Specification of Source §1 Infinite Loop Detection—2021 edition

Joey Chen

National University of Singapore
School of Computing

March 1, 2026

1 Introduction

In this report we detail the full specification and method behind the Infinite Loop Detector.
We will first begin by introducing the notion of a *State Transition System* and use it to define and derive results on infinite loops.
Next, we will derive a denotational semantics for *Source* programs where we will be able to extract Symbolic information about the programs, which can then be used to build the state transition system.
The sets will then be used by the *analyzer* to prove the existence of an infinite loop, using theorems in the first section.

2 Objectives

Our main goal for this project is to create a tool to detect infinite loops for students in the NUS module CS1101S, where students learn basic programming skills and CS fundamentals.
Due to Turing's theorem, it is impossible to create a program that detects every instance of nontermination. To circumvent this issue, we can make use of the fact that programs students write for this module are fairly simplistic.
Typical programs students write that encounter infinite loops tend to have a fairly simple control-flow structure that lends itself smoothly to analysis. We conducted a small survey and curated a list of common infinite loops. You can find them here: https://tinyurl.com/source-infloops
Since this tool is meant to aid in students' learning, we will prioritize having no false positives. To ensure this, every detected infinite loop will implicitly contain a proof that the program will not terminate. More details in section 3 below.
Another desirable would be to provide helpful error messages to the students, so that they are able to quickly debug and learn from their mistakes.

3 Preliminaries (Math)

3.1 Definitions

We will adapt the notation in **[CS02]**, but modify it slightly to suit our purposes.
A *state transition system* $\Sigma =< Names, Var : Names \to V_{name}, T >$ consists of the following components:

- $Names$, a finite set of function names, e.g. 'fac', or 'fib', etc. We define a special name $*$ called the *termination symbol*, to denote termination.
- Var, a function which takes as input a function name, and returns a set of global variables used in the function, and variables used as arguments to the function. A *state* σ_{name} is an assignment to all variables in V_{name}. An *assertion* is a first-order formula over a set of variables.
- T, a finite set of *transitions*

A transition is a tuple $< f, g, q, p >$, where:

- $f \in Names$ is the function name of the caller function.
- $g \in Names \cup \{*\}$ is either the function name of the callee function, or a special *terminate symbol*.
- q is an assertion over $Var(f)$ representing a sufficient condition for the transition to be executed.
- p is an assertion over $Var(f) \cup primed(Var(g))$, denoting the relation between the variables in the caller and the callee. We define a special set function, $primed$ that adds primes to all variable names to avoid any name clashes, i.e. $primed : x \mapsto x'$

A *program path* is a nonempty sequence $(< name_1, \sigma_1 >, < name_2, \sigma_2 >, ...)$ of pairs containing a function name $name_i$ and a corresponding state in $Var(name_i)$, such that:
For each adjacent pair $< name_i, \sigma_i >$ and $< name_{i+1}, \sigma_{i+1} >$, there exists $\tau =< f, g, q, p > \in T$ and $name_i = f,\ name_{i+1} = g, \sigma_i \models q, (\sigma_i, \sigma_{i+1}) \models p$.
The only exception is when g is the termination symbol, then the sequence ends.
A *computation* is a triple $< \Sigma, start, \Omega >$ where

- Σ is a state transition system
- $start$ is one of the names in Σ
- Ω is a state of $Var(start)$. $start$ and Ω represent the first function call in the computation.

Example 3.1 *Consider the function fib:*

```
function fib(x) {
  if(x===0 || x===1) {
    return 1;
  } else {
    return fib(x-1) + fib(x-2);
  }
}
```

The corresponding state transition system for fib will be: $\Sigma :=< Names, Var : Names \to V_{name}, T >$ *where*

$$Names = \{fib\}$$

$$Var(fib) = \{x\}$$

$$\begin{aligned} T = \{ &< fib, fib, (x > 1), (x' = x - 1) >, \\ &< fib, fib, (x > 1), (x' = x - 2) >, \\ &< fib, fib, (x < 0), (x' = x - 1) >, \\ &< fib, fib, (x < 0), (x' = x - 2) >, \\ &< fib, *, x = 0, \top >, \\ &< fib, *, x = 1, \top > \\ \} \end{aligned}$$

A program path will be:

$$(< fib, x = 4 >, < fib, x = 2 >, < *, \top >)$$

$$(< fib, x = 4 >, < fib, x = 3 >, < fib, x = 1 >, < *, \top >)$$

Note that different paths may start with the same pair. An infinite loop happens if any one of the paths are infinite.

