0368.4162 Program Analysis Mooly Sagiv

**Lecture 1 – An Overview to Program Analysis**

22 February 2011 Spring 2011

By Moria Abadi and Priel Levy

**1 Static Analysis**

**Definition 1.1**: Static analysis is process that operates on computer program and automatically infers static properties which hold on every execution. Dynamic program analysis, on the other hand, automatically infers properties of the program while it is being executed.

**Example 1.1:**

Consider in the following C code. We want to find variables with constant value at a given program location:

The compiler doesn’t know the value that is given by getc(), so it can’t know if the if statement is being executed or the else one. But in both of the cases, and without dependency in the input, the value of z is 44 (62+8 = (-7)2-5) so in the last line of the program will be printed “44”.

*int p(int x){*

*return x \*x ;*

*}*

*void main()*

*{*

*int z;*

*if (getc())*

*z = p(6) + 8;*

*else z = p(-7) -5;*

*printf (z);*

*}*

**Example 1.2:** a bit more complicated:

*int x*

*void p(a) {*

*read (c);*

*if c > 0 {*

*a = a -2;*

*p(a);*

*a = a + 2;*

*}*

*x = -2 \* a + 5;*

*print (x);*

*}*

*void main {*

*p(7);*

*print(x);*

*}*

Apparently, the variable x gets different values throughout the program, due to the recursion of p with different values of a. But, in each time that a is subtracted by 2, it is then added to 2, so the following invariant holds: x = -2\*a0+5, where a0 is the value that was given to a in the function call. So, in the last line of the program, it holds that x = 2\*(-7)+5 = 9 and “9” is printed.

The previous codes are based on an iterative progression – starting from a good solution, and each step making more approximations till coming to fixed point in which the code is “stabilized”.

**2 Iterative Approximations**

Constant propagation program holds for tracking after possible program’s states accordingly to the control statements that exist.

**Definition 2.1:** A state in a program is mapping *f*: V-> D U {?}, where V is the set of all the variables in the program and D is the possible values for them. The “?” means that the value of variable is unknown at this specific location. We refer to “?” as the default value (the initial one). In order to approximate the static solution for the program, we go over the code, command after command, and infer the next state according to the preceding one.

**Example 2.1:**

*int z=3;*

*if (x ==1)*

*{*

*while (getc() != ‘\n’)*

*{*

*x = x+1;*

*}*

*}*

The suitable control flow graph(CFG):

1. Assigning: z = 3 2. Splitting: check all the possible paths

[x->?, y->?, z->?] [x->?, y->?, z->3]

Z = 3

If (x==1)

true false

[x->?, y->?, z->3] [x->1, y->?, z->3] [x->?, y->?, z->3

3. while loop: keep fixing the values, if needed:

true false

[x->1, y->?, z->3] [x->?, y->?, z->3]

while(getc() != ‘\n’)

while(getc() != ‘\n’)

[x->1, y->?, z->3] [x->?, y->?, z->3]

[x->1, y->?, z->3] [x->?, y->?, z->3]

x= x +1

x= x +1

[x->2, y->?, z->3] [x->?, y->?, z->3]

Thus, in this case, we get different values in the true label and the false label, therefore the value of x determined to be “?”.

**Example 2.2:** [x->?, y->?, z->?]

z = 3

[x->?, y->?, z->3]

While (x>0)

[x->?, y->?, z->3] , [x->?, y->?, z->3]

(Two cases – the while loop is either true or false)

true false

If(x==1)

[x->1, y->?, z->3] [x->?, y->?, z->3]

y = z+4

y=7

[x->1, y->7, z->3] [x->?, y->7, z->3]

assert y==7

[x->?, y->7, z->3] (Merging of the two states)

**3 Memory Leakage**

Algorithm that will be presented later will enable to find potential memory leaks. For example, the following program reverses a list. There is potential leakage of address pointed to by “head” that can be detected in compile time (even though the length of the list is unknown). If we change the order of “head=n”, “rev=head”, the analysis will soundly prove that the program does not include memory leaks:

**Example 3.1:**

*List reverse(Element \*head)*

*{*

*List rev, n;  
rev = NULL;*

*while (head != NULL) {  
 n = head →next;*

*head → next = rev;   
 head = n;*

*rev = head;*

*}  
return rev;*

*}*

Another example of program that searches till finding blank, in which there is potential buffer overrun:

**Example 3.2:**

*void foo(char \*s )*

*{*

*while ( \*s != ‘ ‘ ) //bug when s is NULL or there isn’t any blank .*

*s++;*

*\*s = 0;*

*}*

In order to avoid it, we have to make assumption on the function’s input. For instance, we can assume that s is a null terminates string so the code will be:

*void foo(char \*s) @require string(s)*

*{*

*while ( \*s != ‘ ‘&& \*s != 0)*

*s++;*

*\*s = 0;*

}

The @require is an indication to the algorithm to do 2 things:

1. To refer s as null terminated string.
2. To make sure that in any place that foo is called, this condition holds.

It can be proven at compile time that in this case there will be no buffer overruns.

