DSL for writing Compiler Optimizations

Liyi Li (liyili2), Everett Hildenbrandt (hildenb2)

Problem

Translating compiler optimizations from academic papers to an LLVM C++ implementation can be tricky. When this translation is done by hand, there can be no guarantee that it is faithful to the original optimization. This can lead to wasted developer time tracking down bugs in optimization implementations.

To solve this, we'd like to define a domain specific language (DSL) for writing optimizations in. A developer using such a DSL should be able to clearly state exactly the necessary information (and nothing more) to implement the optimization; afterwards the DSL should be compiled into an executable form either by generating C++ for use with LLVM or by making an executable program which operates directly on the LLVM IR syntax.

Ideally, one could write down exactly the "psuedo-code" used in academic papers to specify the optimization and that would be enough to generate the entire optimization. You wouldn't have to learn C++ to do this. By disentangling algorithm design from implementation, we can also avoid any dependencies on LLVM for algorithms written in an optimization DSL. Simply by providing another "backend" for our DSL, we could target other languages.

Previous Work

See bottom for more details on references.

Alive Peephole Optimizations[1]: Lopes et al. define a DSL based on templates for specifying peephole optimizations on the LLVM IR (these templates resemple rewriting logic specifications). Additionally, pre-conditions on the optimizations can be specified (so that the templates do not over-match). Their implementation verifies the transformation correct using an external SMT solver, then generates C++ code which corresponds to the specified optimization.

PTRANS using VeriF-OPT[2]: Mansky et al. define a DSL for writing CFG-based (control flow graph) optimizations. This framework is not specific to LLVM, though they provide a "MiniLLVM" sample language for demonstration. Rather, this work demonstrates the key point that LLVM function optimizations are a specific instance of CFG-based optimizations. The PTRANS DSL is based on CTL (computational-tree logic), which takes quite a bit of familiarity to use correctly.

Project

In this project, we will define a DSL to allow user to use this language to define most optimizations in CFG including Peephole optimizations. This DSL will be entirely generic in the sense that it will work on a general version of CFG which is allowed to instantiated to other languages optimizations which can use these CFGs. We will try to define the syntax and transition rules of the DSL in K and translate it into a version in Isabelle by using our existing semantic preserving translator from K to Isabelle. In this case, we can verify the correctness of the framework of the DSL in Isabelle and automatically generate an optimization tool in Haskell, which is based on the DSL with LLVM semantics defined in Isabelle. The automatic translated code can give us the semantic equivalent translation from the code in Isabelle, which makes the tool in Haskell more convincing. This tool will allow users to generate optimized LLVM programs directly. We can also use the clang compiler to compile the tool in Haskell to a version in LLVM. Another thought is to actually compile the Haskell code to a version in C++ and try to fit the C++ code into the current LLVM framework, which seems a harder task.

The project can be broken down into (roughly) four stages:

- Define LLVM-semantics in K-framework (mostly done)
- Define K-semantics for simple LLVM rewrites ("peephole" optimizations)
- Define K-semantics for CFG rewrite DSL (with PTRANS in mind)
- Implement key optimizations for demonstration purposes

LLVM IR Rewrites

This will involve work similar to the Alive project mentioned above so that we can rewrite individual instructions. However, if we define these rewrite rules in the K-framework we can automatically have an executable backend which performs the rewrites for us. Additionally, we can verify the optimizations by exporting the K rewrite rules (along with the LLVM semantics) to Isabelle.

The LLVM semantics are defined (mostly, though not tested extensively) in the K-framework. We can instantiated our DSL language to a specific language, LLVM and show that we can do optimizations on LLVM programs correctly and efficiently.

CFG Rewrites

The PTRANS DSL (and VeriF-OPT framework) is a powerful CFG-optimization specification and execution framework, but is hard to use because it requires extensive knowledge of CTL. We will make a wrapper DSL which is de-sugared to the CTL used by PTRANS. This gives us the verification power of PTRANS (for optimization correctness verification), but a more manageable specification language for optimization developers to use.

The CFG-rewrite framework specified by PTRANS is language-agnostic. We'll have to tie it back to LLVM by specifying the correct syntax and semantics of LLVM for Isabelle (which PTRANS uses). For this, we can

export the K-semantics of both LLVM and the CFG-rewriter DSL to Isabelle and have Isabelle take care of generating correctness proofs and translation executables.

Implement Optimizations

To demonstrate the correctness of our approach, we will write a few key optimizations in our DSL and prove their correctness (as well as analyzing their performance). Suggested optimizations are peephole optimizations (which we can compare against the Alive implementation), SROA (which we can compare against the LLVM built-in implementation), and perhaps automatic parallelization optimizations.

