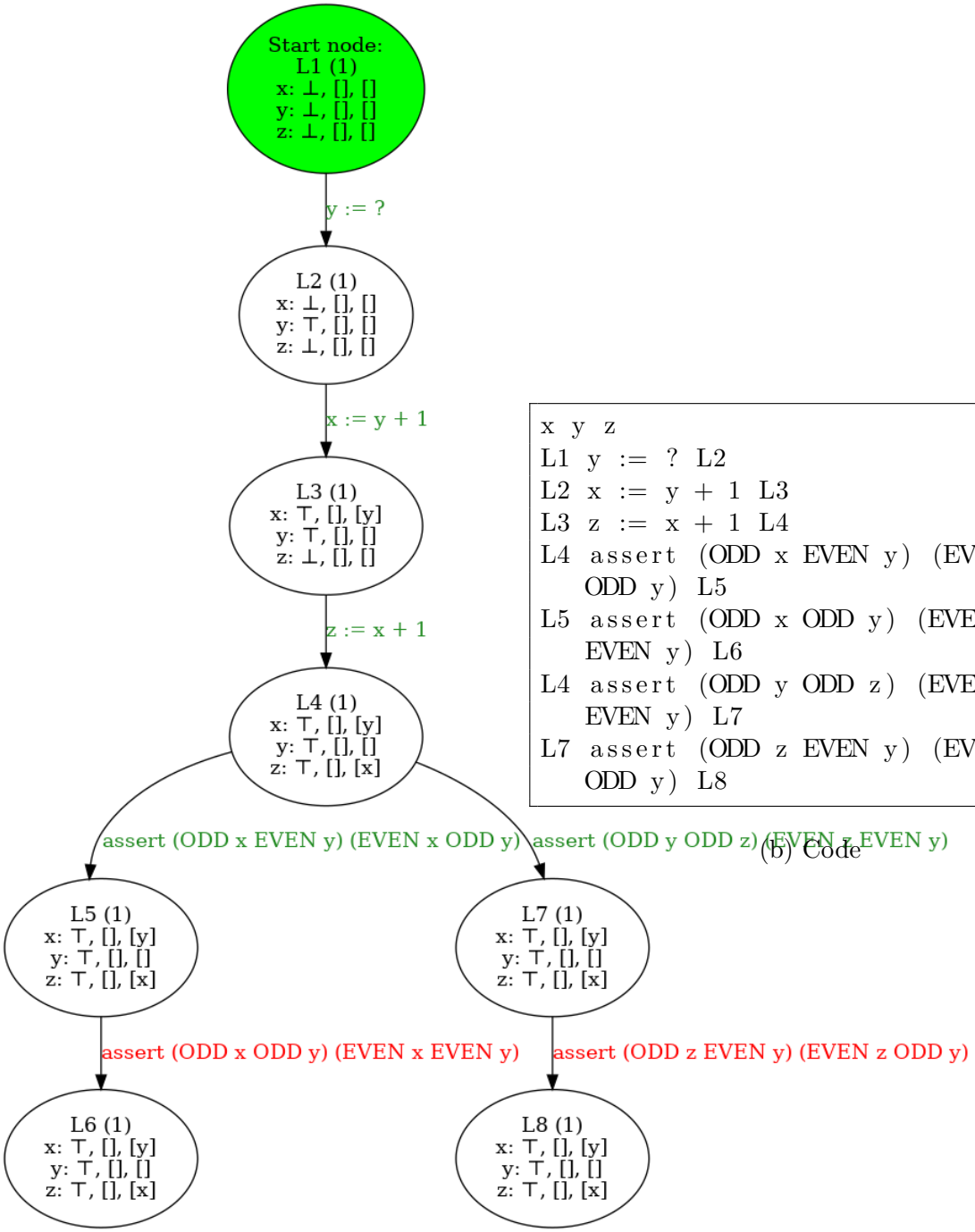
```
pipenv run analyze --type parity docs/parity/examples/example2.code
```

Example 3 - Deducing opposite parity

This example demonstrates that analysis is also able to track opposite parity relation between variables. Here, since x is assigned $y+1$, do don't know its parity but we know it is the opposite of y. We're able to assert that exactly one of x, y has to be odd (assertion on $L4 \to L5$ edge) and that is impossible for x and y to be of similar parity ($L5 \to L6$ edge).

Additionally, we're able to assert that y and z are of similar parity by using knowledge available about relation of x and y.

Disclaimer, in this example code branches without `assume` nodes, they are removed for brevity, and would have not affected the analysis.



(a) Control flow graph

```
x y z
L1 y := ? L2
L2 x := y + 1 L3
L3 z := x + 1 L4
L4 assert (ODD x EVEN y) (EVEN x
   ODD y) L5
L5 assert (ODD x ODD y) (EVEN x
   EVEN y) L6
L4 assert (ODD y ODD z) (EVEN z
   EVEN y) L7
L7 assert (ODD z EVEN y) (EVEN z
   ODD y) L8
```

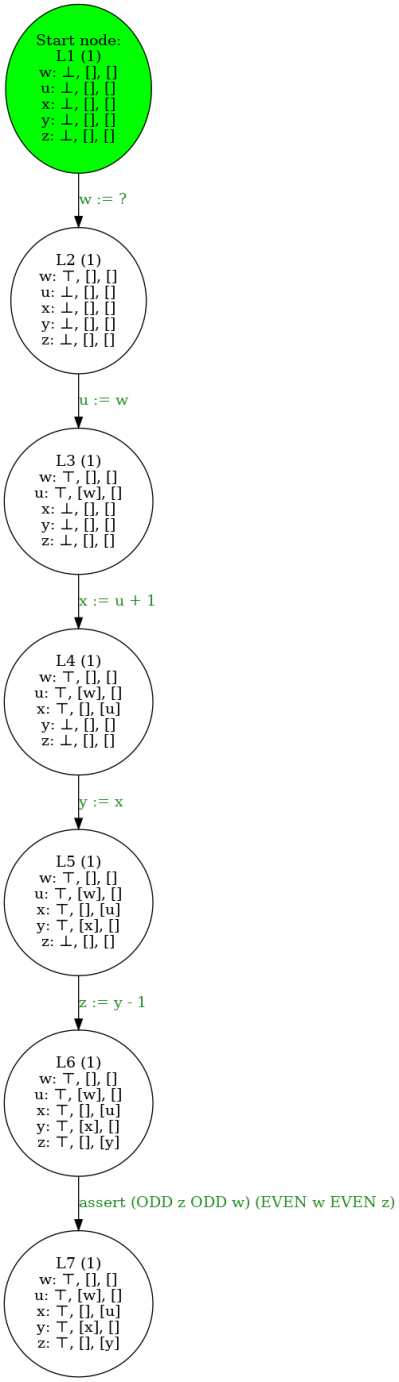
(b) Code

```
pipenv run analyze --type parity docs/parity/examples/example3.code
```

Example 4 - Transitive relations between variables

This examples demonstrates the advantages of using a SAT solver. This example has 5 variables. A series of assignments creates relation of similar or anti-similar parity between pairs of variables.

A chain of 4 links is stored in the abstract state: $w-u$, $u-x$, $x-y$ and $y-z$. Analysis is able to deduce the relation between z and w.



(a) Control flow graph

```
w u x y z
L1 w := ? L2
L2 u := w L3
L3 x := u + 1 L4
L4 y := x L5
L5 z := y - 1 L6
L6 assert (ODD z ODD w) (EVEN w
    EVEN z) L7
```

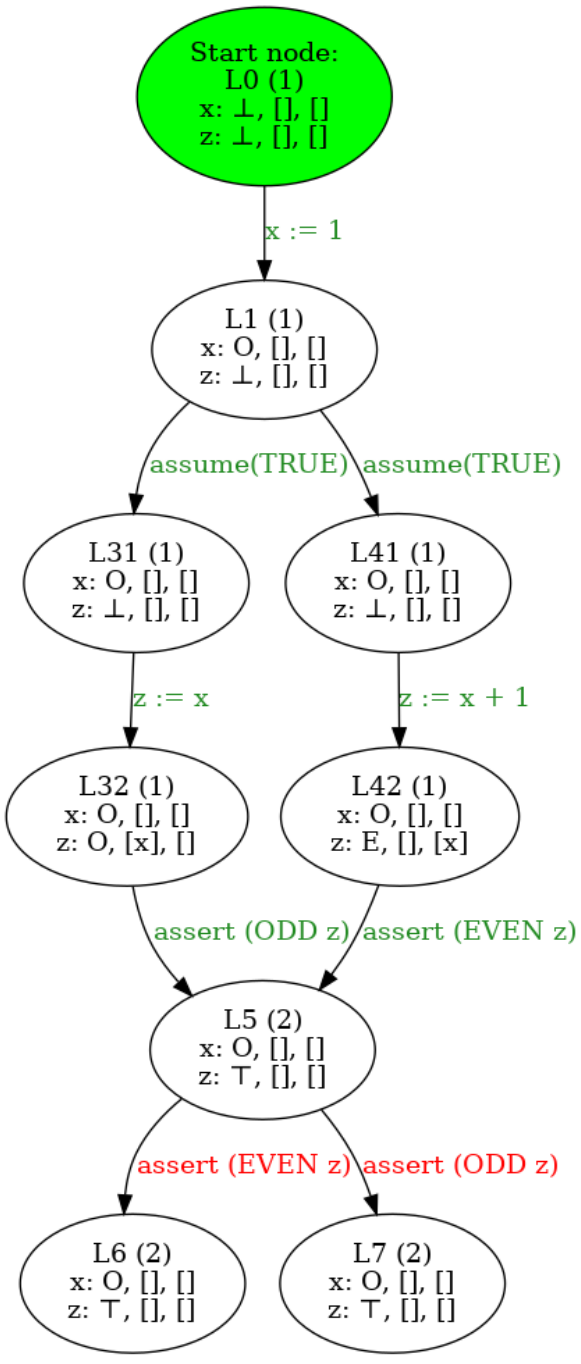
(b) Code

```
pipenv run analyze --type parity docs/parity/examples/example4.code
```

Example 5 - Dealing with conflicts between branches

This example shows that analysis is able to deal with conflict after merging branches. In this example, x is known to be odd. Then code branches to two paths, one setting z to x and other to $x + 1$.