**4 Live variables**

**Definition 4.1:** A variable is live at the exit from a label if there exists a path from the label to a use of the variable that does not re-define the variable.

**Example 4.1:**

There are points in the program that the variables a and c are simultaneously alive, and the variables b and c are simultaneously alive, but there are not any points in which the variables a and b are simultaneously alive This useful information help the compiler to allocate registers efficiency by reusing the same register for a and b.

The liveliness graph is:

/\* c\*/

*L0: a = 0*

/\* a, c\*/

*L1: b = a+1*

/\*b, c\*/

*c = c+b*

/\*b, c\*/

*a = a\*2*

/\*a, c\*/

*if (c< n) goto L1*

/\*c\*/

*return c*

* 1. **Other examples program analyses of static analysis**
* Reaching definitions: a reaching definition for a given instruction is another instruction, the target variable of which may reach the given instruction without an intervening assignment. For example, in the following code:

*L1: y := 3*

*L2: x := y*

L1 is a reaching definition at L2. In the following, example, however:

*L1 : y := 3*

*L2 : y := 4*

*L3 : x := y*

L1 is no longer a reaching definition at L3, because L2 kills its reach.

* Expressions that are ``available''
* Dead code: is code in the [source code](http://en.wikipedia.org/wiki/Source_code) of a program which is executed but whose result is never used in any other computation. Dead code analysis can be performed using [live variable analysis](http://en.wikipedia.org/wiki/Live_variable_analysis)
* Pointer variables never point into the same location
* Points in the program in which it is safe to free an object, based on live variable analysis.
* Statements that can be executed in parallel
* An access to a variable which must be in cache – we can use static analysis to know what is in the cache.

Integer intervals: The idea is given a path through a CFG and the associated collection of constraints it remains, decide about the solution strategies to be applied. For each variable symbol xwe consider interval valuation *I*(*x*) = [*x, x*]. The partial order *>* is given by the subset relation between intervals; the smallest and largest elements are *{}* (empty interval) and [*−∞, ∞*], respectively

* 1. **The Need for Static Analysis**

As the advance of architecture and the language, the need for static analysis becomes stronger:

* Compilers: advanced computer architectures, high level programming languages (functional, OO, garbage collected, concurrent).
* Software Productivity Tools: compile time debugging (stronger type checking for C, array bound violations, identify dangling pointers, generate test cases, and generate certification proofs).
* Program understanding
  1. **Challenges in Static Analysis**
* Correctness & precision: there are static analysis problems can be reduced the halting problem (undecidable), thus can’t have exact solution – soundness or completeness might be damaged.
* Efficiency & scaling: As there are exist many big programs, with millions lines of code, such program with exponential time can hardly be applicable. The analysis may yield good results with small programs, but isn’t useful if it is not scalable.

In C compilers, the language was designed to reduce the need for optimizations and static analysis and give the programmer full control over performance (order of evaluation, storage, registers). Nowadays, C compilers spend most of the compilation time in static analysis – which no programmer can do in logical time due to complexity and size of programs.

1. **Foundation of static analysis**

Static analysis can be viewed as interpreting the program over an “abstract domain”. The abstract domain represents sets of possible states in the program. The chosen of the abstraction depends on the property that is analyzed.

Static analysis guarantees sound results: every identified constant is indeed a constant but not every constant is identified as such. Namely, every optimization is correct, but it is not sure that this is the optimal.

**Example 5.1:** Abstract interpretation casting out nines

It is possible to divide the natural numbers into 9 equivalence classes - 0, 1, 2, 3, 4, 5, 6, 7, 8.

The division is as following: if n<9, his equivalence class is n, else his equivalence class is the sum of his digits (recursively). In this way, we can use an abstract calculator that operates on the equivalence classes of the numbers instead of their real values, and report an error if the values do not match (if the values do match, we cannot infer that the values are equal). Consider the following query: “123 \* 457 + 76543 = 132654$?” so left side is 123\*457 + 76543 = 6 \* 7 + 7 =6 + 7 = 4, right side is 3 -> report an error (as 4!=3).

The soundness is being kept due to algebra properties:   
(10a + b) mod 9 = (a + b) mod 9  
(a+b) mod 9 = (a mod 9) + (b mod 9)  
(a\*b) mod 9 = (a mod 9) \* (b mod 9)

This abstract calculator is soundness: if it reports an error, such error does exist, but it is possible that there is an error which the calculator will not report.**Example 5.2:** Even/Odd abstract interpretation

The aim is to determine if an integer variable is even or odd at a given program location.

