InterprocStack analyzer for recursive programs with finite-type and numerical variables

Bertrand Jeannet

Contents

1 Invoking InterprocStack

You can try an online version there.

The executable (`interprocstack` or `interprocstack.opt` is invoked as follows:

```
interprocstack <options> <inputfile>
```

The input file should be a valid "Simple" program. The options are:

`-debug <int>` debug level, from 0 (lowest) to 4 (highest). Default is 0.

`-domain {box|octagon|polka|polkastrict|polkaeq|ppl|pplstrict|pplgrid|polkagrid}`
abstract domain to use (default: polka). All domains supported by the Apron library can be specified:

box	intervals
octagon	octagons
polka,ppl	topologically closed convex polyhedra
polkastrict,pplstrict	possibly non-closed convex polyhedra
polkaeq	linear equalities
pplgrid	linear congruences
polkagrid	reduced product of linear congruences and Polka convex polyhedra

`-depth <int>` depth of recursive iterations (default 2, may only be more). See Fixpoint library documentation.

`-guided <bool>` if true, guided analysis of Gopand and Reps (default: false). See Analyzer

`-widening <bool><int><int><int>` specifies usage of widening first heuristics, delay and frequency of widening, and nb. of descending steps (default: false 1 1 2). See Fixpoint

`-analysis <('f'|'b')+>` sequence of forward and backward analyses to perform (default "f")

2 The "Simple" language

Porgram Sample:

```
/* Procedure definition */
proc MC(b:bool,n:int) returns (r:int)
var t1:int, t2:int, b1:bool;
begin
  if (b != (n>100)) then fail; endif;
  if b then
     r = n-10;
  else
     t1 = n + 11;
     b1 = t1>100;
     t2 = MC1(b1,t1);
     b1 = t2>100;
     r = MC1(b1,t2);
  endif;
end
/* Main procedure */
var x:int, y:int,b:bool;
begin
  b = x>100;
  y = MC(b,x);
end
```

2.1 Syntax and informal semantics

2.1.1 Program.

<program>	::=	[typedef <type_def>$^+$]	enumerated types definition
		<procedure_def>*	definition of procedures
		[var <vars_decl> ;]	local variables of the main procedure
		begin <statement>$^+$ end	body of the main procedure
<type_def>	::=	id = enum { id (, id)* } ;	

<procedure_def>	::=	proc id (<vars_decl>) returns (<vars_decl>)	signature, with names of input and output formal parameter
		[var <vars_decl> ;]	(other) local variables
		begin <statement>$^+$ end	body

A procedure definition first declares a list of formal input parameters, then a list of formal output parameters (which are local variables that defines the tuple of returned values), then a list of local variables, and then a body. A procedure ends with its last statement.

<vars_decl>	::=	ϵ | id : <type> (, id : <type>)*	
<type>	::=	bool	Boolean type
	|	uint[<integer>]	unsigned integers encoded with <integer> bits
	|	sint[<integer>]	signed integers encoded with <integer> bits
	|	id	Name of an enumerated type
	|	int | real	numerical types

Variables are either of finite-type, or unbounded integer or real type. Finite types includes Boolean, bounded integers encoded with the given numbe rof bits, and enumerated types. Real variables can behave as floating-point variables by modifying the exact semantics of arithmetic operations (see below).

2.1.2 Instructions.

<statement>	::=	skip ;
	|	halt ;
	|	fail ;
	|	assume <expr> ;
	|	id = random ;
	|	id = <expr> ;
	|	id = id (<var_list>) ;
	|	(<var_list>) = id (<var_list>) ;
	|	if <bexpr> then <statement>$^+$ [else <statement>$^+$] endif ;
	|	while <bexpr> do <statement>$^+$ done ;
<var_list>	::=	ϵ | id (, id)*

The `skip` operation does nothing. The `halt` operation halts the program execution. The `fail` has exactly the same operational semantics, but, in addition, indicates a final control point for backward analysis (see below).