Inside each branch z's parity is known (and is opposite to the other), however, when branches merge, the conflict is removed.



(a) Control flow graph

```
x z
L0 x := 1 L1
L1 assume (TRUE) L31
L1 assume (TRUE) L41
L31 z := x L32
L32 assert (ODD z) L5
L41 z := x + 1 L42
L42 assert (EVEN z) L5
L5 assert (EVEN z) L6
L5 assert (ODD z) L7
```

(b) Code

```
pipenv run analyze --type parity docs/parity/examples/example5.code
```

Summation analysis

Overview

In our implementation of summation analysis we tried to capture the following aspects of execution:

- Storing known constant values of variable or sets of variables.
- Storing differences between pairs of variables.

We used an SMT solver to check whether information provided by our abstract state conflicts with assertion predicates or not.

Abstract Domain

Our abstract state is defined as a 2-tuple of the from (S, D). Where:

- $S : 2^{Symbols} \to \mathbb{Z} \cup \{\bot, \top\}$ - a mapping that tracks constant values of sets of variables.
- $D : (Symbols \times Symbols) \to \mathbb{Z} \cup \{\bot, \top\}$ - a matrix that tracks difference between values of variables.

  For example, if x is known to be greater than y by 2, then $D[x, y] = 2$ and $D[y, x] = -2$.

Our abstract domain is then all possible assignments to the 2-tuple.

In order to control the complexity of analysis, implementation limits S element. While algorithm assumes any element of $2^{Symbols}$, code only initializes $\bigcup_{1 \le i < k} Symbols^i$, i.e. sets of up to k elements.

We'll define $\sqcup$ and $\sqcap$ between elements of $\mathbb{Z} \cup \{\bot, \top\}$ as we have defined them for the lattice we seen in class, for example, $\sqcup$ is defined as:

- $K \sqcup \bot = K$
- $K \sqcup \top = \top$
- $K \sqcup K = K$
- $K \sqcup K' = \top$
- $\bot \sqcup \top = \top$

We'll also define the above $\sqcup$ commutative, meaning that $a \sqcup b \equiv b \sqcup a$.

The join operation is defined as following: given $\alpha_1 = (S_1, D_1)$ and $\alpha_2 = (S_2, D_2)$ abstract states:

$$\alpha_1 \sqcup \alpha_2 = (S_1 \sqcup S_2, D_1 \sqcup D_2)$$

$$\forall s \in Symbols :$$

$$(S_1 \sqcup S_2)[s] = \left\{ S_1[s] \sqcup S_2[s], \quad s \in Symbols \right\}$$

$$(D_1 \sqcup D_2)[s_1, s_2] = \left\{ D_1[s_1, s_2] \sqcup D_2[s_1, s_2], \quad s \in Symbols \right\}$$

When verifying `assert PRED` statements, our analysis computes satisfiability of the following formula: $state_\alpha \wedge \neg pred$. We'll detail how $state_\alpha$ is obtained from $\alpha = (S, D)$:

$$state_\alpha = sums_\alpha \wedge diff_\alpha$$
$$sums_\alpha = \bigwedge \{s_1 + \ldots + s_n = v \mid S[\{s_1, \ldots, s_n\}] = v \wedge v \in \mathbb{Z}\}$$

In this model, atoms are variables of the program. $sums_\alpha$ is the set of clauses that restricts variables to hold the sums known in state.

$$diff_\alpha = \bigwedge \{x - y = D[x, y] \mid x, y \in Symbols \wedge D[x, y] \in \mathbb{Z}\}$$

$diff\alpha$ expresses the known differences between variables.

With the above formula, and the formula derived from the `assert` statement, we use the SMT solver to check if there is an assignment that satisfies the state yet violates the assertion predicate. If such assignment exists, we deem that assertion as invalid.

Example

Given variables x, y, z and the following state:

$$S[\{x\}] = \top \quad S[\{y\}] = 15 \quad S[\{z\}] = \top$$
$$S[\{x, z\}] = 30 \quad S[\{x, y\}] = \top \quad S[\{y, z\}] = \top$$
$$D[x, y] = -5 \quad D[x, z] = -10 \quad D[y, z] = \top$$

The state formula would be:

$$state_\alpha = \overbrace{(y = 15) \wedge (x + z = 30)}^{sums_\alpha} \wedge \overbrace{(x - y = -5) \wedge (x - z = -10)}^{diff_\alpha}$$

Abstract Transformers

We'll now define the transformers we used. All nodes are initialized with the $\alpha_\perp$ state, defined as:

$$\alpha_\perp = (S_\perp, D_\perp)$$
$$\forall s_1, s_2 \in Symbols, s' \in 2^{Symbols} : S_\perp[s'] = \emptyset \quad D_\perp[s_1, s_2] = \emptyset$$

For any statement we'll denote the tagged mappings as mappings after the transformation:

$$[\![\texttt{stmt}]\!]^\sharp((S, D)) = (S', D')$$

For `skip` statement, the state is not modified:

$$S' = S, \quad D' = D$$

For `i := ?` statement, all information regarding i is removed. Both sums that include i (with concrete values), and i's difference to other variables:

$$S'[x] = \begin{Bmatrix} S[x], & i \notin x \\ \top, & i \in x \end{Bmatrix}$$
$$D'[s_1, s_2] = \begin{Bmatrix} D[s_1, s_2], & s_1 \neq i \wedge s_2 \neq i \\ \top, & \text{otherwise} \end{Bmatrix}$$

For `i := K` statement, the transformer is split into 2 cases:

- If i's previous value is known, i.e. $d = (K - S[\{i\}]) \in \mathbb{Z}$, then d is used to augment the i's differences and sums:

$$S'[x] = \begin{Bmatrix} S[x], & i \notin x \\ S[x] + d, & i \in x \end{Bmatrix}$$

$$D'[s_1, s_2] = \begin{Bmatrix} 0, & s_1 = s_2 = i \\ D[s_1, s_2] + d, & s_1 = i \\ D[s_1, s_2], & \text{otherwise} \end{Bmatrix}$$

- Otherwise, I'd value is unknown, therefore its d is unavailable as well. In this case, we remove values of known sums that include i and reset it's deltas from other variables.

$$S'[x] = \begin{Bmatrix} K, & i = \{i\} \\ \top, & i \in x \wedge |x| > 1 \\ S[x], & i \notin x \end{Bmatrix}$$

$$D'[s_1, s_2] = \begin{Bmatrix} 0, & s_1 = s_2 = i \\ \top, & s_1 = i \\ D[s_1, s_2], & \text{otherwise} \end{Bmatrix}$$

For `i := i` the state is not transformed.
For `i := j` statement, we again, break transformation into 2 cases:

- If i's old and j's current values are known, then we can use $d = (S[\{j\}] - S[\{i\}])$ to augment known sums and deltas. The transformation is:

$$S'[x] = \begin{Bmatrix} S[x], & i \notin x \\ S[x] + d, & i \in x \end{Bmatrix}$$

$$D'[s_1, s_2] = \begin{Bmatrix} 0, & s_1 = i \wedge s_2 = j \\ D[s_1, s_2] + d, & s_1 = i \\ D[s_1, s_2], & \text{otherwise} \end{Bmatrix}$$

- Otherwise, delta is unknown and we have to reset sums and deltas of i (other than with j):

$$S'[x] = \begin{Bmatrix} S[x], & i \notin x \\ \top, & i \in x \end{Bmatrix}$$

$$D'[s_1, s_2] = \begin{Bmatrix} 0, & s_1 = i \wedge s_2 = j \\ \top, & s_1 = i \\ D[s_1, s_2], & \text{otherwise} \end{Bmatrix}$$

For `i := i + 1`, sums involving i are increased by 1 if known, and i's differences from other variables are adjusted by 1 as well:

$$S'[x] = \begin{Bmatrix} S[x], & i \notin x \\ S[x] + 1, & i \in x \end{Bmatrix}$$

$$D'[s_1, s_2] = \begin{Bmatrix} D[s_1, s_2] + 1, & s_1 = i \\ D[s_1, s_2], & \text{otherwise} \end{Bmatrix}$$

The transformer for `i := i - 1` is similar with opposite 1 signs .