Consider the following code:

*while (x !=1) do {*

*if (x %2) == 0{*

*x := x / 2;*

*}*

*else {*

*x := x \* 3 + 1;*

*assert (x %2 ==0);*

*}*

*}*

Let’s define abstract domain {O, E, ?}. “E” states for an even number, “O” for an odd number, and “?” states for unknown. Now, we can use algebra properties: if x%2=0, then x is even, else, it is odd. Moreover, product of two odd numbers is odd, and sum of two odd numbers is even.

x = ?

From this static analysis, we can conclude that:

-when reaching the assert statement, x is even, so the assertion is executed.

- if the program terminates, x is odd (because 1 is odd).

-If x is odd in the current iteration, it will be even in the next one, so we can add in the else statement the command x = x/2 and it is still equivalent to the original program.

While(x!=1)

x = ?

x= ?

If(x%2==0)

true false

x = E x = O

x = x/2

x = x\*3 + 1 +1+1+2

x= ? x = E

The static analysis can’t determine if the program halts. This is an open problem.

1. **Abstract interpretation**

**Definition 6.1:** Abstract interpretation is a general methodology for calculating analyses rather than just specifying them and then relying on a posteriori validation.

So abstraction is housing the concrete set in one smaller one. The other direction is called concretization.

Let L be the concrete domain of the concrete semantics. The idea behind the abstraction is to find a “small” domain M, which is description of the elements of L. Then instead of performing the analysis over L, the analysis over M is performed. To express the relationships between L and M it is customary to use abstraction function  : L -> M giving a representation of elements of L as elements of M and a concretization function  : M->L that expresses the meaning of elements of M in terms of elements of L. For this and have to be related somehow. The relation between and is called Galois Insertion and is demonstrated at example 6.1. The Galois Insertion is defined as (L,,,M) between (L, ) and (M,) if and only if

: L -> M and : M->L are monotone functions that satisfy:

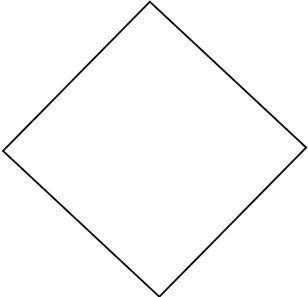
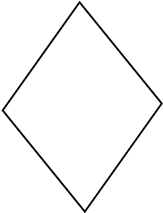
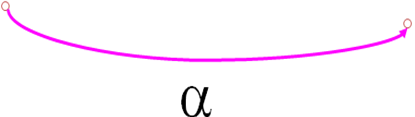
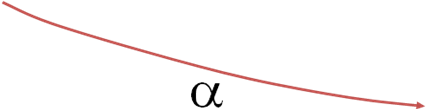
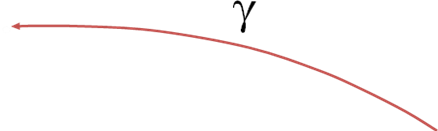
1. l.l
2. = l.l

.In addition the connection between L and M expresses that we do not lose safety by going back and forth between the two domains although we may lose precision. In the case of (1) this ensures that if we start with an element l in L we can first find description (l) of it in M and next determine which element ((l)) of L that describes (l). This need not to be l but it will be a safe approximation of l, i.e. l ((l)) . From (2) we can see there that the precision is not lost by first doing concretization and then abstraction. As a consequence M cannot contain elements that do not describe elements of L

**Example 6.1:** the abstraction/concretization model

Abstract

Concrete

 Sets of stores descriptors of sets of stores

Now let’s return to example 5.2, and present it according to our model: sets that contain only even numbers will be mapped to “E”, sets that contain only odd numbers will be mapped to “O”, and sets that contain either even numbers or odd ones will be mapped to “?”. This way, some information is lost, as they are different sets that are mapped to the same element (for example {1} and {3, 5, 7, 9, 11…}), but the model that obtained is much simpler.

For every command in a program, it is possible to find its abstract meaning by:

* Translation into the suitable concrete sets that it represents (concretization).
* Apply the operator on each element.
* Abstraction.

**6.1: abstract transformer**

Concrete Representation

Concrete Representation

Operational semantics

Concretization Abstraction

Abstract semantics

Abstract Representation

Abstract Representation

**Example 6.3:**  abstract interpretation of product and sum.

+’ ? O E \*’ ? O E

? ? ? ? ? ? ? E

O ? E O O ? O E

E ? O E E E E E

These abstract tables have only 3 values and conservatively summarize the multiplication and addition tables (that are infinity).

**Example 6.4:** rule of signs

The aim is safely identify the sign of variables at every program location.

The abstract representation is {P, N, ?} and defined by:

α (C) = P if all elements in C are positive

N if all elements in C are negative

? else

Concretization defined by the following function:

γ (a) = {0, 1, 2,…} if a = P

{-1, -2, -3,…} if a = N

Z (the set of integers) else (a = ?)

For example, abstract (conservative) semantics of \* is the following:

\*# P N ?

P P N ?

N N P ?

? ? ? ?