We will use the following theorem to motivate our definition of termination and infinite loops.

Theorem 3.1 *The length of a program path is finite iff it contains the terminal symbol *.*

Proof:
finite $\Leftarrow$ contains $*$: by definition, once $*$ is encountered, the sequence ends.
finite $\Rightarrow$ contains $*$: consider otherwise, i.e. $*$ is not encountered, then let n be the length of the sequence since it is finite. Now, let $< f_n, \sigma_n >$ be the n^{th}, i.e. last element of the sequence. Then by hypothesis, f_n is not $*$ and by definition, there will be another element $< f_{n+1}, \sigma_{n+1} >$ in the program path, so the length of the program path is greater than n, which is a contradiction. □

We say a program path *terminates* if it contains the termination symbol $*$. A computation *terminates* if all the possible program paths starting with $< start, \Omega >$ terminate. A computation is said to go into an *infinite loop* if it does not terminate.
We will prove that a computation goes into an infinite loop by constructing a program path that does not have the termination symbol.

3.2 Immediate Results

From these definitions, we can already describe two very common classes of infinite loops: forgetting the base case and forgetting to update the state.

Theorem 3.2 *If for some f, $\forall < f, g, q, p > \in T,\ f = g$ then any program path containing f does not terminate.*

This corresponds to the case where there is a missing base case.
Proof: For some program path $< g_1, \sigma_1 >, < g_2, \sigma_2 >, ..., < f, \sigma_i >$, clearly $*$ is not present, and the path has not terminated yet. By the hypothesis, the next call in the path will be $< f, \sigma_{i+1} >$. By induction, we can construct an infinitely long program path. □

Example 3.2 *Consider the function fac:*

```
function fac(x) {
  return x*fac(x-1);
}
```

The corresponding transition set is

$$T = \{< fac, fac, \top, (x' = x - 1) >\}$$

Using the previous theorem, any program path will be infinite, i.e.

$$(< fac, x = 1 >, < fac, x = 0 >, < fac, x = -1 >, ...)$$

Theorem 3.3 *If for some f, $\exists < f, g, q, p > \in T, \exists \sigma s.t.\ f = g, \sigma \models q, (\sigma, primed(\sigma)) \models p$, then any comptation containing $< f, \sigma >$ in one of its paths does not terminate.*

This corresponds to the case where the state is not updated between function calls.
Proof: For some program path $< g_1, \sigma_1 >, < g_2, \sigma_2 >, ..., < f, \sigma >$, clearly $*$ is not present, and the path has not terminated yet. By the hypothesis, we can construct a program path which will contain $< f, \sigma >$ as the next call. By induction, we can construct an infinitely long program path.
□

Example 3.3 *Consider the function fac:*

```
function fac(x) {
  if(x===0) {
    return 1;
  } else {
    return x*fac(x);
  }
}
```

The corresponding transition set is

$$\begin{aligned} T = \{ & < fac, fac, (x > 1), (x' = x) >, \\ & < fac, fac, (x < 1), (x' = x) >, \\ & < fac, *, x = 0, \top >, \\ & \} \end{aligned}$$

Using the previous theorem, we can construct the infinite program path:

$$(< fac, x = 1 >, < fac, x = 1 >, < fac, x = 1 >, ...)$$

3.3 Countdown Functions

From an observation of simple programs in introductory program courses, most of the control flow statements are limited to inequalities involving an integer variable and constant.
Integer inequalities involving only one variable are also very easy to manipulate algebraically, and every inequality can be written in the form $x < c$ where x is an integer variable and c is an integer constant. For example $x \geq 1$ is equivalent to $-x < -2$
A common pattern in introductory programming exercises is managing control flow with something we call a 'countdown variable'. The example below will better illustrate this point.

```
function fac(x) {
  if(x===0) {
    return 1;
  } else {
    return fac(x-1);
  }
}
```

In this code snippet, we notice 3 things:

1. A single integer value x is used for control flow, depending on whether $x = 0$ or $x \neq 0$.
2. One branch of the if statement terminates the program, while the other calls fac recursively, but with its argument $x - 1$.
3. If $x < 0$, the program will go into an infinite loop.

