CAnDL: A Domain Specific Language for Compiler Analysis

Philip Ginsbach
The University of Edinburgh
Edinburgh, UK
philip.ginsbach@ed.ac.uk

Lewis Crawford
The University of Edinburgh
Edinburgh, UK
s1203531@sms.ed.ac.uk

Michael F. P. O'Boyle
The University of Edinburgh
Edinburgh, UK
mob@ed.ac.uk

Abstract

Optimizing compilers require sophisticated program analysis and transformations to exploit modern hardware. Implementing the appropriate analysis for a compiler optimization is a time consuming activity. For example, in LLVM, tens of thousands of lines of code are required to detect appropriate places to apply peephole optimizations. It is a barrier to the rapid prototyping and evaluation of new optimizations.

In this paper we present the Compiler Analysis Description Language (CAnDL), a domain specific language for compiler analysis. CAnDL is a constraint based language that operates over LLVM's intermediate representation. The compiler developer writes a CAnDL program, which is then compiled by the CAnDL compiler into a C++ LLVM pass. It provides a uniform manner in which to describe compiler analysis and can be applied to a range of compiler analysis problems, reducing code length and complexity.

We implemented and evaluated CAnDL on a number of real world use cases: eliminating redundant operations; graphics code optimization; identifying static control flow regions. In all cases were we able to express the analysis more briefly than competing approaches.

CCS Concepts • **Software and its engineering → Specification languages**; *Constraint and logic languages*;

Keywords LLVM; optimization; constraint programming

ACM Reference Format:
Philip Ginsbach, Lewis Crawford, and Michael F. P. O'Boyle. 2018. CAnDL: A Domain Specific Language for Compiler Analysis. In *Proceedings of 27th International Conference on Compiler Construction (CC'18).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3178372.3179515

CC'18, February 24–25, 2018, Vienna, Austria

ACM ISBN 978-1-4503-5644-2/18/02…$15.00
https://doi.org/10.1145/3178372.3179515

1 Introduction

Compilers are complex pieces of software responsible for the generation of efficient code. This requires the careful application of a variety of optimizations. Most optimizations can be split in two parts: analysis and transformation. The analysis is required to check for applicability and legality and often contains most of the complexity. For example, simple peephole optimizations in the LLVM `instcombine` pass contain approximately 30000 lines of complex C++ code, despite the transformations being simple.

This complexity is an impediment to the implementation of new compiler passes, preventing the rapid prototyping of new ideas. Ideally, we would like a simpler way of describing such analysis that reduces boiler-plate code and opens the way for new compiler optimization innovation.

In this paper we present CAnDL, a domain specific language for compiler analysis. It is a constraint programming language that operates on LLVM's SSA-based IR. Instead of writing compiler analysis code inside the main codebase of the compiler infrastructure, it enables compiler writers to specify optimization functionality external to LLVM. The CAnDL compiler generates a C++ function that is linked in and acts as an LLVM pass. The formulation of optimizing transformations in CAnDL is faster, simpler and less error prone than writing them in C++. It has a strong emphasis on modularity, which enables debugging and the formulation of highly readable code.

We focus on programmer productivity and do not investigate formal methods for proving the correctness of optimization passes in this paper. However, previous work such as [13] suggests that CAnDL would also be also suitable for automatic verification.

We demonstrate the usefulness and generality of CAnDL via a number of use cases from different domains, including: standard LLVM optimization passes, custom optimizations for graphics shader programs and the detection of static control-flow regions for polyhedral program transformation.

The core contributions of this papers are:

- The specification of a domain specific language for writing compiler analysis.
- The implementation of the language inside of LLVM.
- The evaluation of the approach on different compiler analysis case studies.

(a) CAnDL program:

```
1 Constraint SqrtOfSquare
2 ( opcode{sqrt_call} = call
3 ∧ {sqrt_call}.args[0]  = {sqrt_fn}
4 ∧ function_name{sqrt_fn} = sqrt
5 ∧ {sqrt_call}.args[1]  = {square}
6 ∧ opcode{square} = fmul
7 ∧ {square}.args[0] = {a}
8 ∧ {square}.args[1] = {a})
9 End
```

(b) C program code:

```
1 double example(double a, double b) { return sqrt(a*a) + sqrt(b*b); }
```

(c) Resulting LLVM IR:

```
1  define double @example(
2   double %0,
3   double %1) {
4   %3 = fmul double %0, %0
5   %4 = call double @sqrt(%3)
6   %5 = fmul double %1, %1
7   %6 = call double @sqrt(%5)
8   %7 = fadd double %4, %6
9   ret double %7 }
10 declare double @sqrt(double)
```

(d) First solution:

```
a = %0

square = %3
sqrt_call = %4

sqrt_fn = @sqrt
```

(e) Second solution:

```
a = %1

square = %5
sqrt_call = %6

sqrt_fn = @sqrt
```

(f) C++ transformation code:

```
1 void transform(map<string,Value*> solution, Function* abs) {
2     ReplaceInstWithInst(
3         dyn_cast<Instruction>(solution["sqrt_call"]),
4         CallInst::Create(abs, {solution["a"]}));
5 }
```

(g) Transformed LLVM IR after DCE:

```
1 define double @example(double %0, double %1) {
2   %3 = call double @abs(double %0)
3   %4 = call double @abs(double %1)
4   %5 = fadd double %3, %4
5   ret double %5 }
```

Figure 1. Demonstration of a simple CAnDL program

2 Example

As a motivating example, assume that we want to use the algebraic property of the square root in Equation 1 to perform a floating point optimization (assuming the fast-math flag).

$$\forall a \in \mathbb{R}: \ \sqrt{a * a} = |a| \tag{1}$$

In order to apply this transformation, the compiler must detect occurrences of $\sqrt{a * a}$ in the IR code and replace them with a call to the `abs` function. The generation of the new function call is trivial, but the detection of even a simple pattern like $\sqrt{a * a}$ requires some care when implementing it manually in a complex code base such as LLVM.

The current approach would be to implement it as part of the `instcombine` pass, which already extends to almost 30000 lines of C++ code. This code makes heavy use of raw pointers and dynamic type casts. This is an impediment to compiler development and as previous work has shown [15, 26], it inevitably leads to buggy code.

Figure 1 shows how this optimization can be implemented with CAnDL. In (a), we can see the CAnDL program for the analysis. It states that a section of LLVM code is eligible for optimization if seven individual constraints simultaneously hold on the values `sqrt_call`, `sqrt_func`, `square`, `a`. The lines 2-8 each stipulate one of these constrains and they are joined together with the logical conjunction operator.

