IncA$_L$: A DSL for Incremental Program Analysis with Lattices

Tamás Szabó
itemis, Germany /
TU Delft, The Netherlands
tamas.szabo@itemis.de

Markus Voelter
independent / itemis,
Germany
voelter@acm.org

Sebastian Erdweg
TU Delft, The Netherlands
s.t.erdweg@tudelft.nl

ABSTRACT

We describe IncA$_{\mathrm{L}}$, a DSL for incremental lattice-based program analyses. IncA$_{\mathrm{L}}$ is an extension of our previous work, IncA, which supported relational program analyses, that has been used for practically relevant analyses on industrial code bases. IncA$_{\mathrm{L}}$ improves the expressive power of IncA by adding support for synthesis of data, enabling, for example, incremental execution of interval analysis.

1. INTRODUCTION

Program analyses are at the core of refactorings and error checking in integrated development environments, and they are also fundamental for optimizations in compilers. On the one hand, program analyses should be precise (the various *-sensitivity properties, such as flow- and context-sensitivity), but on the other hand, they should also be fast and memory efficient. The literature documents a vast amount of approaches, that deliver *either* on precision [2] *or* on performance [1]; delivering on both is still a challenge.

In prior work [3], we introduced a domain-specific language (DSL) called IncA for defining efficient incremental program analyses that update their result as the subject program changes. IncA can express relational analyses, i.e., analyses that create relations between nodes in the abstract syntax tree (AST) of the subject program. We implemented several relevant analyses this way (e.g. control flow analysis, see later), and industrial experience and systematic experiments show that it scales to large code bases as documented in [3].

However, there are many analyses that require the synthesis of *new* data, and not just the creation of relationships between *existing* data. Examples include interval analysis or type state analysis. To fill this gap, in our current work, we created IncA$_{\mathrm{L}}$, an evolution of IncA, that supports lattice-based analyses. The choice for lattices is motivated by two reasons: (1) lattices directly represent varying precision through their structure and the definition of lattice operations, and (2) the monotonicity of lattice operations is crucial fixpoint computations that are common in program analyses frameworks (including IncA$_{\mathrm{L}}$).

In this paper, we document the evolution of IncA and IncA$_{\mathrm{L}}$ into an expressive and efficient incremental program analysis framework. We discuss key challenges that we have already solved, present the state of our current work around lattice-based analyses, and identify open challenges.

2. FROM IncA TO IncA$_{\mathrm{L}}$ AND BEYOND

The vexing trade-off between expressivity and performance governs the development of IncA: developers want increasing expressiveness in the language, , while, at the same time, retaining incrementality which is crucial for efficiency.

2.1 Past: Relational Analyses with IncA

Relational analyses rely on establishing relationships between AST nodes of the subject program. Graph patterns are a natural choice for representing such relations because they prescribe an expected structure on a graph. Efficient incremental graph pattern matching algorithms and libraries are available [4]. Typically, these algorithms allow *either* recursive graph patterns *or* subject programs with cycles but not both. However, practically relevant program analysis require both.

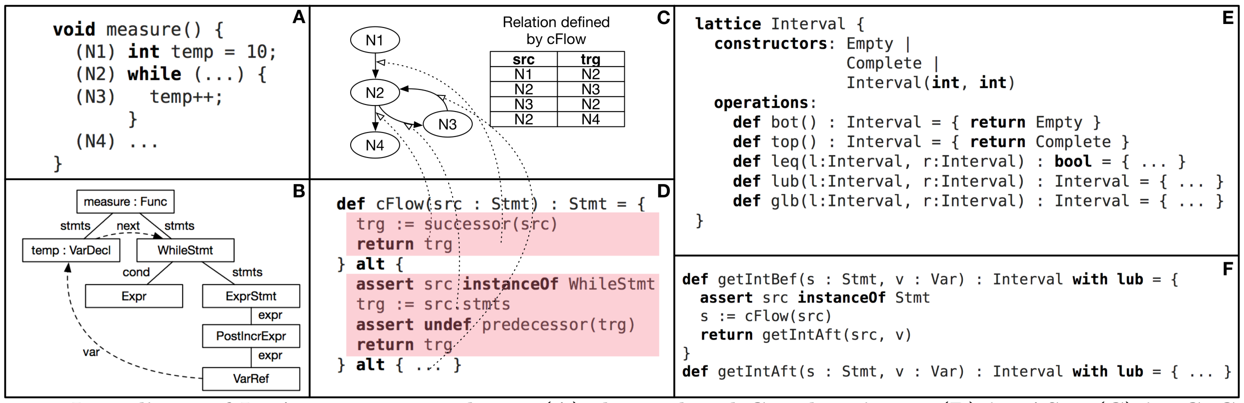
Thus, a key early challenge for IncA was to support recursive analyses on recursive programs in the face of both insertions and deletions. Our past work focused on developing IncA as a language, and the extension of an existing incremental graph pattern matching framework to support the language. As described in [3], our solution scales to industrial code bases, and supports several real-world analyses.