For `i := j + 1` and `i := j - 1` statements we consider two cases, depending on knowledge of $S[\{j\}]$ and $S[\{i\}]$. The minus case is similar except for minor sign adjustments.

- If $d = S[\{j\}] - S[\{i\}]$ is a concrete value, then:

$$S'[x] = \begin{Bmatrix} S[x], & i \notin x \\ S[x] + d + 1, & i \in x \end{Bmatrix}$$

$$D'[s_1, s_2] = \begin{Bmatrix} 1, & s_1 = i \wedge s_2 = j \\ D[s_1, s_2] + (d+1), & s_1 = i \\ D[s_1, s_2], & \text{otherwise} \end{Bmatrix}$$

- Otherwise, if d is not a concrete value, we have to remove information involving i form the state:

$$S'[x] = \begin{Bmatrix} S[x], & i \notin x \\ \top, & i \in x \end{Bmatrix}$$

$$D'[s_1, s_2] = \begin{Bmatrix} 1, & s_1 = i \wedge s_2 = j \\ \top, & s_1 = i \\ D[s_1, s_2], & \text{otherwise} \end{Bmatrix}$$

For `assume TRUE` statements the state is not altered, for `assume FALSE` statements, state is transformed into bottom state ($\sqcup$ - neutral) so chaotic iteration will treat it as a dead end.

For `assume i = j` statements, if i and j are known to be of different value (either through constant propagation or their de ltas), statement is treated as `assume FALSE`, otherwise, state is transformed to include that delta between i and j is 0:

$$S' = S$$

$$D'[s_1, s_2] = \begin{Bmatrix} 0, & s_1 = i \wedge s_2 = j \\ D[s_1, s_2], & \text{otherwise} \end{Bmatrix}$$

For `assume i != j` and `assume i != K` statements state is checked for conflicts, i.e. if variable i is of value K or equal ot j, then state is transformed to $\alpha_\perp$ and statement is treated as `assume FALSE`.

For `assume i = K` statements, state is checked for conflict with the already known value of i. If there's a conflict, statement is treated as `assume FALSE`, otherwise, constant value is stored in the state:

$$S'[x] = \begin{Bmatrix} K, & x = \{i\} \\ S[x], & \text{otherwise} \end{Bmatrix}$$

$$D' = D$$

Finally, for `assert PRED` statements, transformer is similar to `skip`, state is not transformed.

In addition to the aforementioned transformations, after each such transformation and join operation, the abstract state is augmented by deducing new deltas and sums.

- *Deduction of deltas from singleton sums* - in transformers such as assignment of constant value to a variable, we learn new singleton sum value. When we do that, we immediately know its deltas from from all other singletons we know constant values of. We use this post-transform augmentation to discover new delta values:

$$\begin{aligned} S' =& S \\ D'[s_1, s_2] =& \begin{cases} S[s_1] - S[s_2], & D[s_1, s_2] \notin \mathbb{Z} \wedge S[\{s_1\}], S[\{s_2\}] \in \mathbb{Z} \\ D[s_1, s_2], & \text{otherwise} \end{cases} \end{aligned}$$

- *Deduction of deltas from other known deltas* - for any 3 variables a, b, c it holds that $(a-b)+(b-c) = a-c$. We can use that to deduce extra deltas from already known ones:

$$\begin{aligned} S' =& S \\ D'[s_1, s_2] =& \begin{cases} D[s_1, s_3] + D[s_3, s_2], & D[s_1, s_2] \notin \mathbb{Z} \wedge \exists s_3 : D[s_1, s_3], D[s_3, s_2] \in \mathbb{Z} \\ D[s_1, s_2], & \text{otherwise} \end{cases} \end{aligned}$$

- *Deduction of singletons from deltas* - given 2 variables, a and b with information that $a = K$ and $b - a = N$ we can deduce $b = K - N$. This post transformation tries to deduce the unknown variable values from known variable values and their deltas.

$$\begin{aligned} S'[x] =& \begin{cases} S[\{j\}] + D[i, j], & x = \{i\} \wedge S[x] \notin \mathbb{Z} \wedge \exists j : D[i, j], S[\{j\}] \in \mathbb{Z} \\ S[x], & \text{otherwise} \end{cases} \\ D' =& D \end{aligned}$$

- *Deduction of sum combinations* - when new sum information appears, we can use it to provide/deduce new sums. For instance a, b and c, d can be used to deduce sum on a, b, c, d.

$$\begin{aligned} S'[x] =& \begin{cases} S[y] + S[z], & S[x] \notin \mathbb{Z} \wedge \exists y, z : S[y], S[z] \in \mathbb{Z} \wedge y \cup z = x \wedge y \cap z = \emptyset \\ S[x], & \text{otherwise} \end{cases} \\ D' =& D \end{aligned}$$

- *Deduction of sub sums* - just like we can deduce value of a from $b = N$ and $a-b = K$, we can deduce $a+b$ from $a+b+c+d+e = K$ and $c+d+e = N$. This transformation uses already known sums to deduce more sub sums.

$$\begin{aligned} S'[x] =& \begin{cases} S[z] - S[y], & S[x] \notin \mathbb{Z} \wedge \exists y, z : S[y], S[z] \in \mathbb{Z} \wedge x = z \setminus y \wedge y \cap x = \emptyset \\ S[x], & \text{otherwise} \end{cases} \\ D' =& D \end{aligned}$$

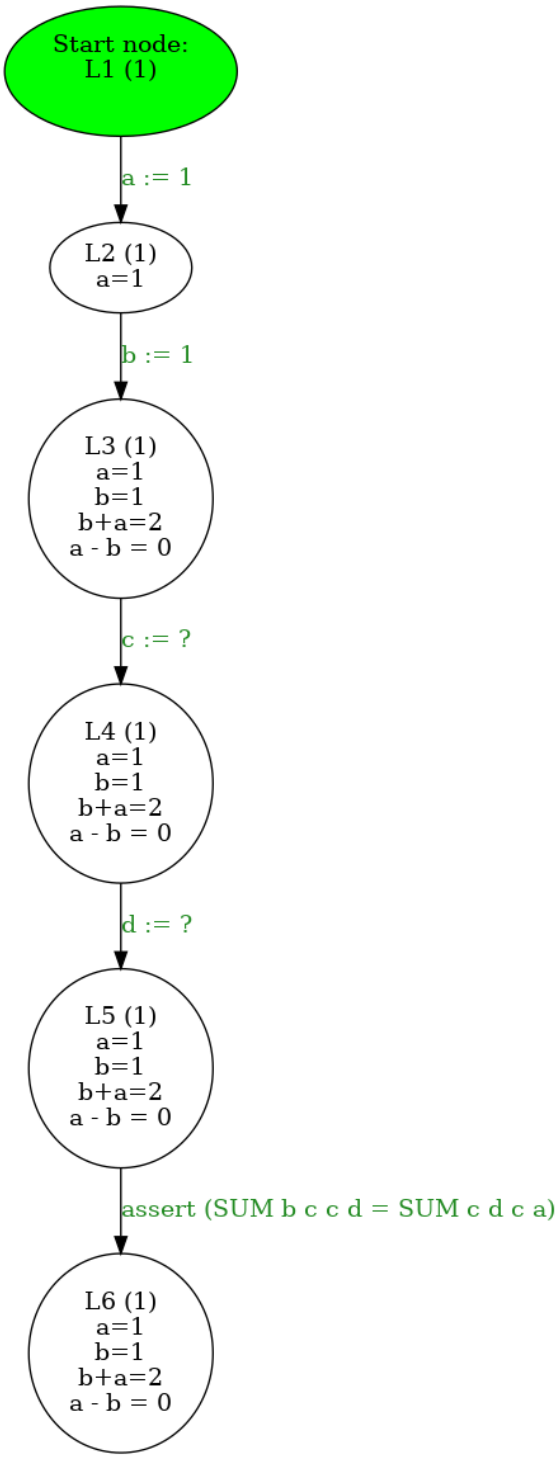
The above transformations are applied in a loop until a fixed point is reached. Process is monotone because:

1. Transformers modify values of S, D mappings.
2. S, D mappings are finite and of constant size for the duration of analysis.
3. Only values of $\top, \bot$ are transformed, the result is always an integer.
4. Thus, each iteration, there's less and less values to operate on.

Test programs

Example 1 - Solving equations with unknowns

In the first example we showcase the abilities provided 'for free' by our SMT solver. In the following example we compare two sums where each has an unknown variable twice. The SMT solver is successfull in applying basic algebra to cancel out same terms.