This CAnDL program can be compiled with our CAnDL compiler into LLVM analysis functionality and (b)-(f) show the results of running it on an example program. In (b), we can see a simple C program that calls the `sqrt` function twice with squares of floating point values. Below this, in (c), we see LLVM IR that is generated from the C code. I involves two `fmul` instructions to generate the squares via a floating point multiplication and two calls to the `sqrt` function with the respective result.

Note that for each application of the `sqrt` function there is a call instruction *e.g.* `%4 = call ...` and the actual `sqrt` function is the first argument of these call instructions.

The CAnDL program detects two opportunities to apply the transformation, shown in (d) and (e). Each of the two solutions assigns values from within the IR code to each of the variables in the CAnDL program such that all constraints are fulfilled. Let us look in detail at the first solution.

- `%4` is assigned to `sqrt_call`. It is a function call (Line 2 of CAnDL program) to the `sqrt` function (Lines 3 and 4 of CAnDL program).
- `@sqrt` is assigned to `sqrt_func`. It is the standard library function `sqrt` (Line 4 of CAnDL program).
- `%3` is assigned to `square`. It is the second argument of `%4` (`sqrt_call`) and it is an `fmul` instruction (Lines 5 and 6 of CAnDL program).
- `%0` is assigned to `a`. It is the first and second argument of `%3` (`square`) (lines 7-8 of CAnDL program).

With the analysis functionality provided by CAnDL, the transformation is now simple. In Figure 1 (f) we can see how the result of the analysis in the form of a C++ dictionary *std::map<std::string,llvm::Value*>* contains all the required information. A new function call to `abs` is generated with the solution for `a` as the only argument. This instruction then replaces the original call instruction that was captured in `sqrt_call`. After standard dead code elimination this results in the optimized code shown in (g).

Although this is a small example, it illustrates the main steps in our scheme. In practice, however, we wish to detect more complex code using constraints on control and data flow structure. In the next section we introduce a powerful description language that is capable of defining a wide class of analysis.

3 Language Specification

The Compiler Analysis Description Language (CAnDL) is a domain specific constraint programming language for the specification of compiler analysis passes. Individual CAnDL programs define computational structures to be exploited by optimizing code transformations. These structures are specified as constraints on single static assignment (SSA) form representations of programs.

Structures can scale from simple instruction patterns that are suited for peephole optimizations over basic control flow structures such as loops to complex algorithmic concepts such as stencil codes with arbitrary kernel functions or code regions suitable for polyhedral analysis.

The basic building blocks of CAnDL programs are well known compiler analysis tools, such as constraints on data and control flow, data types and instruction opcodes. On top of these low level constraints, CAnDL employs powerful mechanisms for modularity and encapsulation that allow the construction of complex programs.

3.1 High Level Structure of CAnDL Programs

An individual CAnDL program contains a set of constraint formulas that are bound to identifiers. As we already saw in Figure 1 (a), the syntax for this is as follows:

Constraint ⟨**s**⟩ *formula* **End**

For the description of CAnDL syntax we use these notational conventions: terminal symbols are **bold**, non-terminals are *italic*, ⟨**s**⟩ is an identifier (alphanumeric string) and ⟨**n**⟩ is an integer literal.

We already saw in Figure 1 (a) that logical conjunctions can be used to combine *formula*s. More generally, a *formula* can be any of the following:

atomic | *conjunction* | *disjunction*
| *foreach* | *forany* | *include* | *collect*

The basis of every CAnDL program are *atomic* constraints. For example in Figure 1 (a), lines 2-8 each specify individual atomic constraints. Atomic constraints are bound together by logical connectives ∧ and ∨ (*conjunction* and *disjunction*) and other higher level constructs. These include two kinds of loop structures (*foreach*, *forany*), as well as a system for modularity (`include`). Lastly, the *collect* construct allows for the formulation of more complex constraints that require the ∀ quantifier.

3.2 Atomic Constraints

The first type of constraint is an *atomic* constraint based on *variable*s. *Variable*s in CAnDL correspond to instructions and values in LLVM IR. Given some IR code, all occurring values can be assigned to the *variable*s of a given constraint formula. This includes instructions, globals, constants and function parameters. Syntactically, a variable is simply an identifier in curly brackets.

CAnDL uses the following atomic constraints:

data_type *variable* = **int**
data_type *variable* = **float**

This restricts the data type to integer or floating point.

ir_type *variable* = **literal**
ir_type *variable* = **argument**
ir_type *variable* = **instruction**

This restricts the type of IR node that is allowed to compile time constants, function arguments or instructions.

opcode *variable* = ⟨**s**⟩

This restricts the value instructions of the specified opcode.

function_name *variable* = ⟨**s**⟩

This restricts the variable to be a specified standard function (i.e. the `sqrt` function).

variable = variable
variable **!=** variable

This enforces two variables to have the same/not the same value. This is a shallow comparison, i.e. it compares whether two variables represent the same IR node.

control_origin *variable*
data_origin *variable*

The value is an origin of control (function entry) or data (function argument, `load` instruction, inpure function call).

variable.**arg[** ⟨**n**⟩] = *variable*
variable ∈ *variable*.**args**

There data flow from one value to the next.

variable -> *variable* Φ *variable*

The left value has to reach the right value, which is a phi node, via the middle value, which is a jump instruction.

domination(*variable*,*variable***)**
strict_domination(*variable*,*variable***)**

Both values have to be instructions and the first dominates the second in the control flow graph.

calculated_from(*varlist*,*varlist*,*variable***)**

Varlist is a set of one or multiple *variables*. Any path from one of the entries in the first *varlist* to the single *variable* argument has to pass through at least one of the entries in the second *varlist*. All paths in the union of the data flow and control dependence graph are considered. We will see later how this is useful to specify kernel functions for e.g. stencil calculations.

There are some *atomic*s that we omit for space reasons. The set of *atomic*s that CAnDL supports is easily extensible. Possible additions include constraints on function attributes, value constraints on literals etc.