Now theabstract (conservative) interpretation is:

x:= x\*y

{…,<-88, -2>,.}

{…,<176, -2>…}

Operational semantics

Concretization Abstraction

Abstract semantics

<N, N>

<P, N>

x:= x\*#y

The concretization leads to sets that have infinity values on which we have to apply the operator. For instance, if there are two variables x, y, and their abstract representation is

<N, N> (both of them are negative) and the operator is x:= x\*y then:

|  |  |
| --- | --- |
| Before | After |
| <-1, -1> | <1, -1> |
| <-1, -2> | <2, -2> |
| <-1, -3> | <3, -3> |
| <-8, -60> | <480, -60> |
| <-80, -2> | <160, -2> |
| … | … |

In this case, it is easy to see that all the possibilities lead to a positive x. Now if we apply the abstraction, we get the new state <P, N>.

The abstract oes not necessarily create a homomorphism structure, for example, if the state is <P, N> and the operator is x:=x\*y, each concrete value that we choose will reach the state <N, P> or the state <P, P>, but applying the abstract operator will lead to the state <?, P>.

x:= x+y

<-1, 7>

<-8, 7>

Operational semantics

Concretization Abstraction

Abstract semantics

<N, P>

<?, P>

<N, P>

x:= x+y

In this conservative approach, we cannot miss states, namely, the set that created in the abstract path contain the set that created in the concrete path (in the above model, <N, P> <= <?, P>). However, it can consider superfluous states which cannot occur in any execution.

**Example 6.5:**

The compiler doesn’t know if the if statement is executed. But in both of the cases (if/else) the value of z is 12.

*x = 5;*

*y = 7;*

*if (getc())*

*y = x + 2;*

*z = x +y;*

**Example 6.6:** (more interesting)

The value of z in the last line is 5. But, according to the static analysis, its value is “?” as we merge the following states:

x->3, y->2, z->?

x->2, y->3, z->?

And get x->?, y->?, z->?

*if (getc())* /\* x->?, y->?, z->? \*/ *x= 3 ; y = 2;* /\* x->3, y->2, z->? \*/

*else*

*x =2; y = 3;* /\* x->2, y->3, z->? \*/

*z = x +y;*

1. **Undecidability Issues**

It is undecidable if a program point is reachable in some execution. Furthermore, some static analysis problems are undecidable even if the program conditions are ignored.

**Example 7.1**

*while (getc()) {*

*if (getc()) x\_1 = x\_1 + 1;*

*if (getc()) x\_2 = x\_2 + 1;*

*...*

*if (getc()) x\_n = x\_n + 1;*

*}*

*y = truncate (1/ (1 + p2(x\_1, x\_2, ..., x\_n))*

The problem “is y = 0 in the end of the program” is equivalent to the problem “is p has root over the natural numbers”, problem that known as the tenth problem of Hilbert and is undecidable.

**7.1 Coping with undecidabilty**

* Loop free programs
* Simple static properties
* Interactive solutions – the compiler may ask the programmer for his purpose.
* Conservative estimations:
  + Every enabled transformation cannot change the meaning of the code but some transformations are no enabled.
  + Non optimal code.
  + Every potential error is caught but some “false alarms” may be issued.

The constant propagation has analogies with numerical analysis: approximation the exact semantics, more precision can be obtained with greater approximations and computational costs.

**7.2 Optimality Criteria**

* Precise (with respect to a subset of the programs).
* Precise under the assumption that all paths are executable (statically exact).
* Relatively optimal with respect to the chosen abstract domain.

**7.3 Relation to Program Verification**

|  |  |
| --- | --- |
| **Program analysis** | **Program verification** |
| Fully automatic | Requires specification and loop invariants |
| Applicable to a programming language | Program specific |
| Can be very imprecise | Relative complete |
| May yield false alarms | Provide counter examples |
|  | Provide useful documentation |
|  | Can be mechanized using theorem provers. |

**7.4 Runtime vs. static testing:**

|  |  |  |
| --- | --- | --- |
|  | **Runtime** | **Abstract** |
| **input** | Need for the program’s input | No need for the program’s input |
| **Effectiveness** | Missed errors | False alarms  Locate rare errors |
| **cost** | Proportional to program’s execution | Proportional to program’s  size |

1. **Successful Tools of Static Analysis**

**8.1 SLAM**

The SLAM analysis engine forms the core of a new tool called Static Driver Veriﬁer (SDV) that systematically analyzes the source code of Windows device drivers against a set of rules that deﬁne what it means for a device driver to properly interact with the Windows operating system kernel.

The goal of the SLAM project is to automatically check that a program respects a set of safety properties of the interfaces it uses.

Safety properties are the class of properties that state that “something bad does not happen". An example is requiring that a lock is never released without first being acquired. Given a program and a safety property, the tool will validate that the code respects the property, or the tool will find an execution path that shows how the code violates the property.

