Real-Time Feedback through Incremental Program Analysis

Tamás Szabó
itemis AG / TU Delft
tamas.szabo@itemis.de

Sebastian Erdweg
TU Delft
s.t.erdweg@tudelft.nl

Markus Voelter
independent / itemis AG
voelter@acm.org

ABSTRACT

Program analyses support software developers, for example, through error detection, code-quality assurance, and by enabling compiler optimizations and refactorings. To provide real-time feedback to developers within IDEs, an analysis must run efficiently even if the analyzed code base is large.

To achieve this goal, we present a domain-specific language called IncA for the definition of *efficient incremental program analyses* that update their result as the program changes. IncA compiles analyses into graph patterns and relies on existing incremental matching algorithms. To scale IncA analyses to large programs, we describe optimizations that reduce caching and prune change propagation. Using IncA, we have developed incremental control flow and points-to analysis for C, well-formedness checks for DSLs, and 10 FindBugs checks for Java. Our evaluation demonstrates significant speedups for all analyses compared to their non-incremental counterparts.

1. INTRODUCTION

Static program analysis is the basis of compiler optimizations and IDE features such as error detection or behavior-preserving refactorings. Program analyses trade off precision for runtime performance and memory use, and there is a large body of research on techniques that enable increasingly precise and efficient analyses. For example, by giving up flow-sensitivity and context-sensitivity, points-to analysis can analyze millions of lines of Java code in under a minute [8].

Our work improves the performance of program analyses through incrementality: When part of the code changes (for example, through user edits), we only reanalyze the changed part, plus all the code whose analysis result depends on the changed results. This way, incremental program analysis can provide significant improvements compared to reanalyzing the whole code base from scratch. Such incremental analyses are useful in IDEs that perform real-time analysis upon user edits and in continuous integration servers that continuously analyze an evolving code base.

© 2016 ACM. ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

We present IncA, a domain-specific language (DSL) for the definition of efficient incremental program analyses, as well as an optimizing compiler and a runtime system for IncA. Conceptually, IncA represents computations as *graph patterns* [27] on top of the abstract syntax tree (AST) of the analyzed program. Graph patterns express relationships between AST nodes, for example, to describe the control flow between individual statements of the analyzed program. The IncA compiler translates a user-defined program analysis into a set of interconnected graph patterns. The IncA runtime system maintains the analysis results incrementally by performing incremental graph pattern matching.

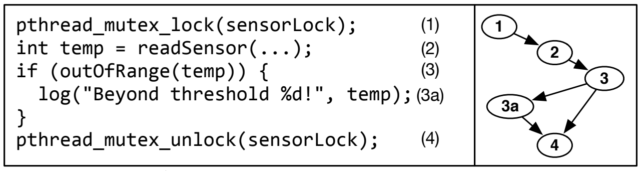
Semantically, a graph pattern describes a set of tuples that relate program entities. However, programming with graph patterns is difficult for many developers, because instead of mapping input to output, they construct tuples for all related entities by splitting, joining and filtering sets of tuples. IncA introduces *pattern functions* to abstract from graph patterns: Instead of operating on sets, a pattern function takes a single input and either rejects it or computes a corresponding output. This way, a pattern function defines what we call the primary linearization of the underlying graph pattern, for example, in the style of a forward or backward analysis. Our compiler translates pattern functions into regular graph patterns.

We have implemented IncA on top of JetBrains MPS,[1] an IDE that relies on projectional editing, where code changes occur in the form of user-issued AST change requests. This aligns well with incrementalization because each AST change triggers an incremental update of the analysis results. The design and implementation of IncA is independent of the analyzed language and we developed a generic architecture for the integration of IncA into other IDEs.

To evaluate IncA, we implemented incremental analyses for C and Java programs and measured their runtime performance. For C, we developed incremental control flow analysis, flow-sensitive points-to analysis, and domain-specific well-formedness checks. For Java, we reimplemented 10 FindBugs analyses [16]. Our evaluation shows that IncA-based incremental program analyses yield significant speedups without introducing unacceptable memory or initialization overheads. In summary, we make the following contributions:

- We introduce IncA, a language for the definition of incremental program analyses that is independent of the analyzed language. Our compiler translates IncA code into graph patterns (Section 3).
- We implement compiler optimizations for IncA that

[1] https://www.jetbrains.com/mps



```
pthread_mutex_lock(sensorLock);          (1)
int temp = readSensor(...);              (2)
if (outOfRange(temp)) {                  (3)
  log("Beyond threshold %d!", temp);     (3a)
}
pthread_mutex_unlock(sensorLock);        (4)
```

Figure 1: A simple C program and its control flow

reduce the memory required for incrementalization and the time required for change propagation (Section 4).

- We describe the runtime system of IncA as a generic architecture that allows the integration of IncA analyses into different IDEs. Technically, the runtime system employs incremental graph pattern matching on the AST of the analyzed program (Section 5).
- We develop three incremental analyses for C, including an incremental flow-sensitive points-to analysis, and reimplement 10 FindBugs analyses for Java (Section 6).
- We evaluate the memory and runtime performance of IncA through extensive case studies on real-world C and Java software projects (Section 7). We show that program analyses developed with IncA provide real-time feedback and scale to large code bases.

We reuse the IncQuery [33] incremental evaluator in our system; thus the evaluator is not our contribution. There exist several incremental algorithms for a particular analysis [6, 22, 14, 23] and frameworks for a particular class of analyses [2, 11, 37] in the literature. In contrast to these solutions, our analysis framework is not bound to a specific class of analyses (Section 6) and employs incremental graph pattern matching for program analysis as a novel approach.

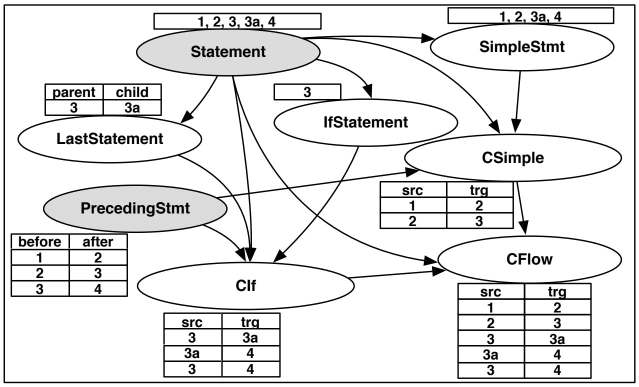
2. INCREMENTAL PROGRAM ANALYSIS BY EXAMPLE

Using control flow analysis for C as an example, this section explains the problem of incremental program analysis and illustrates our solution.

Control flow analysis 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 points-to analysis.

The input of control flow analysis is the AST of the program and its output is a control flow graph (CFG) [21]. As an example, consider the C program and its control flow in Figure 1. Each node in the CFG represents a statement in the program. A source statement is connected to a target statement in the CFG if there exists an execution trace in which the execution of the source statement immediately precedes the execution of the target statement. Note how control can flow from statement 3 both to 3a and 4 in Figure 1, depending on the condition of the if statement, which is why the CFG contains edges from 3 to 3a and from 3 to 4.

The result of our control flow analysis is a set of CFG edges. In this section, we restrict ourselves to a subset of C with if statements and simple statements that entail a sequential control flow such as assignments. We define the relation CFlow for the edges of the CFG and we compute it as the union of two helper relations. (1) **CSimple** consists of those control flow edges (src,trg) where src is a simple statement that syntactically precedes trg in the source code. (2) **CIf** consists of control flow edges (src,trg) that lead into or out of an if statement or its branches, which is the case if (a) src syntactically precedes trg and src is an if statement without an else part, (b) src is an if statement and trg is the first statement of one of its branches, or (c) src is the last statement of one of the branches of an if that syntactically precedes trg.