3.3 Range Constraints

Building on top of the basic conjunction and disjunction constructs, there are range based versions that operate on arrays of variables.

formula **foreach** ⟨**s**⟩ = *index* .. *index*

formula **forany** ⟨**s**⟩ = *index* .. *index*

These constructs allow the repeated application of a formula according to some range of indices. This is demonstrated by Figure 2, which shows two equivalent CAnDL programs, one formulated with `foreach` and one without. In both cases, the program specifies an array of five variables with data flow from each element to the next. We can see how the `foreach` loop can be expanded similar to loop unrolling.

```
Constraint ValueChain
 {element[i]} ∈ {element[i+1]}.args foreach i=0..4
End
```

```
Constraint ValueChain
( {element[0]} ∈ {element[1]}.args
∧ {element[1]} ∈ {element[2]}.args
∧ {element[2]} ∈ {element[3]}.args
∧ {element[3]} ∈ {element[4]}.args)
End
```

Figure 2. Expansion of range constraints in CAnDL

3.3.1 Modularity

Modularity is central to the CAnDL programming language, and it is achieved using the *include* construct.

include ⟨**s**⟩
[(*variable* -> *variable* {, *variable* -> *variable*})]
[@ *variable*]

Note that the syntax in square brackets is optional and the syntax in curly brackets can be repeated. The basic version of *include*, without the optional structures, is simple. It copies the *formula* that corresponds to the identifier verbatim into another *formula*. If [@ *variable*] is specified, then all the variable names of the inserted constraint formula are prefixed with the given variable name, separated with a dot. The other optional syntax is used to rename individual *variable*s in the included *formula*.

Figure 3 illustrates this with two equivalent programs. Both programs specify an addition of four values, first adding pairwise and then adding the intermediate results. We can see in the first listing that a *formula* for the addition of two values is bound to the name `Sum`. This is then included three times in another *formula* names `SumOfSums`. Using the optional grammatical constructs, the formula operates on a different set of *variable*s each time such that the third addition takes the results of the previous two as input.

```
Constraint Sum
( opcode{out} = add
∧ {out}.args[0] = {in1}
∧ {out}.args[1] = {in2})
End
Constraint SumOfSums
( include Sum@{sum1}
∧ include Sum@{sum2}
∧ include Sum({sum1.out}->{in1},{sum2.out}->{in2}))
End
```

```
Constraint SumOfSums
( opcode{sum1.out} = add ∧
∧ {sum1.out}.args[0] = {sum1.in1}
∧ {sum1.out}.args[1] = {sum1.in2}
∧ opcode{sum2.out} = add
∧ {sum2.out}.args[0] = {sum2.in1}
∧ {sum2.out}.args[1] = {sum2.in2}
∧ opcode{out} = add
∧ {out}.args[0] = {sum1.out}
∧ {out}.args[1] = {sum2.out})
End
```

Figure 3. Expansion of Inheritance in CAnDL

3.3.2 Collect

The *collect* construct is used to capture all possible solutions of a given formula. It is used to implement constraints that require the logical ∀ quantifier. For example, it can be used to guarantee that all memory accesses in a loop use affine index computations. The grammar is simple but the semantics require some elaboration.

collect ⟨**s**⟩ *index formula*

In Figure 4, the variables `arg[0],...,arg[N-1]` are constraint to contain all data dependences of `ins`. The first argument of *collect* specifies the name of an index variable that is used to detect which variables belong to the collected set. In this example we want all solutions of `arg[i]` for a given value of `ins`. The second argument gives an upper bound to the amount of collected variables, in this case we leave it unspecified by using the symbol `N`.

```
Constraint CollectArguments
( ir_type{ins} = instruction
∧ collect i N ({arg[i]} ∈ {ins}.args))
End
```

Figure 4. Simple collect example in CAnDL

We can now extend this example to show how *collect* can be used to implement quantifiers. Consider that we want to detect instructions with only floating point data dependences. Formulating this involves the ∀ quantifier, as it is equivalent to the following equation.

$$\forall v\colon\ v \in I.\text{args} \implies \text{data\_type}(v) = \text{float} \qquad (2)$$

We can rewrite this to an equivalent formulation on sets.

$$S_1 := \{v \mid v \in I.\text{args}\} \subset S_2 := \{v \mid \text{data\_type}(v) = \text{float}\}$$

Now we can apply the following equivalences:

$$S_1 \subset S_2 \Leftrightarrow S_1 = S_1 \cap S_2$$
$$\Leftrightarrow \exists S\colon S = S_1 \wedge S = S_1 \cap S_2$$

This means that if we constraint a set S to be equal to both S_1 and $S_1 \cap S_2$ at the same time, the constraints are satisfiable if and only if the implication in Equation 2 holds.

This condition can be expressed in CAnDL, as is shown in Figure 5. With the first *collect* statement in line 3, we constrain the set `arg` to be equal to S_1 and with the second one in lines 4-5 we constrain it to be $S_1 \cap S_2$ as well. Note that we were from the onset only interested in the values that qualify for `ins`. The set `arg` was only introduced to further constraint `ins`, not because we actually wanted to know the values that it contains.

```
1 Constraint FloatingPointInstruction
2 ( ir_type{ins} = instruction
3 ∧ collect i N ( {ins} ∈ {arg[i]}.args)
4 ∧ collect i N ( {ins} ∈ {arg[i]}.args
5                ∧ data_type{arg[i]} = float))
6 End
```

Figure 5. Collect Example in CAnDL

The exact same approach can be used to e.g. restrict all array accesses in a loop to be affine in the loop iterators. This can be achieved by first *collect*ing all memory accesses (i.e. all `load` and `store` instrutions) and then using another *collect* statement to stipulate affine calculations for the indices.

3.4 Expressing Larger Structures

The modularity of CAnDL allows the creation of a library of building blocks that are shared by multiple CAnDL programs. We will now give an overview about how these can be defined with CAnDL.

Important building blocks include control flow structures such as single entry single exit regions and loops. These are standard in compiler analysis and the implementation in CAnDL is straightforward. A for loop involves a comparison of the loop iterator with the end of the iteration space. In order to be valid, this value has to be determined before the loop is entered, it isn't allowed to change from loop iteration to iteration. This leaves it to be either a function argument, an actual constant or an instruction that strictly dominates the loop entry. This is expressed in Figure 6. Note that this formula is to be included into larger CAnDL programs, as the `begin` variable is under specified otherwise.

```
1 Constraint LocalConst
2 ( ir_type{value} = literal
3 ∨ ir_type{value} = argument
4 ∨ strict_domination({value}, {begin}))
5 End
```

Figure 6. LocalConst in CAnDL

Another class of important building blocks are different categories of memory access. These form a hierarchy of restrictiveness and include multidimensional array access and array access that is affine in some loop iterators. LLVM strictly separates memory access from pointer computations, which means that CAnDL only has to concern itself with pointer computations here. In general it is required that the base pointer is `LocalConst` in order to avoid pointer chases. The index computation can then be described using the data flow and instruction opcode restrictions.

