Creation of a Static Analysis Algorithm Using Ad Hoc Programming Languages

Dmitry Khalansky
ITMO University
Saint Petersburg, Russia
rouanth@gmail.com

Arthur Lazdin
ITMO University
Saint Petersburg, Russia
lazdin@yandex.ru

Dmitry Mouromtsev
ITMO University
Saint Petersburg, Russia
d.muromtsev@gmail.com

***Abstract*—The complexity of software grows every year, and while there are many programming techniques and new languages that accommodate the need to provide high abstractions, still many languages that require attention to low-level details are in use as of yet. In order to avoid tedious debugging which needs time that could be spent on dealing with high-level logic, static analysis of source code can be used to more efficiently find common problems. We have studied the process of creation of algorithms for static analysis tools by building a simple value range analysis mechanism, that is, a way to detect some cases of integers not matching a predicate involving arithmetic and comparison operations. This algorithm provides means to detect possible division by zero and integer overflow and is easily extended to find cases of out-of-bounds addressing of containers. While there is a multitude of value range analysis mechanisms that are more sophisticated by orders of magnitude, the works in which they are presented focus on the properties of the resulting tools such as estimated amount of false positives, performance, memory usage, or soundness. We, on the other hand, are going to present the process of extension of static analysis algorithm from ground up. An ad hoc programming language is developed in multiple stages to separate the creation of algorithm from numerous details of its implementation which would necessarily arise were we to build it on a real-world language.**

I. Introduction

There are many pitfalls awaiting those who wish to develop a static code analyser. For example, one may try to work on an existing and mature language, and this often leads nowhere since too often languages, evolving over time, develop a large set of corner cases which must all be accounted for during analysis. This is a problem because algorithms used for static analysis are often non-trivial and require rapid prototyping which can often be hindered by necessity to view the algorithm full context of the language.

One way to mitigate this is to work not with source code directly but with some intermediate form used internally by existing tools for the language such as compilers or interpreters; sometimes even libraries for building representations of source code are provided, which can also be used. This allows one to bypass parsing, which often is a complicated task itself. The main drawback when compared to manual parsing is the possible loss of information which is commonly stripped such as comments, so it's a step forward.

We, however, assert that even better approach is introduction of a language designed specifically to test a static analysis algorithm. The main benefits as we see them are:

- Iterative development. When using an existing language infrastructure, even the first version of the algorithm must take into account most of the language constructs even if this support consists of just a vague outline. Testing such an algorithm can be cumbersome since it isn't always obvious whether there's an error in the algorithm itself or its incompleteness constitutes the problem. On the other hand, when language accompanies algorithm, each construct can be added after the existing ones have been thoroughly tested, allowing the researcher to eliminate bugs more quickly.

- Portability between languages. Often multiple languages have a common set of constructs with behaviour that's more or less alike. If an algorithm is language-agnostic, it can be applied to multiple languages with little effort. If the algorithm is developed with a specific language in mind, however, it can be more difficult to separate it from the details not inherent to the semantics of the source code but influenced by the design of the language.

- More deep insight. When developing an algorithm for an existing language, one should conform completely to its semantics. While the creation of algorithm itself can be a very challenging and creative task, we firmly believe that having the ability to modify the language to check how these changes would influence the algorithm significantly enhances the understanding of numerous small details about the language semantics and means to analyse it.

- Easier formal verification. The semantics of most mature languages are difficult to formalize in terms of automated verification assistants, which limits the ability to reason about algorithms designed specifically for them. On the other hand, the semantics of ad hoc languages are often straightforward, which allows to spend most of the effort on verification of algorithms rather than description of the language.

With this in mind, we set out to show the process of development of a simple value range analysis mechanism by using a more sophisticated language at each step of the process, enhancing it as the algorithm evolves.

II. Development of static analyser

A. Type system

Before defining a set of constructions the language will possess, we need to determine what kinds of data should it operate on.