(a) Control flow graph

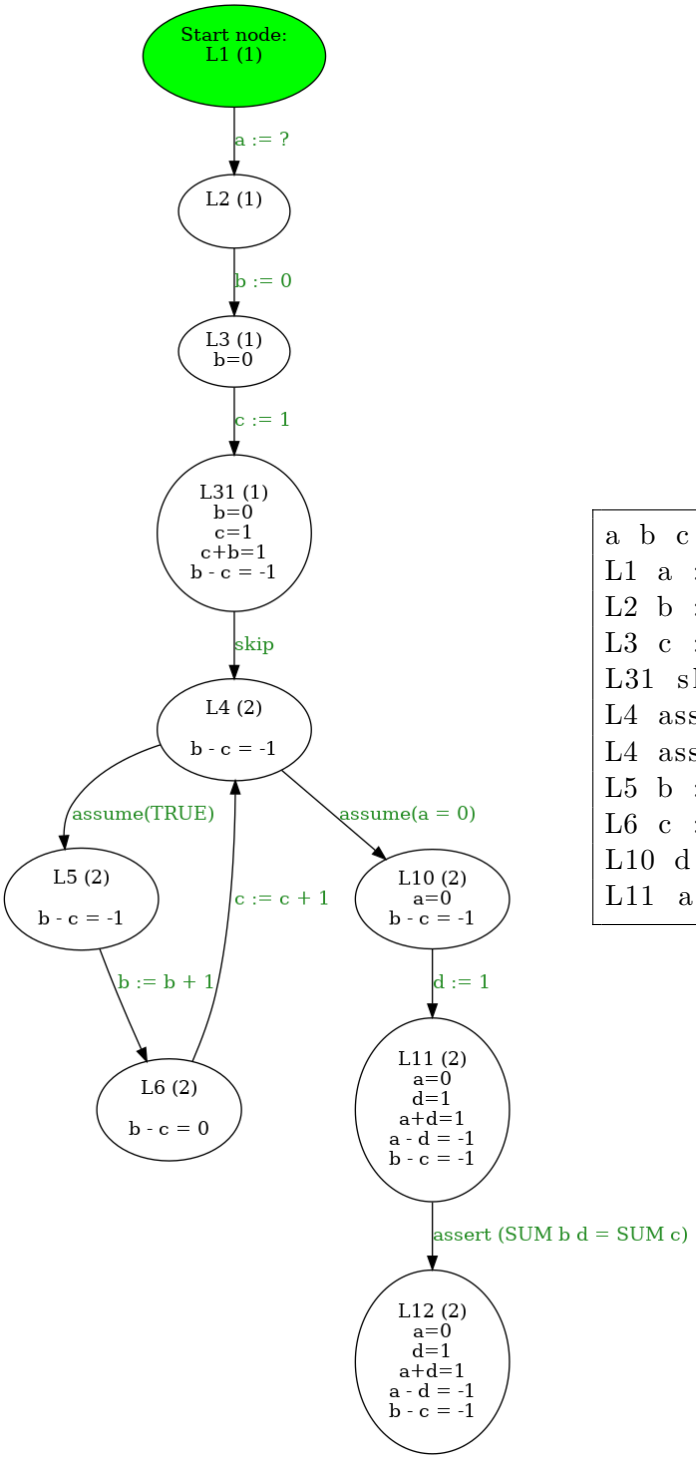
```
a b c d
L1 a := 1 L2
L2 b := 1 L3
L3 c := ? L4
L4 d := ? L5
L5 assert (SUM b c c d = SUM c d c
    a) L6
```

(b) Code

```
pipenv run analyze --type sum docs/sum/examples/example1.code
```

Example 2 - Tracking variable deltas

In the next example we show the usefulness of tracking deltas between variables. The example contains begins with setting $b = 0$ and $c = 0$, then program iterates a random number of times, incremeting b and c each iteration. The analysis is able to deduce that after the loop, b and c still differ by 1.



(a) Control flow graph

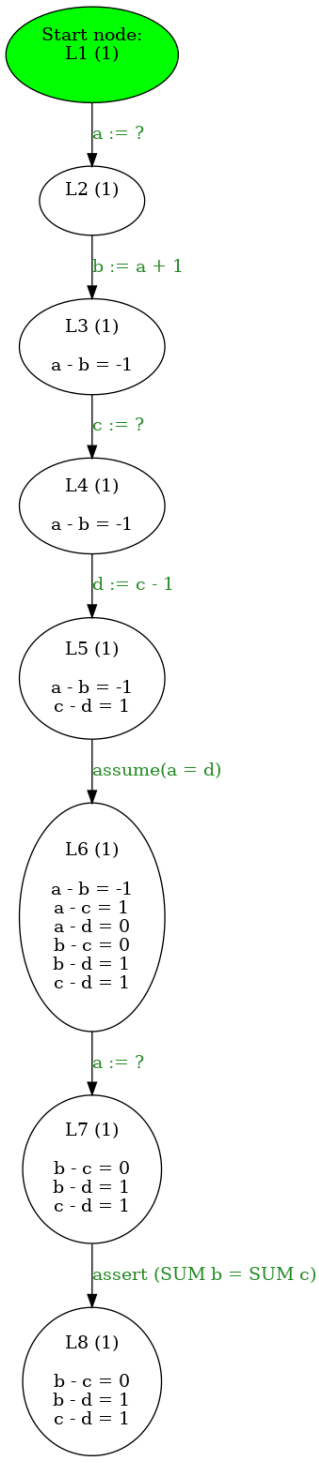
```
a b c d
L1 a := ? L2
L2 b := 0 L3
L3 c := 1 L31
L31 skip L4
L4 assume (TRUE) L5
L4 assume (a = 0) L10
L5 b := b + 1 L6
L6 c := c + 1 L4
L10 d := 1 L11
L11 assert (SUM b d = SUM c) L12
```

(b) Code

```
pipenv run analyze --type sum docs/sum/examples/example2.code
```

Example 3 - Deducing new deltas

Next example demonstrates the deduction of deltas between variables with no direct assigment or relation. In the example we establish deltas $a - b = -1$ and $c - d = 1$. Them we add the information that $a = d$. The analysis is able to deduce all the deltas among a, b, c, d and finally assert that $b = c$ even when we wipe info about a.



(a) Control flow graph

```
a b c d
L1 a := ? L2
L2 b := a + 1 L3
L3 c := ? L4
L4 d := c - 1 L5
L5 assume (a = d) L6
L6 a := ? L7
L7 assert (SUM b = SUM c) L8
```

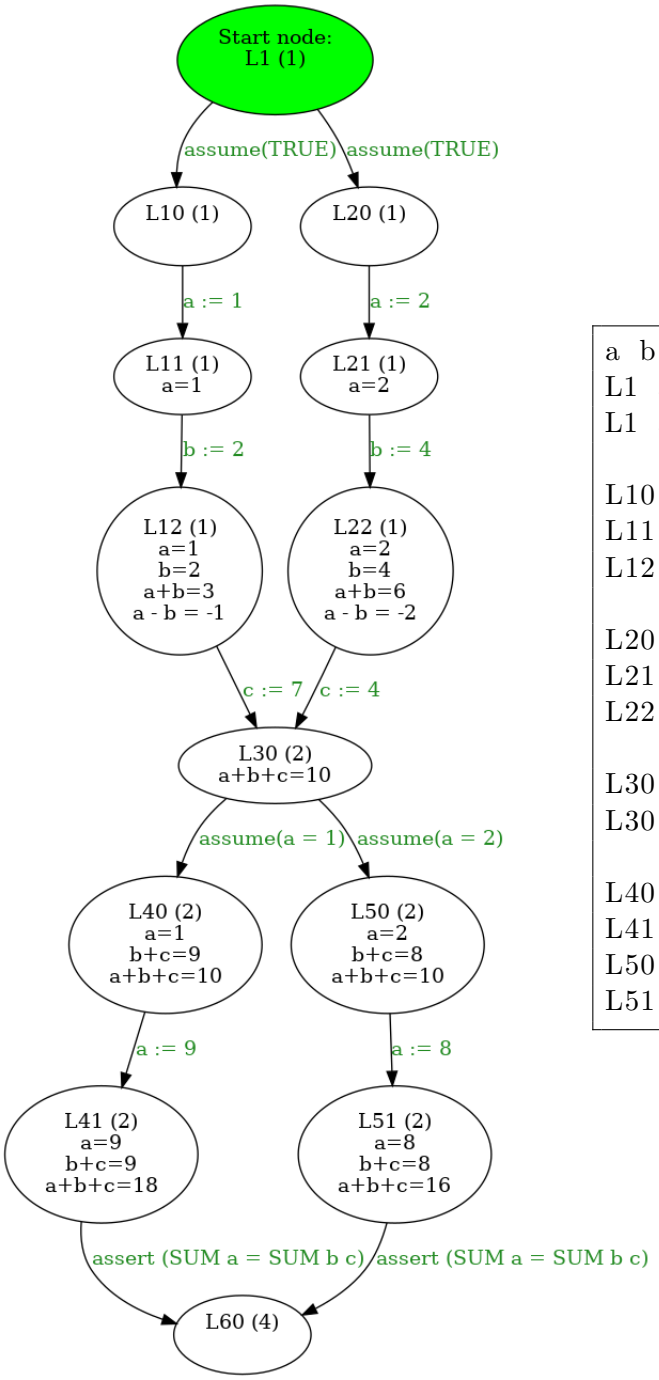
(b) Code

```
pipenv run analyze --type sum docs/sum/examples/example3.code
```

Example 4 - Deducing values of sub-sums

This example shows the deduction of sums from other available sums. This program branches into two path, where on both, variables are set such that $a + b + c = 10$, but no two variables are set to the same value. Upon merging, only the sum remains in the state.