We can generalize this to include functions where control flow is dependent only on an inequality involving a single integer variable and an integer constant, and this variable is only incremented/decremented by another integer constant in each recursive function call.
For convenience we define $T_{name} = \{< f, g, q, p > \in T \mid f = name\}$

Theorem 3.4 *If for some f, $x \in Var(f)$, $T_f = \{< f, f, x \neq c, x' = x - d >, < f, *, x = c, >\}$, where $c, d \in \mathbb{Z}, d > 0$, then any program path containing $< f, x < c >$ will not terminate.*

Proof: For some program path $< g_1, \sigma_1 >, < g_2, \sigma_2 >, ..., < f, x < c >$, clearly $*$ is not present, and the path has not terminated yet. let c_1 be the value of x, and $c_{i+1} = c_i - d$. Then, since $c_1 < c$, and $c_{i+1} < c$, $\forall i c_i < c$. Then using the first transition in the transition set, we can construct an infinitely long program path $< g_1, \sigma_1 >, ..., < f, x = c_1 >, < f, x = c_2 > ....$

□

Corollary 3.1 *If for some f, $x \in Var(f)$, $T_f = \{< f, f, x \neq c, x' = x + d >, < f, *, x = c, >\}$, where $c, d \in \mathbb{Z}, d > 0$, then any program path containing $< f, x > c >$ will not terminate.*

Proof: simply repeat the previous proof, replacing $-$ with $+$, and $<$ with $>$. □

4 Implementation

Now that we have a mathematical foundation for our analysis of infinite loops, we only need to link it to the syntax of Source programs. Our goal will be to first obtain the function transition set of the program, then test it for nontermination.
The implementation will consist of 3 stages: the symbolic executor, the serializer and the analyzer.
The symbolic executor will convert the Source AST into another tree-like structure as an intermediate value, which will then be used by the serializer to build the function transition set.
The analyzer will then use the function transition sets to produce domains where the functions will enter an infinite loop, using the theorems above.
The process goes something like this:
$S_4 \rightarrow_{SymbolicExecutor} Symbol \rightarrow_{Serializer} T(f) \rightarrow_{Analyzer} D$

4.1 Symbolic Execution

The symbolic executor will in essence, summarize the AST into a tree of symbols which contain algebraic data about the program.

4.1.1 Symbols

We will introduce a few symbols that we use below.

- A *Literal Value symbol* will be used to represent literal integer or boolean values. These symbols will be used mainly for intermediate computation and evaluation.

  We will denote this symbol by $LV(x)$ where x is a literal value, for example $LV(4)$ represents the number 4.

- The *Number symbol* will be used to represent algebraic expressions involving the sum of an integer variable and a constant, for example $x + 1$.

  We will denote number symbols by $N(x, c, sign)$ where x is the name of the variable, c is the constant value, and $sign$ is the sign of x (1 for positive or -1 for negative). e.g. we will denote $x + 1$ by $N(x, 1, 1)$.

- The *Inequality symbol* will be used to represent equalities or inequalities between an integer variable and a constant, for example $x < 20$. We will denote inequality symbols by $Ineq(x, a, direction)$, where x is the name of the variable, a is the constant on the right hand side, and $direction$ is -1 for $<$, 0 for $=$, and 1 for $>$.

- A *Logical symbol* will be used to represent conjunctions or disjunctions between boolean symbols. We will denote logical symbols by $L(B_1, B_2, type)$ where B_1 and B_2 are the boolean symbols, and $type$ is either $conjunction$ or $disjunction$. An example is $L(Ineq(x, -1, -1), Ineq(x, 1, 1), conjun$ for $x < -1 \wedge x > 1$.