Since we're only truly interested in numbers when analysing the value ranges, it makes sense not to provide sum types or product types, strings, hash maps and many other types typical of languages used in the real world. We also don't treat boolean values as a separate type, instead using non-zero and zero integers to represent truth and falsehood respectively. As for arrays, since we're only interested in them when they are subject to possible out-of-bounds addressing, they can be represented as a number n of elements in them, with insertion and deletion being respectively the increment and decrement of n, and their addressing stated as an assertion that the index doesn't exceed n.

Furthermore, we will only support integral numbers. Implementing rational numbers as pairs of integers would make the algorithm more cumbersome without adding much value since relations between rational numbers are easily expressed with operations on integers, which shall be covered. Introduction of general floating-point types, one the other hand, would make the algorithm significantly more complex. Due to the imprecise nature of floating-point numbers, it's difficult to reason about them formally. Therefore, we declare them to be out of scope of this work.

Now, we need two kinds of integral numbers. The first one (hereafter "infinite integers") is just the type of numbers up to infinity. For these, our goal shall be finding the maximal value they may take in the course of program execution and determining the machine type to be used as their representation in memory. The second kind (hereafter "modular integers") is of integers modulo some power of 2. Their maximal value shall also be checked in order to detect possible overflows. While it's true that they may be desirable in some cases, our language doesn't support them. We don't see, however, how it would be difficult to implement them since it would only take one type parameter and a simple branching in the algorithm to suppress the warning about possible overflow for a variable.

B. Operations on numbers

Given that the only two kinds of types in our language are both numeric, it seems natural to proceed by defining the operations on integers available in our language.

First and foremost, there is a well-defined conversion between the two kinds. We can for the sake of this discussion treat infinite integers as integers modulo infinity. Then conversion from a number modulo n to modulo m is just changing the type parameter if $m > n$ and taking this number modulo m otherwise.

Next, arithmetic operations of addition, subtraction, multiplication, and division, work as expected on infinite integers. Modular integers have these operations as well, and division doesn't take modular inverse of its divisor— instead, modular integers are treated exactly like infinite ones in this regard. This is what we're accustomed to and encounter in the overwhelming majority of programming languages supporting division, even the ones that recognise that the numbers are stored in memory in registers modulo some integer. Arithmetics between integers of different types is impossible without explicit conversion.

Next, another difference between the infinite and modular integers is that we add to the latter support of bitwise operations. It's easy to see that in our case bitwise negation of infinite numbers isn't well defined. Despite the fact that conjunction and disjunction, on the other hand, are, we don't see how these operations would be useful for infinite numbers. We expect fixed-length numbers to be used when the need to store option flags or binary structures arises, and there isn't much reason to perform bitwise operations on numbers of arbitrary lengths.

Of course, we need to compare our numbers as well. We introduce two operators for this, "equals" and "less". When comparing two numbers with different modules, we take the lesser module and compare them using it. The result has type of integer modulo 2.

Now, it would be tempting to create a language which supports only the operations listed above and assignments of variables. But there wouldn't be much to analyse this way since all presented operations are well-defined. Therefore, we need to introduce some form of non-determinism, and so we add a **rand** operator which returns an arbitrary integer in $[0; +\infty)$.

Careful analysis reveals that there is a restriction we must impose in order to make the system consistent. We can't assign **rand** to an infinite number directly because it would mean that the number requires arbitrarily large representation in memory. In order to have an infinite number with a random value, we must first convert the result of **rand** to a modular number, and only then convert it to an infinite one. Thus, we only support random numbers modulo some power of two.