In the following diagram we can see the general architecture of the system:



As shown in the above diagram, the essential points of the process, as implemented by SLAM, are: (1) the automated creation of a Boolean program abstraction of an instrumented C program that contains information relevant to the property under consideration; (2) model checking of the Boolean program to determine the absence or presence of errors; (3) the validation of a counterexample trace to determine whether or not it is a feasible trace of the C program. The last step can either produces a validated counterexample trace or a proof that the trace is invalid (a provably false alarm), in which case information is added to the abstraction to rule out the false alarm. In SLAM the Abstraction, Model Checker and Trace Validation are called C2BP, BEBOP and NEWOTON respectively.

**8.1.1 The main algorithm**

The main algorithm is demonstrated in the following diagram:

1. Run C2BP to construct Boolean Program BP

2. Run Bebop on BP. If no counterexample OK

3. If counterexample run Newton to check feasibility. If feasible, then BUG

4. If un-feasible, refine the abstraction with new predicates derived from the counterexample (the paths of counterexample are eliminated from consideration) and goto 1.

C program

SLIC rules

Boolean Program

Abstract Trace

Concrete Program Trace

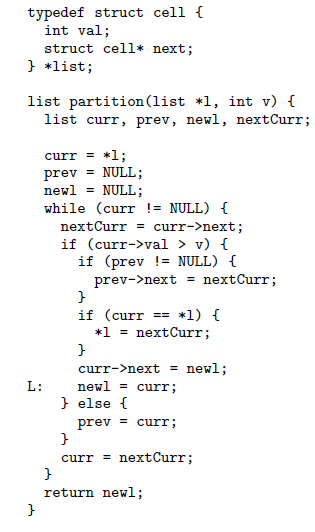
👍

**8.1.2 Some Characteristics of SLAM**

Since property checking is undecidable, the SLAM refinement algorithm may not converge. In addition, it may terminate with a “don't know" answer due to the incompleteness of the underlying theorem provers. However, experiments show that, it usually converges in a few iterations with a definite answer. One reason for this is that the properties checked are very control-intensive, and have relatively simple dependencies on data of the C program in the following sense: any feasible execution path of the C program is a feasible execution path of the Boolean program. Of course, there may be feasible execution paths of the Boolean program that are infeasible in the C program. Such paths can lead to imprecision in subsequent model checking.

**Example 8.1**

For the following program:



Bebop outputs the following invariant:



representing the reachable states at label L from a Boolean program generated from the above program (with some abstraction).

Because C2bp is sound, this Boolean function is also an invariant over the state of the C program at label L.

**8.2 Astrée**

Astrée is a static analyzer for large embedded synchronous safety-critical software. Astrée automatically computes supersets of the possible values in synchronous C programs at every program point. Thus, if Astrée does not report any bad behavior, it proves that no such behavior can happen whatever the inputs of the C program.

**8.2.1 Errors reported by Astrée**

Astrée can automatically report on a number of errors. The kind of errors which are currently reported by Astrée stems from the first end-user requirements. They wanted to see what could be proved without going through the expensive process of producing formal specifications. The least one can expect from a critical software is that the code never produces fatal errors, such as divisions by zero. Another common requirement is that the language is never used in cases where the result is stated as "undefined". For example, this is the case of out-of-bound array accesses, or integer overflows.

The errors which are reported are:

* out-of-bound array accesses
* integer division by zero
* floating point operations overflows and invalid operations (resulting in

IEEE floating values Inf and NaN)

* integer arithmetic wrap around behavior (occurring mainly in overflows)
* casts that result in wrap around operations (when the target type is too small to contain a value).

Once having a superset of the possible values of all program variables at each program point Astrée can use some user-defined known facts and report on arbitrary user defined assertions (written in C) on the software.

* + 1. **Some Characteristics of Astrée**

Astrée was developed to prove the absence of run-time errors for a specific class of synchronous C programs. As expected, it will be quite efficient and precise on the difficulties raised by this class of programs and may be weak on other aspects of the language. One restriction of the class of C programs for which Astrée was designed is that it does not contain any dynamic memory allocation, string manipulation and very restricted pointers. That allows for a fast and precise memory analysis which would not be possible otherwise.

On the other hand, the class of analyzed C programs contains large programs (hundreds of thousands of lines of code), with a huge number of global interdependent variables (about 10000 for a 100,000 lines program). This makes it hard to be efficient and precise, and specific algorithms and heuristics have been developed to keep the complexity of Astrée low (not far above linear in the number of lines of codes and the number of global variables).

As is necessary for many critical software, Astrée deals well with complex control using thousands of boolean variables. In addition, Astrée makes a sound analysis of floating values computations (as described in IEEE Computer Society, 1985), taking into account all possible rounding errors.

ASTRÉE was able to prove completely automatically the absence of any run time errors in the primary flight control software of the Airbus A340 fly-by-wire system, a program of 132,000 lines of C.