- A *Boolean symbol* is an inequality symbol or a logical symbol.

- The *Branch symbol* will be used when processing if statements, and will contain a logical symbol for the condition, and 2 symbols for the consequent and alternate statements respectively. We will denote branch symbols by $Br(B, Sym_1, Sym_2)$ where B is a boolean or skip symbol, and Sym_1 and Sym_2 are arbitrary symbols.

- The *Sequence symbol* will be used to denote a list of symbols executed consecutively. We will denote it with $Seq(Sym_1, ..., Sym_n)$ where $Sym_1, ..., Sym_n$ are arbitrary symbols.

- A *Function symbol* will be used to denote a function call and its arguments. We will denote it by $F(f, x_1, ..., x_n)$ where f is the name of the function, and $x_1, ..., x_n$ are its parameters.

- The *Termination symbol* will be used to denote that the program will terminate. Similarly to section 2, we will use $*$ for the termination symbol.

Since we are only interested in the control-flow structure of the program, we will introduce a *skip* symbol to denote that we do not care about the result of evaluating the statement. For expressions we are unable to analyze, we will also use the skip symbol to denote that we are ignoring this expression.
If the skip symbol is used in the construction of another symbol, we will return another skip symbol to denote that the construction does not make sense. Two exceptions are the branch and sequence symbols. To those familiar with functional programming, the skip symbol is similar to the *Optional* or *Maybe* monads used in languages such as Scala and Haskell.
We summarize the symbols below.

Symbol	Explanation
Num	The algebraic expression $x + sign \cdot c$
Inequality	The set $\{x \mid sign \cdot x < sign \cdot c\}$
LV	literal values
Logical	Conjunction or disjunction of B_1 and B_2 respectively depending on $type$
BoolSym	Boolean Symbols from **Logical + Inequality**
Branch	Symbol for conditional expressions
Sequence	A sequence of symbols
Function	Function symbol
Terminate	Termination symbol
Skip	Symbolizes we ignore this expression

For brevity when the values inside the symbols are not used we will omit them (e.g. F for $F(f, Sym_1, ...Sym_n)$

4.1.2 Store

To keep track of variable declarations, we will introduce the notion of a *store*. Note that this is a simplified version of the store used in the CS4215 notes.
A Store Σ will be used to store symbols, indexed by identifiers (i.e. strings). We will define an operation, $\Sigma[l \leftarrow v]$, which denotes a store that works like Σ, except that $\Sigma(l) = v$

4.1.3 Symbolic Executor

Semantic Domain	Definition	Explanation
Num	**Id * Int * Bool**	Number Symbol
Inequality	**Id * Int * Bool**	Inequality symbol
LV	**Int + Bool**	literal values
Logical	**BoolSym * BoolSym * Bool**	Logical symbol
BoolSym	**Inequality + Logical**	Boolean Symbols
Branch	**BoolSym * Sym * Sym**	Branch symbol
Sequence	**Sym * ... * Sym**	Sequence symbols
Function	**Id * Sym * ... * Sym**	Function symbol
Terminate	$\{*\}$	Termination symbol
Skip	$\{Skip\}$	Skip symbol
Symbol	**Num + LV + BoolSym + Branch + Sequence + Function + Termination + Skip**	Set of all possible symbols
Store	$Id \rightarrow Symbol$	store of symbols

The semantic function $\cdot|\cdot \rightarrow \cdot$ is defined as a three-argument relation:

$$\cdot|\cdot \rightarrow \cdot : \textbf{Store} * \textbf{Source §1} \rightarrow \textbf{Symbol} * \textbf{Store}$$

Most of the time, we will not be using the store, so we will use abbreviated symantic function where Σ is implicit (TODO??)

$$\cdot \rightarrow \cdot : \textbf{Source §1} \rightarrow \textbf{Symbol}$$

$$E \rightarrow Sym := \Sigma | E \rightarrow (Sym, \Sigma)$$

When we encounter literal values, we simply wrap them with the LV tag.

$$\frac{}{\texttt{true} \to LV(true)} \qquad \frac{}{\texttt{false} \to LV(false)} \qquad \frac{}{n \to LV(n)}$$

