ICAI 2023
12th International Conference on Applied Informatics
Eger, Hungary, 2023

A Survey of Dataflow Analyses in Clang

Kristóf Umann, Zoltán Porkoláb

ELTE, Eötvös Lóránd University, Budapest, Hungary. Faculty of Informatics,
Department of Programming Languages and Compilers
szelethus@inf.elte.hu, gsd@inf.elte.hu

Abstract

Despite Clang being one of the most popular compilers for the C family of languages, it once was surprisingly unfriendly towards dataflow analyses. As it turns out, there are infrastructural barriers from implementing them as easily as it is for its backend, LLVM. While this appears to be changing, the cumbersome implemenetation of the few dataflow algorithms that were in Clang present an interesting case study. This paper presents their struggles and clever solutions.

1. Introduction

Even basic optimizations demand the use of some data flow algorithms, making them a rather fundamental element of compiler design. Despite this, Clang is rather unfriendly towards implementing them. This is partly due to Clang's infrastructure design – it is not a standalone tool, but rather the frontend compiler for the LLVM backend compiler. While data flow analyses for optimizations are not a concern, they are still a great asset for emitting diagnostics, meaning that there still is a need for a sensible data flow infrastructure.

2. Obstacles

A common theme among data flow algorithms is looking for gens (generations, or in other words "reads" or "loads") or kills (in other words "writes" or "stores") of either variables or values. Under the term $GEN[s]$ set or $KILL[s]$ set, we mean a set of variables generated, or killed at a given statement s. Similarly, these sets can be defined on the domain of CFGBlocks (nodes of the control flow graph) –

$GEN[B]$ is the set of variables generated by one of the statements in CFGBlock B.

Many data flow analyses are defined with a fixpoint algorithm on GEN/KILL sets. In literature, we assume that these sets are precalculated, but that is rarely a case for Clang. Also, most data flow algorithms are defined and showcased on either some simple pseudo code, or an instruction set in a three address form:

```
x <- y + z
```

This tells us that variables y and z are read, and x is written. This is a very low level of representation that Clang's AST can't express this easily – The use of a variable is denoted by the AST node $DeclRefExprs$, but that on its own does not tell us whether the variable is read or written. [5, 7] all mention this a very significant difficulty in Clang: only the surrounding context, and even that with great difficulty can tell that. LLVM IR is largely void of this problem (it is similar to in this regard to the code snippet above), but in [6] Chris Lattner provides a lengthy reasoning why it would be next to impossible to lower to LLVM IR, conduct analysis, but still produce diagnostics in the original source code. LLVM IR is still tempting enough though that we have ongoing projects to get some use out of it [2].

This poses another problem: in literature, transfer functions are responsible for the flow of information through a statement. For liveness, the transfer function for assignment statements like `x = a;` would tell that a should be added to a so called liveness set, and x should be removed. In Clang however, we can not always define the same rule for the same statement, since we need the context around the it.

On another note, suppose you need to conduct data flow analysis on the following code in clang, and need to identify variables in the code:

```
void A::foo(int u) {
  int i;
  this->a = 5;
  S s;
  s.a.b.x.z = 3;
  s.a->get().b = 6;
}
```

The parameter and u and i are easily identifiable, as they are both fundamentally typed. Describing the implicit *this* parameter is also easy with Clang's toolset, but not when we are talking about fields. If an object is encapsulated in another object, possibly multiple times, Clang's toolset runs thin, and makes the prospect of pointer analysis all the more difficult.

3. Conclusion

In our full paper, we discuss specific dataflow implementations: Liveness analysis [8], UninitializedVariables analysis, Thread Safety analysis [5] and Lifetime analysis [3].

[1, chapter 9.3] describes a dataflow analysis framework as follows: it is a $(D, V, \wedge, F)$ quadruple, where D is the direction of the dataflow, V is the lattice, which includes the domain of the analysis, $\wedge$ is the meet operator, $F : V \to V$, a family of transfer functions.

Out of these 4, we have D factored out [4], and F, if we regard statement visitors as such. As for the other two, an agreement on the most efficient data structure with a merge/intersect operation would tie things together. All dataflow algorithms above need to manage sets for each statement in the CFG, and these sets are usually created from one another. Even for bitvector analyses, such as Uninitialized variables and thread safety, they implement their own custom container.

Support for the fields of a record are only supported by thread safety and lifetime. In fact, thread safety implements a custom intermediate language for this purpose.

For GEN/KILL analyses, such as liveness, uninitialized variables and reaching definitions, it should be possible to factor a lot of knowledge out about what statements read/write a variable. However, these analyses don't always agree on what is a GEN or KILL, so any attempt at refactoring must be configurable to some extent.

As of the writing this paper, a new kind of dataflow analysis infrastructure is being implemented in Clang – in our full paper, we intend to discuss how well does it fulfill the hopes set out for it.

References

[1] A. V. Aho, R. Sethi, J. D. Ullman: *Compilers principles, techniques, and tools*, Reading, MA: Addison-Wesley, 1986.

[2] A. Dergachev: *Get info from the LLVM IR for precision*, http://lists.llvm.org/pipermail/cfe-dev/2020-August/066537.html, 2020, (visited on 07/21/2022).

[3] M. Gehre, G. Horváth: *Implementing the C++ Core Guidelines' Lifetime Safety Profile in Clang*, https://www.youtube.com/watch?v=VynWy0Ib6Bk&ab_channel=LLVM, 2019, (visited on 07/21/2022).

[4] G. Horváth: *Factor two worklist implementations out*, https://reviews.llvm.org/D72380, 2020, (visited on 07/25/2022).

[5] D. Hutchins, A. Ballman, D. Sutherland: *C/C++ Thread Safety Analysis*, in: 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation, IEEE, Sept. 2014, doi: 10.1109/scam.2014.34, url: https://doi.org/10.1109%2Fscam.2014.34.

[6] C. Lattner: *LLVM Dev meeting: Slides & Minutes from the Static Analyzer BoF*, https://lists.llvm.org/pipermail/cfe-dev/2015-November/045872.html, 2015, (visited on 07/21/2022).

[7] C. Lattner, T. Shpeisman: *MLIR: Multi-Level Intermediate Representation for Compiler Infrastructure*, https://www.youtube.com/watch?v=qzljG6DKgic&ab_channel=LLVM, 2019, (visited on 07/21/2022).

[8] K. Umann: *An in-depth look at Liveness Analysis in the Static Analyzer*, https://lists.llvm.org/pipermail/cfe-dev/2020-July/066330.html, 2020, (visited on 07/21/2022).