**8.2.3 Examples of Astrée analysis**

Example 1

*/\* boolean.c \*/*

*typedef enum {FALSE = 0, TRUE = 1} BOOLEAN;*

*BOOLEAN B;*

*void main () {*

*unsigned int X, Y;*

*while (1) {*

*/\* ... \*/*

*B = (X == 0);*

*/\* ... \*/*

*if (!B) {*

*Y = 1 / X; // No alarm reported here*

*};*

*/\* ... \*/*

*};}*

Yields no warning (thanks to the relationship automatically determined between *B* and *X*), thus proving the absence of any run-time error (integer divide-by-zero can never happen when executing this program).

Example 2

*/\* float-error.c \*/*

*void main ()*

*{ float x, y, z, r;*

*x = 1.000000019e+38;*

*y = x + 1.0e21;*

*z = x - 1.0e21;*

*r = y - z;*

*printf("%f\n", r);*

*}*

One would expect these programs to print 2.0e21 and 2.0, respectively, based on the reasoning that (x+a)-(x-a) = 2a. However, this reasoning turns out to be wrong due to rounding errors.

Example 3

*/\* float.c \*/*

*void main () {*

*float x,y,z;*

*if ((x < -1.0e38) || (x > 1.0e38)) return;*

*while (1) {*

*y = x+1.0e21;*

*z = x-1.0e21;*

*x = y-z;*

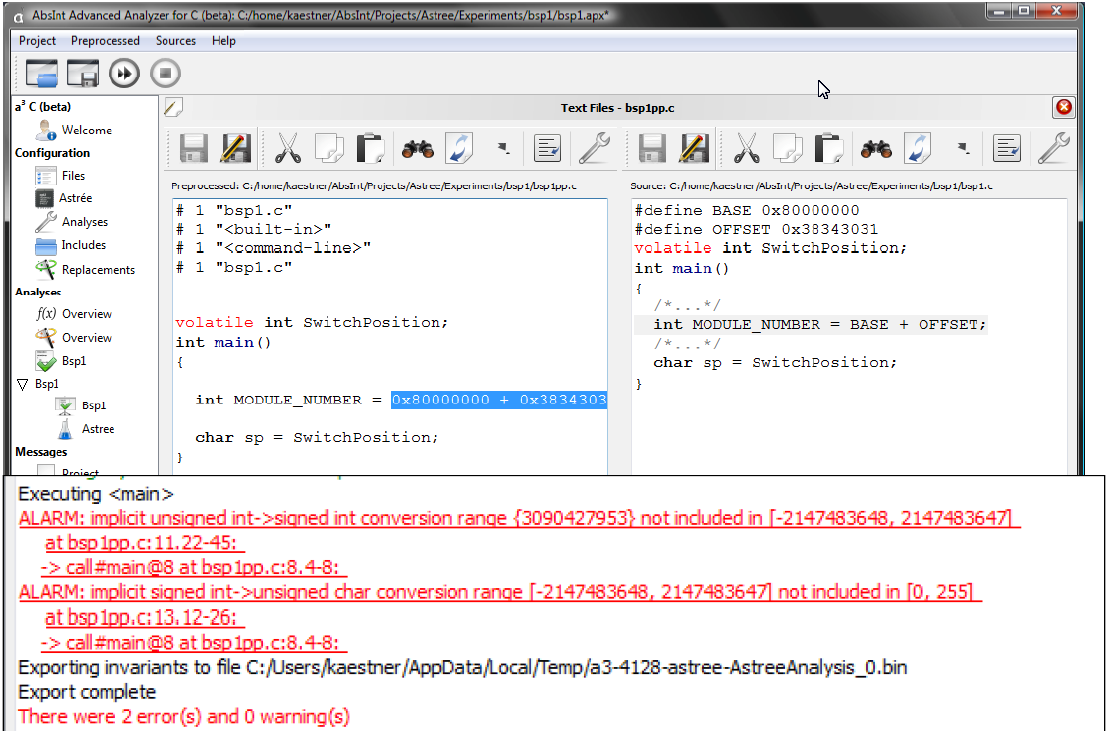
*}*

*}*

Astrée proves that the above program is free of run-time errors when running on a machine with floats on 32 bits.

Example 4

Screen short of Astrée:



**8.3 TVLA**

TVLA (Three-Valued-Logic Analyzer) is a research tool implementing a system for executing static-analysis from the semantic of a given program.

In the “real” execution each program state is represented by the memory. TVLA represents program state as logical structure and program statements as logical formulas.

TVLA automatically generates intermediate representation of the program which is a control flow graph with edges between the logical formulas representing the statements.

**8.3.1 Concrete Program State**

Program state represents the possible states of the memory (heap). It is represented using 2-valued logical structure:

Structure in logical representation:

The logical structure is a pair of (U, I) where U is the universe that represents the heap elements, i.e. the memory stores and I is an interpretation function that maps each predicate to its interpretation. The unary predicates encode the contents of variables and the binary predicates encode the contents of pointer valued fields.