For identifiers, we scan through the store and return the corresponding symbol, if found.

$$\frac{}{\Sigma | x \to (\Sigma(x), \Sigma)}$$

For constant declarations, we simply update the store.

$$\frac{E \to Sym}{\Sigma | \texttt{const} x = E \texttt{ ;} \to (v, \Sigma[x \leftarrow Sym])}$$

For sequences of statements.

$$\frac{\Sigma | E_1 \to (Sym_1, \Sigma^{(1)}) \qquad ... \qquad \Sigma^{(n-1)} | E_1 \to (Sym_n, \Sigma^{(n)})}{\Sigma | E_1 \texttt{; } ... \texttt{ ; } E_n \texttt{;} \to (Seq(Sym_1, \ ... \ , Sym_n), \Sigma^{(n)})}$$

For blocks we revert changes to the store once we exit.

$$\frac{\Sigma | E \to (Sym, \Sigma')}{\Sigma | E \to (Sym, \Sigma)}$$

For if statements and conditional operators, we wrap the branch test, consequent and alternative in their respective locations in a new branch symbol.

$$\frac{E_1 \to L_1 \qquad E_2 \to Sym_1 \qquad E_3 \to Sym_1}{if(E_1)\{E_2\} \ else \ \{E_3\} \to Br(L_1, Sym_1, Sym_2)}$$

$$\frac{E_1 \to L_1 \qquad E_2 \to Sym_1 \qquad E_3 \to Sym_1}{E_1 ? E_2 : E_3 \to Br(L_1, Sym_1, Sym_2)}$$

Similarly for call statements, we simply wrap the function name and its arguments in a new function symbol.

$$\frac{E_1 \to Sym_1, E_2 \to Sym_2 ... E_n \to Sym_n}{f(E_1, E_2, ..., E_n) \to F(f, Sym_1, Sym_2, ...Sym_n)}$$ where f is an identifier for some function

For return statements, we want to replace the symbol with a termination symbol if the expression does not include any function calls. To do this, we define a function, $terminal : Symbol \to Bool$ as

follows.

$$terminal(x) = \begin{cases} \texttt{false}, & x \in Function \\ terminal(x_1) \wedge ... \wedge terminal(x_n) & x = Seq(x_1, ..., x_n) \\ terminal(x_1) \wedge terminal(x_2) & x = Br(B, x_1, x_2) \\ \texttt{true} & otherwise \end{cases}$$

$$\frac{E \to Sym}{return\ E \to Sym}\ terminal(Sym)$$

$$\frac{E \to Sym}{return\ E \to *}\ \neg terminal(Sym)$$

For unary operators, we carry out the respective operations on the literal value, or symbols.

$$\frac{E \to LV(x)}{!E \to LV(\neg x)} \qquad \frac{E \to LV(x)}{-E \to LV(-x)}$$

$$\frac{E \to Bool}{-E \to BoolNegate(Bool)} \qquad \frac{E \to Num(x, c, positive)}{-E \to Num(x, -c, negative)}$$

Binary operators are more cumbersome as there are many cases to consider. We only allow addition and subtraction between a number symbol and literal value, or between two literal values.

$$\frac{E_1 \to LV(a) \qquad E_2 \to LV(b)}{E_1 + E_2 \to LV(a + b)} \qquad \frac{E_1 \to LV(a) \qquad E_2 \to LV(b)}{E_1 - E_2 \to LV(a - b)}$$

$$\frac{E_1 \to N(x, c, sign) \qquad E_2 \to LV(a)}{E_1 + E_2 \to N(x, c + a, sign)} \qquad \frac{E_1 \to LV(a) \qquad E_2 \to N(x, c, sign)}{E_1 + E_2 \to N(x, c + a, sign)}$$

$$\frac{E_1 \to N(x, c, sign) \qquad E_2 \to LV(a)}{E_1 - E_2 \to N(x, c - a, sign)} \qquad \frac{E_1 \to LV(a) \qquad E_2 \to N(x, c, sign)}{E_1 - E_2 \to N(x, a - c, -sign)}$$

For inequality operators, we only allow inequalities between number symbols and literal values.