1, 2, 3, 3a, 4

1, 2, 3a, 4

3

parent	child
3	3a

before	after
1	2
2	3
3	4

src	trg
1	2
2	3

src	trg
3	3a
3a	4
3	4

src	trg
1	2
2	3
3	3a
3a	4
3	4

Figure 2: Computation graph of the CFG analysis evaluation

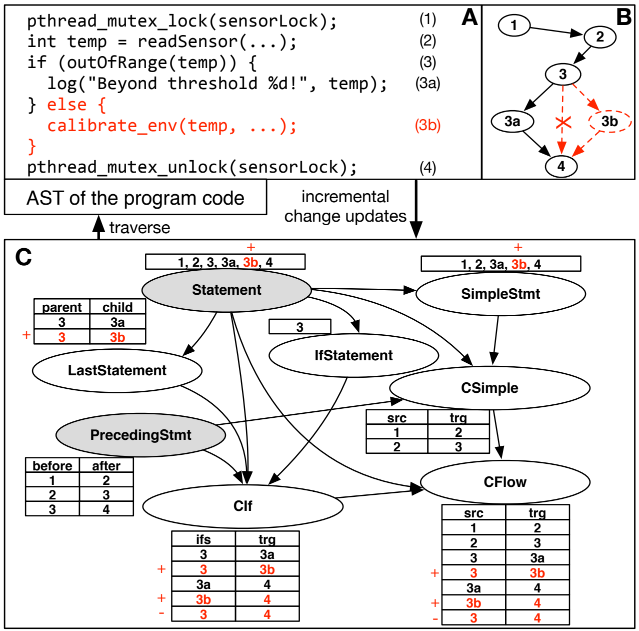
In our example, CSimple has two elements (1,2) and (2,3), while CIf has three elements (3,3a), (3a,4), and (3,4). The union of the two relations constitutes CFlow.

Evaluation with graph patterns To compute the elements of the relations CSimple, CIf, and CFlow, we need to traverse the AST and discover the relevant structural relations between the AST nodes and their attributes. A natural representation of such computations is a *graph pattern* [27]. Much like our informal description of the relations above, a graph pattern describes sets of related entities through structural constraints on AST nodes and on instances of other graph patterns.

Given a set of interconnected graph patterns, we can compute their results by using a *computation graph* as illustrated in Figure 2 for our example. A computation graph consists of two kinds of nodes, each of which yields a set of related entities. *Input nodes* (grey) represent the AST structure directly, do not perform any computation, and do not depend on any other node. In Figure 2, Statement and PrecedingStatement are the only input nodes; the former enumerates the nodes of type Statement from the AST and the latter enumerates the pairs of statements where the first statement syntactically precedes the other. *Computation nodes* (white) use the results of input nodes and other computation nodes to relate program entities. For example, node CSimple uses information from node Statement, SimpleStatement, and PrecedingStatement to identify statements related through simple control flow. Node CFlow combines the results from two computation nodes to produce the complete CFG.

Incremental control flow analysis We encode program analyses as graph patterns and computation graphs because this provides a good basis for incrementalizing the computation. Suppose the user modifies the analyzed program and adds an else branch 3b to the if statement as illustrated in Figure 3. This invalidates the old CFG. Using graph patterns and computation graphs, instead of recomputing a new CFG from scratch, we can *incrementally update* the existing CFG.

To this end, computation graphs use memoization and perform incremental graph pattern matching: When code gets changed, we send change events to the input nodes of the computation graph. The nodes then transitively propagate changes to all dependent computation nodes and trigger the



```
pthread_mutex_lock(sensorLock);            (1)
int temp = readSensor(...);                (2)
if (outOfRange(temp)) {                    (3)
  log("Beyond threshold %d!", temp);       (3a)
} else {
  calibrate_env(temp, ...);                (3b)
}
pthread_mutex_unlock(sensorLock);          (4)
```

Statement: 1, 2, 3, 3a, 3b, 4 (+ 3b)

SimpleStmt: 1, 2, 3a, 3b, 4 (+ 3b)

IfStatement: 3

	parent	child
	3	3a
+	3	3b

before	after
1	2
2	3
3	4

src	trg
1	2
2	3

	ifs	trg
	3	3a
+	3	3b
	3a	4
+	3b	4
-	3	4

	src	trg
	1	2
	2	3
	3	3a
+	3	3b
	3a	4
+	3b	4
-	3	4

Figure 3: Example analysis evaluation; (A) example C program code after modification, (B) CFG of the program code, (C) computation graph for incremental evaluation

reanalysis of changed program entities. This way, we avoid any reanalysis of unchanged parts of the program.

In our example, the introduction of the `else` branch triggers the following changes in the computation graph (as shown in Figure 3):

- Addition of `3b` to `Statement` and `SimpleStatement`.
- Removal of `(3,4)` from `CIf` because `if` statement `3` now has an `else` branch. That is, the conditional treatment is exhaustive and control is guaranteed to pass through one of the branches before reaching `4`.
- Addition of `(3,3b)` to `CIf` because `3b` is the first statement of the `else` branch.
- Addition of `(3,3b)` to `LastStatement` because `3b` is the last statement of `else` branch of `if` statement `3`.
- Addition of `(3b,4)` to `CIf` because `(3,3b)` was added to `LastStatement` and `3` precedes `4`.
- Propagation of all changes from `CIf` to `CFlow`.

In our implementation based on projectional editing, we receive change events for user-issued AST changes directly from the IDE. Fine-grained AST change notifications align perfectly with incrementalization and there is no need for a potentially costly parsing step to compute AST differences. After an AST change, a projectional editor derives a new projection from the AST and displays it to the programmer.

A DSL for program analysis A DSL for program analysis based on graph patterns is a good semantic basis for incrementally analyzing programs. However, programming with graph patterns is difficult because graph patterns operate on sets of related program entities, using operations such as filter, map, and cross product. This is far removed from the usual way of defining program analyses as forward or backward analyses [21]. Additionally, considerable anecdotal evidence at company DB1 and DB2[2], where developers use the IncQuery graph pattern language [33] for commercial projects, shows that developers would rather rely on more familiar core abstractions; functions, distinguishing input from output, direction in the function body and assignments. This experience was the main driver when we designed a DSL called IncA which abstracts from graph patterns.

```
def cFlow(trg : Statement) : Statement = {
  return cSimple(trg)
} alt {
  return cIf(trg)
}

def cSimple(trg : Statement): SimpleStatement ={
  src := precedingStatement(trg)
  assert src instanceOf SimpleStatement
  return src
}

def cIf(trg : Statement): Statement = {
  src := precedingStatement(trg)
  assert src instanceOf IfStatement
  return lastStatement(src)
} alt {
  src := precedingStatement(trg)
  assert src instanceOf IfStatement
  assert undef src.else
  return src
} alt {
  assert undef precedingStatement(trg)
  parent := trg.parent
  assert parent instanceOf IfStatement
  return parent
}
```

Figure 4: Control flow analysis for a subset of C in IncA

Instead of operating on sets of program entities, IncA supports the definition of program analyses using what we call *pattern functions*. A pattern function is a linearized graph pattern: It takes a single tuple of inputs and either rejects the input or produces a single tuple of outputs through a sequence of statements in the body of the pattern function. Semantically, a pattern function corresponds to a graph pattern that relates the program entities that occur as either input or output. In particular, the separation of entities into input and output and the order of statements in the body of a pattern function are semantically irrelevant. Nevertheless, linearization simplifies the definition of program analyses. Since many different linearizations of the same graph pattern are possible, we say a pattern function defines the *primary linearization* of the underlying graph pattern. We illustrate IncA through the definition of the control flow analysis for our subset of C, where the linearization enables us to write the definition in the style of a backward analysis.

Figure 4 shows the IncA code that defines the control flow analysis. We show three pattern functions that follow the informal description of the corresponding relations above. Function `cFlow` takes a target statement `trg` and finds and returns the control flow predecessors of `trg` using functions `cSimple` and `cIf`. We write **alt** to provide alternative results for a pattern function.

Function `cSimple` queries pattern `precedingStatement` for `trg`. The function can return multiple results, but IncA abstracts over this. Next, `cSimple` asserts that the predecessor of `trg` is indeed a simple statement. If it is, `cSimple` yields this statement as a control flow predecessor. Otherwise, `cSimple` fails for `trg` and does not yield any output.