The `assume expr` instruction is equivalent to `if expr then skip; else halt;` . It may be used to abstract non-deterministic assignements. For instance, the non-deterministic assignement of a value between 0 and 2 to a variable `x` can be written as

```
x = random;
assume x>=0 and x<=2;
```

A call to a procedure with a single return value can be written either as `x = P(...)` or `(x)=P(...)`. A call to a procedure with multiple return values is written as `(x,y) = P(a,b)`. Input parameters are passed by value. Formal input parameters in a procedure are assumed immutable.

Conditional and iteration constructions have the usual semantics.

2.1.3 Expressions.

Expressions.

<expr>	::=	random	random value (type deduced from the context)
	|	<bexpr>	Boolean expression
	|	<iexpr>	bounded integer expression
	|	<eexpr>	enumerated type expression
	|	<nexpr>	numerical expression

Boolean expressions.

```
<bexpr>      ::=  true | false
                | id                                          variable
                | <constraint>
                | not <bexpr>
                | <bexpr> (or | and) <bexpr>
                | ( <bexpr> )
                | if <bexpr> then <bexpr> else <bexpr>
<constraint> ::=  id == <expr>
                  <iexpr> (>= | > | <= | <) <iexpr>
                  <nexpr> (>= | > | <= | <) <nexpr>
```

Atoms of Boolean expressions are Boolean constants or equality constraints (on any type) or inequality constraints over bounded integer or numerical variables.

Bounded integer and enumerated expressions.

```
<iexpr>  ::=  uint[<integer>](<integer>)      constant (second integer) of given type
            | sint[<integer>](<integer>)      constant (second integer) of given type
            | id                              variable
            | <iexpr> (+|-|*) <iexpr>
            | ( <iexpr> )
            | if <bexpr> then <iexpr> else <iexpr>
<eexpr>  ::=  id                              variable or constant (label)
            | ( <eexpr> )
            | if <bexpr> then <eexpr> else <eexpr>
```

Example:

```
/* Illustrate enumerated and bounded integer types */
typedef t = enum  a,b,c,d,e ;

var x:sint[4],y:sint[4], z:t;

begin
  x = sint[4](2);
  y = random;
  z = a;
  if x==y then
    x = x + x;
    x = x-sint[4](3)*y;
    z = b;
  endif;
  if x<=y then
    z = e;
  endif;
  if z==d then z=c; endif;
end
```

Numerical expressions.

```
<nexpr>   ::=  <coeff>
            |  id                                              variable
            |  <unop> <nexpr>   |  <nexpr> <binop> <nexpr>
            |  ( <nexpr> )
            |  if <bexpr> then <nexpr> else nexpr
<binop>   ::=  (+|-|*|/|%)[_(i|f|d|l|q)[,(n|0|+oo|-oo)]]
<unop>    ::=  -
            |  (cast|sqrt)[_(i|f|d|l|q)[,(n|0|+oo|-oo)]]
<coeff>   ::=  <float>   |  <rational>
<float>   ::=  C/OCaml floating-point number syntax
<rational>::=  <integer>   |  <integer>/<integer>
```

The syntax of numerical expressions is the same as the one used in the APRON interactive toplevel (in OCaml language).

The priorities between (Boolean and arithmetic) operators is **{** `*`,`/` **}** > **{** `+`,`-` **}** > **{** `<`, `<=`, `=`, `>=`,`>` **}** > **{ not }** > **{ and }** > **{ or }**

By default, numerical expressions are evaluated using exact arithmetic on real numbers (even if some involved variables are integer), and when assigned to a variable, filtered by the variable domain (when it is integers). For instance, the assignement

```
x = 3/4;
```

is equivalent to the `halt` instruction if `x` is declared as integer, because the rational 3/4 is not an integer value. The problem here is that the instruction is not well-typed (even if this is not checked). If one want to emulate the semantics of the integer division operation, with rounding to zero, one should write

```
x = 3 /_i,0 4;
```

Indeed, most arithmetic operators have optional qualifiers that allows to modify their semantics. The first qualifier indicates a numerical type, the second one the rounding mode.