$$\frac{E_1 \to N(x, c, sign) \qquad E_2 \to LV(a)}{E_1 < E_2 \to Ineq(x, a - c, sign)} \qquad \frac{E_1 \to LV(a) \qquad E_2 \to N(x, c, sign)}{E_1 < E_2 \to Ineq(x, a - c, -sign)}$$

We can make use of the previous definitions to define the rest of the inequalitites more succintly. For convenience, we define the function $neg : BoolSym \to BoolSym$ to allow us to negate integer inequalities.

$$neg(sym) = \begin{cases} Ineq(x, c, -sign), & sym = Ineq(x, c, sign), sign \neq 0 \\ L(Ineq(x, c, -1), Ineq(x, c, 1), conjunction), & sym = Ineq(x, c, sign), sign = 0 \\ L(neg(S_1), neg(S_2), disjunction), & sym = L(S_1, S_2, conjunction) \\ L(neg(S_1), neg(S_2), conjunction), & sym = L(S_1, S_2, disjunction) \end{cases}$$

$$\frac{E_1 < E_2 \to Ineq(x, a, -sign)}{E_1 > E_2 \to Ineq(x, a, sign)} \qquad \frac{E_1 < E_2 \to Ineq(x, a+1, sign)}{E_1 <= E_2 \to Ineq(x, a, -sign)}$$

$$\frac{E_1 < E_2 \to Ineq(x, a-1, sign)}{E_1 >= E_2 \to Ineq(x, a, -sign)} \qquad \frac{E_1 < E_2 \to Ineq(x, a, sign)}{E_1 === E_2 \to Ineq(x, a, 0)}$$

$$\frac{E_1 < E_2 \to Ineq(x, a, sign)}{E_1 ! == E_2 \to Neg(Ineq(x, a, 0))}$$

For logical symbols, we have to take into account short-circuiting. We do this by explicitly creating a branch symbol with the desired behaviour.

$$\frac{E_1 \to B_1 \qquad E_2 \to B_2}{E_1 \;\&\&\; E_2 \to Br(B_1, B_2, LV(false))} \qquad \frac{E_1 \to B_1 \qquad E_2 \to B_2}{E_1 \;||\; E_2 \to Br(B_1, LV(true), B_2)}$$

Lastly, we catch all other statements which do not fall under any of the previously defined rules under the following rule which simply produces a skip symbol.

$$\frac{E}{E \to Skip}\text{E does not have a rule}$$

Example 4.1 *Consider the function fac:*

```
function fac(x) {
  if(x===0) {
    return 1;
  } else {
    return x*fac(x-1);
  }
}
```

The symbolic executor will produce the following symbol tree:

```
Seq(
  Branch(
    Ineq(x,0),
    Seq(*),
    Seq(
      F(fac,N(x,-1))
    )
  )
)
```

4.2 Serializer

After the symbolic executor is run, we will obtain a tree of symbols. We need one more step to turn this tree into a transition set before we can analyze it.
We define a new function

$$\cdot \mid \cdot \mapsto \cdot : Id * Sym \to T$$

inductively as follows:

$$\frac{}{Sym \mapsto_f \{< f, *, \top, >\}} terminal(Sym)$$

$$\frac{}{F(g, N(x_1, c, sign), N(x_2, c, sign), ...) \mapsto_f \{< f, g, true, p >\}} p = "sign \cdot x_1 = c \wedge sign \cdot x_2 = c \wedge ..."$$

$$\frac{E_1 \mapsto_f \{< f, g_1, q_1, p_1 >, ...\} \quad E_2 \mapsto_f \{< f, g_{n+1}, q_{n+1}, p_{n+1} >, ...\}}{Br(C, E_1, E_2) \mapsto_f \{< f, g_1, q_1 \wedge C, p_1 >, ...\} \cup \{< f, g_{n+1}, q_{n+1} \wedge \neg C, p_{n+1} >, ...\}}$$

$$\frac{E_1 \mapsto_f S_1 \quad ... \quad E_n \mapsto_f S_n}{Seq(E_1, ..., E_n) \mapsto_f S_1 \cup ... \cup S_n}$$

In practice, it will be convenient to re-use our symbols in our representation of transition sets, and we will do just that. Let $t =< F_1, F_2, B >$ be a representation of a transition, where

- F_1 is the function symbol of the caller function.
- F_2 is the function symbol of the callee function, or the termination symbol.
- B is either a boolean symbol, or a null value.