Function `cIf` is composed of three alternatives that com-

[2] information removed due to double blind review

$$
\begin{array}{lll}
\text{(module)} & m & ::= \textbf{module}\ n\ \textbf{import}\ \overline{n}\ \{\overline{f}\} \\
\text{(function)} & f & ::= \mathit{vis}\ \textbf{def}\ n(\overline{n : T}) : \overline{T} = \overline{a} \\
\text{(visibility)} & \mathit{vis} & ::= \textbf{private} \mid \textbf{public} \\
\text{(alternative)} & a & ::= \overline{s} \\
\text{(statement)} & s & ::= \overline{n} := e \mid \textbf{assert}\ c \mid \textbf{return}\ e \\
\text{(condition)} & c & ::= e == e \mid e\ != e \mid e\ \textbf{instanceOf}\ T \mid \\
 & & \quad\ \ e\ \textbf{not instanceOf}\ T \mid \textbf{undef}\ e \\
\text{(expression)} & e & ::= n \mid l \mid e.n \mid n(\overline{e}) \mid n^{+}(e) \\
\text{(literal)} & l & ::= \text{number} \mid \text{string} \mid \text{enum} \mid \text{boolean} \\
\text{(type)} & T & ::= \text{AST node type} \\
\text{(name)} & n & ::= \text{name}
\end{array}
$$

Figure 5: Syntax of IncA

pute the control flow when an `if` statement is involved. IncA provides direct access to the AST structure. For example, in the second alternative, `src.else` represents the `else` branch of the if statement `src`. As shown by `cIf`, IncA also supports the language construct **undef** to ascertain that a given pattern-function call does not yield any result or that an AST node is undefined.

Our compiler translates pattern functions into regular graph patterns, but also analyzes the pattern functions to perform optimizations. In particular, our compiler determines which parts of an AST are irrelevant for an analysis and uses this information to prune the propagation of change notifications and to reduce caching (Section 4).

3. INCREMENTAL PROGRAM ANALYSIS WITH IncA

In this section we discuss IncA in greater detail. Specifically, we describe the syntax of IncA and explain how we translate its syntactic constructs into graph patterns.

3.1 Syntax of IncA

Figure 5 shows the syntax of IncA. We write $\overline{a}$ for a sequence of a elements, which includes the empty list.

A *module* groups related pattern functions. Modules can import other modules to gain access to their public pattern functions. Functions are public by default, but can be marked private to restrict their visibility to the containing module.

A *pattern function* represents the primary linearization of a graph pattern. A function takes a tuple of typed input parameters and either rejects the input or produces an output tuple. A function may define multiple *alternative* linearizations for the same input.

Each alternative consists of a sequence of *statements*, ending with a return statement that defines the output tuple. An assignment statement binds variables to the components of a tuple. An assertion statement aborts the computation and rejects the current input if the condition fails. As *conditions* we support equality, inequality, AST node-type membership, AST node-type exclusion, and undefinedness testing.

An *expression* can refer to a variable, represent a value literal, refer to a property of an AST node, call another pattern function, or compute the transitive closure of another pattern function. For the transitive closure, the called pattern function must take a single input and provide a single output. The evaluation of such expressions yields all intermediate outputs, but IncA's syntax abstracts over this and permits the use as if there was only a single definite output.

3.2 Compilation to Graph Patterns

The theory of graph patterns is well established [27] and we define the semantics of IncA through translation to graph patterns. A graph pattern consists of pattern variables and constraints over these variables. The constraints can refer to other graph patterns. We use the following constraints:

- `Entity(v,T)` holds if variable `v` has type `T`.
- `Relation(l,v1,v2)` holds if there is an edge labeled `l` between variables `v1` and `v2`.
- `Eq(v1,v2)` and `Neq(v1,v2)` hold if variables `v1` and `v2` point to the same/different element.
- $\texttt{PC(p,}\overline{\texttt{v}}\texttt{)}$ and $\texttt{NPC(p,}\overline{\texttt{v}}\texttt{)}$ hold if the pattern `p` accepts/rejects the tuple $\overline{\texttt{v}}$.
- `TC(p,v1,v2)` holds if the transitive closure of the binary pattern `p` contains the tuple `(v1,v2)`.
- $\texttt{Alt(}\overline{\texttt{p}}\texttt{,}\overline{\texttt{v}}\texttt{)}$ holds if any of the patterns in $\overline{\texttt{p}}$ accepts the tuple $\overline{\texttt{v}}$.

We map IncA language constructs to graph-pattern constraints as follows, starting with expressions. An IncA variable becomes a pattern variable and a literal becomes a pattern variable with an `Eq` constraint. A property access `e.n` translates to a constraint `Relation(n,e,v)`, where `v` is a fresh pattern variable that represents the result of the lookup. A function call $\texttt{f(}\overline{\texttt{e}}\texttt{)}$ becomes a constraint $\texttt{PC(f,}\overline{\texttt{e}}\,\overline{\texttt{v}}\texttt{)}$, where $\overline{\texttt{v}}$ are fresh and represent the result of the call (the notation $\overline{\texttt{e}}\,\overline{\texttt{v}}$ means concatenation). For transitive closure, we use the `TC` constraint.

Next, we translate IncA conditions to constraints. Equality and inequality straightforwardly translate to `Eq` and `Neq` constraints, and node-type membership test with `instanceof` becomes an `Entity` constraint. Negative conditions require helper patterns and calls to them with `NPC`. The helper function contains an `Entity` for a `not instanceOf` while it contains the corresponding subpattern for the expression in case of an `undef`. The function call with an `undef` is an exception, because it is directly translated to an `NPC`.

For statements, we proceed as follows. An assertion simply promotes the constraints of its condition. An assignment matches up the variables on the left-hand side with the variables that result from the expression on the right-hand side. We generate `Eq` constraints for the pairs of variables from the two sides. To represent the output tuple of a function, we use designated variables. We handle a return statement as an assignment to these variables.

Finally, we collect the functions from a module and its transitively imported modules and generate graph patterns for the functions and their alternatives. Function input parameters become pattern variables $\overline{\texttt{v}_i}$ and we create designated pattern variables $\overline{\texttt{v}_o}$ for the output tuple. The type annotations on input and output of a function become `Entity` constraints over $\overline{\texttt{v}_i}\,\overline{\texttt{v}_o}$. We combine the patterns $\overline{\texttt{p}}$ of a function's alternatives using an alternative constraint $\texttt{Alt(}\overline{\texttt{p}}\texttt{,}\overline{\texttt{v}_i}\,\overline{\texttt{v}_o}\texttt{)}$.

This translation process also shows the driving forces for the abstraction from graph patterns. Graph patterns are verbose, because they require explicit variables for *all* intermediate expressions, explicit `Entity` for type constraints and helper patterns for negative conditions. Additionally, they are nondirectional which is in contrast to the usual directional nature of program analyses.

To achieve incrementality, we reuse the incremental graph pattern-matching implementation that is part of IncQuery [33], but there are also many other alternatives for matching [27]. The expressive power of our program analysis language (and of the core IncQuery engine) is classified as FO(LFP) [25, 17] which stands for First Order logic extended with the Least Fixed Point operator. We believe that this formalism is expressive enough for a wide variety of program analysis as we demonstrate in Section 6.

4. COMPILER OPTIMIZATIONS FOR IncA

The performance of IncA depends on both the memory required for caching as well as the efficiency of change propagation in the computation graph. In this section, we describe compiler optimizations for IncA that improve both of them.

Our approach for performance improvement relies on the observation that the evaluation of a program analysis usually only depends on a relatively small part of an AST. For incremental analysis, this means that many AST changes do not affect the analysis result and can be safely ignored. We can use this observation to improve caching and change propagation. There is no need to write a changed element to the input nodes of the computation graph if it is known to be irrelevant for the analysis. This saves both memory and time, because by discarding irrelevant input nodes we avoid subsequent change propagation in the computation graph, which in turn also avoids caching of unneeded results.

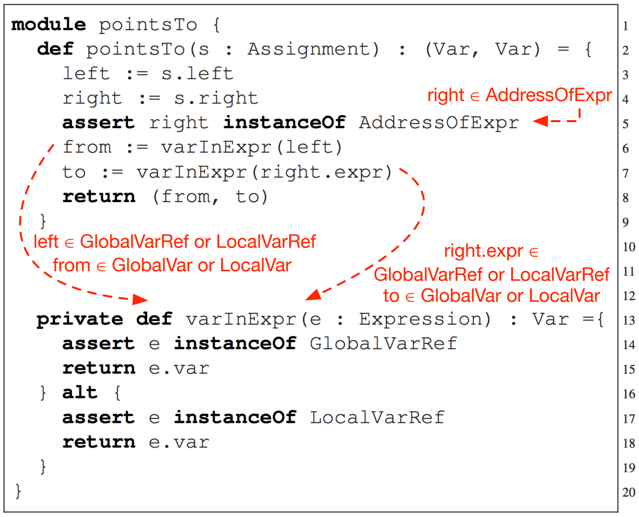
To make use of this observation in practice, we must be able to distinguish relevant from irrelevant changes. To this end, we have developed an analysis for IncA programs, that is, an *analysis of the program analysis* code. Our analysis inspects the IncA code to compute a conservative approximation of all changes that can affect the result of the IncA program. We analyze each module of the IncA program with its transitive imports separately because the IncA compiler creates one computation graph for each module.