- By default, operations have an exact arithmetic semantics in the real numbers (even if involved variables are of integer).

  The type qualifiers modify this default semantics. Their meaning is as follows:

i	integer semantics
f	IEEE754 32 bits floating-point semantics
d	IEEE754 64 bits floating-point semantics
l	IEEE754 80 bits extended floating-point semantics
q	IEEE754 128 bits floating-point semantics

- By default, the rounding mode is “any” (this applies only in non-real semantics), which allows to emulate all the following rounding modes:

n	nearest
0	towards zero
+oo	towards infinity
-oo	towards minus infinity

Examples of numerical expressions.

We illustrate the syntax and (approximated) semantics of numerival expressions with the following program, annotated by InterprocStack using the command

```
interprocstack.opt -domain polka -colorize false examples/numerical.spl
```

```
proc integer () returns () var a : int, b : int, c : int;
begin
  /* (L5 C5) top */
  a = 5; /* (L6 C8) [|a-5=0|] */
  b = 2; /* (L7 C8) [|b-2=0; a-5=0|] */
  if brandom then
     /* (L8 C17) [|b-2=0; a-5=0|] */
     c = a / b; /* (L9 C14) bottom */
  else
    /* (L10 C6) [|b-2=0; a-5=0|] */
    c = a /_i,? b; /* (L11 C16) [|b-2=0; a-5=0; -c+3>=0; c-2>=0|] */
    c = a /_i,0 b; /* (L12 C18) [|c-2=0; b-2=0; a-5=0|] */
    c = a /_i,-oo b; /* (L13 C20) [|c-2=0; b-2=0; a-5=0|] */
    c = a /_i,+oo b; /* (L14 C20) [|c-3=0; b-2=0; a-5=0|] */
    c = a /_i,n b; /* (L15 C18) [|b-2=0; a-5=0; -c+3>=0; c-2>=0|] */
    c = a %_i,? b; /* (L16 C16) [|c-1=0; b-2=0; a-5=0|] */
  endif; /* (L17 C8) [|c-1=0; b-2=0; a-5=0|] */
end

proc exact () returns (z : real) var x : real, y : real;
begin
  /* (L22 C5) top */
  x = 5; /* (L23 C8) [|x-5=0|] */
  y = 2; /* (L24 C8) [|y-2=0; x-5=0|] */
  z = x / y; /* (L25 C10) [|2z-5=0; y-2=0; x-5=0|] */
  y = 0.1; /* (L26 C10)
              [|2z-5=0; 3602879701896396 8y-3602879701896397=0; x-5=0|] */
  z = x + y; /* (L27 C10)
                [|3602879701896396 8z-1837468647967162 37=0;
                  3602879701896396 8y-3602879701896397=0; x-5=0|] */
  z = z - y; /* (L28 C10)
                [|z-5=0; 3602879701896396 8y-3602879701896397=0; x-5=0|] */
end

proc floating () returns (z : real) var x : real, y : real;
begin
  /* (L33 C5) top */
  x = 5; /* (L34 C8) [|x-5=0|] */
  y = 2; /* (L35 C8) [|y-2=0; x-5=0|] */
  z = x /_f,? y; /* (L36 C13)
                    [|y-2=0; x-5=0;
                      -7136238463529799405291429847247475681913733 12z
                      +1784059828558929176909397126420998565333565441>=0;
                      7136238463529799405291429847247475681913733 12z
                      -1784059403205970525736317797202739275623301119>=0|] */
  y = 0.1; /* (L37 C10)
              [|3602879701896396 8y-3602879701896397=0; x-5=0;
                -7136238463529799405291429847247475681913733 12z
                +1784059828558929176909397126420998565333565441>=0;
                7136238463529799405291429847247475681913733 12z
                -1784059403205970525736317797202739275623301119>=0|] */
  z = x +_f,? y; /* (L38 C14)
                    [|3602879701896396 8y-3602879701896397=0; x-5=0;
                      -7136238463529799405291429847247475681913733 12z
```

```
                    +3639482050260215524856578735848702239922192385>=0;
                    7136238463529799405291429847247475681913733l2z
                    -3639481182540179876463495959770156714984210431>=0|] */
  z = z -_f,? y; /* (L39 C14)
                  [|36028797018963968y-3602879701896397=0; x-5=0;
                    -5986310706507378352962293074805895248510699696029696z
                    +2993156088286294306052268890496798408637685236865433 7>=0;
                    5986310706507378352962293074805895248510699696029696z
                    -2993154618221170818913589023617374447727566952962457 7>=0|] */
end

var z : real
begin
  /* (L43 C5) top */
  () = integer(); /* (L44 C17) top */
  z = exact(); /* (L45 C14) [z-5=0|] */
  z = floating(); /* (L46 C17)
                   [|-5986310706507378352962293074805895248510699696029696z
                     +2993156088286294306052268890496798408637685236865433 7>=0;
                     5986310706507378352962293074805895248510699696029696z
                     -2993154618221170818913589023617374447727566952962457 7>=0|] */
end
```