One can see that this representation is isomorphic to our previous definition of a transition, $< f, g, q, p >$ as f and g can be inferred from F_1 and F_2 respectively, q can be inferred from B, with the null value meaning $q = \top$ p can be inferred by comparing the arguments in F_1 and F_2.

Example 4.2 *Consider the output from the previous example:*

```
Seq(
  Branch(
    Ineq(x,0),
    Seq(*),
    Seq(
      F(fac,N(x,-1))
    )
  )
)
```

The Serializer will produce the following:

```
[
  { caller: F(fac, N(x, 0)),
    callee: F(fac, N(x, -1)),
    condition: Ineq(x, 0, -1)
  },
  { caller: F(fac, N(x, 0)),
    callee: F(fac, N(x, -1)),
    condition: Ineq(x, 0, 1)
```

```
  },
  { caller: F(fac, N(x, 0)),
    callee: *,
    condition: Ineq(x, 0, 0)
  },
]
```

Which corresponds to the transition set:

$$\begin{aligned} T = \{ & < fac, fac, x > 0, (x' = x - 1) > \\ & < fac, fac, x < 0, (x' = x - 1) > \\ & < fac, *, x = 0, \top > \\ \} & \end{aligned}$$

4.3 Analyzer

Now we are ready to use the transition sets to detect infinite loops. The idea is to:

1. Get a list of computations that result in an infinite loop.
2. Extract the start symbol from that computation.
3. Add an additional branch at the start of the original function which throws an error if the conditions in the start symbol are satisfied.

To do this, we will apply our theorems in section 3 to the transition set, and modify the AST of the program accordingly. We illustrate with an example:

Example 4.3 *For the factorial function,*

```
function fac(x) {
  if(x===0) {
    return 1;
  } else {
    return x*fac(x);
  }
}
```

We will insert a line to throw an error.

```
function fac(x) {
  if(x<0){error ("infinite loop...")}else{}
  if(x===0) {
    return 1;
  } else {
    return x*fac(x);
  }
}
```

Since we know exactly which theorem we used in proving the infinite loop, we can write descriptive error messages for the student.

Note that since we are editing the program at the AST level, we can freely change the SourceLocation of the nodes we add, so that the error message produced points to the line of the desired function call.

4.4 Proof of finiteness

We have one more problem we need to address before we can use our infinite loop detector, that is, whether the infinite loop detector itself will run into an infinite loop!

Theorem 4.1

The symbolic executor will terminate in finite time for finite inputs of code.
Proof: Firstly, note that for every rule in the symbolic executor being evaluated on statement E, the program will either

1. returns a symbol immediately, or
2. recursively call the symbolic executor again, but on code that has length strictly less than statement E.

If we assume that the symbolic executor goes into an infinite loop, by observation 2 above, this would imply that we started with an infinitely long input of code, which is a contradiction.

□

Proof that the serializer is finite can be obtained using a similar argument as the above theorem. The Analyzer also only takes time linear to the length of the transition sets, since the theorems we currently have only require a linear scan through the transition sets.

5 Further work

Currently, our work is limited by

1. **Theory**: Since the focus for this project was on Source 1, we did not develop any theory which would work for Source 2 and beyond. One idea for Source 2 would be to model lists by using their lengths as size functions using the *size change principle* in **[LJBA01]**.
2. **Lack of theorems**: The theorems in section 3 only cover a very small subset of infinite loops, and does not cover several commonly used patterns such as mutal recursion.
3. **Symbolic Executor is not expressive enough**: The symbolic executor is not able to turn many expressions into symbols. For instance, multiplication is not supported at all. Being able to represent more programs symbolically will allow us to prove more theorems about infinite loops on them. There is still a need to keep the symbolic executor simple though, so a delicate balance needs to be kept.
4. **Theorems in analyzer are hard to implement**: Manipulating transitions sets as they are represented in Typescript records gets quite cumbersome as the complexity of the theorems increase, and this will lead to many bugs.