Next, program branches again, assuming the value of a that was given in one of the branches. Analysis is able to decude the value of $b + c$ based on information available in the state.



(a) Control flow graph

```
a b c
L1 assume (TRUE) L10
L1 assume (TRUE) L20

L10 a := 1 L11
L11 b := 2 L12
L12 c := 7 L30

L20 a := 2 L21
L21 b := 4 L22
L22 c := 4 L30

L30 assume (a = 1) L40
L30 assume (a = 2) L50

L40 a := 9 L41
L41 assert (SUM a = SUM b c) L60
L50 a := 8 L51
L51 assert (SUM a = SUM b c) L60
```

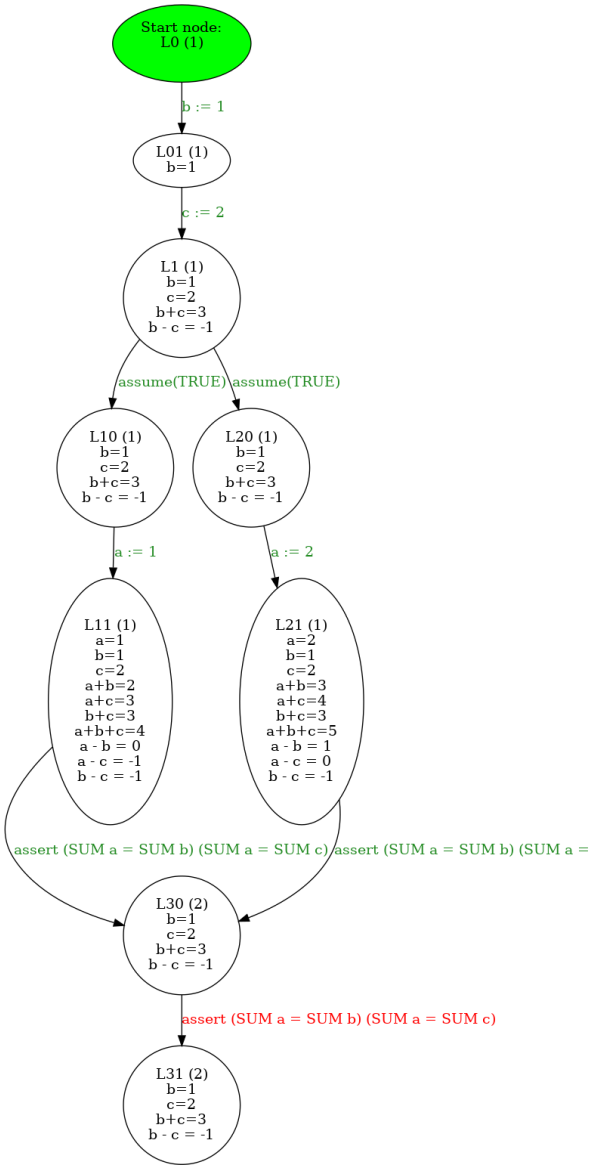
(b) Code

```
pipenv run analyze --type sum docs/sum/examples/example4.code
```

Example 5 - Conflicts between branches

This example shows one of the weaknesses of the analysis. Our analysis uses a very simple lattice of values for sums and delta, this means that we either have a concrete value for a given sum, a $\top$ or a $\bot$. This means that when program branches, and there's a conflict between concrete values, value is assumed to be $\top$ and information is lost.

This example asserts that value of a is either 2 or 3. It is indeed true because each of the branches assigns one of those values, and asserts hold on the branches. Once the branches merge, value of a becomes $\top$ and the same assertion that held on both of the branches is no longer proveable.



(a) Control flow graph

```
a b c

L0 b := 1 L01
L01 c := 2 L1
L1 assume (TRUE) L10
L1 assume (TRUE) L20

L10 a := 1 L11
L11 assert (SUM a = SUM b) (SUM a
    = SUM c) L30

L20 a := 2 L21
L21 assert (SUM a = SUM b) (SUM a
    = SUM c) L30

L30 assert (SUM a = SUM b) (SUM a
    = SUM c) L31
```

(b) Code

```
pipenv run analyze --type sum docs/sum/examples/example5.code
```

Shape analysis

Overview

Our implementation of shape analysis is based on the paper by Sagiv, Reps and Wilhelm "Parametric Shape Analysis via 3-Valued Logic", with modifications to allow for symbolic length analysis between nodes in a linked list. We try to capture the original aspects of execution that are discussed in the paper, along with the addition of abstract length calculations between nodes in a list.

- Storing instrumentation predicates for a node in the heap (concrete or abstract) as defined in the paper - such as reachability predicate. The predicates are evaluated in 3-valued logic.
- Storing a symbolic size for an abstract node.
- Storing an array of possible heap structures according to the current execution state.

With such information, we are able to go over all possible structures and verify that our `assert` statements hold in each of them.

Abstract Domain

Our abstract state is defined as a set of possible heap structures. Each structure $S = \langle U^S, \tau^S, s^S \rangle$ consists of the following:

- U^S - a set of individuals (heap nodes) that can be concrete or abstract.
- τ^S - a set of k-ary predicates τ^S from a k-tuple of individuals to a value in 3-valued logic ($\{0, 1, 1/2\}$).
- $s \in U^S \to Expr$ - a size function that maps an individual to a symbolic algebraic expression.

New possible structures are added during a $focus$ operation or in a loop head join operation. Structures are removed during a *coerce* operation. These operations are described in the next sections. Thus, our abstract domain is all the possible sets of heap structures.

Three Valued Logic

The logical operators are naturally extended to support 3-valued logic in a manner described in class and in the paper:

- $1 \vee 1/2 = 1$
- $0 \vee 1/2 = 1/2$
- $1/2 \vee 1/2 = 1/2$
- $1 \wedge 1/2 = 1/2$
- $0 \wedge 1/2 = 0$
- $1/2 \wedge 1/2 = 1/2$
- $\neg 1/2 = 1/2$

The Set of Predicates

The set of k-ary predicates that we keep for each structure is as follows:

- Unary predicate $x \in U^S \to \{0, 1, 1/2\}$ for each variable $x \in Symbols$ in the program, meaning variable x points to individual.
- Unary predicate $r_x \in U^S \to \{0, 1, 1/2\}$ for each variable $x \in Symbols$ in the program, meaning individual is reached from variable x.
- Unary predicate $c \in U^S \to \{0, 1, 1/2\}$, meaning individual is on a directed cycle.
- Unary predicate $is \in U^S \to \{0, 1, 1/2\}$, meaning individual is pointed to by two or more nodes.
- Unary predicate $sm \in U^S \to \{0, 1, 1/2\}$, meaning individual is a summary (abstract) node.
- Binary predicate $n \in U^S \times U^S \to \{0, 1, 1/2\}$ meaning there is an edge pointing from one individual to another.

We note that owing to the implentation from the paper, the analysis supports an unlimited number of heap shared nodes (where $is(v) \in \{1/2, 1\}$).

Also, the analysis detects null referencing (by attempting to call `n` field of a null variable) and cycles in the linked list. In the former case, the structure becomes invalid and removed from the set. (More on adding/removing structures in the focus/coerce section).

Arbitrary sizes

Let us define A as the set of arbitrary sizes discovered along the execution of the program. An arbitrary size $a \in \mathbb{N}^+$ is an algebraic variable we attach to the size of an abstract node.

We divide arbitrary sizes into two types: temporal and permanent. A permanent size p_i has a scope that can span the whole program, while a temporal size t_i lives inside the loop where it was first introduced into the analysis. We use this distinction later in the coerce stage. Let us define A_p and A_t as the set of permanent arbitrary sizes and temporal arbitrary sizes respectively.

We define $Expr$ as the set of algebraic expressions with variables from A. Such algebraic expression outputs an integer value that represents the size of a node.

For example let $A = p_1, t_1$, $s(v) = e_1 \in Expr$ and $s(u) = e_2 \in Expr$ where $e_1 = p_1 - 3$ and $e_2 = p_1 + t_1 - 2$. This means that the sizes of v and u are bound to p_1 and the size of u is also bound to t_1. We cannot say much about the exact sizes of the nodes but we know that u is bigger than v by $s(u) - s(v) = (p_1 + t_1 - 2) - (p_1 - 3) = t_1 + 1$.

We define two functions on expressions: $even \in Expr \to \{0, 1\}$ and $len \in U^S \times U^S \to Expr$. $even$ tries to deduce parity by checking if there is an even factor in the expression. len sums expressions by walking on the path from one node to the other (inclusive).

Join Operation

In non `assume` statements join operation is simply the set union of the possible structures. In `assume` statements (which are "loop heads" admittedly), we perform set union but may add an extra structure that introduces an arbitrary size. Join operation is described in more detail in the next section.