The special role of **rand** also allows us to speculate that it mustn't be considered a number at all. It's obvious why it isn't a modular number. But there isn't much sense in using it as an infinite one as well. Consider division by an arbitrary number: it doesn't change the distribution of values except for possible change of sign which, given that we have modular arithmetics, is irrelevant. Dividing by 3, for example, still gives us the equally possible values $[0; \infty)$. Next, addition and subtraction do shift the distribution, but we still have to use the result modulo some number. Multiplication does have a meaning, but only in a handful of cases. $kx \mod n$ has the same distribution of values as $x \mod n$ if k and n are relatively prime. If they aren't, for example, if k is also a power of 2, the difference is notable as it reduces the set of possible values from $[0; n)$ to $\{0\} \cup \{k\} \cup \{2k\} \cdots \left\{\left(\frac{n}{k} - 1\right) k\right\}$. It is easy to verify that the same result can be achieved using operations outside the initial conversion to modular numbers. The only thing that could be done is division by the random number which gives us a weighted distribution. But the expected value of division by a random number in $[0; \infty)$ is zero. So without loss of generality we assume that **rand** doesn't support any operation except for conversion and treat it as if it had a separate type. If the need arises, it is easy even if cumbersome to expand algorithms to account for arithmetics on random values.

We shall use the familiar infix notations with the same operator precedence as in C. Naturally, the precedence can be overridden using placement of brackets. Type casting operators have higher precedence then everything else, with conversion to modular numbers having higher precedence then conversion to infinite numbers.

With that out of the way, let's build the ABNF grammar of the language subset that we described.

⟨*expr*⟩ ::= ⟨*expr*⟩ [SP] ⟨*infix-op*⟩ [SP] ⟨*expr*⟩
| ⟨*var-name*⟩
| ⟨*number*⟩
| '~' ⟨*expr*⟩
| 'inf' SP ⟨*expr*⟩
| ⟨*expr*⟩ SP 'bits' SP ⟨*mod-number*⟩
| '(' [SP] ⟨*expr*⟩ [SP] ')'
| 'rand'

⟨*var-name*⟩ ::= '@' ALPHA [alnum]

⟨*alnum*⟩ ::= ALPHA [alnum]
| DIGIT [alnum]

⟨*white*⟩ ::= ' '
| '\t'
| '\n'

⟨*infix-op*⟩ ::= '+'
| '-'
| '*'
| '/'
| '&'
| '|'
| '-'
| '<'
| '='

⟨*mod-number*⟩ ::= ⟨*non-zero-digit*⟩ [natural-number]

⟨*natural-number*⟩ ::= ⟨*digit*⟩ [natural-number]

⟨*non-zero-digit*⟩ ::= a number from 1 to 9

⟨*digit*⟩ ::= 0
| ⟨*non-zero-digit*⟩

An example of an expression matching the '<expr>' rule is:

```
inf ~ ((32 & @x bits 6) = 0)
```

Here, we take the value of @x modulo 64 (because $2^6 = 64$), perform bitwise conjunction with 32, thus taking its 5th bit, compare it to zero, which gives us a number modulo 2, invert, and convert it to an infinite number. The result is an infinite number equalling to 1 if the 5th bit of @x is set and 0 otherwise.

C. Algorithm for arithmetics

This isn't much of a language, but it's enough to be reasoned about.

Our goal is finding out which values expressions can take.

We don't have a way to access variables yet, so we shall ignore them for the time being.

The naive way to do this would be considering each operator separately, defining rules for various operations. While this would work on the current stage, it would need a complete rewriting later, when we introduce variables. For example, consider the following expression:

```
@x − 1 ∗ (@x + 2)
```

For any casual observer it's obvious that for any value of @x the result would be -2. But if @x has range $[0; 15]$, then the naive solution gives us

```
[0; 15] − [1] ∗ ([0; 15] + [2]) =
[0; 15] − [1] ∗ [2; 17] =
[0; 15] − [2; 17] =
[−15; 13]
```

Obviously, a more clever solution is needed.

The only possible source of non-determinism is the **rand** operator. Therefore, for each use of the operator, we introduce a unique handler. All subsequent operations over the handlers only modify their ranges but don't discard the source. For our previous example, let's say that @x has its value as a result of taking a random number R modulo 16. Then we have

```
{R mod 16} − {1} ∗ ({R mod 16} + {2}) =
{R mod 16} − {1} ∗ {R mod 16 + 2} =
{R mod 16} − {1 ∗ (R mod 16 + 2)} =
{R mod 16 − 1 ∗ (R mod 16 + 2)}
```

Now checking all the possible values of R mod 16 gives us the correct result, a single number 2.