Structure in graphical representation:

-Individuals of the universe are represented by circles with names inside.

-A unary predicate p is represented in the graph by having a solid arrow from the predicate name p to node u if p(u)=true.

- A binary predicate n is represented in the graph by having a solid arrow labeled n between each pair of the relation.

**8.3.2 Examples of structures for program in C with two variables x and y and the following list structure.**

*Typedef struct node {*

*struct node \*n;*

*int data;*

*} \*List*;

There are 3 predicates:

x(u), y(u) (true, if the variable points to the object u)

n(u1,u2) (true, if field n of object u1 points to the object u2)

Example 1

The concrete program state is: x points to a list with two members and y points to the last list element. Then the structure is:

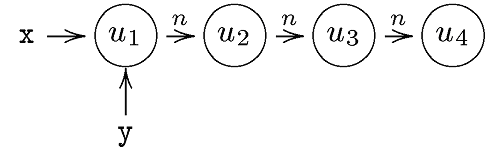
x

n

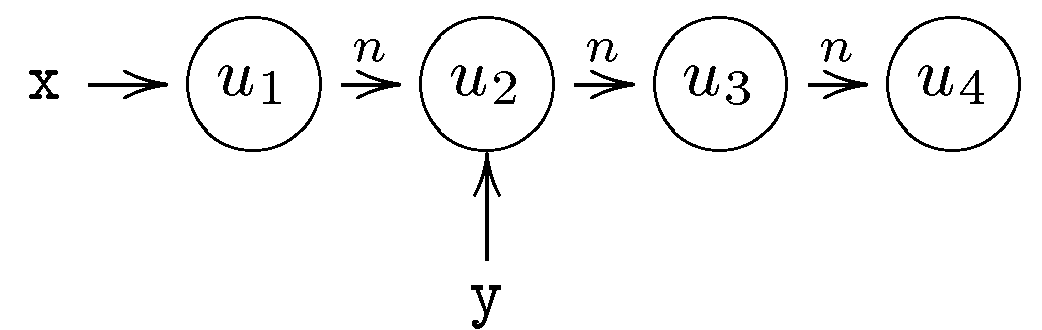
y

Example 2

The concrete program state before the statement x = y->n is that x and y point to a list with 4 elements .



Then the concrete program state after the statement x = y->n will be



**8.3.3 Examples of structures for the program, which contains 3 variables: x, t and root and two pointer fields left and right.**

There are 5 predicates

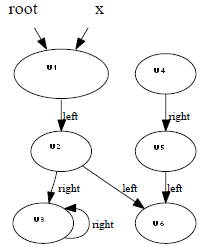
t(u), x(u), root(u) (true if the variable points to the object u)

left(u1,u2) (true if field left of object u1 points to the object u2)

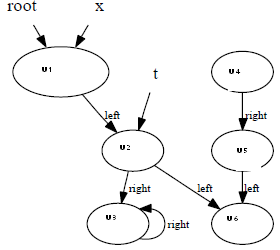
right(u1,u2).

Example 1

The concrete program state before the statement t=x.left is:



Then the concrete program state after the statement t=x.left will be



**8.3.2 Running the example with the concrete semantics**

The following is the control flow graph representing the program:

return x

x = t

t =malloc(..);

t→next=x;

x = NULL

T

F

The structure contains the unary predicates x(u) and t(u) and binary predicate next(u1,u2).

1. At the beginning there is the empty structure.

2.The assignment x=null does not change the memory state, therefore also after this statement the concrete state is represented by the empty structure.

3. After t=malloc(…) the memory state is: one element is pointed by t.

4. After t->next=x the state is not changed because the interpretation of the predicate x is empty, therefore the concrete state after this statement remains

tt

5. After the statement x=t the concrete state is

t

The following picture represents the memory states (from left to right) after executing the program statements from this point:

t

n

t

x

n

x

t

x

x

n

t

n

t

x

x

t

n

n

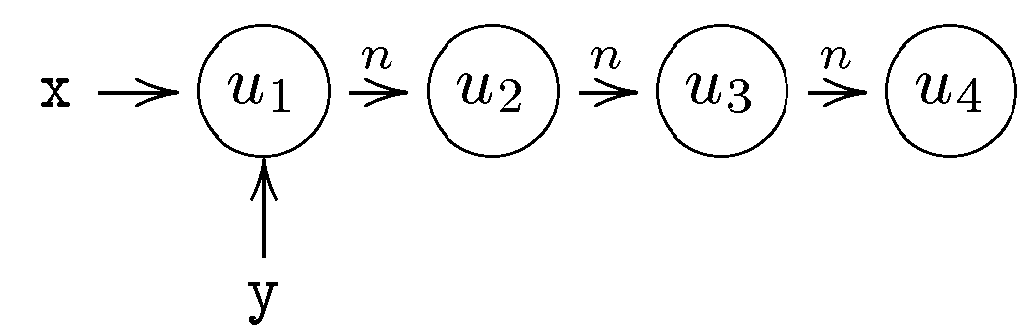
**8.3.3 The abstract semantics**

The abstract memory stores by TVLA are 3-valued logical structures.