Example IDEs and compilers use control flow analysis to reason about the execution order of statements in a program and as a building block for further analyses such as the interval analysis described later. Fig 1 (B) shows the AST of the program in Fig 1 (A). Fig 1 (C) shows the control flow graph (CFG) of that program.

The IncA control flow analysis uses pattern functions to encode relations between AST nodes of the subject program. Fig 1 (D) shows the `cFlow` function that takes as input a node of type `Stmt` and returns another `Stmt`. A pattern function can have several alternative bodies that each encode a way of obtaining the output(s) from the input(s), thus defining a relation between the program nodes. For example, the second alternative derives the `N2-N3` edge by first navigating to the statements in the body of the `while`, and then returning the statement that has no predecessor (note the `undef` construct for negation) because control would first flow to the first statement in the loop body. In IncA, the result of a program analysis consists of tuples of a relation as shown in Fig 1 (C).

2.2 Current: Lattices in IncA$_{\mathrm{L}}$

IncA was limited to relating AST nodes because of the restricted expressivity of the IncA language, and because of missing capabilities in the runtime system. In turn, analyses that require the synthesis of new data could not be expressed, and thus could not benefit from IncA's incrementality. A prime example is an interval analysis that derives the potential ranges of values (the synthesized data) of program variables. Given the code snippet in Fig 1 (A), assuming



```
void measure() {
  (N1) int temp = 10;
  (N2) while (...) {
  (N3)   temp++;
       }
  (N4) ...
}
```

src	trg
N1	N2
N2	N3
N3	N2
N2	N4

```
def cFlow(src : Stmt) : Stmt = {
  trg := successor(src)
  return trg
} alt {
  assert src instanceOf WhileStmt
  trg := src.stmts
  assert undef predecessor(trg)
  return trg
} alt { ... }
```

```
lattice Interval {
  constructors: Empty |
                Complete |
                Interval(int, int)
  operations:
    def bot() : Interval = { return Empty }
    def top() : Interval = { return Complete }
    def leq(l:Interval, r:Interval) : bool = { ... }
    def lub(l:Interval, r:Interval) : Interval = { ... }
    def glb(l:Interval, r:Interval) : Interval = { ... }
}
```

```
def getIntBef(s : Stmt, v : Var) : Interval with lub = {
  assert src instanceOf Stmt
  s := cFlow(src)
  return getIntAft(src, v)
}
def getIntAft(s : Stmt, v : Var) : Interval with lub = { ... }
```

Figure 1: Ingredients of IncA$_{\mathrm{L}}$ program analyses: (A) the analyzed C code snippet, (B) its AST, (C) its CFG, (D) IncA code for control flow analysis, (E) definition of the interval lattice in IncA$_{\mathrm{L}}$, and (F) IncA$_{\mathrm{L}}$ code for interval analysis. The solid lines in (B) represent containment edges, while the dashed lines represent references to other AST nodes. The dotted lines between (C) and (D) show the alternative body that derives the respective CFG edge.

that the interval analysis does not know how many times the loop will be executed, it would associate with `temp` the $[10, 10]$ interval at `N1`, $[10, \infty)$ at `N2`/`N4`, and $[11, \infty)$ at `N3`.