This form also allows us to perform basic arithmetic transformations. One of them is usage of distributivity of multiplication and bitwise conjunction. It enables us to group terms that are alike and eliminate them.

With that in mind, we can internally represent every expression as a sum type with the following constructors:

- **ARand**— constructor representing **rand** operator, with a single parameter being the modulus.
- **AConst**— constructor for numeric literals, having the value as its single parameter.
- **AVar**— constructor for getting a value of a variable, its parameter being the unique identifier of the variable.
- **AInf**— constructor for conversion operator to an infinite number, accepts a single parameter— underlying expression.
- **AMod**— conversion to a modular number, accepting two parameters: the number of bits and the underlying expression.
- **APlus**, **AMinus**, **AMult**, **ADiv**, **AAnd**, **AOr**, **AEq**, **ALt**— infix operators, each accepting two expressions.
- **ANeg**— prefix operator for bitwise negation, having the underlying expression as a parameter.

This way, the example above can be represented as

```
AInf (ANeg
  (AEq
    (AAnd
      (AConst 32)
      (AMod 6 (AVar 'x')))
    (AConst 0)))
```

Now we can reason about how various constructs affect the value ranges.

- **ARand**— as specified above, the result of **rand** can be any number in $[0;n)$, where n is a modulus.
- **AConst**— the range a constant can take is just its value.
- **AInf**— the range is unaffected.
- **AMod**— this conversion limits the range to the values in $[0;n)$ where n is the type parameter. Furthermore, the value range of the underlying expression— let's call it $[a;b]$— is processed as follows. First, the greatest multiple of n less than or equal to a is determined using the formula $\lfloor \frac{a}{n} \rfloor \cdot n$. Next, the least multiple of n greater than b is found with $\lceil \frac{b}{n} \rceil n$. Then the range $\left[\lfloor \frac{a}{n} \rfloor \cdot n; \lceil \frac{b}{n} \rceil n\right)$ is split into chunks of size n. For each chunk we create a vector of bits of length n, setting each bit to true if the corresponding number in this chunk is in the range and to false otherwise. The value range after the conversion is determined by analysis of bitwise disjunction of these vectors.
  An example of this procedure is as follows: let's say we have an expression of range $[5;7] \cup \{10\}$ and wish to find the range of its value modulo 4. First, we find the minimal range which includes this one but is delimited by multiples of 4, which is $[4;12]$. We split it in chunks of four elements as follows: $[4;8)$, $[8;12)$. The corresponding vectors are 0111 and 0010. Their disjunction is 0111. Thus, the value range we're looking for is $[2;4]$.
- **ANeg**— for every part of value range, we find the value of bitwise negation of its borders, swap them, and take their union.
  For example, if $[5;7] \cup [10;14]$ has type of integer modulo 32, then in binary these ranges are represented as $[00101;00111] \cup [01010;01110]$. Negating them gives us $[\ 00111;\ 00101] \cup [\ 01110;\ 01010]$, which is $[11000;11010] \cup [10001;10101]$. In decimal, $[24;26] \cup [17;21]$.
- **AEq**— the ranges of the operands are compared. If the intersection of the ranges is non-empty, then their **AEq** can be 1, otherwise it's always 0. If both ranges consist of the same single element, then their **AEq** can't be 0 and is always 1.
- **ALt**— in a similar vein, if the greatest element in range of the left operand is less than the least element in range of the right operand, **ALt** is always 1; if the least element in range of the left operand is greater than the greatest element in range of the right operand, **ALt** is always 0; otherwise, it can be both 0 and 1.
- **APlus** and **AMinus**— if subsets of value ranges are represented as $[a_1;a_2]$ and $[b_1;b_2]$, then the range of results of addition of the corresponding values is $[a_1+b_1;a_2+b_2]$. For subtraction it's $[a_1-b_2;a_2-b_1]$. If the addition or subtraction is performed on modulo numbers, we need to behave like **AMod** wraps the results, but see the discussion of possible overflows below.
- All the other operations are binary arithmetic and bitwise operations, and for them one should take the Cartesian product of ranges of their operands and apply the operator to each tuple.