As a first approximation of the relevant changes, we can collect the *declared types* of all mentioned AST nodes from the functions. For example, consider again the control flow analysis in Figure 4. The IncA code refers to AST nodes of type `Statement`, `SimpleStatement`, `IfStatement`, and `ElsePart` (via `src.else`). Since type `Statement` is a supertype of the other two statement types, a sound approximation is given by the set of changes that modify nodes of type `Statement` or `ElsePart`. Accordingly, when incrementally executing the control flow analysis in the IncA runtime system, we can ignore changes to expressions, function declarations, and others. Note that the type information is not an additional assumption about the analysis, it is part of the IncA code. The subtyping relationship between the AST node types is determined by the analyzed program's language.

Using the declared types of AST nodes is a good starting point for optimization; however it is rather imprecise. To improve the effectiveness of the optimization, we can take type assertions (`instanceOf`) into account to determine the actually relevant AST node types. Consider an excerpt of a points-to analysis for C shown in Figure 6. Function `pointsTo` computes pairs of C variables for assignments of the form `u = &v`. It uses function `varInExpr` to reject expressions that are not variables (such as additions or multiplications) and otherwise to extract the variable of an expression.

If we only consider the declared types and ignore type assertions, we have to assume that a change to any node of type `Assignment`, `Var`, or `Expression` affects the analysis result. However, we can apply the following reasoning to narrow the set of relevant changes in Figure 6:

- Initially, both variables `left` and `right` have type `Expression` according to the properties of `Assignment`.
- The type assertion at line 5 constrains the type of `right` to the more specific type `AddressOfExpr`.
- The function call in line 6 calls `varInExpr` with a reference to the variable `left`. Function `varInExpr` has two alternatives, the first one restricting `e` to `GlobalVarRef`, while the second one restricts it to `LocalVarRef`. Expressions that satisfy neither restriction are rejected by `varInExpr`. So the call in line 6 only succeeds, and thus contribute to the analysis result, if `left` is a `GlobalVarRef` or a `LocalVarRef`. Moreover, the type of variable `from` must be `GlobalVar` or `LocalVar`.
- The same reasoning applies to the function call at line 7 for `right.expr` and variable `to`.



```
1  module pointsTo {
2    def pointsTo(s : Assignment) : (Var, Var) = {
3      left := s.left
4      right := s.right
5      assert right instanceOf AddressOfExpr
6      from := varInExpr(left)
7      to := varInExpr(right.expr)
8      return (from, to)
9    }
10
11
12
13   private def varInExpr(e : Expression) : Var ={
14     assert e instanceOf GlobalVarRef
15     return e.var
16   } alt {
17     assert e instanceOf LocalVarRef
18     return e.var
19   }
20 }
```

right ∈ AddressOfExpr
left ∈ GlobalVarRef or LocalVarRef
from ∈ GlobalVar or LocalVar
right.expr ∈
GlobalVarRef or LocalVarRef
to ∈ GlobalVar or LocalVar

Figure 6: Identifying relevant AST node types in IncA

This reasoning shows that we only need to observe changes to statement nodes of type `Assignment`, to expression nodes of type `AdressOfExpr`, `GlobalVarRef`, and `LocalVarRef` and to variable nodes of type `GlobalVar` and `LocalVar`. Considering that, for example, the mbeddr dialect of C with its language extensions has more than 200 different kinds of expressions, our analysis can yield significant improvements for the memory and runtime performance of an IncA analysis. We empirically confirm this in Section 7.

We now provide a detailed description of our interprocedural data-flow analysis for IncA programs:

Traverse call chains We perform a depth-first traversal over the pattern-function call graph, starting at publicly visible functions. The analysis is interprocedural, that is, we pass type information from the caller to the callee. This means that different call chains may result in different type constraints for parameters and variables; thus we analyze all call chains separately.

Type intersection During traversal, for each alternative of a pattern function, we collect the type constraints from assertions and function calls for each parameter and local variable. An alternative can only succeed if each parameter and variable satisfies all type constraints. This corresponds to constructing the intersection of all type constraints for each parameter and variable.

Type union A pattern function succeeds if at least one of its alternatives succeeds. Thus, it is sufficient if the parameters satisfy the type constraints of at least one alternative. We approximate this by constructing the union of the type

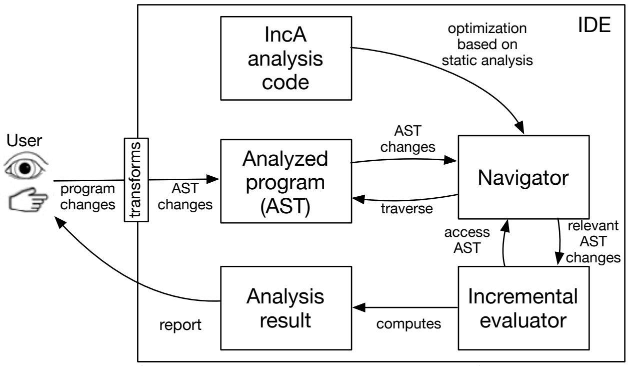


Figure 7: Architecture for integrating IncA into IDEs

constraints for parameters and variables from all alternatives.

Optimization The analysis result is the union of the type constraints for the different call chains. Only changes that affect nodes of these types can affect the analysis result. We prune the propagation of irrelevant changes, thus avoiding the computation and caching of irrelevant pattern matches.

As opposed to using a library for program analysis, an advantage of specifying the program analysis as DSL code is that the analysis itself becomes analyzable which enables our optimization. Additionally, our optimization is generic as it is not bound to the IncA DSL, for example, *it could be applied on the graph patterns* as well.

5. TECHNICAL REALIZATION AND IDE INTEGRATION

This section elaborates on the architecture of IncA's runtime system. We identify the components that enable the integration of incremental program analyses into IDEs and explain their interactions.

5.1 Overview

Figure 7 shows the architecture for integrating IncA into an IDE. The analysis code is an IncA program that we run on the analyzed program. As we detail in Section 5.3, our architecture requires that the IDE translates user edits into corresponding AST change notifications, which trigger the incremental analysis.

The navigator is the entry point of the incremental analysis and gets notified about AST changes by the IDE. It acts as an adapter between the IDE and the incremental evaluator. In particular, it allows the evaluator to traverse the input AST during the initialization of the computation graph. Later, when the navigator receives an AST change notification, it notifies the incremental evaluator about *relevant* AST changes, as determined by our compiler optimization (Section 4). Since the navigator knows the IDE-internal AST representation, it constitutes a language-independent but IDE-specific component.

The incremental evaluator is responsible for the incremental maintenance of the analysis results. The component is independent of the IDE and of the analyzed language. The evaluator uses the navigator to navigate in the AST and to receive notifications about AST changes; this way, the evaluator does not depend on the internals of the IDE.

The incrementally maintained results of a program analysis can be used to inform the user about new errors or as part of a refactoring in the IDE. This connection closes the loop between the user and our system and shows how incremental analysis supports interactive development.

5.2 Implementation for MPS

We instantiated the architecture for the Meta Programming System (MPS) using IncQuery [33] as the incremental evaluator. Both tools are available as open-source software, as is our implementation.[3]

MPS is an IDE that uses *projectional editing* [36] instead of a parser-based approach. When editing the program in a projectional editor, every user edit (for example, inserting an operator) directly corresponds to an AST change. After an AST change, a projectional editor renders new projection from the changed AST based on projection rules of the AST nodes and displays it to the programmer. Projectional editing is well-suited for incremental program analysis because the user's edits directly correspond to incremental AST changes and no incremental parsing is necessary.