Test suite

Along our project development, we will use the same test suite to of LLVM semantics to test our optimization tool. The test suite will involve 6000 LLVM programs which mostly are translated from real C programs. The basic idea of the testing is to test whether or not a program will preserve the semantic behavior of the original programs. Whether or not the DSL can catch all the described and desired behaviors are another things that we want to test but we will test it by manually drawing some selected programs and comparing it with the CFGs generated by our DSL. By using this test suite, we can show our tool is useful and is able to handle most LLVM programs.

Proposed DSL Syntax

Here is the first draft (still rough) of the DSL we are proposing. It contains both instruction patterns (which would be used for peephole optimizations) as well as graph patterns (using CTL connectives to describe allowable graph structures).

```
(* CTL spatial quantifiers *)
(*-------------------------*)
syntax CTLQuant     ::= "A"                                    (* for all paths *)
                      | "E"                                    (* exists a path *)

(* CTL Logic Connectives *)
(*----------------------*)
syntax CTL  ::= InstPattern
              | "--" CTLQuant ">" InstPattern                  (* `next` *)
              | InstPattern "<" CTLQuant "--"                  (* backwards `next` *)
              | "--" CTLQuant InstPattern "->" InstPattern     (* `until` *)
              | InstPattern "<-" CTLQuant InstPattern "--"     (* backwards `until` *)
              | "--" CTLQuant "->"                             (* `eventually` *)
              | "<-" CTLQuant "--"                             (* backwards `eventually` *)

(* Instruction Patterns *)
```

```
(*----------------------*)
syntax InstPattern  ::= Term                                (* domain specific *)
                      | CTL                                 (* inclusion *)
                      | PatternName List{Var}               (* allow definitions *)
                      | Var ":" InstPattern                 (* pretty predicates *)
                      | "not" InstPattern                   (* `not` *)
                      | InstPattern " " InstPattern [left]  (* `and` *)
                      | InstPattern "|" InstPattern [left]  (* `or` *)

(* Syntax for defining patterns *)
(*------------------------------*)
syntax PatternDef   ::= "pattern" "[" PatternName List{Var} "]" ":" InstPattern

(* Transformation rule syntax *)
(*----------------------------*)
syntax Rule         ::= InstPattern "=>" InstPattern
                      | InstPattern "=>" InstPattern if InstPattern

(* Define a transformation rule *)
(*------------------------------*)
syntax RuleDef      ::= "rule" "[" RuleName "]" ":" List{Rule}

(* How to build composite transformations *)
(*----------------------------------------*)
syntax Strategy     ::= RuleName                            (* apply rule once *)
                      | StrategyName                        (* apply strategy once *)
                      | Strategy "*"                        (* apply many *)
                      | Strategy ";" Strategy [left]        (* sequence *)

(* Name a list of rules as belonging to a transformation *)
(*-------------------------------------------------------*)
syntax StrategyDef  ::= "strategy" "[" StrategyName "]" ":" Strategy
```

CTL Arrows

CTL is powerful for reasoning about directed graph structures, but is unwieldy to write. We've provided "CTL-arrow" patterns which match exactly the semantics of the typical CTL `until` and `next` operators (we've provided both forward and backwards arrows). Because the arrows are CTL formulae, we can think of them as matching an instruction where the CTL formula represented by the arrow is satisfied at that point in the CFG (or any other directed graph structure).

Example: Dominance

To claim that CTL is useful for CFG-based optimizations, we must show we can express simple concepts like "dominance" and "dominance frontiers". In the following example, we show how we could eliminate parts of the CFG which we know will throw an error because they are dominated by a divide-by-zero instruction.

```
pattern [dominatedBy I] : I <-A--

rule [removeError] : dominatedBy (c = E:Exp / 0) => noop
```

Here, we define a pattern `dominatedBy I`, which will match some other instruction if it is dominated by the instruction `I`. The pattern used is `I <-A--`, which in CTL means "any instruction where all backwards paths go through instruction `I`". When instantiated with `c = E / 0`, this means "any instruction where all backwards paths go through an instruction which calculates an expression `E` divided by `0` and assigns it to some variable `c`".

We also make the rule `removeError` which will rewrite an instruction to a `noop` if it is dominated by `c = E:Exp / 0`.

Example: Dominance Frontier

We could also define a pattern which matches instructions in the dominance frontier of another pattern.

```
pattern [inDominanceFrontierOf I] : dominatedBy I <E-- not dominatedBy I

(* Make the ` ` an explicit `and` *)
pattern [inDominanceFrontierOf I] : (dominatedBy I <E--) and (not dominatedBy I)
```

Here we've used the fact that in our grammar the space character ' is an implicitandoperation. If we made it an explicitand, we would get the second pattern shown above. The second part of the pattern,not dominatedBy I, will match only if the current instruction is not dominated by the instructionI. The first part,(dominatedBy I) <E–, will only match if there exists a predecessor of the current instruction which *is* dominated byI'.