To enable the capture of higher order functions such as stencils or reduction operations, we need to handle arbitrary kernel functions. Kernel functions are sections of code that are side-effect free and can be separated out. Identifying side-effect free code is useful in many types of compiler optimization. It can be expressed in CAnDL, as shown in Figure 7.

```
1 Constraint KernelFunction
2 ( collect i N
3 ∧ ( include LocalConst({outer}->{begin}@{closure[i]}
4   ∧ ir_type{closure[i].value} != literal
5   ∧ {closure[i].use} ∈ {closure[i].value}.args
6   ∧ domination({inner}, {closure[i].use}))
7 ∧ collect i N data_origin{tainted1[i]}
8 ∧ collect i N
9 ∧ ( domination({outer}, {tainted2[i]})
10  ∧ strict_domination({tainted2[i]}, {inner}))
11 ∧ calculated_from(
12   {tainted1[0..N],tainted2[0..N]},
13   {origin[0..N],closure[0..N].value,input[0..N]},
14   {output}))
15 End
```

Figure 7. CAnDL Formulation of Kernel Functions

Essentially, this set of contraints captures the computation of a single `output` value from a set of specified `input` values as well as a set of automatically captured `closure` variables. The computation for `output` should be such that it can be replaced with a function call that is free of side effects and takes only the `input` and `closure` variables as arguments.

In addition to these variables, there are two non-obvious additional variables involved: `outer` and `inner`. These set boundaries in the control flow as follows: Closure values have to be computed before `outer` and the computation that results in outer is performed after `inner`. Usually these will be derived from loops nests, where `outer` is the entry to the outermost loop and `inner` is the entry to the innermost loop. The actual core constraint is then a generalized dominance relationship in the program dependence graph.

The use of the sets `tainted1` and `tainted2` makes sure that no impure functions are used in computing `output` and the entire computation is performed after `inner` (ruling out e.g. loop carried variabls from outer loops).

4 Implementation

CAnDL interacts with the LLVM framework, as shown in Figure 8. CAnDL programs are read by the CAnDL compiler, which then generates C++ source code to implement the specified LLVM analysis functionality. This code depends on a generic backtracking solver, which is incorporated into the main LLVM code base. We will see in the evaluation section this this solver adds little compile-time overhead in practice. The generated code is compiled and linked together with the existing LLVM libraries to make LLVM optimization passes available in the clang compiler. [1]

The generated analysis passes use the solver to search for the specified computational structures and output the found instances into report files, as well as making them available to ensuing transformation passes.

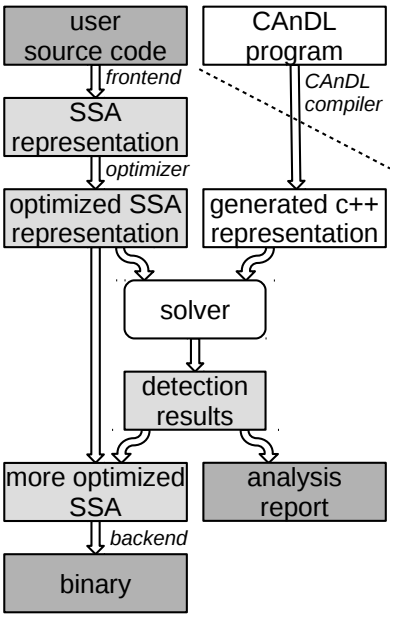

Figure 8. CAnDL in the LLVM/clang build system

4.1 The CAnDL Compiler

The CAnDL compiler is responsible for generating C++ code from CAnDL programs. An overview of its flow is shown in Figure 9. The frontend reads in CAnDL source code and builds an abstract syntax tree. This syntax tree is simplified in two steps to eliminate some of the higher order constructs of CAnDL. The `inheritance` clauses are replaced using standard function inlining after the contained variables have been transformed accordingly. Also, `foreach` and `forany` statements are lowered to conjunctions and disjunctions by duplicating the contained constraint code and then renaming the contained variable names appropriately for each iteration.

[1] Our impementation of CAnDL is available as open source on https://github.com/cc18ginsbach.

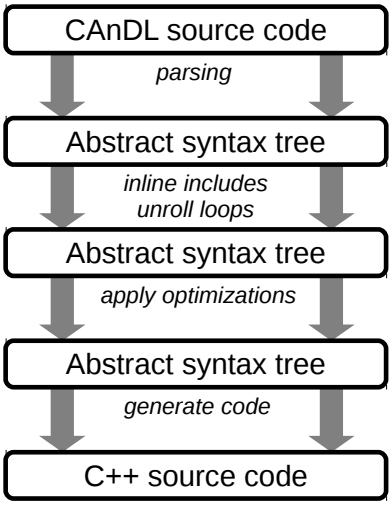

Figure 9. CAnDL compiler flow

This is equivalent to complete loop unrolling. From this point onwards, variable names are treated as flat strings. The remaining core language now consists only of atomics, conjunctions, disjunctions and collections.

The CAnDL compiler applies a set of basic optimizations to speed up the solving process using the later generated C++ code. For example, nested conjunctions and disjunctions are flattened wherever possible.

Finally, the compiler generates the C++ source code. This essentially involves constructing the constraint problem as a graph structure that is accessible to the solver.

4.1.1 C++ Code Generation

We demonstrate the code generation process in Figure 10 with an example. Each of the atomic constraints results in a line of C++ code that constructs an object of a corresponding class: In this case, the three involved atomic constraints are implemented by `AddInstruction`, `FirstArgument` and `SecondArgument`. For constraints that involve more than one variable, these objects are instantiated as shared pointers.

The compiler then generates similar objects for the more complex `conjunction`, `disjunction` and `collect` structures. In our example, this only affects the variable `addition`, which is part of a `conjunction` clause. This results in an additional object construction that instantiates the `And` class corresponding to the ∧ operator in CAnDL. The `select` function is used here to specify which variable of a constraint is being operated on. In this case, `addition` is the second variable in lines 3-4 of the CAnDL program, so we use `select<1>`.

Finally, the generated objects are inserted into a vector together with the corresponding variable names. This vector is then passed to the solver.

```
Constraint SimpleAddition
( opcode{addition} = add
∧ {addition}.args[0] = {left}
∧ {addition}.args[1] = {right})
End
```

```
auto constr0 = AddInstruction(context);
auto constr1 = make_shared<FirstArgument>(context);
auto constr2 = make_shared<SecondArgument>(context);
auto constr3 = And(constr0, select<1>(constr1),
                            select<1>(constr2));

vector<pair<string,SolverAtomContainer>> result(3);
result.emplace_back("addition", constr3);
result.emplace_back("left", select<0>(constr1));
result.emplace_back("right", select<0>(constr2));
```

Figure 10. C++ source code generation

4.2 The Solver

The solver takes LLVM IR code and a graph representation of the constraint problem as constructed by the generated code. We saw in Figure 10 that this representation comes in the form of a vector of labeled instances of a class called `SolverAtomContainer`. This class wraps around the class `SolverAtom` that is defined in Figure 11.

```
class SolverAtom {
public:
  virtual SkipResult skip_invalid(unsigned& c)const;

  virtual void begin();
  virtual void fixate(unsigned c);
  virtual void resume();
};
```

Figure 11. The SolverAtom interface.