Let's see how these rules apply to our example if `@x` in $\{0\} \cup [100;127]$ (for the sake of this discussion we ignore the fact that we've decided to use random number handlers instead of ranges).

```
inf ~ ((32 & @x bits 6) = 0)
inf ~ ((32 & {0}, [100; 127] bits 6) = 0)
inf ~ ((32 & {0}, [36; 63]) = 0)
inf ~ (({0}, {32}) = 0)
inf ~ [0; 1]
inf [0; 1]
[0; 1]
```

Why is this useful if we iterate through the possible values of random numbers? The answer is that it isn't uncommon for expressions not to depend on a single random number in multiple places. Thus, in many cases we can isolate the parts where each random number is present, iterate over its values, and then use the resulting ranges instead.

Consider the expression

```
(@x + 2) * (@y − 5) < @z + @a
```

Here we assume that all the variables are independent from each other in terms of used random handlers. Let's say each variable has a value range of $[0;2^{32})$. Then the approach using random handlers would require us to iterate $\left(2^{32}\right)^4$ times. On the other hand, using the range-transforming rules we only need to perform $\left(2^{32}\right)^2 + 2^{32} + 2^{32}$, which is approximately a square of the time we would need using the aforementioned approach.

Now we have to consider that, if an arithmetic operation is performed on a modulo number, an overflow or division by zero can occur. For such a case, we need to mark the resulting set of values as being possibly invalid. Mechanisms of varying complexity can be introduced to inform user about the possible error. We shall simply store a boolean value denoting the possible error an propagate it to the top of the tree representing expression, that is, an error bit is set for a node if an invalid operation can occur in it or any of its children.

The resulting algorithm is as follows:

1) Perform high-level optimisations such as reductions, rewriting using distributivity and associativity, or other similar operations on expressions. We omit the details here for the sake of briefness since they are rarely relevant and can introduce only marginal improvements in terms of speed. While it's possible to write code such as subtraction of a variable from itself, this capability is rarely used in real programs.

When the same random handler occurs multiple times in the expression, usually it isn't subject to trivial optimisations.

2) Build a list of all the random number handlers used in the expression, counting the number of occurrences of each one.
3) For each handler, traverse the expression and mark the subtrees in which it is present. Then, starting from the top, descend the tree once again using these marks and replace either the first binary operator both operands of which are marked, or if it doesn't exist, the random number itself with the corresponding ranges, possibly in terms of other random numbers. If an overflow or division by zero possibly occurs during the traversal, set the error bit on these ranges.
4) Once all the handlers are purged, use the more efficient algorithm to find the value range of the whole expression. If an error possibly occurs, set the error bit of the result.

D. Imperative assignments

Now that we have expressions, let's put them to use and write a language that allows us to chain assignment operations. An example program in such a language could be as follows:

```
@x = rand mod 16;
@y = @x * 2;
@z = @x & @y
```

The first question that arises is of storage: how are we going to represent the state of the variables? A partial answer has been proposed in the previous section: we have to store the expressions in terms of random number handlers.

For this, we need to replace all the variable reads with their underlying expressions while preserving their type, otherwise on the next write to a variable all the values of variables using it would invalidate. And, as we've mentioned before, **rand** operators aren't substituted as-is. Rather, they're replaced by a unique identifier on their use, and then variable substitutions operate on these.

So for variable value storage we can use the same type as the one for describing expressions with the exception that **ARand** must accept a parameter which is its unique identifier. Our solution is to make **ARand** parametric in the original definition, too, but to provide arbitrary parameter during parsing and ignore it until it's replaced by the identifier on the stage of analysis.