Verifying Assert Statements

When verifying `assert PRED` statements, our analysis computes satisfiability of the following formula: $state_\alpha \wedge \neg pred$. $state_\alpha$ is obtained from the set of possible structures at a given state. The formula for each structure s_α is a disjunction of formulae with the form $f \rightarrow g$, where f encodes a property of the structure and g encodes the corresponding predicate to assert.

$$
\begin{aligned}
state_\alpha &= \bigvee_{s \in S} s_\alpha \\
s_\alpha &= \bigwedge_{x \in Symbols} (\exists v.x(v) = 1 \rightarrow NotNull_x) \\
&\wedge \bigwedge_{x \in Symbols} (\neg(\exists v.x(v) = 1) \rightarrow Null_x) \\
&\wedge \bigwedge_{x,y \in Symbols} ((\exists v.x(v) = 1 \wedge y(v) = 1) \rightarrow Equals_{xy}) \\
&\wedge \bigwedge_{x,y \in Symbols} (\neg(\exists v.x(v) = 1 \wedge y(v) = 1) \rightarrow NotEquals_{xy}) \\
&\wedge \bigwedge_{x,y \in Symbols} ((\exists v,u.x(v) = 1 \wedge y(u) = 1 \wedge n(v,u) = 1) \rightarrow EqualsNext_{xy}) \\
&\wedge \bigwedge_{x,y \in Symbols} (\neg(\exists v,u.x(v) = 1 \wedge y(u) = 1 \wedge n(v,u) = 1) \rightarrow NotEqualsNext_{xy}) \\
&\wedge \bigwedge_{x,y \in Symbols} ((\exists v.y(v) = 1 \wedge r_x(v) = 1) \rightarrow Reach_{xy}) \\
&\wedge \bigwedge_{x,y \in Symbols} ((\exists v,u.x(v) = 1 \wedge y(u) = 1 \wedge r_x(u) \wedge even(len(v,u)) \rightarrow Even_{xy}) \\
&\wedge \bigwedge_{x,y \in Symbols} ((\exists v,u.x(v) = 1 \wedge y(u) = 1 \wedge r_x(u) \wedge \neg even(len(v,u)) \rightarrow Odd_{xy}) \\
&\wedge \bigwedge_{x,y,z,w \in Symbols} ((\exists v,u.x(v) = 1 \wedge y(u) = 1 \wedge r_x(u) \wedge \exists t,s.z(t) = 1 \wedge w(s) = 1 \wedge r_z(s)) \rightarrow LenEqual
\end{aligned}
$$

In this model, atoms are derived from the counterpart `assert` statement. For example, the statement `x != y.n` is translated to $NotEqualsNext_{xy}$ and the statement `LEN x y = LEN z y` is translated to $LenEquals_{xyzy}$.

We use the SMT solver to check if there is an assignment that satisfies $state_\alpha \wedge \neg pred$.

Theorem 1. $\exists M$ s.t. $M \models state_\alpha \wedge \neg pred \iff$ `assert PRED` is invalid

Proof. If such exists, it means there is a structure s where all formulae in the disjunction s_α are satisfied, including $f \rightarrow pred$. But as $\neg pred$ is true, f must be false, which means that the corresponding property does not hold in this structure and the assertion is invalid.

In the opposite direction, if no assignment exist, for all structures there is an unsatisfied formula. Considering that in each structure the atoms are independant of each other, any formula can become satisfied by setting the relevant atom to true, except the formula $f \rightarrow pred$. It means that in all structures $f \rightarrow pred$ is false as $\neg pred$ is true, which means that f is true and the corresponding property holds. □

Join Operation

The join operation between two states S_1 and S_2 is naturally defined as the set union of the possible structures of each state $Structures_{S_1}$ and $Structures_{S_2}$ respectively.

Another function of the join procedure is to perform an *embed* operation on each structure before the union. *embed* merges elements in U^S that give identical results on every unary predicate. The resulting node from a merge of u and v is a summary node with the same unary predicates values. n predicate is updated and may take a $1/2$ value if there were differences between u and v. Summary node's size is updated as the sum of $size(u)$ and $size(v)$.

Equality between structures is decided by comparing the predicates in each structure in linear time, avoiding the computationaly heavy graph isomorphism problem. In our implementation, size expressions are compared along with the predicates.

Loop Join Algorithm

In loop joins (`assume` statements), a compare function that ignores sizes is used instead. This way, two structures that are identical except maybe the sizes attached to their nodes can be used to add a new structure with an arbitrary size.

The algorithm receives along with the two sets of structures, an arbitrary value a, and only in case of `assume` statement. This value is injected into the sizes of the nodes implying that they have an unknown size, alas bound to the number of iterations.

In order to avoid adding this value again and again we verify beforehand that it doesn't already exist in the structure. In our implementation we do it with an arbitrary sizes stack.

Algorithm 0: Join Operation

Data: $Structures_{S_1}$,$Structures_{S_2}$,$arbitrary\_size$
for $st_2 \in Structures_{S_2}$ **do**
  $embed(st_2)$
end
if *arbitrary_size is Null* **then**
  return $Structures_{S_1} \cup Structures_{S_2}$
end
Set $\leftarrow Structures_{S_1}$;
for $st_2 \in Structures_{S_2}$ **do**
  $st_1 \leftarrow FindEqualStructureIgnoreSize(Structures_{S_1})$;
  if st_1 *is Null* **then**
    $Set.append(st_1)$;
  else
    $st_1^{'} \leftarrow copy(st_1)$;
    if $arbitrary\_size \notin arbitrary\_sizes(st_1^{'})$ **then**
      for $v \in st_1, u \in st_2$ *such that* $size_{st_1}(v) \neq size_{st_2}(u)$ **do**
        $size_{st_1^{'}}(v) = arbitrary\_size \cdot (size_{st_2}(u) - size_{st_1}(v))$;
      end
      $arbitrary\_sizes(st_1^{'}).push(arbitrary\_size)$;
      $Set.append(st_1^{'})$;
    end
  end
end
return *Set*

Example

For example, assume that a structure S_1 in the state of the `assume` statement contains two abstract summary nodes u and v with sizes 2 and 3 respectively. Now, an identical structure S_2 is spotted at the join, but with $size(u) = 3$ and $size(v) = 5$.

We deduce that during the execution in the control flow graph, one node was added to u and two were added to v, before it visited current statement again. Since it is a "loop head", we assume that in the next visit, again one node will be added to the equivalent of u and two to the equivalent v. So we copy S_1, set $size(u) = 2 + a$ and $size(v) = 3 + 2a$ and add it to the set.

Abstract Transformers

The transformers used in the shape analysis are implemented exactly as described in the paper and in the class presentation (where they are refered as update formulae). There are update formulae for all the predicates x, r_x (for every $x \in Symbols$), n, c and is, for each statement.

The only addition to the original algorithm is that in `x := new` statement, we set $size(v) = 1$ where v is the new individual that `x` points to ($x(v) = 1$).

Focus and Coerce Operations

There are two operations that control the number of possible structures in the set, and are described in detail in the paper: $focus$ and *coerce*. $focus$ operation adds possible structures before the update formulae to avoid having variables pointing at summary nodes and avoid moving along n fields whose value in the abstract state is 1/2. *coerce* operation acts after the update formulae and tries to repair structures with the help of the predicates. If a repair is impossible, it removes the structure from the set. In our implementation of the analysis, $focus$ operation from paper was left untouched, whereas an important section was added at the end of *coerce* operation.

Coerce By Size

One of the repairs that original *coerce* tries is to concretisize a summary node $v \in U^S$ once it detects that:

- There is a variable $x \in Symbols$ such that $x(v) = 1$.
- There is another individual $u \in U^S$ such that $n(u, v) = 1$.
- There is another individual $u \in U^S$ such that $n(v, u) = 1$ and $is(u) = 0$.

After the concretization we have a "lost" number of nodes that equals to the previous size $size_{old}(v)$ minus one. Therefore, we need to reclaim this size elsewhere in the structure.

Our first attempt is to see if there is an equal node after v in the list that can take the size. The rationale behind this is that we have duplicated v earlier during $focus$ operation. This happens if the statement is of the form `x:=y.n`, it holds that $y(u) = 1$ for some $u \in U^S$ and since v is a summary node and it holds that $n(u, v) = 1/2$. (see right row in figure 15 from paper).

If such next node doesn't exist, it means the end of our linked list is reached (see middle row in figure 15 from paper). Now we attempt to extract the latest temporary arbitrary variable from the old size and get its value by solving the equation $size_{old}(v) = 1$.

The rationale behind this is that we have "fast-forwarded" to the end of the list and we no longer need the temporary variable. Once we have extracted the variable, we substitute it with the computed value. If, however, the variable is not part of $size_{old}(v)$, we declare that there is no way to redeem the "lost" size and repair the structure and hence we remove it from the set.

Algorithm 1: Coerce by Size Operation

Data: v_c, $size_{old}(v_c)$
$u \leftarrow GetNextSummaryNode(v_c)$;
if *u is not Null* **then**
  $size(u) \leftarrow size(u) - 1$;
  return *True*
else
  $a \leftarrow arbitrary\_sizes.peek()$;
  if $a \notin A_t \vee t \notin variables(size_{old}(v_c))$ **then**
    return *False*
  end
  $[\![a]\!] \leftarrow Solve("size_{old}(v_c) = 1", a)$;
  for $w \in U^S$ **do**
    $size(w) = size(w)[[\![a]\!]/a]$;
    if $ExpressionNegativityCheck(size(w))$ **then**
      return *False*
    end
  end
  $arbitrary\_sizes.pop(a)$;
  return *True*
end

Expression Negativity Check

Another issue that might occur here is that after the substitution, a size becomes negative. It may happen if the analysis wrongly assumes that we have reached the end of a longer list than the one we are in fact iterating. In this case the temporal variable is assigned a value larger than it should. If we detect such case, we immediately remove this structure.

