Compiler Analysis as Constraint Solving: Taming The Complexity With CAnDL

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

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

Abstract

Optimizing compilers require sophisticated program analysis to select suitable optimizing transformations. Implementing analysis functionality is difficult and time consuming. For example, in LLVM, tens of thousands of lines of code are required to detect opportunities for peephole optimizations.

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.

A very simple CAnDL program is shown in Figure 1. The aim is to implement an optimization pass that applies the algebraic property of the square root function in Equation 1 to perform a floating point optimization (we 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.

In Figure 1 (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.

The CAnDL program detects two opportunities to apply the transformation, which are shown in (d) and (e). Both solutions assign values from within the IR code to each of the variables in the CAnDL program such that all constraints are fulfilled. The transformation functionality (f) that generates the optimized code in (g) follows directly.

While this simple example illustrated the main steps in our scheme, we wish to detect much more complex structures in practice. CAnDL has powerful modularity functionality built in that allowed us to formulate common computational idioms such as:

- sparse and dense linear algebra
- generalized reductions and histograms
- stencil kernels

We used the generated analysis functionality as the base for code transformations that map C/C++ programs onto heterogeneous hardware and achieved speedups from 1.26× to over 20× on embedded and discrete GPU configurations.

(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