Our system reuses the incremental graph pattern matching component of IncQuery. This component realizes the computation graph presented in Section 2. The component expects the graph patterns to be specified using the Java API PSystem.[4] We implemented the IncA compiler to translate graph patterns into a PSystem specification. After startup, IncQuery uses the navigator to initialize its computation graph and to retrieve AST changes.

5.3 Applicability in other IDEs

The applicability of our solution is ultimately determined by the granularity of an IDE's incremental change notifications as the analyzed program changes. We see three kinds of IDEs where our solution can be applied efficiently. (1) Projectional IDEs (like MPS) where code manipulations directly correspond to incremental AST change events. (2) Graphical IDEs, which, like projectional IDEs, also directly manipulate structures and can thus easily derive AST changes after a change. (3) Finally, textual IDEs that are backed by an incremental parser (cf. [24] for example incremental parsers and the Eco language workbench [7] which relies on an incremental parser). In this case, the degree of incrementality that our approach can achieve depends on the granularity of the incremental AST differences the parser can provide.

6. CASE STUDIES

To validate our approach, we have used IncA to implement three program analyses for mbeddr C and one program analysis for Java. mbeddr C [35] is an extensible C dialect and IDE for embedded software built on top of MPS. This section describes the program analyses and gives details about their implementation, while Section 7 contains the performance evaluation. We show only the most interesting parts of the implementations; the full implementation is available online.[5]

6.1 Control Flow Analysis

The introductory example in Section 2 already gave an intuition about control flow analysis. The incremental construction of a control flow graph (CFG) is an important building block for incremental, flow-sensitive analyses such as the flow-sensitive points-to analysis described next. These two

[3] information removed due to double blind review

[4] https://wiki.eclipse.org/EMFIncQuery/DeveloperDocumentation/PSystem

[5] information removed due to double blind review

$$\frac{}{u \rightarrow v}\; u = \&v \quad (1) \qquad \frac{x \rightarrow v}{u \rightarrow v}\; u = x \quad (2) \qquad \frac{x \rightarrow y,\; y \rightarrow v}{u \rightarrow v}\; u = {}^{*}x \quad (3) \qquad \frac{x \rightarrow u,\; y \rightarrow v}{u \rightarrow v}\; {}^{*}x = y \quad (4)$$

Figure 8: Andersen's rules for points-to analysis

analyses combined enable further precise analyses such as uninitialized read and unused assignment analysis.

We implemented an incremental control flow analysis that handles all of mbeddr C, including conditionals (`if` and `switch`), loops (`for`, `while`, and `do while`), and jumps (`break` and `continue`). The implementation follows the style of the introductory example, extending the `cFlow` relation with further alternatives to handle all control statements. The complete control flow analysis produces a CFG where the nodes not only represent statements of the program, but also other control flow points like the alternative `case` branches of a `switch` statement or the `else if` parts of an `if` statement. To this end, we defined an interface `ICFGNode` as supertype for all nodes that can appear in the CFG. We use this type in the points-to analysis as well.

6.2 Points-to Analysis

Our second case study is a points-to analysis for mbeddr C. Given a variable that stores a pointer, the goal of a points-to analysis is to identify the possible targets of the variable. There is a vast amount of research in this area [19, 29, 38], because the precision of points-to analysis directly benefits optimizations such as lock elision in a concurrent system and program analyses such as uninitialized read analysis.

We represent the result of a points-to analysis as a relation `PointsTo(from,to)`, which consists of those tuples where both `from` and `to` are variables and `from` potentially points to `to` at runtime. A points-to analysis is sound if `PointsTo` is a conservative approximation of the actual targets of variables in the program. A well-known algorithm for computing the `PointsTo` relation is Andersen's algorithm [18]. The algorithm considers four basic kinds of assignments as shown in Figure 8 and derives the points-to relation for the whole program from them.

Our points-to analysis in IncA builds on Andersen's rules but extends them in three ways. First, by implementing the analysis in IncA we immediately improve the run time after code changes through incrementality. Second, we add flow-sensitivity by building on top of our incremental control flow analysis. Third, we do not require the code to only use the four kinds of assignments in Andersen's rules, rather support all of mbeddr C except pointer arithmetics.

Figure 9 shows an excerpt of the points-to analysis in IncA. We use three main pattern functions for flow-sensitive points-to analysis. Function `pointsToBefore(n,u)` computes the potential targets of variable `u` before the execution of node `n` of the CFG. We compute `pointsToBefore` by promoting the bindings of `pointsToAfter` along the control flow graph. Function `pointsToAt(n)` computes the effect of node `n` in the form of changed bindings `(x,y)`. We describe function `pointsToAt` in greater detail below. Function `pointsToAfter(n,u)` yields the potential targets of variable `u` after the execution of node `n`. If there is no new binding at all at node `n` (first alternative) or the new binding has no effect on variable `u` (second alternative), `pointsToAfter` retains the targets described by `pointsToBefore`. Otherwise, if node `n` affects variable `u` (third alternative), `pointsToAfter` yields the new targets of `u`.

A single CFG node can contain multiple assignments due to expression nesting as in `(x=&u)+(y=&v)`. Function `pointsToAt(n)` gathers all assignments that occur in node `n` and computes the corresponding points-to tuples. Given an assignment, we can extract the pointer variable `u` from the assignment's left hand side expression and the pointed-to variable `v` from the right with Andersen's rules. However, depending on the side of the assignment, the computation is different. `pointsToAt` calls `varInExprLeft` to look up the pointer variable and `varInExprRight` to obtain the pointed-to variable and returns them as a new points-to tuple.

```
def pointsToBefore(n:ICFGNode, u:Var) : Var = {
  pred := cFlow(n)
  return pointsToAfter(pred, u)
}

def pointsToAfter(n:ICFGNode, u:Var) : Var = {
  assert undef pointsToAt(n)
  return pointsToBefore(n, u)
} alt {
  (x, y) := pointsToAt(n)
  assert x != u
  return pointsToBefore(n, u)
} alt {
  (x, y) := pointsToAt(n)
  assert x == u
  return y
}

def pointsToAt(n : ICFGNode) : (Var, Var) = {
  (lhs, rhs) := assignmentAt(n)
  u := varInExprLeft(lhs)
  v := varInExprRight(rhs)
  return (u, v)
}
```

Figure 9: Flow-sensitive points-to analysis in IncA

Function `varInExprLeft` in Figure 10 computes the pointer variable of an assignment's left-hand side expression. If the expression is a plain variable reference, function `varInExpr` from Figure 6 extracts the variable. Otherwise, based on the fourth Andersen rule (Figure 8), a pointer dereferencing `*e` requires the lookup of the points-to targets of `e`. Function `varInExprLeft` calls itself recursively on `e` to handle multiple dereferencings `**e`, until finally reaching a plain variable. We look up the points-to targets of the dereferenced variable `u` by calling `pointsToBefore`, using `ancestorCFGNode` to retrieve the surrounding CFG node. Function `varInExprRight` implements three alternatives corresponding to the first three Andersen rules (in order). The implementation for dereferencing (second alternative) is identical to `varInExprLeft` and the other two alternatives are straightforward.

6.3 Well-formedness Checks for mbeddr C

We implemented four well-formedness checks for mbeddr C and its language extensions. While the control flow analysis and the points-to analysis inspected each function declarations in separation, our well-formedness checks require global knowledge about the source code. This case study shows that IncA scales to support whole-program analyses. The checks are as follows:

CYCLE mbeddr C provides modules for organizing code. This check detects cyclic module dependencies.

GLOBAL This check detects conflicting global variables with the same name across modules.

REC This check detects recursive functions by construction

```
def varInExprLeft(lhs : Expression) : Var = {
  return varInExpr(lhs)
} alt {
  assert lhs instanceOf DerefExpr
  e := lhs.expression
  u := varInExprLeft(e)
  n := ancestorCFGNode(lhs)
  v := pointsToBefore(n, u)
  return v
}

def varInExprRight(rhs : Expression) : Var = {
  assert rhs instanceOf AddressOfExpr
  e := rhs.expression
  return varInExpr(e)
} alt {
  u := varInExpr(rhs)
  n := ancestorCFGNode(rhs)
  v := pointsToBefore(n, u)
  return v
} alt {
  assert rhs instanceOf DerefExpr
  e := rhs.expression
  u := varInExprRight(e)
  n := ancestorCFGNode(rhs)
  v := pointsToBefore(n, u)
  return v
}
```

Figure 10: Extracting points-to variables in IncA

```
def CI_CONFUSED_INHERITANCE(c: Class): Void = {
  assert c.isFinal == true
  member := c.member
  assert member instanceOf Field
  assert member.visibility instanceOf Protected
}
```

Figure 11: FindBugs `CI_CONFUSED_INHERITANCE` in IncA

and inspection of a call graph. In embedded systems with constrained memory, the stack space required for recursive functions is often unacceptable.