Lattices naturally represent the varying precision of the synthesized data (e.g., the jump to ∞ is a reduction in precision). Analysis developers design the lattice structure to fit the precision requirement of a given use case (e.g., live error checking can accept lower precision as long as it is fast; a compiler optimization requires the opposite), and define lattice operations (least upper bound, greatest lower bound, fixpoint accelerators such as widening) in a way that obeys the desired degree of over-/under-approximation. For example, for the interval analysis, one may decide to allow a specific number of loop iterations, and, if the analysis does not converge to a fixpoint interval during those iterations, then widen it to the top of the lattice.

IncA$_{\mathrm{L}}$ adds user-definable lattices to IncA; the key challenge here is to support aggregate computations on synthesised data in the presence of recursive analyses and subject programs (shown below). We exploit the monotonicity of the lattice operators for fixpoint-based computations.

Example Fig 1 (E) shows the `Interval` lattice expressed in IncA$_{\mathrm{L}}$. Fig 1 (F) shows a part of the interval analysis for C in IncA$_{\mathrm{L}}$. It consists of two recursively dependent pattern functions `getIntBef` and `getIntAft`. `getIntBef` takes a `Stmt s` and a `Var`, and returns `Interval` that holds the potential range of values for the variable before `s`. `getIntAft` returns the interval after `s`. `getIntBef` uses the previously shown `cFlow` function to obtain the control flow predecessor(s) for `s`, and it returns the interval that was assigned to the given variable after the execution of the predecessor(s) as computed by `getIntAft`. The potential for having multiple CFG predecessors leads us to the requirement for aggregation: instead of tracking individual intervals, we typically want to combine them based on the lattice's `lub` or `glb` operator. For instance, the initial interval for `temp` was $[10, 10]$, and, after the first evaluation of the loop body, we derived a new interval $[11, 11]$. This is propagated back to the loop head through the `N3-N2` CFG edge. We now have to aggregate these two intervals, leading to $[10, 11]$. In IncA$_{\mathrm{L}}$, the aggregation is controlled by annotations on lattice types as shown in Fig 1 (F). Both functions use the `lub` annotation which means that the runtime system uses the least upper bound operator to aggregate intervals. This example shows that IncA$_{\mathrm{L}}$ is capable of expressing lattice-based analyses and incrementalize their evaluation. However, the degree of incrementality is suboptimal as explained next.

2.3 Future: Deltas Between Lattice Values

So far, the unit of incrementalization in IncA$_{\mathrm{L}}$ was the tuple of a relation (Fig 1 (C)); the increments in the analysis result were represented as insertions and deletions of whole tuples. Consider now the tuples associated with the loop body: `(N3, temp,` $[10, 10]$`)`, `(N3, temp,` $[10, 11]$`)`, ..., `(N3, temp,` $[10, \infty)$`)`. Only the lattice value changes! However, IncA$_{\mathrm{L}}$ represents the increments as a delete of the previous tuple and an insert of the new tuple. This prevents incremental reuse of previous computations. If we change the initializer at `N1` from `10` to `11`, IncA$_{\mathrm{L}}$ cannot reuse results from previous computations because, not understanding deltas between lattice values, it treats them as individual values without any relationship to one another. Consequently, IncA$_{\mathrm{L}}$ will perform a re-analysis of the loop. In our future work we will work on automatically deriving the delta representation for a lattice, leading to better incremental reuse of previous computations, and, in turn, better performance.

3. TALK OUTLINE

The talk will describe the evolution of IncA and IncA$_{\mathrm{L}}$ with real-world examples. We first give a tour of the IncA DSL, its capabilities in comparison to other tools, and our performance benchmarks. Then, we show the algorithmic challenges of incrementalizing lattice-based computations in IncA$_{\mathrm{L}}$. Finally, we highlight the research challenge of identifying the delta representation between lattice values in more detail.

4. REFERENCES

[1] A. Egyed. Instant Consistency Checking for the UML. ICSE, 2006.

[2] Y. Smaragdakis and M. Bravenboer. Using Datalog for Fast and Easy Program Analysis. Datalog, 2011.

[3] T. Szabó, S. Erdweg, and M. Voelter. IncA: A DSL for the Definition of Incremental Program Analyses. ASE, 2016.

[4] Z. Ujhelyi, G. Bergmann, Ábel Hegedüs, Ákos Horváth, B. Izsó, I. Ráth, Z. Szatmári, and D. Varró. EMF-IncQuery: An integrated development environment for live model queries. *SCP*, 2015.