It is straightforward now to specify the language constructs.

- **CAsgn**— assignment operation. Accepts two parameters: the unique identifier of the variable and the expression assigned to it.
- **CSeq**— chaining of assignments. Accepts two parameters which are the expressions being chained.

For example, the example above could be expressed using these terms as follows:

```
CSeq (CAsgn 'x' (AMod ARand 10))
     (CSeq (CAsgn 'y'
                (AMult (AVar 'x')
                       (AConst 2)))
           (CAsgn 'z'
                (AAnd  (AVar 'x')
                       (AVar 'y'))))
```

It may be difficult to see at first why would we need to introduce a separate construct for chaining when we could have a simple list of assignments. But expanding the language with flow control structures should allow a unified treatment of single statements and chains of them. We could, of course, instead of building trees of **CSeq** use a constructor accepting not two arguments but a single list of them. But brief consideration reveals that **CSeq** is actually a linked list itself.

Now, let's formalise the operational semantics of the new language.

First, we introduce an environment which is a mapping from variable identifiers to their values. At first, the environment is empty. What shall we do if the program accesses a variable that hasn't been set yet? The choice is arbitrary for our discussion, so we just claim that such programs are invalid. We don't have a way to express the incorrectness of a program; more robust system is required for this. We assume that our algorithm deals only with well-formed programs.

Next, we need to have a list of possible ranges for each variable at each point of execution in order to determine the maximal and minimal values that it takes. This leads to another restriction: it's prohibited to assign to a variable values that have different types. For example, if a variable x has been an integer modulo 256, it can't at a later point become an integer modulo 16 or an infinite integer. We consider only the programs that are correct in this regard.

We define an evaluation function which takes a tree representing the program, an environment, and calls itself recursively, traversing the tree using the following principles:

- **CAsgn**
  1) Replace each **ARand** with one having a correct unique identifier.
  2) Traverse the expression being assigned, replacing occurrences of **AVar** with contents of environment for the respective identifiers.
  3) Perform the algorithm discussed in the previous section on the result. Add the range to the list of possible values corresponding to the variable.
  4) Check the error bit; if it's set, notify user.
  5) Put the result into the environment with the name of the variable being assigned.
- **CSeq**
  1) Take its first subtree, run the evaluation function with the same environment— say, e_1— on it. Take the resulting environment and call it e_2.
  2) Take the second subtree, run the evaluation function on it with environment e_2. The resulting environment is e_3.
  3) Return e_3 as the new environment.

It's easy to see that **CSeq** is an associative operation, that is, for all statements a, b, c, we have CSeq a (CSeq b c) $=$ CSeq (CSeq a b) c. Because of this, we can build the tree representing a program by nesting **CSeq** arbitrarily.

The ABNF of the resulting language is just the previous one with an additional rule:

$\langle statement \rangle$ ::= $\langle var\text{-}name \rangle$ [SP] '=' [SP] $\langle expr \rangle$
  | $\langle statement \rangle$ [SP] ';' [SP] $\langle statement \rangle$

E. Introducing conditionals

While this language could have its use, we shall see that it isn't difficult to add conditionals to it, making it a little closer to what we're used to see.

First, we define a new statement called **CIf** with three parameters: the conditional expression and two statements. Its operational semantics is as follows:

1) Evaluate the conditional expression with the current environment.
2) If the result is 0, evaluate the second statement with the same environment. Otherwise, evaluate the first. The result of evaluating **CIf** is the same as the resulting environment of the chosen statement.

An example of a program based on **CIf** is:

```
@a = rand mod 16;
@b = rand mod 32;
if (@b mod 16 == @a) then
        @c = @a mod 32 * 2
else
        @c = @a mod 32 + 1
fi;
@d = @c == 0
```

As is evident, we don't have a separate scope for variables initialized in the statements inside of **CIf**. This could lead to a problem. If a variable hasn't been assigned to before **CIf** but is initialized in one of the branches but not the other, we're left with a program that could be correct in one set of cases but not the other. We need to keep this in mind and prohibit programs that perform initialization in just a single branch of **CIf**.