The motivation for this interface is as follows: The solver operates on unsigned integers, using the `skip_invalid` method to search for partial solutions. The corresponding LLVM values are numbered consecutively and the unsigned integers simply represent indices into that enumeration. When `skip_invalid` returns `FAIL`, the solver backtracks. The other member functions `begin`, `fixate` and `resume` allow implementations of the `SolverAtom` interface to do bookkeeping.

Pseudocode for the backtracking constraint solver is shown in Figure 12. The array of `SolverAtom`s is named `atom`, `solution` is an array of integers that is incrementally filled with a solution to the constraint problem. In line 3, we can see that the `skip_invalid` method is used to find the next candidate solution for the `i`th element of the solution, taking into account all the previously established elements `solution[0..i-1]`. The candidate solution is directly stored in `solution[i]`, which is passed by reference as shown in Figure 11. If the step was successful, then the solver either returns the solution if it is complete or it moves on the the next element (line 11), otherwise it backtracks (line 16). The member function `fixate`, `begin` and `resume` are called appropriately in lines 6, 12 and 17.

```
1  i := 0
2  while i >= 0:
3      result := atom[i].skip_invalid(solution[i])
4
5      if result = SkipResult::PASS:
6          atom[i].fixate(solution[i])
7
8          if i+1 = N:
9              return solution
10         else:
11             i := i+1
12             solver_atom[i].begin()
13             solution[i] := 0
14
15     if result = SkipResult::FAIL:
16         i := i-1
17         atom[i]->resume()
18         solution[i] := solution[i]+1
```

Figure 12. Pseudocode of the backtracking solver

We can see that the solver is generic and requires no knowledge of LLVM or the structure of variable names in CAnDL. Furthermore, the solver is unaware not only of the underlying LLVM structure and the corresponding atomic constraints, but also of the `conjunction`, `disjunction` and `collect` constructs.

The complexity of the solving process lies almost entirely in the implementation of the `SolverAtom` interface for the different atomic constraints and their interactions. For simple constraints like the **is an add instruction** statement, this is straightforward and `skip_invalid` only has to test whether the value at index n in the function is an add instruction or not. This can be implemented as shown in Figure 13.

```
class AddInstruction : public SolverAtom {
public:
  AddInstruction(Context&) { ... }

  SkipResult skip_invalid(unsigned& c) const
  {
      if(c >= value_list->size())
        return SkipResult::FAIL;

      if(auto inst = dyn_cast<Instruction>
                       ((*value_list)[c]))
      {
        if(inst->getOpcode() == Instruction::Add)
          return SkipResult::PASS;
      }

      c=c+1;
      return SkipResult::CHANGE;
  }

  void begin() {}
  void fixate(unsigned c) {}
  void resume() {}

private:
  shared_ptr<vector<Value*>> value_list;
};
```

Figure 13. SolverAtom for additions

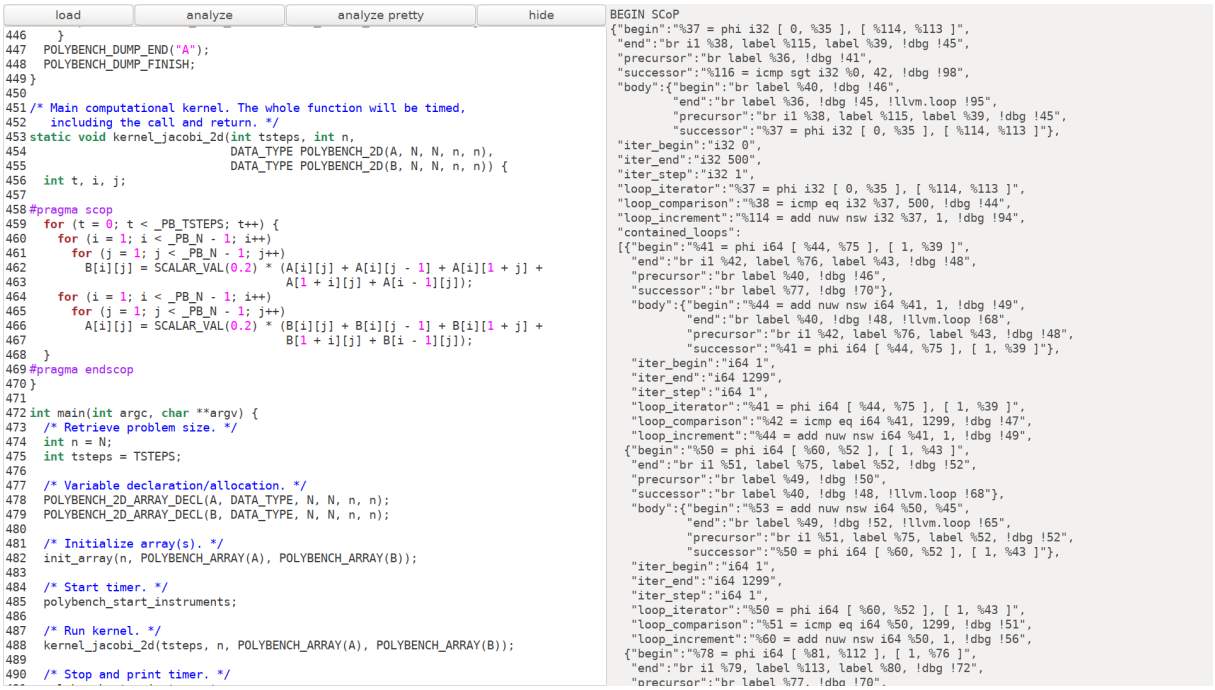

Figure 14. Interactive CAnDL test tool

Left hand panel shows a SCoP in Polybench jacobi-2d, the right hand panel show the corresponding constraint solution

4.3 Developer Tools

CAnDL makes writing compiler analysis passes easier, but reasoning about the semantics of compiler IR still remains difficult and the correctness of CAnDL programs can only be guaranteed with thorough testing. It is important to keep in mind that CAnDL is targeted at expert compiler developers.

In order to make debugging of CAnDL programs more feasible, we developed supporting tools. Most importantly, this includes an interactive gui, where developers can test out corner cases of CAnDL programs to find false positives and false negatives. This gui is shown in Figure 14, the example is from one of the use cases presented in a section 5.

In the left column, we can see part of a C program from the PolyBench benchmark suite, which implements a two-dimensional Jacobi stencil. The gui is configured to look for static control flow regions (SCoPs), as described later in one of the use cases. The user has clicked the "analyze" button, which triggered the analysis to run and prints the results in the right column.