The algorithm to detect a negative expression is described below. First we check if there are permanent arbitrary variables in the expression, and if yes, we check their coefficients. Any negative coefficient is enough to invalidate the structure. For example, we consider $p_1 - p_2$ negative, while $p_1 - 2 \cdot t_1$ not. If there are no permanent variables, but only temporal, we check only for their coefficients. For example $-4 \cdot t_1$ is negative. In the last case, the expression is a constant where the condition is obvious.

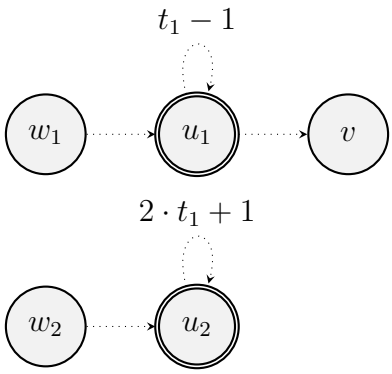
Algorithm 2: Expression Negativity Check

Data: $expr \in Expr$
if $A_p \cap variables(expr) \neq \emptyset$ **then**
  for $var \in A_p \cap variables(expr)$ **do**
    if $coefficient(var, expr) < 0$ **then**
      return *True*
    end
  end
  return *False*
else if $A_t \cap variables(expr) \neq \emptyset$ **then**
  for $var \in A_t \cap variables(expr)$ **do**
    if $coefficient(var, expr) < 0$ **then**
      return *True*
    end
  end
  return *False*
else
  return $expr < 0$
end

Example

For example, assume that during *coerce* we have concretized v, whose previous size was $p_1 - t_1 + 3$. We see that there is no next node in the list that is equal to v, so we reach the conclusion that end of the iteration has come and t_1 is going to be replaced with the solution to the equation $p_1 - t_1 + 3 = 1$, which is $t_1 = p_1 + 2$.

Assume we have two nodes u_1 and u_2 that depend on t_1, such that $size(u_1) = t_1 - 1$ and $size(u_2) = 2 \cdot t_1 + 1$. u_1 is a summary node in the same linked list of v that accumulated the nodes behind v during the iteration. u_2 is a summary node that accumulated nodes at a double rate in each iteration (see diagram below). After the substitution, we get that $size(u_1) = p_1 + 1$ and $size(u_2) = 2 \cdot p_1 + 5$. w_1 and w_2 are not affected by the substitution at all.



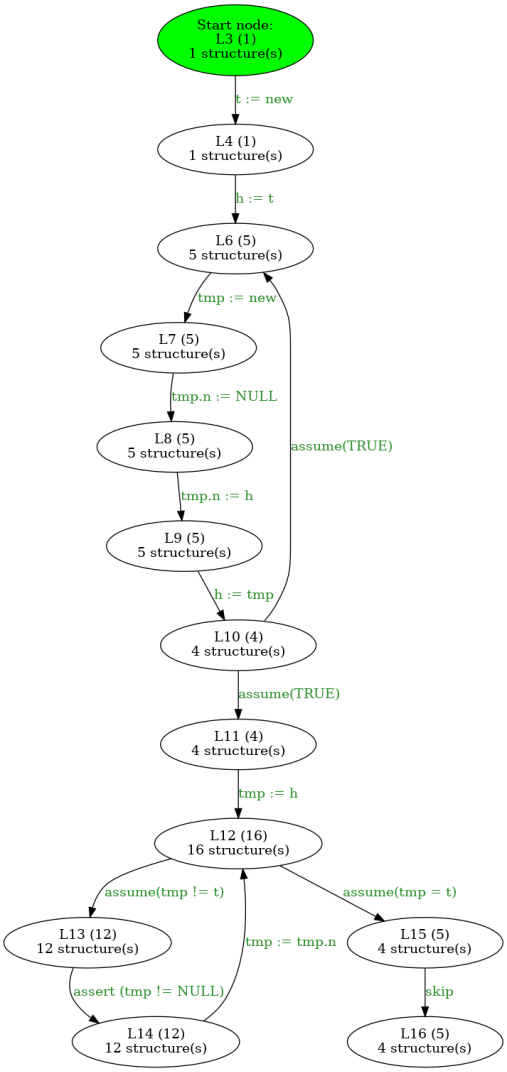
Example Structure

Test programs

Example 1 - Asserting pointer is not NULL while iterating

First example of the shape analysis is taken from the presentation in class. First we create a singly linked list with an arbitrary length, and then we iterate over the list, asserting that `tmp` variable is not NULL during the iteration.

Inside the nodes of the control flow graph we have the number of valid structures in the abstract state. The graphs of the structures themselves are attached separately in a designated folder for each state. Inside the nodes of a structure graph we have as a label a unique index of the node (part of the internal implementation) and the size expression in parentheses.



(a) Control flow graph

```
h t tmp
L3 t := new L4
L4 h := t L6
L6 tmp := new L7
L7 tmp.n := NULL L8
L8 tmp.n := h L9
L9 h := tmp L10
L10 assume(TRUE) L6
L10 assume(TRUE) L11
L11 tmp := h L12
L12 assume(tmp != t) L13
L12 assume(tmp = t) L15
L13 assert(tmp != NULL) L14
L14 tmp := tmp.n L12
L15 skip L16
```

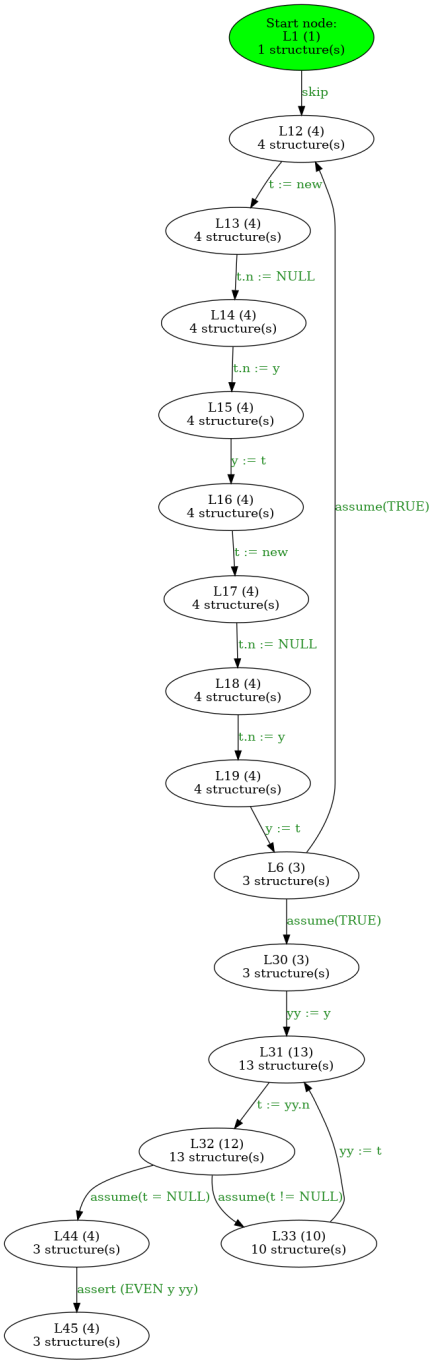
(b) Code

```
pipenv run analyze --type shape docs/shape/examples/example1.code
```

Example 2 - Asserting length between nodes is even

This example demonstrates that analysis is able to deduce parity of length between nodes. We create a singly linked list with an even number of nodes by creating at each iteration two nodes at once.

In the end we can assert that the length between `y` and `yy` is even. If we look closely at the set of structures, we find a structure with a node with size $2 + 2 \cdot PL6$. $PL6$ is a permanent arbitrary size labeled by the line number of the `assume` statement. Because $2 + 2 \cdot PL6 = 2(1 + PL6)$, and $PL6$ is a positive integer, we deduce that this size is in fact always even.