2.2 Formal semantics (to be updated)

We give here the standard operational semantics of the "Simple" language.

2.2.1 Semantics domains.

They are summarized by the following table:

$\boldsymbol{k} \in \boldsymbol{K}$	program's control points
$\boldsymbol{\epsilon} \in \mathbf{Env} = \mathbf{Var} \to \mathbb{R}$	environments
$(\boldsymbol{k}, \boldsymbol{\epsilon}) \in \boldsymbol{K} \times \mathbf{Env}$	activation records
$\boldsymbol{s} \in \boldsymbol{S} = (\boldsymbol{K} \times \mathbf{Env})^+$	program states, which are non-empty sta

An environment is defined by a function assigning a value to a set of variables **Var** visible in the current scope.

A program state is a non-empty stack of activation records. A stack is viewed as a word where the last letter corresponds to the top of the stack.

2.2.2 Semantics of expressions.

The semantics of a numerical expression `nexpr` is a function

$$[\![\texttt{nexpr}]\!]^{\boldsymbol{n}} : \mathbf{Env} \to \wp(\mathbb{R})$$

We refer to [1] for the precise definition of this semantic function.

The semantics of a Boolean expression `bexpr` is a function

$$[\![\texttt{bexpr}]\!]^{\boldsymbol{b}} : \mathbf{Env} \to \wp(\mathbb{B})$$

which is defined inductively as follows:

$$\begin{array}{rcl}
[\![\texttt{true}]\!]^b & = & \lambda\epsilon\,.\,\{true\} \\
[\![\texttt{false}]\!]^b & = & \lambda\epsilon\,.\,\{false\} \\
[\![\texttt{brandom}]\!]^b & = & \lambda\epsilon\,.\,\{true, false\} \\
[\![\texttt{not } e]\!]^b & = & \lambda\epsilon\,.\,\{\neg b \mid b \in [\![e]\!]^b(\epsilon)\} \\
[\![e_1 \texttt{ and } e_2]\!]^b & = & \lambda\epsilon\,.\,\{b_1 \wedge b_2 \mid b_1 \in [\![e_1]\!]^b(\epsilon) \wedge b_2 \in [\![e_2]\!]^b(\epsilon)\} \\
[\![e_1 \texttt{ or } e_2]\!]^b & = & \lambda\epsilon\,.\,\{b_1 \vee b_2 \mid b_1 \in [\![e_1]\!]^b(\epsilon) \wedge b_2 \in [\![e_2]\!]^b(\epsilon)\} \\
[\![e_1 \texttt{ op } e_2]\!]^b & = & [\![(e_1 - e_2) \texttt{ op } 0]\!]^b \\
[\![e = 0]\!]^b & = & \lambda\epsilon\,.\,(0 \in [\![e]\!]^n(\epsilon)) \\
[\![e \geq 0]\!]^b & = & \lambda\epsilon\,.\,([\![e]\!]^n(\epsilon) \cap [0, \infty[\neq \emptyset) \\
[\![e \leq 0]\!]^b & = & \lambda\epsilon\,.\,([\![e]\!]^n(\epsilon) \cap]-\infty, 0] \neq \emptyset) \\
[\![e > 0]\!]^b & = & \lambda\epsilon\,.\,([\![e]\!]^n(\epsilon) \cap]0, \infty] \neq \emptyset) \\
[\![e < 0]\!]^b & = & \lambda\epsilon\,.\,([\![e]\!]^n(\epsilon) \cap]-\infty, 0[\neq \emptyset)
\end{array}$$