The solver found a SCoP in the IR code (corresponding to lines 459-468 of the C program). The text on the right shows the hierarchical structure of the solution, with IR values assigned to every variable. The corresponding C code entities can be recovered using the debug information that is contained in the generated LLVM IR code. By modifying the C code, the developer can now test the detection and e.g. verify that no SCoP is detected if irregular control flow is introduced.

5 Case Studies

We evaluate the usefulness of CAnDL on three real life use cases. We first use it for simple peephole optimizations. We then apply CAnDL to graphics shader code optimization. Finally, we demonstrate the detection of static control flow parts (SCoPs) for polyhedral code analysis. Where possible, we compare the number of lines of CAnDL code, its program coverage and performance against prior approaches.

5.1 Simple Optimizations

Arithmetic simplifications in LLVM are implemented in the `instcombine` pass. One example of this is the standard factorization optimization that uses the law of distributivity to simplify integer calculations as shown in Equation 3. It is implemented in 203 lines of code and furthermore uses supporting functionality provided by `instcombine`.

$$a * b + a * c \rightarrow a * (b + c) \tag{3}$$

This analysis problem can be formulated in CAnDL as shown in Figure 15. The CAnDL program even captures a much larger class of opportunities for factorization than `instcombine`, which require to first apply associative and commutative laws to reorder the values. This is for example needed in the case of Equation 4 and only partially supported by LLVM with the additional `reassociate` pass.

$$a * b + c + d * a * e- > a * (b + d * e) + c \tag{4}$$

```
Constraint ComplexFactorization
( opcode{value} = add
∧ {value}.args[0] = {sum1.value}
∧ {value}.args[1] = {sum2.value}
∧ include SumChain at {sum1}
∧ {mul1.value} = {sum1.last_factor}
∧ include MulChain at {mul1}
∧ {mul1.last_factor} = {mul2.last_factor}
∧ include SumChain at {sum2}
∧ {mul2.value} = {sum2.last_factor}
∧ include MulChain at {mul2})
End
```

Figure 15. ComplexFactorization in CAnDL

We evaluated the program in Figure 15 against the default factorization optimization in LLVM's `instcombine` on three benchmark suites: the sequential C versions of NPB, the C versions of Parboil and the OpenMP C/C++ versions of Rodinia. We annotated the existing LLVM `instcombine` pass such that it reports every time that it successfully applies the `tryFactorization` function.

We compiled all the individual benchmark programs in the three benchmark suites, which consist of 94915 lines of code in total. For each benchmark suite, we summed up all factorizations that were reported. We also measured LLVM's total compilation time.

We then disabled the standard LLVM optimization and instead used the CAnDL generated detection functionality. We compiled the same application programs reporting the number of factorizations found and again measured the total compilation time this time using CAnDL. Note that this compilation time includes all the other passes within LLVM as well as the CAnDL generated path.

5.1.1 Results

	LLVM	CAnDL
Lines of Code	203	12
Detected in NPB	1	1 + 2
Detected in Parboil	0	0 + 1
Detected in	24	24 + 4
Total Compilation time	152.2s	152.2s+7.8s

Figure 16. Factorizations LLVM vs CAnDL

The results are shown in Figure 16. In general, the impact of simple peephole optimizations is small and in two of the benchmark sets we find only very small numbers. LLVM was unable to perform any factorization in the entire Parboil suite. However, the Rodinia suite contains more opportunities, mostly in the Particlefilter and Mummergpu programs.

In all three benchmarks suites, our scheme finds the same factorization opportunities as the `instcombine` pass plus an additional 7 cases. With only 12 lines of CAnDL code, we were able to capture more factorization opportunities than LLVM did using two hundred lines of code.

Using CAnDL on large complex benchmark suites only increased total compilation time by 5%, demonstrating its use as a prototyping tool.

5.2 Graphics Shader Optimizations

Graphics computations often involve arithmetic on vectors of single precision floating point values that represent either vertex positions in space or color values. Common graphics shader compilers utilize the LLVM framework using the LunarGLASS project.

For real shader code, LLVM misses an opportunity for the associative reordering of floating point computations. Although such reordering is problematic in general, it is applicable in the domain of graphics processing.

There are often products of multiple floating point vectors, where several of the factors are actually scalars that were hoisted to vectors. By reordering the factors and delaying the hoisting to vectors, some of the vector products can be simplified to scalar products, as shown in the following equation.

$$\begin{aligned}\vec{x} &= \vec{a} *_v \vec{b} *_v \text{vec3}(c) *_v \vec{d} *_v \text{vec3}(e)\\ &= \text{vec3}(c * e) * \vec{a} *_v \vec{b} *_v \vec{d}\end{aligned}$$

We implemented the required analysis functionality for this optimization with CAnDL, as shown in Figure 17. The included `VectorMulChain` program discovers chains of floating point vector multiplications in the IR code and uses the variables `factors` and `partials` such that

$$\texttt{partials}[0] = \texttt{factors}[0]$$
$$\texttt{partials}[i+1] = \texttt{partials}[i] \times \texttt{factors}[i+1].$$

The `VectorMulChain` program furthermore guarantees that this is a chain of maximal length by checking that neither of the first two factors are multiplications and the last factor is not used in any multiplication. `ScalarHoist` detects the hoisting of scalars to vectors and this is used to collect all hoisted factors into the array `hoisted`. In a last step, all other factors are collected into the array `nonhoisted`.

```
Constraint FloatingPointAssociativeReorder
( include VectorMulChain and
∧ collect j N
∧ ( {hoisted[k]} = {factors[i]} forsome i=0..N
  ∧ include ScalarHoist({hoisted[j]}->{out},
                        {scalar[j]}->{in})@{hoist[j]})
∧ collect j N
  ( {nonhoisted[j]}  = {factors[i]} forsome i=0..N
  ∧ {nonhoisted[j]} != {hoisted[i]} forall  i=0..N))
End
```

Figure 17. CAnDL code for vectorized multiplication chains

The corresponding transformation pass simply generates all the appropriate scalar and vector multiplications and replaces the old result with this newly generated one. We evaluated the performance impact on the CFXBench 4.0 on the Qualcomm Adreno 530 GPU.

5.2.1 Results

The optimization was relevant to 8 of the fragment shaders in GFXBench 4.0. The number of lines of code needed and the resulting performance impact are shown in Figure 18 and Figure 19. A total of 19 opportunities for the optimization to be applied were detected. Although the performance impact was moderate with $1-4\%$ speedup on eight of the fragment shaders, it shows how new analysis can be rapidly prototyped and evaluated with only a few lines of code.