(a) Control flow graph

```
y yy t
L1 skip L12
L12 t := new L13
L13 t.n := NULL L14
L14 t.n := y L15
L15 y := t L16
L16 t := new L17
L17 t.n := NULL L18
L18 t.n := y L19
L19 y := t L6
L6 assume(TRUE) L12
L6 assume(TRUE) L30
L30 yy := y L31
L31 t := yy.n L32
L32 assume(t = NULL) L44
L32 assume(t != NULL) L33
L33 yy := t L31
L44 assert (EVEN y yy) L45
```

(b) Code

```
pipenv run analyze --type shape docs/shape/examples/example2.code
```

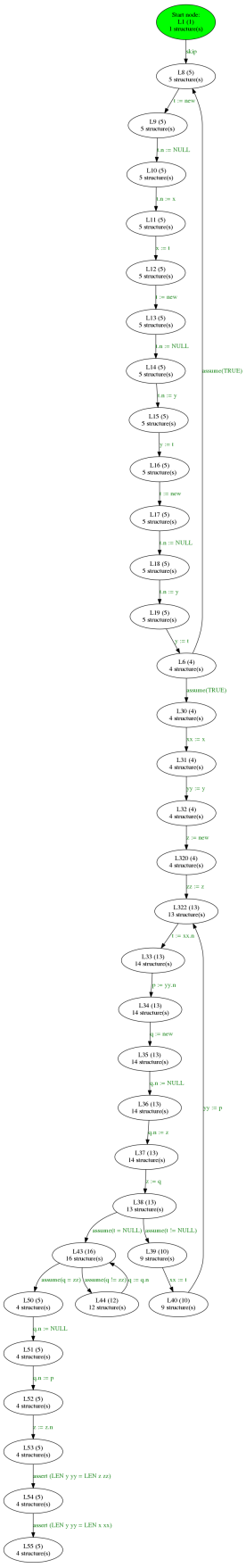
Example 3 - Asserting two lists have same length 1

This example demonstrates that analysis is able to track sizes of nodes in different linked lists during iteration. First we create two singly linked lists pointed to by `x` and `y`, where the list pointed to by `y` is double the size of the list pointed to by `x`.

Now we iterate over the `x` and `y` lists simultaneously until `x` list end is reached, creating a new list pointed to by `z` on the way.

Then we iterate over `z` list to reach its end. We can assert at this point that `LEN y yy` is equals to `LEN z zz` (and also to `LEN x xx`). Note that `yy` is in the "middle" of `y` list, and until the end of this list we have exactly the same number of nodes as we have passed.

In the structures list, we find a structure where we see that `LEN z zz` is $1 + (PL6 + 1) + 1 = PL6 + 3$, same as `LEN y yy`. Note that length from `yy` to the end of the list is also $PL6 + 3$.



(a) Control flow graph

```
x y z xx yy zz t p q
L1 skip L8
L8 t := new L9
L9 t.n := NULL L10
L10 t.n := x L11
L11 x := t L12
L12 t := new L13
L13 t.n := NULL L14
L14 t.n := y L15
L15 y := t L16
L16 t := new L17
L17 t.n := NULL L18
L18 t.n := y L19
L19 y := t L6
L6 assume(TRUE) L8
L6 assume(TRUE) L30
L30 xx := x L31
L31 yy := y L32
L32 z := new L320
L320 zz := z L322
L322 t := xx.n L33
L33 p := yy.n L34
L34 q := new L35
L35 q.n := NULL L36
L36 q.n := z L37
L37 z := q L38
L38 assume(t = NULL) L43
L38 assume(t != NULL) L39
L39 xx := t L40
L40 yy := p L322
L43 assume (q = zz) L50
L43 assume (q != zz) L44
L44 q := q.n L43
L50 q.n := NULL L51
L51 q.n := p L52
L52 z := z.n L53
L53 assert (LEN y yy = LEN z zz)
   L54
L54 assert (LEN y yy = LEN x xx)
   L55
```

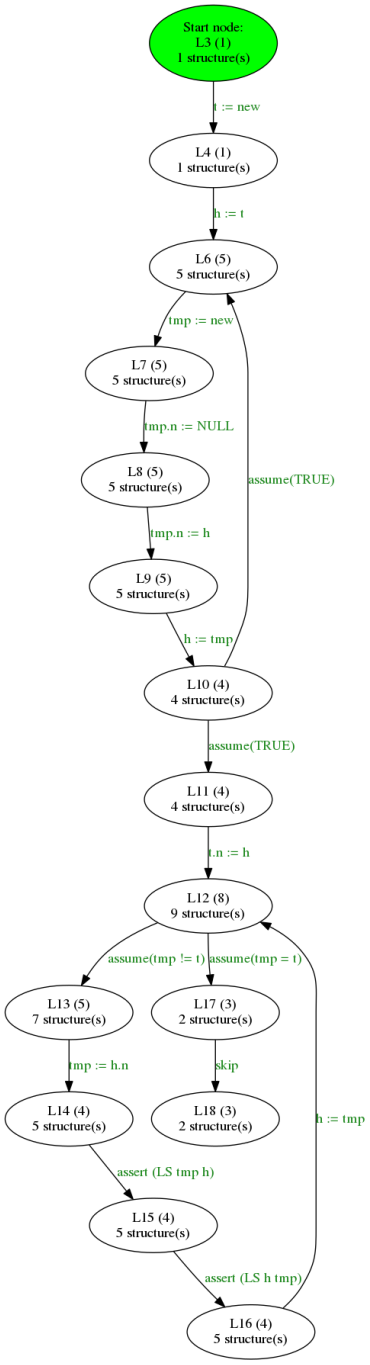
(b) Code

```
pipenv run analyze --type shape docs/shape/examples/example3.code
```

Example 4 - Asserting nodes are reachable in a cycle

This examples demonstrates the ability of the analysis to deduce reachability and cycle properties. Here we create a singly linked list of arbitrary length and afterwards point its head to its tail, creating a cycle.

We iterate over the list, incrementing `tmp` and `h` pointers, asserting on the way that `h` is reachable from `tmp` and vice versa owing to the fact that they are both on a cycle.



(a) Control flow graph

```
h  t  tmp
L3  t  :=  new  L4
L4  h  :=  t  L6
L6  tmp  :=  new  L7
L7  tmp.n  :=  NULL  L8
L8  tmp.n  :=  h  L9
L9  h  :=  tmp  L10
L10  assume(TRUE)  L6
L10  assume(TRUE)  L11
L11  t.n  :=  h  L12
L12  assume(tmp  !=  t)  L13
L12  assume(tmp  =  t)  L17
L13  tmp  :=  h.n  L14
L14  assert(LS  tmp  h)  L15
L15  assert(LS  h  tmp)  L16
L16  h  :=  tmp  L12
L17  skip  L18
```

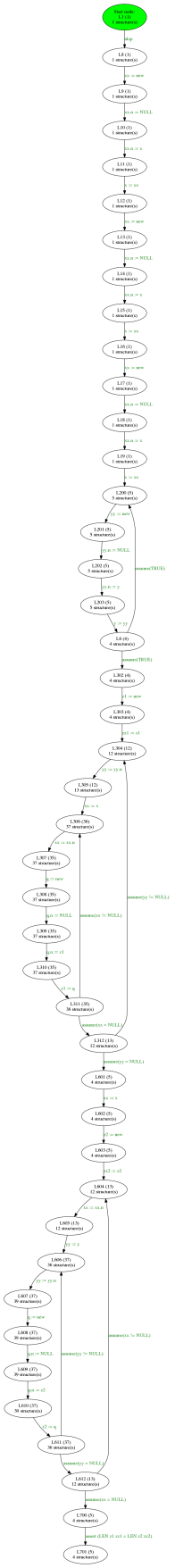
(b) Code

```
pipenv run analyze --type shape docs/shape/examples/example4.code
```

Example 5 - Asserting two lists have same length 2

This example shows again that it can deduce lengths between nodes in different linked lists created in different ways.

Here we create two lists, one of length 3 (pointed to by `x`) and one of an arbitrary length (pointed to by `y`). Now we create two lists (pointed to by `z1` and `z2`). `z1` list is created by first going over `y` list and in each iteration going over `x` list in a nested loop, adding a new node to `z1`. `z2` list is created by first going over `x` list and in each iteration going over `y` list, adding a new node to `z2`. Both lists are of length $3(PL7+2)+1$ in the end.



(a) Control flow graph

```
x y z1 z2 q zz1 zz2 xx yy
L1 skip L8
L8 xx := new L9
L9 xx.n := NULL L10
L10 xx.n := x L11
L11 x := xx L12
L12 xx := new L13
L13 xx.n := NULL L14
L14 xx.n := x L15
L15 x := xx L16
L16 xx := new L17
L17 xx.n := NULL L18
L18 xx.n := x L19
L19 x := xx L200
L200 yy := new L201
L201 yy.n := NULL L202
L202 yy.n := y L203
L203 y := yy L6
L6 assume(TRUE) L200
L6 assume(TRUE) L302
L302 z1 := new L303
L303 zz1 := z1 L304
L304 yy := yy.n L305
L305 xx := x L306
L306 xx := xx.n L307
L307 q := new L308
L308 q.n := NULL L309
L309 q.n := z1 L310
L310 z1 := q L311
L311 assume(xx = NULL) L312
L311 assume(xx != NULL) L306
L312 assume(yy = NULL) L601
L312 assume(yy != NULL) L304
L601 xx := x L602
L602 z2 := new L603
L603 zz2 := z2 L604
L604 xx := xx.n L605
L605 yy := y L606
L606 yy := yy.n L607
L607 q := new L608
L608 q.n := NULL L609
L609 q.n := z2 L610
L610 z2 := q L611
L611 assume(yy = NULL) L612
L611 assume(yy != NULL) L606
L612 assume(xx = NULL) L700
L612 assume(xx != NULL) L604
L700 assert(LEN z1 zz1 = LEN z2
    zz2) L701
```

(b) Code

```
pipenv run analyze --type shape docs/shape/examples/example5.code
```