The semantics of a Boolean expression can be extended as a predicate on environments as follows:

$$\begin{array}{rcccl}
[\![\texttt{bexpr}]\!]^b & : & \mathbf{Env} & \rightarrow & \mathbb{B} \\
 & & \epsilon & \mapsto & (true \in [\![\texttt{bexpr}]\!]^b)
\end{array}$$

2.2.3 Semantics of programs

We define the semantics of a program as a discrete dynamical system $(S, S_0, \rightarrow)$ where S is the state-space of the program (see above), S_0 is the set of initial states, and $\rightarrow$ is the transition relation between states.

If k_0 denotes the starting point of the main procedure, then S_0 is defined as

$$S_0 = \{k_0\} \times \mathbf{Env}$$

This reflects the hypothesis that variables are uninitialized (read: they may contain arbitrary values) at start.

The transition relation is defined as follows:

$$\frac{(k)\ \texttt{skip}\ (k')}{\omega \cdot (k, \epsilon) \rightarrow \omega \cdot (k', \epsilon)} \qquad \frac{(k)\ \texttt{assume}\ bexpr\ (k') \quad true \in [\![bexpr]\!]^b(\epsilon)}{\omega \cdot (k, \epsilon) \rightarrow \omega \cdot (k', \epsilon)}$$

$$\frac{(k)\ \texttt{x=random}\ (k') \quad \forall y \in \mathbf{Var} \setminus \{x\} : \epsilon'(y) = \epsilon(y)}{\omega \cdot (k, \epsilon) \rightarrow \omega \cdot (k', \epsilon')}$$

$$\frac{(k)\ \texttt{x=nexpr}\ (k') \quad v \in [\![nexpr]\!]^n(\epsilon)}{\omega \cdot (k, \epsilon) \rightarrow \omega \cdot (k', \epsilon[x \mapsto v])}$$

$$\frac{(k)\ \texttt{if bexpr then}\ (k_1) \ldots (k_1')\ \texttt{else}\ (k_2) \ldots (k_2')\ ;\ (k')}{\begin{array}{c}(k, \epsilon) \rightarrow (k_1, \epsilon)\ \texttt{if}\ true \in [\![bexpr]\!]^b(\epsilon) \\ (k, \epsilon) \rightarrow (k_2, \epsilon)\ \texttt{if}\ false \in [\![bexpr]\!]^b(\epsilon) \\ (k_1', \epsilon) \rightarrow (k', \epsilon) \\ (k_2', \epsilon) \rightarrow (k', \epsilon)\end{array}}$$

$$\frac{(k)\ \texttt{while bexpr do}\ (k_1) \ldots (k_1')\ \texttt{done}\ (k_2)}{\begin{array}{c}(k, \epsilon) \rightarrow (k_1, \epsilon)\ \texttt{if}\ true \in [\![bexpr]\!]^b(\epsilon) \\ (k, \epsilon) \rightarrow (k_2, \epsilon)\ \texttt{if}\ false \in [\![bexpr]\!]^b(\epsilon) \\ (k_1', \epsilon) \rightarrow (k, \epsilon)\end{array}}$$