	LLVM	CAnDL
Lines of Code	Not implemented	10
Detected in GFX 4.0	-	19

Figure 18. Shader optimization LLVM vs CAnDL

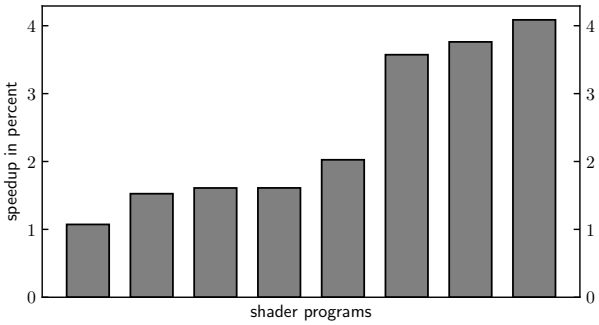

Figure 19. Speedup on Qualcomm Adreno 530

5.3 Polyhedral SCoPs

The polyhedral model allows compilers to utilize powerful mathematical reasoning to detect parallelism opportunities in sequential code and to implement sophisticated code transformations for well structured nested loops. It is currently applicable to Static Control Flow Parts (SCoPs) with affine data dependencies. Detecting SCoPs is fundamental for any later polyhedral optimization.

Implementations of the polyhedral model may differ in their precise definition of SCoPs. We implemented SCoP detection functionality in CAnDL and compared against the Polly implementation in LLVM. We rely on the definition of Semantic SCoPs in [8]. The constraints for SCoPs can be broken into several components:

Structured Control Flow SCoPs require well structured control flow. Technically speaking, this means that every conditional jump within the corresponding piece of IR is accounted for by for loops and conditionals. We enforce this with the `collect` statement as demonstrated in Figure 5. We use `collect` with CAnDL programs `ForLoop` and `Conditional` that describe the control flow of for loops and conditionals and extract the involved conditional jump instructions. We then use another `collect` to verify that these are indeed all conditional jumps within the potential SCoP.

Once we have established the control flow, we can use the iterators that are involved in the loops to define affine integer computations in the loop. This is done in a brute force fashion with a recursive constraint program. Using this analysis we then check that the iteration domain of all the for loops is well behaved, i.e. the boundaries are affine in the loop iterators.

Affine Memory Access We want to make sure that all memory access in the SCoP is affine. For this to be true we have to verify that for each load and store instruction, the base pointer is loop invariant and the index is calculated affinely. The loop invariant base pointer is easily guaranteed using the `LocalConst` program from Figure 6.

Checking the index calculations is more involved and is again based on the `collect` method that was demonstrated in Figure 5. We use the `collect` construct to find all of the affine memory accesses in all the loop nests. We then use collect all `load` and `store` instructions and verify that both collections are identical.

We evaluated our detection of SCoPs on the PolyBench suite. For both our method as well as for Polly, we counted how many of the computational kernels in the benchmark suite are captured by the analysis.

5.3.1 Results

As is visible in Figure 20, we capture all the SCoPs that Polly was able to detect. There is some postprocessing of the generated constraint solutions required to achieve this. Firstly, our results are not in the jscop format that Polly uses but contain the raw constraint solution as shown on the right side of Figure 14. Also, our CAnDL implementation does not merge consecutive outer level loops into SCoPs of maximum size. To compare the results, we extracted the detected loops from our report files, manually grouped them into maximum size SCoPs and verified that they fully cover the SCoPs detected by Polly.

For lines of code, we compared our version with the amount of code in Polly's `ScopDetection.cpp` pass. We are able to detect the same number of SCoPs with much fewer lines of code. Note that the LoC count that we give for our SCoP program does not include all CAnDL code involved in the detection of polyhedral regions. We consider the code that is not specific to this idiom (such as loop structures) to be part of the CAnDL standard library. In the same way we did not account for e.g the ScalarEvolution pass when counting the lines for Polly.

By having a high level representation of SCoPs, we allow polyhedral compiler researchers to explore the impact of relaxing or tightening the exact definition of SCoPs in a straightforward manner, enabling rapid prototyping.

	Polly	CAnDL
Lines of Code	1903	45
Detected in datamining	2	2
Detected in Linear-algebra	19	19
Detected in medley	3	3
Detected in stencils	6	6

Figure 20. SCoPs detected Polly vs CAnDL

6 Related Work

There is a large body of work that uses constraints for program analysis. Constraint systems have long been used in program analysis for classical purposes such as dataflow analysis and type inference [1]. In [9] they show how constraints can be used to for some existing analysis problems. It can also be used to assist generation of programs from specifications [20].

There is considerable work on formally verifying existing compiler transformations using SMT solvers and theorem provers that operate on IR code, among them [28]. Recent domain specific language for formally verifying compiler optimizations, such as Alive [13] operate on single static assignment compiler IR as well. However, it has no support for control flow and is limited to simple peephole optimizations.

Further approaches to generating compiler optimizations that focus on formal verification instead of programmer productivity include Rhodium [11], PEC [10] and Gospel [23]. CompCert [17] verifies peephole optimizations directly on x86 assembly code and LifeJacket [18] proves the correctness of floating point optimizations in LLVM, as does Alive-FP [16], which also generates C++ code. The authors of [21] investigate the automatic generation of optimization transformations from examples. None of these approaches however, tackle the issue of efficiently writing arbitrary compiler analysis passes.

In [14], the authors present a specification language for program analysis functionality called PAG that is based on abstract interpretation. Domain specific languages for the conception of optimization passes have also been studied before using tree rewrites among them [19]. In [12], the authors propose the domain specific language OPTRAN for matching patterns in attributed abstract syntax trees of Pascal programs. These patterns can then be automatically replaced by semantically equivalent, more efficient implementations. Generic programming concepts can also be used to generate optimization passes, as demonstrated by [24]. These schemes however are not able to work at the IR level essential for compiler implementation and do not scale beyond simple functions.

As opposed to code transformation techniques based on the LLVM ASTMatcher and LibTooling [25], we work on compiler intermediate representation and are independent from the compiler frontend. This makes us integrate well with the existing optimization infrastructure. allows us to be language independent and makes our approach robust to shallow syntactic changes.

Another language for implementing optimizations that emphasizes programmer productivity is OPTIMIX [3, 4], based on graph rewrite rules. OPTIMIX programs are compiled into C code that performs the specified transformation. A domain specific language for the generation of optimization transformations was also used in the CoSy compiler [2]. These are simple rewrite engines and have no knowledge of global program constraints.