COMP mbeddr C supports interfaces and composable components. This check detects components that fail to implement all functions declared by their interfaces.

6.4 FindBugs for Java

FindBugs [16] is a well-known suite of predefined patterns to detect potential bugs in Java code. To show that our system is independent of the analyzed language, we implemented 10 FindBugs analyses in IncA for MPS' Java dialect. Apart from a few language extensions, this Java language is identical to the original Java language.

As an example, we show the implementation of the `CI_CONFUSED_INHERITANCE` rule in Figure 11. The rule detects final Java classes that have at least one protected field. Since the class is final, it cannot be subclassed and the field should be private or public. The implementation is a direct encoding of the description, but runs incrementally thanks to IncA.

7. PERFORMANCE EVALUATION

In this section, we present the performance evaluation of our system for the case studies introduced in Section 6. We answer the following questions:

(Q1) Run time: How does the run time of IncA program analyses compare to their non-incremental counterpart?

(Q2) Memory: How does the memory requirement of IncA analyses compare to their non-incremental counterpart?

(Q3) Optimization impact: How does our optimization (Section 4) affect the run time and memory requirements?

7.1 Evaluation Setup

For each case study, we start with an initial code base (introduced below). After the initial, non-incremental run of the analysis, we programmatically make 100 updates of the code base and run the analysis after each update. Each update consists of 1 to 20 random code changes, such as duplicating a statement, deleting a statement or function, renaming a local variable, or introducing a new module import statement. Note that certain code changes may result in uncompilable code, for example because of a broken reference to a variable. IncA handles such cases and simply ignores the broken code parts. Once the code is fixed with incremental changes, the runtime system updates the analysis results. This approach allows us to imitate how a user would modify the source code. We measure the wall-clock time of processing the initial code and of processing each update step. For the memory measurement, we call the garbage collector after each analysis run and measure the required heap memory. We subtract the heap memory used before running the first analysis to obtain the memory usage of our system. We repeat each measurement five times and discard the results of the first and second run to account for Java VM warm-up.

As the first benchmark, we run the control flow and points-to analysis on the Toyota ITC code[6], a collection of C code snippets with intentional bugs to test the precision of static analysis tools. The code base comprises about 15,000 lines of C code. We compare the performance of the incremental analyses to a non-incremental flow-sensitive points-to analyses that was already available in mbeddr. The two analyses produce exactly the same results on the benchmark.

The second benchmark was running the well-formedness checks on a commercial Smart Meter software implemented in mbeddr C [34]. A smart meter is an electric meter that continuously records the consumption of electrical power, calculates derived quantities, and sends the data back to the utility provider for monitoring and billing. The whole project comprises about 44,000 lines of mbeddr C code. For comparison, we implemented non-incremental well-formedness checks in MPS Java that produce exactly the same results.

Running the 10 FindBugs checks consituted the third benchmark; we ran it on the Java implementation of the mbeddr importer which is responsible for migrating legacy C code to mbeddr C. The importer comprises about 10,000 lines of MPS Java code. Because of the MPS Java code base, we would need to generate textualized Java code after every code change to be able to use the original FindBugs tool. For our large code base this is impractical, and thus for this benchmark we do not have a non-incremental counterpart.

We ran the benchmarks on a 64-bit OSX 10.10.3 machine with an Intel Core i7 2.5 GHz processor and 16 GB of RAM, using MPS version 3.3 and Java 1.8.0_65. The raw data and the sequence of code changes are available online.[7]

7.2 Run Time Measurements (Q1 & Q3)

Figure 12 (A) - (C) show the results of our run time measurements. For points-to analysis and well-formedness checks we show the results of the incremental and non-incremental

[6] https://github.com/regehr/itc-benchmarks

[7] information removed due to double blind review

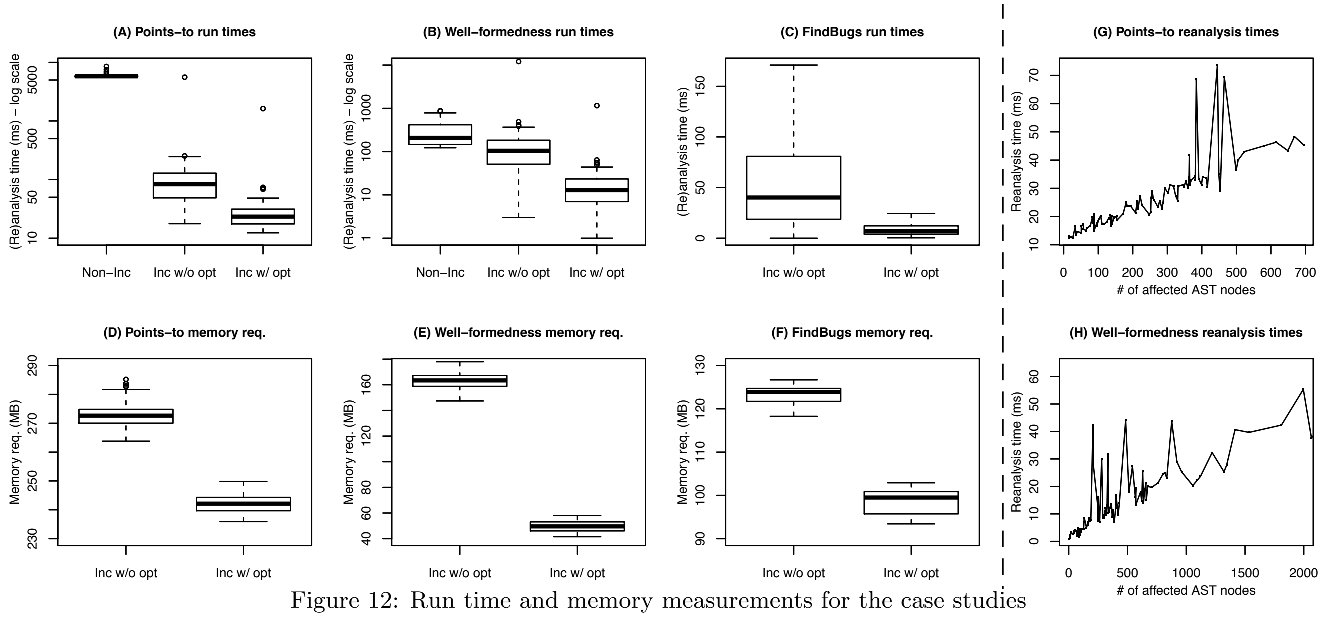


Figure 12: Run time and memory measurements for the case studies

solutions using a logarithmic scale, for FindBugs we only show the incremental results on a linear scale.

Box plots (A) and (B) immediately show that incremental program analyses in IncA perform significantly better than non-incremental analyses. Box plots (A) - (C) also show that the optimization has a significant impact on the run time. The following table summarizes the median run time data:

	Non-inc.	Inc w/o opt		Inc w/ opt	
		init	update	init	update
Points-to (A)	5.8s	5.6s	83.2ms	1.6s	23.3ms
W.-form. (B)	209ms	12.1s	104.8ms	1.2s	12.8ms
FindBugs (C)	n/a	4.5s	40.2ms	2.3s	7ms

The initialization times, especially for the optimized versions, do not pose an unduly run time requirement considering that loading large programs into MPS also takes a few seconds. After initialization, incremental analysis without optimization achieves speedups of A=70x and B=2x compared to non-incremental analysis. With optimization, we even achieve speedups of A=249x and B=16x. Indeed, the optimization accounts for an additional speedup of 2 - 10x for initialization and of 3 - 8x for change-processing time. IncA and its optimization provide good runtime performance across analyses and for both analyzed languages.