$$\frac{\begin{array}{c}(k)\ \texttt{(y0,...,yN)=f(x0,...xM)}\ (k')\\ \epsilon_{\mathbf{start}}(\vec{pin}) = \epsilon(\vec{x})\\ \epsilon' = \epsilon[\vec{y} \mapsto \epsilon_{\mathbf{exit}}(\vec{pout})\end{array}}{\begin{array}{c}\omega \cdot (k, \epsilon) \longrightarrow \omega \cdot (k', \epsilon) \cdot (k^f_{\mathbf{start}}, \epsilon_{\mathbf{start}})\\ \omega \cdot (k', \epsilon) \cdot (k^f_{\mathbf{exit}}, \epsilon_{\mathbf{exit}}) \longrightarrow \omega \cdot (k', \epsilon')\end{array}}$$

3 What InterprocStack does for you

InterprocStack takes as input a "Simple" program, annotates it with control points, perform forward and/or backward analysis, and print the annotated programs with invariants associated with each control point, like this:

```
Annotated program after forward analysis:
var x : int, y : int, z : int
begin
  /* (L5 C5) top */
  assume z >= 10 and z <= 20; /* (L6 C25) [|-z+20>=0; z-10>=0|] */
  x = 0; /* (L7 C8) [|x=0; -z+20>=0; z-10>=0|] */
  y = 0; /* (L7 C13) [|-3x+y=0; -x+z+1>=0; -z+20>=0; z-10>=0; x>=0|] */
  while x <= z do
    /* (L8 C17) [|-3x+y=0; -x+z>=0; -z+20>=0; z-10>=0; x>=0|] */
    x = x + 1; /* (L9 C12)
                  [|-3x+y+3=0; -x+z+1>=0; -z+20>=0; z-10>=0; x-1>=0|] */
    y = y + 3; /* (L10 C12) [|-3x+y=0; -x+z+1>=0; -z+20>=0; z-10>=0; x-1>=0|] */
  done; /* (L11 C7) [|-3x+y=0; -x+z+1=0; -x+21>=0; x-11>=0|] */
  if y >= 42 then
    /* (L12 C15) [|-3x+y=0; -x+z+1=0; -x+21>=0; x-14>=0|] */
    fail; /* (L13 C9) bottom */
  endif; /* (L14 C8) [|-3x+y=0; -x+z+1=0; -3x+41>=0; x-11>=0|] */
end
```

Invariants

An *invariant* $\boldsymbol{I}$ is a set of environments:

$$\boldsymbol{I} \in \wp(\mathbf{Env})$$

The result of the analyses performed by InterprocStack is the projection $\boldsymbol{p(X)}$ of sets of stacks on their top elements. The result of this projection is a function that maps control point to invariants:

$$\begin{array}{rcl} \boldsymbol{p} : \wp((\boldsymbol{K} \times \mathbf{Env})^+) & \longrightarrow & (\boldsymbol{K} \to \wp(\mathbf{Env}) \\ \boldsymbol{X} & \longmapsto & \boldsymbol{\lambda k} \,.\, \{\epsilon \mid \omega \cdot (k, \epsilon) \in \boldsymbol{X}\} \end{array}$$

Intuitively, the invariant on numerical variables associated with each program point, as returned by InterprocStack, indicates the possible values for these variables at each program point, for any stack tail.

Single forward/reachability analysis

A forward analysis compute the set of reachable states of the program, which is defined as:

$$\boldsymbol{Reach} = \{s \in \boldsymbol{S} \mid s_0 \in \boldsymbol{S_0} \wedge s_0 \to^* s\}$$

InterprocStack returns an overapproximation of $\boldsymbol{p(Reach)}$.

Single backward/coreachability analysis

A backward analysis computes the set of states from which one can reach a `fail` instruction.