Although there is much room for improvement, we would like to emphasize that this tool is meant to be an aid to students, and should not become a hinderence to them. Therefore maintaining performance and a 0 false positive rate should still be prioritized over new features.

5.1 More theorems

Another commonly encountered pattern is mutual recursion.

Example 5.1

We will briefly state an approach on how to detect infinite loops in the above example. Note that a typical computation for the above goes as follows:

$$(< even, x = 2 >, < odd, x = 1 >, < even, x = 0 >, *)$$

We will only encounter an infinite loop for negative values of x, similarly to the factorial function:

$$(< even, x = -1 >, < odd, x = -2 >, < even, x = -3 >, ...)$$

The idea on how to handle this is to split the computation up into 2 sequences, one for each function, i.e.

1. $$(< even, x = -1 >, < even, x = -3 >, ...)$$
2. $$(< odd, x = -2 >, < odd, x = -4 >, ...)$$

Notice that if we can prove that neither of the sequences terminate, the original sequence will also not terminate (since it will not contain the * symbol). We can also prove the nontermination of each sequence using the theorem on countdown functions.
However, implementing this involves a lot of work and book keeping, and unfortunately, we did not have time to complete this before the project deadline. The next section elaborates on how this can be improved.

5.2 Alternative implementation of analyzer

As mentioned previously, manually manipulating the transition sets is very cumbersome and will lead to bugs. It may be fruitful to use another programming language to implement the theorem checking, or even a domain specific language.
As a proof of concept, here's a quick (simplified) implementation of some of the above theorems (including mutual recursion) in Prolog.

```
% online demo: https://swish.swi-prolog.org/p/source-infloops.pl
:- use_module(library(clpfd)).

transition(even, term, X, _) :- X #= 0.
transition(even, odd, X, Y) :- X #\= 0, Y #= X - 1.

transition(odd, term, X, _) :- X #= 0.
transition(odd, even, X, Y) :- X #\= 0, Y #= X - 1.

transition(fac, term, X, _) :- X #= 0.
transition(fac, fac, X, Y) :- X #\= 0, Y #= X - 1.

transition(f, f, X, Y) :- Y #= X-1.

hasBaseCase(X) :- transition(X,Y,_,_), X\=Y.
noBaseCase(X) :- \+ (hasBaseCase(X)).

infCountdown(X, Y) :- transition(X, term, C1,
                      transition(X, X, Z, 0), Z #> C1, Y #< C1
                      \+ (transition(X, term, Y, _)).

infCountdown(X, Y) :- transition(X, term, C1, _),
                      transition(X, X, Z, 0), Z #< C1, Y #> C1,
                      \+ (transition(X, term, Y, _)).

nStepTransition(1, X, Y, Z, W) :- transition(X, Y, Z, W).
nStepTransition(N, X, Y, Z, W) :- N \= 1, M is N-1,
                                  nStepTransition(M, X, A, Z, B),
                                  transition(A, Y, B, W).

infCountdown2(X, Y) :- transition(X, term, C1, _),
                       nStepTransition(2, X, X, Z, 0), Z #> C1, Y #< C1,
                       \+ (transition(X, term, Y, _)),
                       \+ nStepTransition(2, X, term, Y, 0).

% ?- noBaseCase(f).           % true
% ?- infCountdown(fac, X).    % X in inf .. -1
% ?- infCountdown2(even, X).  % X in inf .. -1
```

As shown above, constraint logic programming finds a good fit in this project.

5.3 Imperative programming

Another improvement will be to provide support for Source 3. This is very difficult as having impure functions will complicate the theory.

6 References

[CS02] Michael Cólon and Henny Sipma. Practical methods for proving pro-gram termination. In *Proceedings of the 14th International Confer-ence on Computer Aided Verification*, CAV 02, page 442454, Berlin,Heidelberg, 2002. Springer-Verlag.
[LJBA01] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. *SIGPLAN Not.*,36(3):8192, January 2001.