Figure 12 (G) and (H) show the points-to and well-formedness analysis times as a function of the input change size (the number of deleted or added AST nodes). IncA scales well, because the plots remain roughly linear with increasing input change sizes. We conclude:

(Q1) Incremental program analysis with IncA provides significant speedups of up to 249x compared to non-incremental analysis. The analyses scale linearly with increasing input change sizes.
(Q3) The IncA compiler optimization improves initialization time by 2–10x and change-processing time by 3–8x.

7.3 Memory Measurements (Q2 & Q3)

Figure 12 (D) - (F) show the results of our memory measurements. We measured the memory required by incrementality for each of the 100 update steps and created the box plots from this data. We summarize the median memory requirements in the following table:

	Inc w/o opt	Inc w/ opt
Points-to (D)	273MB	242MB
Well-formed (E)	163MB	50MB
FindBugs (F)	124MB	100MB

The points-to analysis is the most complex case study and it has the biggest memory requirement, because its computation graph caches major part of the input AST to compute a complete CFG and to handle a large variety of assignments, including nested ones. The optimization helps to reduce the memory requirement with A=11%, B=69%, and C=19%. Based on our experience with real-world usage of MPS, the IDE typically requires about 1.2GB of memory. This means that the optimized analyses have a memory overhead of A=20%, B=4%, and C=8% relative to MPS. We conclude:

(Q2) For real world scenarios, incremental program analysis with IncA requires an acceptable amount of 20% additional memory for our most complex case study.
(Q3) The IncA compiler optimization reduces the required memory by 11 - 69%.

8. RELATED WORK

Improving the performance of program analyses has attracted a lot of research, because of their widespread use in compilers and IDEs. In particular, the application of incrementality to speed up program analyses has a long tradition. While we present a *framework* for incremental program analyses, even the *manual* enhancement of specific program analyses to support incrementality is still subject to research.

Specialized algorithms Yur et al. propose an algorithm for incremental flow-sensitive and context-sensitive points-to analysis [38]. The paper assumes that the CFG is incrementally constructed. This in itself is a challenging problem which we address as part of our work. Saha and Ramakrishnan propose an incremental flow-insensitive points-to analysis for a subset of C using logic programming [29]. Lu et al.

present a context-sensitive and field-sensitive points-to analysis for Java based on context-free language reachability [19].

All of the previous approaches target only a particular analysis (e.g. points-to analysis). There are also several algorithms [6, 22, 14, 23] tailored specifically to handle a specific *class* of analyses (e.g. data-flow analyses) in an incremental way. In contrast to those, IncA goes one step further because it is not specific to any particular class of analyses, as we showed in Section 6.

The previously mentioned algorithms come with their own incrementalization engine that could play the role of the incremental evaluator in our architecture (although limiting expressivity to a particular class of analyses). Our contributions are generic enough to be applicable on top of these algorithms as well thus improving their efficiency with our optimizations enabled by DSL code.

Analysis Frameworks In the context of model-driven development, numerous systems incrementally reapply consistency rules on models [9, 12, 4]. These systems are only incremental in the sense that they *selectively* reapply affected consistency rules after a change; once selected, the rules run *non-incrementally* on the *whole* input. This form of incrementalization is not practical for complex program analyses or for large inputs. In contrast, our system incrementally reanalyzes only the truly affected program entities when the program gets changed.

EMF-IncQuery is a framework for incremental querying of EMF models [33], which is, like IncA, based on the EMF-independent IncQuery evaluator. We have generalized EMF-IncQuery in several ways. First, we have designed the IncA DSL for the description of program analyses that abstracts from graph patterns and supports more conventional descriptions in the style of forward or backward analyses. Second, we extended our runtime system with the data-flow analysis-based optimization, whereas EMF-IncQuery follows a manual approach to register relevant parts of the metamodel. Third, we describe an architecture that supports the integration in IDEs other than Eclipse. Finally, as a novel approach we employ incremental graph pattern matching for program analysis instead of model queries and demonstrate that the solution scales to real world programs.

Wachsmuth et al. introduce a language-independent task engine for incremental name and type analysis in the Spoofax language workbench [37]. The task engine's evaluation starts by collecting analysis tasks, which are executed in a second step. When the analyzed program changes, the task engine recollects analysis tasks and executes only the new ones. Arzt and Bodden present an algorithm that selectively re-derives changed parts of a program's data-flow graph upon program manipulation, which enables incremental data-flow analysis [2]. As the algorithm is integrated into the Inter-procedural Distributive Environment framework [28], all other data-flow analyses can benefit from incrementality. DOOP [30] is a framework for the points-to analysis of Java code. It supports various forms of context-sensitivity while being flow-insensitive. DOOP builds on Datalog, a logic programming language used in deductive databases, and uses the commercial LogicBlox [1] engine which performs incrementalization. The authors argue that the high-level Datalog language is a perfect fit for specifying points-to analysis. In fact, Datalog and graph patterns are closely related having similar expressive power [17, p. 225]. We argue that our DSL is one step further in the direction to program analyses with the directional pattern functions. We also validate the vision of DOOP's authors by showing that a declarative language is useful not only for points-to but different classes of analyses. In comparison to these solutions, IncA is not bound to a specific class of program analysis (e.g. data-flow or name analysis) as it also supports syntactic/structural analyses such as well-formedness checks or the FindBugs rules. These systems could also be extended with our optimization for the filtering of change events.

Conway et al. propose a system that incrementally checks safety properties encoded as automatons, where CFG edges trigger transitions [5]. The system achieves incrementality by computing and incrementally maintaining a derivation graph that combines the CFG nodes with an automaton state. When a program change occurs, the system selectively re-executes the automaton to compute the changes in the derivation graph. Reps et al. extend the Synthesizer Generator IDE [32] with attribute grammars [26] to define the semantics of languages and implement a runtime system that incrementally re-computes attribute values upon code change. The expressive power of these solutions is limited in comparison to IncA (see Section 3.2) as the former automaton-based approach has the expressive power of regular expressions and the latter one by Reps et al. does not support recursion. The lack of expressiveness would prevent implementing some of our case studies (Section 6).

General-purpose incrementalization i3QL [20] makes incremental computations available as an embedded Scala DSL using an SQL-like syntax. Its runtime system builds on relational algebra and applies optimizations known from database engines, but does not detect and filter irrelevant program changes. Adapton [13] is an ML-style general-purpose language for incremental self-adjusting computations. The runtime system of Adapton tracks memory dependencies in order to retrigger computations when a change to a data structure occurs. Compared to our evaluation, these tools were tested only on small programs with simple analyses thus it is unclear if general-purpose incrementalization frameworks can scale to support complex program analysis of large-scale code bases.

Analysis DSLs Other researchers have proposed DSLs for specific program analyses, but do not provide incrementalization. Examples include the declarative DSL for CFG construction in the JastAdd compiler [31, 10], the data-flow rules of DCFlow [15], fact extraction with DeFacto [3], and the DSL included in MPS for constructing data-flow graphs. In contrast to these, IncA supports the definition and incremental execution of arbitrary program analyses.

9. CONCLUSIONS

We presented IncA, a DSL for the definition and efficient evaluation of incremental program analyses in IDEs. IncA provides a declarative notation for analyses in the style of pattern functions that our optimizing compiler translates into graph patterns. We demonstrated the applicability of IncA by developing incremental program analyses for C and Java. Our performance evaluation shows that IncA provides significant speedups compared to non-incremental analyses and acceptable memory overhead.

We plan to use IncA in the future to enforce secure coding standards (e.g. Misra, CERT) in mbeddr. These standards define well-formedness rules which IncA can check incrementally as the program is edited.

10. REFERENCES

[1] Aref, M., ten Cate, B., Green, T. J., Kimelfeld, B., Olteanu, D., Pasalic, E., Veldhuizen, T. L., and Washburn, G. Design and Implementation of the LogicBlox System. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2015), SIGMOD '15, ACM, pp. 1371–1382.

[2] Arzt, S., and Bodden, E. Reviser: Efficiently Updating IDE-/IFDS-based Data-flow Analyses in Response to Incremental Program Changes. In *Proceedings of the 36th International Conference on Software Engineering* (New York, NY, USA, 2014), ICSE 2014, ACM, pp. 288–298.

[3] Basten, H. J. S., and Klint, P. *Software Language Engineering: First International Conference, SLE 2008, Toulouse, France, September 29-30, 2008. Revised Selected Papers*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, ch. DeFacto: Language-Parametric Fact Extraction from Source Code, pp. 265–284.