Altogether, it will only match instructions that are not dominated by `I` but have a predecessor which is dominated by `I`. Using the fact that space is an implicit `and` operator, we can write it in the quite intuitive structural manner given above.

Usage

We will define patterns to pick out structures in the graph (thus program) that can be transformed. When a pattern matches, part of the pattern will be re-written using a rule (this is when the optimization happens). We can control the order to try rules in using strategies which allows us to sequence rules as well as apply rules as many times as possible.

Example: Constant Propagation

Using this DSL, we imagine defining constant propagation in this way:

```
pattern [termUse c t] : c = t <-A not (c = t')-- uses c

rule [constProp] : I => I[t/c] if I:(termUse c (t:const))

rule [constFold] : c1:const +  c2:const   =>   c1 +Int   c2
                   c1:const *  c2:const   =>   c1 *Int   c2
                   c1:const -  c2:const   =>   c1 -Int   c2
                   c1:const && c2:const   =>   c1 &&Bool c2

strategy [CONST] : (constProp* ; constFold*)*
```

Pattern `termUse` will only match instructions which use `c` (the `uses c` part) and match the pattern `c = t <-A not (c = t')--`. The part `c = t <-A ... --` means that all backwards paths from the instruction must go through an instruction that matches `c = t`. The part `not (c = t')` says that along all such backwards paths, it must *not* match the instruction `c = t'`. This captures the fact that the instruction of interest definitely uses the term `t` stored in variable `c`.

To make the `constProp` rule, we instantiate the pattern `termUse` with the second argument having the additional restriction that it must be a constant, to get the final pattern `I:(termUse c (t:const))`. If we find an instruction `I` which matches this pattern, we replace it with `I[t/c]`, which is the same instruction `I` where every occurrence of `c` has been replaced by the constant `t`.

We also provide a `constFold` rule which simply replaces expressions which have constants in them with the expression actually calculated.

Finally, we provide the strategy `CONST`, which will apply `constProp` as many times as possible, followed by `constFold` as many times as possible. The entire sequence of propagation followed by folding will be applied as many times as possible too.

Example: Dead-code Elimination

We can define dead-code elimination as follows:

```
pattern [deadCode] : c = t --A-> not (uses c)

rule [deadElim] : deadCode => noop

strategy [DEADCODE] : deadElim*
```

Here we define the pattern `deadCode` which matches instructions of the form `c = t` (assignment to `c`) where the pattern `not (uses c)` is matched forever on all forward paths (the `--A->` part dictates "all forward paths").

The rule `deadElim` is simple - if some instruction matches `deadCode`, it is replaced by `noop`. The `DEADCODE` strategy repeatedly applies `DEADCODE` until it cannot anymore.

Using this new rule, we could come up with another strategy which repeatedly performs constant propagation and dead-code elimination.

```
strategy [CONST-DEAD]   : (CONST* ; DEADCODE*)*
```

— Does K already have an Isabelle back end? (I was told it does have a Coq back end, but don't know about Isabelle.) I assume it does, because if not, then exporting the K semantics of LLVM to Isabelle could be far too time-consuming. In any case, I encourage you to think about whether to create one DSL instead of two, and if so, which one.

K does not have an Isabelle back-end yet, but Liyi Li has a tool in shallow embedding a K definition into an Isabelle definition.

Proposed Timetable

March 14

- liyili2: Provide K definitions for PTRANS CFG-rewrites, make sure we can unite K definition of LLVM semantics and CFG-rewrites in PTRANS
- hildenb2: Provide DSL for specifying LLVM IR rewrites (in K)

March 28

- liyili2: Provide "sugared" CFG-rewrite DSL in K
- hildenb2: Implement and test some peephole optimizations, compare with Alive

April 18

- liyili2: Implement generic dataflow analysis algorithm using CFG framework
- hildenb2: Make specific CONST dataflow analysis algorithm, ensure that extending generic algorithm to specific instance is "easy"

References

[1] N. Lopes, D. Menendez, S. Nagarakatte, and J. Regehr, "Provably Correct Peephole Optimizations with Alive," *Proc. of the 36th annual ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*. 2015.

[2] W. Mansky, D. Griffith, and E. Gunter, "Specifying and Executing Optimizations for Parallel Programs," *GRAPH Inspection and Traversal Engineering (GRAPHITE)*. 2014.