We explain the abstract interpretation using the following example:

Suppose the vocabulary contains two unary predicates x, y. Then two elements u1 and u2 are equivalent iff they get the same values on these predicates, i.e. x(u1)=x(u2) and y(u1)=y(u2).

Suppose the concrete structure is list with 4 elements that the head of the list pointed by x and y.



Then there are two equivalent classes:

* u1 which represents the concrete element u1
* u234 which represents the elements u2,u3 and u4.

The concrete u1 will be mapped to the abstract u1 and the concrete u2, u3 and u4 will be mapped to the abstract u234.

The interpretation of the abstract unary predicates is obvious from the construction.

For the concrete binary predicate n the corresponding interpretation of abstract predicate n will be:

- n(u1,u1)=0, because the concrete n(u1,u1) is evaluated to 0.

- n(u1,u234)=1/2, because the concrete n evaluates to 0 for the pairs (u1,u3) and (u1,u4)

and to 1 for the pair (u1,u2) .

- n(u234,u234)=1/2, because the concrete n evaluates to 0 for some pairs from

{u2,u3,u4}x{u2,u3,u4} and to 1 for other pairs from {u2,u3,u4}x{u2,u3,u4}.

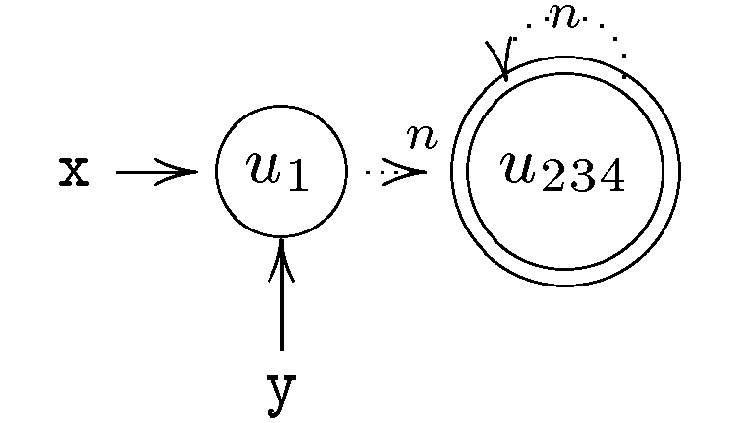
There is an additional unary predicate, called “summary” (sm), which captures whether individuals of the 3-valued structure represent more than one concrete individual. For example sm(u1)=0.

The graphical notation for 3-valued logical structures is derived from the one for 2-valued structures, with the following additions:

- summary nodes are represented by double circles;

- unary and binary predicates with 1/2 value are represented by dotted arrows.

Then the abstract structure of our example is:



The 3-valued logical structures are of bounded size. At every program point TVLA will have abstract state that represents the (possibly infinite) set of 2-valued logical structures according to the concrete semantic.

* + 1. **Running the example in the abstract domain**

x

t

n

nn

xx

t

tn

t

n

x

n

nt

t

x

t

x

x

t

n

nx

x

t

n

n

2

return x xx

x=t

t =malloc(..)

t→next=x;

NULL

T

F

4

6

12

1

5

x

t

n

xx

t

tn

n

x

t

n

3

11

x

t

n

n

t

x

t

x

empty

n

n

The analysis is the same as previously until step 11.

At step 12 the concrete semantics yields:

x

tt

nn

n

However the last two elements get the same value for the predicates x and t, therefore the last two elements are mapped to the same element.

x

t

n

This abstract structure represents arbitrary list with three or more elements, where x and t point to the first element.

Therefore the last iteration starts with the list of arbitrary size. This list can not be bigger, and then at this iteration the analysis reaches fix point.

*List InsertSort(List x) {*

*List r, pr, rn, l, pl; r = x; pr = NULL;*

*while (r != NULL) {*

*l = x; rn = r → n; pl = NULL;*

*while (l != r) {*

*if (l → data > r → data) {*

*pr → n = rn; r → n = l;*

*if (pl = = NULL) x = r;*

*else pl → n = r;*

*r = pr;*

*break;*

*}*

*pl = l; l = l → n;*

*}*

*pr = r; r = rn;*

*}*

*assert sorted[x,n];*

*//assert perm[x, n, x, n];*

*return x;*

*}*

* + 1. **Example of using TVLA**

TVLA can prove that at the end of this program

the list is sorted and it is a permutation of the

original list.

*typedef struct list\_cell {*

*int data;*

*struct list\_cell \*n;*

*} \*List;*

* 1. **Limitations**

TVLA has scalability problems;

it works on programs of size up to 10 screens.