With this out of the way, we define the following mechanism for reasoning about **CIf**:

1) For each statement, find all the variables that are set in it.
2) For each variable in both sets, consider the following cases:
   - It is being set in both branches and has values c_t and c_e respectively. Then the resulting value of the branch is $a \cdot c_t + (a = 0) \cdot c_e$, where a is the condition.
   - It is being set in the first branch but not the second. Then its value is set to $a \cdot c_t + (a = 0) \cdot c_o$ where c_o is the value of the variable in the initial environment.
   - It is being set in the second branch but not the first, then it's value is $a \cdot c_o + (a = 0) \cdot c_e$.
3) Put the resulting ranges for each variables into the resulting environment.
4) Merge the lists of possible value ranges obtained during traversal of both branches for each variable.

For example, assuming the presence of variables `@a` and `@b` in the environment, consider a simple implementation of **max** operation which is defined as follows:

```
if (@a < @b) then
        @c = @b
else
        @c = @a
fi
```

Our analysis reveals the value of `@c` to be $(a < b) \cdot b + (a \geq b) \cdot a$.

We extend the grammar of the language as follows:

$\langle statement \rangle$ ::= $\langle var\text{-}name \rangle$ [SP] '=' [SP] $\langle expr \rangle$
  | $\langle statement \rangle$ [SP] ';' [SP] $\langle statement \rangle$
  | 'if' SP '(' [SP] $\langle expr \rangle$ [SP] ')' SP 'then' SP $\langle statement \rangle$ SP 'else' SP $\langle statement \rangle$ SP 'fi'

F. Terminating loops

It is important to support loops to express the majority of useful programs. However, loops introduce the possibility of programs that never terminate. Even more, it's been proven[2] that there can't exist an algorithm which discerns in general case the programs that never halt. All we can do is reason about separate programs.

For now we shall ignore the existence of loops that may never terminate and instead use the imperative equivalent of total functional programming[3]. The main point is that we can analyse data (as specified in the category theory) in a predetermined number of steps. This allows us to express many useful concepts using only the loops that are guaranteed to terminate.

With that, we introduce the **CFor** statement with three parameters: the name for the counter variable, the expression to represent a number of steps to perform, and a loop body.

The semantics are as follows:

1) Assign 1 to the variable used as a counter.
2) If the variable is greater than the number of steps, exit the loop.
3) Get the environment of the loop body.
4) Assign to the counter variable the current step number.
5) Repeat starting from step 2.

An evident difference from the way counter variables are commonly used is that modifying the variable inside the loop body doesn't change the flow of the program: the number of steps is stored internally and is assigned to the counter each time a new iteration is started. This allows us to be confident that all our programs always terminate.

An example of program using this kind of loop follows:

```
@j = 0;
for @i to (@n) do
        @j = @j + @i
done
```

This program calculates the sum of the first `@n` numbers.

Loops can be reasoned about, but it's nontrivial. Our subset of loops corresponds to recursive definitions, and some of them have the so-called closed forms, that is, arithmetic expressions which, when evaluated, give the same result as the final assignment in the loop. For example, for the loop presented above the closed form is this:

```
@i = @n;
@j = @i * (@i + 1) / 2
```

There are many techniques to find the closed forms of recursive definitions (see "Concrete Mathematics"[5] for concrete examples), but none of them are automatic. The best we can do is to store a library of common closed forms, simplify the expressions in loop bodies and try to find a match. This is out of scope of this work.

So in order to analyse loops, we perform the following:

1) Determine the value range for number of steps (hereafter n).
2) Iterate the loop at least for the number of times corresponding to the least value of n.
3) Calculate the values of all the variables assigned in loop bodies after this iteration. Store them in a list in form of a tuple (x, i, e), where x is the name of the variable assigned, i is the iteration number, and e is the value of the variable at this point.
4) If this iteration has been the last, halt and, for each variable assigned at the loop body, generate an expression of the form $\sum_i (i = n) \cdot e_i$ using the data from the list generated in the previous step where e_i is the value of the variable at iteration i.
5) Repeat from step 2.