[4] Cabot, J., and Teniente, E. Incremental Integrity Checking of UML/OCL Conceptual Schemas. *J. Syst. Softw. 82*, 9 (Sept. 2009), 1459–1478.

[5] Conway, C. L., Namjoshi, K. S., Dams, D., and Edwards, S. A. Incremental Algorithms for Inter-procedural Analysis of Safety Properties. In *Proceedings of the 17th International Conference on Computer Aided Verification* (Berlin, Heidelberg, 2005), CAV'05, Springer-Verlag, pp. 449–461.

[6] Cooper, K. D., and Kennedy, K. Efficient computation of flow insensitive interprocedural summary information. In *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction* (New York, NY, USA, 1984), SIGPLAN '84, ACM, pp. 247–258.

[7] Diekmann, L., and Tratt, L. *Software Language Engineering: 7th International Conference, SLE 2014, Västerås, Sweden, September 15-16, 2014. Proceedings*. Springer International Publishing, Cham, 2014, ch. Eco: A Language Composition Editor, pp. 82–101.

[8] Dietrich, J., Hollingum, N., and Scholz, B. Giga-scale Exhaustive Points-to Analysis for Java in Under a Minute. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (New York, NY, USA, 2015), OOPSLA 2015, ACM, pp. 535–551.

[9] Egyed, A. Instant Consistency Checking for the UML. In *Proceedings of the 28th International Conference on Software Engineering* (New York, NY, USA, 2006), ICSE '06, ACM, pp. 381–390.

[10] Ekman, T., and Hedin, G. The JastAdd System — Modular Extensible Compiler Construction. *Sci. Comput. Program. 69*, 1-3 (Dec. 2007), 14–26.

[11] Erdweg, S., Bračevac, O., Kuci, E., Krebs, M., and Mezini, M. A Co-contextual Formulation of Type Rules and Its Application to Incremental Type Checking. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (New York, NY, USA, 2015), OOPSLA 2015, ACM, pp. 880–897.

[12] Groher, I., Reder, A., and Egyed, A. Incremental Consistency Checking of Dynamic Constraints. In *Fundamental Approaches to Software Engineering*, D. Rosenblum and G. Taentzer, Eds., vol. 6013 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2010, pp. 203–217.

[13] Hammer, M. A., Phang, K. Y., Hicks, M., and Foster, J. S. Adapton: Composable, Demand-driven Incremental Computation. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2014), PLDI '14, ACM, pp. 156–166.

[14] Hermenegildo, M., Puebla, G., Marriott, K., and Stuckey, P. J. Incremental analysis of constraint logic programs. *ACM Trans. Program. Lang. Syst. 22*, 2 (Mar. 2000), 187–223.

[15] Hills, M. Streamlining Control Flow Graph Construction with DCFlow. In *Software Language Engineering*, B. Combemale, D. Pearce, O. Barais, and J. Vinju, Eds., vol. 8706 of *Lecture Notes in Computer Science*. Springer International Publishing, 2014, pp. 322–341.

[16] Hovemeyer, D., and Pugh, W. Finding Bugs is Easy. *SIGPLAN Not. 39*, 12 (Dec. 2004), 92–106.

[17] Immerman, N. *Descriptive complexity*. Springer Science & Business Media, 2012.

[18] Lars Ole, A. *Program Analysis and Specialization of the C Programming Language*. PhD thesis, University of Copenhagen, 1994.

[19] Lu, Y., Shang, L., Xie, X., and Xue, J. An Incremental Points-to Analysis with CFL-Reachability. In *Compiler Construction*, R. Jhala and K. De Bosschere, Eds., vol. 7791 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2013, pp. 61–81.

[20] Mitschke, R., Erdweg, S., Köhler, M., Mezini, M., and Salvaneschi, G. I3QL: Language-Integrated Live Data Views. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014* (2014), pp. 417–432.

[21] Nielson, F., Nielson, H. R., and Hankin, C. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[22] Pollock, L. L., and Soffa, M. L. An incremental version of iterative data flow analysis. *IEEE Trans. Softw. Eng. 15*, 12 (Dec. 1989), 1537–1549.

[23] Puebla, G., and Hermenegildo, M. *Static Analysis: Third International Symposium, SAS '96 Aachen, Germany, September 24–26, 1996 Proceedings*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996, ch. Optimized algorithms for incremental analysis of logic programs, pp. 270–284.

[24] Ramalingam, G., and Reps, T. A categorized bibliography on incremental computation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 1993), POPL '93, ACM, pp. 502–510.

[25] Rensink, A. Representing First-Order Logic Using Graphs. In *Graph Transformations*, H. Ehrig,

G. Engels, F. Parisi-Presicce, and G. Rozenberg, Eds., vol. 3256 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2004, pp. 319–335.

[26] Reps, T., Teitelbaum, T., and Demers, A. Incremental Context-Dependent Analysis for Language-Based Editors. *ACM Trans. Program. Lang. Syst. 5*, 3 (July 1983), 449–477.

[27] Rozenberg, G., Ed. *Handbook of Graph Grammars and Computing by Graph Transformation: Volume I. Foundations*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1997.

[28] Sagiv, M., Reps, T., and Horwitz, S. Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation. *Theor. Comput. Sci. 167*, 1-2 (Oct. 1996), 131–170.

[29] Saha, D., and Ramakrishnan, C. R. Incremental and Demand-driven Points-to Analysis Using Logic Programming. In *Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming* (New York, NY, USA, 2005), PPDP '05, ACM, pp. 117–128.

[30] Smaragdakis, Y., and Bravenboer, M. *Datalog Reloaded: First International Workshop, Datalog 2010, Oxford, UK, March 16-19, 2010. Revised Selected Papers*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, ch. Using Datalog for Fast and Easy Program Analysis, pp. 245–251.

[31] Söderberg, E., Ekman, T., Hedin, G., and Magnusson, E. Extensible intraprocedural flow analysis at the abstract syntax tree level. *Science of Computer Programming 78*, 10 (2013), 1809 – 1827. Special section on Language Descriptions Tools and Applications (LDTA'08 & '09) & Special section on Software Engineering Aspects of Ubiquitous Computing and Ambient Intelligence (UCAmI 2011).

[32] Teitelbaum, T., and Reps, T. The Cornell Program Synthesizer: A Syntax-directed Programming Environment. *Commun. ACM 24*, 9 (Sept. 1981), 563–573.

[33] Ujhelyi, Z., Bergmann, G., Ábel Hegedüs, Ákos Horváth, Izsó, B., Ráth, I., Szatmári, Z., and Varró, D. EMF-IncQuery: An integrated development environment for live model queries. *Science of Computer Programming 98, Part 1*, 0 (2015), 80 – 99. Fifth issue of Experimental Software and Toolkits (EST): A special issue on Academics Modelling with Eclipse (ACME2012).

[34] Voelter, M., Deursen, A. v., Kolb, B., and Eberle, S. Using C Language Extensions for Developing Embedded Software: A Case Study. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (New York, NY, USA, 2015), OOPSLA 2015, ACM, pp. 655–674.

[35] Voelter, M., Ratiu, D., Kolb, B., and Schaetz, B. mbeddr: Instantiating a language workbench in the embedded software domain. *Automated Software Engineering 20*, 3 (2013), 339–390.

[36] Voelter, M., Siegmund, J., Berger, T., and Kolb, B. Towards User-Friendly Projectional Editors. In *Software Language Engineering*, B. Combemale, D. Pearce, O. Barais, and J. Vinju, Eds., vol. 8706 of *Lecture Notes in Computer Science*. Springer International Publishing, 2014, pp. 41–61.

[37] Wachsmuth, G., Konat, G., Vergu, V., Groenewegen, D., and Visser, E. A Language Independent Task Engine for Incremental Name and Type Analysis. In *Software Language Engineering*, M. Erwig, R. Paige, and E. Van Wyk, Eds., vol. 8225 of *Lecture Notes in Computer Science*. Springer International Publishing, 2013, pp. 260–280.

[38] Yur, J.-s., Ryder, B. G., and Landi, W. A. An Incremental Flow- and Context-sensitive Pointer Aliasing Analysis. In *Proceedings of the 21st International Conference on Software Engineering* (New York, NY, USA, 1999), ICSE '99, ACM, pp. 442–451.