The set of states coreachable from a `fail` instruction is defined as:

$$\boldsymbol{Coreach} = \{s \in \boldsymbol{S} \mid s \rightarrow^* s_f \wedge s_f \in \boldsymbol{F}\}$$

with the set of final states

$$\boldsymbol{F} = \{\omega \cdot (k, \epsilon) \mid k \in \boldsymbol{K}_{\mathbf{fail}} \wedge \epsilon \in \mathbf{Env}\}$$

where $\boldsymbol{K}_{\mathbf{fail}}$ is the set of control points just preceding `fail` instructions. InterprocStack returns an overapproximation of $\boldsymbol{p}(\boldsymbol{Coreach})$.

Combined forward/backward analyses

InterprocStack allows to combine forward and backward analysis any number of time.

If $\boldsymbol{X}$ denotes the invariant computed by InterprocStack at the previous step, at the current step:

- a forward analysis will consider the set of initial states

$$\boldsymbol{S_0} \cap \boldsymbol{p}^{-1}(\boldsymbol{X})$$

- a backward analysis will consider the set of final states

$$\boldsymbol{F} \cap \boldsymbol{p}^{-1}(\boldsymbol{X})$$

For instance, after executing the command

```
interprocstack -analysis fb essai.spl
```

on the previous program, we get

```
Annotated program after forward analysis
var x : int, y : int, z : int
begin
  /* (L5 C5) top */
  assume z >= 10 and z <= 20; /* (L6 C25) [|-z+20>=0; z-10>=0|] */
  x = 0; /* (L7 C8) [|x=0; -z+20>=0; z-10>=0|] */
  y = 0; /* (L7 C13) [|-3x+y=0; -x+z+1>=0; -z+20>=0; z-10>=0; x>=0|] */
  while x <= z do
    /* (L8 C17) [|-3x+y=0; -x+z>=0; -z+20>=0; z-10>=0; x>=0|] */
    x = x + 1; /* (L9 C12)
                  [|-3x+y+3=0; -x+z+1>=0; -z+20>=0; z-10>=0; x-1>=0|] */
    y = y + 3; /* (L10 C12) [|-3x+y=0; -x+z+1>=0; -z+20>=0; z-10>=0; x-1>=0|] */
  done; /* (L11 C7) [|-3x+y=0; -x+z+1=0; -x+21>=0; x-11>=0|] */
  if y >= 42 then
    /* (L12 C15) [|-3x+y=0; -x+z+1=0; -x+21>=0; x-14>=0|] */
    fail; /* (L13 C9) bottom */
  endif; /* (L14 C8) [|-3x+y=0; -x+z+1=0; -3x+41>=0; x-11>=0|] */
end

Annotated program after backward analysis
var x : int, y : int, z : int
begin
  /* (L5 C5) [|-z+20>=0; z-13>=0|] */
  assume z >= 10 and z <= 20; /* (L6 C25) [|-z+20>=0; z-13>=0|] */
```

```
  x = 0; /* (L7 C8) [|x=0; -z+20>=0; z-13>=0|] */
  y = 0; /* (L7 C13) [|-3x+y=0; -x+z+1>=0; -z+20>=0; z-13>=0; x>=0|] */
  while x <= z do
    /* (L8 C17) [|-3x+y=0; -x+z>=0; -z+20>=0; z-13>=0; x>=0|] */
    x = x + 1; /* (L9 C12)
                  [|-3x+y+3=0; -x+z+1>=0; -z+20>=0; z-13>=0; x-1>=0|] */
    y = y + 3; /* (L10 C12) [|-3x+y=0; -x+z+1>=0; -z+20>=0; z-13>=0; x-1>=0|] */
  done; /* (L11 C7) [|-3x+y=0; -x+z+1=0; -x+21>=0; x-14>=0|] */
  if y >= 42 then
    /* (L12 C15) [|-3x+y=0; -x+z+1=0; -x+21>=0; x-14>=0|] */
    fail; /* (L13 C9) bottom */
  endif; /* (L14 C8) bottom */
end
```

References

[1] A. Miné. *Domaines numériques abstraits faiblement relationnels (Weakly relational numerical abstract domains)*. PhD thesis, École Normale Supérieure de Paris, 2004. PDF.