To define an ABNF of the resulting language, we only need to modify one rule:

⟨*statement*⟩ ::= ⟨*var-name*⟩ [SP] '=' [SP] ⟨*expr*⟩
| ⟨*statement*⟩ [SP] ';' [SP] ⟨*statement*⟩
| 'if' SP '(' [SP] ⟨*expr*⟩ [SP] ')' SP 'then' SP ⟨*statement*⟩ SP 'else' SP ⟨*statement*⟩ SP 'fi'
| 'for' SP ⟨*var-name*⟩ SP 'to' SP '(' SP ⟨*expr*⟩ SP ')' SP 'do' ⟨*statement*⟩ 'done'

III. Further extensions

One could add weighted probabilities to different values in the value ranges so that user of the algorithm would know which branches would be taken more often.

It is also possible to define a partial order on the set of possible values with comparison operation being the proximity of value ranges to zero. This would make it possible to reason about a wider class of loops and determine their termination.

Functions could be defined as separate programs with some initial variable values and specific types. Functions can be used as expressions.

Complex structures can be built and stored internally as sets of variables, that is, they can be treated as simple syntactic sugar.

Sign bits could be added to modular numbers, rendering them signed. Modifications to arithmetics would need to be implemented, but we don't see how it could be difficult.

General case for loops could be considered by determining variables that change with a fixed period and analysing them: it would suffice to determine the ranges on one period and take their union. This reasoning can be extended to whole groups of variables that form a strongly connected component with regard to mutual modification. All the variables for which a period can't be found should be discarded by saying that they can take any value at all.

IV. Conclusion

In the course of this work, we've used an ad hoc language to reason about simple value range analysis algorithm. The iterative nature of the process has allowed us to highlight several key points which can be applied to value range analysis without the need to consider the semantics of a whole language.

The outline of our discussion has been as follows:

- Decide which sorts of data structures need to be analysed by the algorithm. Reflect this in type system of the language.
- Determine the primary operation that needs to be analysed. Reason about it. Formalize the language semantics involving this operation.
- Gradually add the other desirable parts of the language to increase the resemblance between the experimental language and languages that are targets for the algorithm.

We believe that this approach allows one to efficiently develop static analysis algorithms.

Acknowledgement

We thank Igor Zhirkov, IntelliJ Labs, for providing invaluable advice and literature references throughout the course of this research.

References

[1] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Catalin Hritcu, Vilhelm Sjberg, and Brent Yorgey, "Software Foundations". Electronic textbook, 2016. Version 4.0. Web: http://www.cis.upenn.edu/ bcpierce/sf.

[2] Alan Turing, On "computable numbers, with an application to the Entscheidungsproblem". Proceedings of the London Mathematical Society, Series 2, Volume 42 (1937), pp 230265, doi:10.1112/plms/s2-42.1.230.

[3] Turner, D.A. (2004-07-28), "Total Functional Programming", Journal of Universal Computer Science, 10 (7): 751768, doi:10.3217/jucs-010-07-0751.

[4] Birch, Johnnie; van Engelen, Robert; Gallivan, Kyle, "Value Range Analysis of Conditionally Updated Variables and Pointers". Web: http://www.cs.fsu.edu/ engelen/cpcpaper.pdf

[5] Ronald L. Graham, Donald E. Knuth, Oren Patashnik, "Concrete Mathematics: A Foundation for Computer Science". 2nd edition. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1994. ISBN:0201558025

[6] Patrick Cousot, Nicolas Halbwachs, “Automatic discovery of linear restraints among variables of a program“. In Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 84–97, Tucson, Arizona, 1978. ACM Press, New York.

[7] Ilya Sergey, “Programs and Proofs“. Web: http://ilyasergey.net/pnp/