Other work has investigated the use of constraint solvers for detecting structure in LLMV IR. The authors of [7] use a solver to detect histograms and scalar reductions in order to automatically parallelize code. A different important approach to detecting structure in intermediate code is the polyhedral model. Compilers using this model, such as [8], [5] and [22], capture well behaved loops with affine array accesses and are able to perform advanced loop optimizations. Recent work has investigated performant reduction computations on GPUs with the polyhedral model [6]. However this approach is not easily extensible to other program structures and captures only a very specific class of well behaved programs.

In [27], the authors propose the use of the Web Ontology Language (WOL) for the description of software architectures. This representation enables automatic reasoning and analysis about the interoperability of software architectures.

7 Conclusion

Optimizing compilers require sophisticated program analysis in order to generate performant code. The current way of implementing this functionality manually in programming languages such as C++ is not satisfactory.

The domain specific Compiler Analysis Description Language (CAnDL) provides a more efficient approach. CAnDL programs can be used to automatically generate compiler analysis passes. They are easier to write and reduce the code size and complexity when comparing against manual C++ implementations.

Although CAnDL is based on a constraint programming paradigm and uses a backtracking solver to analyze LLVM IR code, the use of CAnDL results in only moderate compile time increases. Many compiler analysis tasks are suitable for CAnDL programs, from small peephole optimizations to sophisticated program analysis with the polyhedral model.

Future work will investigate how methods from formal verification can be applied to CAnDL in order to guarantee the correctness of compiler optimizations. Also, there is some overlap between what can be formulated in CAnDL and what is provided by LLVM in the form of the ScalarEvolution pass. We are interested as to whether we could use this existing functionality to speed up solving times.

References

[1] Alexander Aiken. Introduction to set constraint-based program analysis. *Sci. Comput. Program.*, 35(2-3):79–111, November 1999.

[2] Martin Alt, Uwe Aßmann, and Hans van Someren. *Cosy compiler phase embedding with the CoSy compiler model*, pages 278–293. Springer Berlin Heidelberg, Berlin, Heidelberg, 1994.

[3] Uwe Aßmann. *How to uniformly specify program analysis and transformation with graph rewrite systems*, pages 121–135. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996.

[4] Uwe Assmann. Optimix - a tool for rewriting and optimizing programs. In *Handbook of Graph Grammars and Computing by Graph Transformation. Volume 2: Applications, Languages and Tools*, pages 307–318. World Scientific, 1998.

[5] Muthu Manikandan Baskaran, J. Ramanujam, and P. Sadayappan. Automatic c-to-cuda code generation for affine programs. In *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction*, CC'10/ETAPS'10, pages 244–263, Berlin, Heidelberg, 2010. Springer-Verlag.

[6] Michael Kruse Chandan Reddy and Albert Cohen. Reduction drawing: Language constructs and polyhedral compilation for reductions on gpus. In *Proceedings of the 25rd International Conference on Parallel Architectures and Compilation*, PACT '16, 2016.

[7] Philip Ginsbach and Michael F. P. O'Boyle. Discovery and exploitation of general reductions: A constraint based approach. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, pages 269–280, Piscataway, NJ, USA, 2017. IEEE Press.

[8] Tobias Grosser, Armin Größlinger, and Christian Lengauer. Polly - performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(4), 2012.

[9] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. Program analysis as constraint solving. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 281–292, New York, NY, USA, 2008. ACM.

[10] Sudipta Kundu, Zachary Tatlock, and Sorin Lerner. Proving optimizations correct using parameterized program equivalence. *SIGPLAN Not.*, 44(6):327–337, June 2009.

[11] Sorin Lerner, Todd Millstein, Erika Rice, and Craig Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 364–377, New York, NY, USA, 2005. ACM.

[12] Peter Lipps, Ulrich Möncke, and Reinhard Wilhelm. *OPTRAN - A language/system for the specification of program transformations: System overview and experiences*, pages 52–65. Springer Berlin Heidelberg, Berlin, Heidelberg, 1989.

[13] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. Provably correct peephole optimizations with alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 22–32, New York, NY, USA, 2015. ACM.

[14] Florian Martin. Pag – an efficient program analyzer generator. *International Journal on Software Tools for Technology Transfer*, 2(1):46–67, Nov 1998.

[15] David Menendez and Santosh Nagarakatte. Alive-infer: Data-driven precondition inference for peephole optimizations in llvm. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 49–63, New York, NY, USA, 2017. ACM.

[16] David Menendez, Santosh Nagarakatte, and Aarti Gupta. *Alive-FP: Automated Verification of Floating Point Based Peephole Optimizations in LLVM*, pages 317–337. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[17] Eric Mullen, Daryl Zuniga, Zachary Tatlock, and Dan Grossman. Verified peephole optimizations for compcert. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 448–461, New York, NY, USA, 2016. ACM.

[18] Andres Nötzli and Fraser Brown. Lifejacket: Verifying precise floating-point optimizations in llvm. In *Proceedings of the 5th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, SOAP 2016, pages 24–29, New York, NY, USA, 2016. ACM.

[19] Karina Olmos and Eelco Visser. Composing source-to-source data-flow transformations with rewriting strategies and dependent dynamic rewrite rules. In *Proceedings of the 14th International Conference on Compiler Construction*, CC'05, pages 204–220, Berlin, Heidelberg, 2005. Springer-Verlag.

[20] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. From program verification to program synthesis. *SIGPLAN Not.*, 45(1):313–326, January 2010.

[21] Ross Tate, Michael Stepp, and Sorin Lerner. Generating compiler optimizations from proofs. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 389–402, New York, NY, USA, 2010. ACM.

[22] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for cuda. *ACM Trans. Archit. Code Optim.*, 9(4):54:1–54:23, January 2013.

[23] Deborah L. Whitfield and Mary Lou Soffa. An approach for exploring code improving transformations. *ACM Trans. Program. Lang. Syst.*, 19(6):1053–1084, November 1997.

[24] Jeremiah James Willcock, Andrew Lumsdaine, and Daniel J. Quinlan. Reusable, generic program analyses and transformations. In *Proceedings of the Eighth International Conference on Generative Programming and Component Engineering*, GPCE '09, pages 5–14, New York, NY, USA, 2009. ACM.

[25] Vanya Yaneva, Ajitha Rajan, and Christophe Dubach. *Compiler-Assisted Test Acceleration on GPUs for Embedded Software*, pages 35–45. ACM, 7 2017.

[26] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. *SIGPLAN Not.*, 46(6):283–294, June 2011.

[27] Eric Yuan. Towards ontology-based software architecture representations. In *Proceedings of the 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering*, ECASE '17, pages 21–27, Piscataway, NJ, USA, 2017. IEEE Press.

[28] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. Formalizing the llvm intermediate representation for verified program transformations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 427–440, New York, NY, USA, 2012. ACM.