Correct by Construction Optimizing Compiler

Liyi Li (liyili2), Everett Hildenbrandt (hildenb2)

Project

Algorithmic specifications of compiler optimizations often come with correctness proofs, but their implementations do not; instead bugs that surface in tests are uncovered and fixed afterward. Perhaps we could have a correct by construction optimizing compiler, where the algorithmic proof of correctness carries over to the compiler implementation.

To have a correct by construction compiler, we could generate the correct C++ (or another appropriate low-level language) directly from the mathematical description of the algorithm; the trusted code base here is the translator from the algorithmic language to the execution language. Alternatively, we could execute the high-level language directly; the execution environment of the high-level algorithmic language becomes the trusted code base.

We experiment with the idea of a correct by construction optimizing compiler by providing a tool for writing CFG based optimizations in a high-level algorithmic language. Our high-level language directly executes instead of generating low-level code. The focus has been CGF based optimizations; in general it should be possible to create rewriting logic DSLs for many different classes of optimizations.

Existing Work

Specifying and Executing Optimizations for Parallel Programs (2014): Mansky et al. define the PTRANS graph-rewriting optimization framework using three languages. One language describes simple graph transformations which always preserve the well-formedness of GCFGs[1]. The second language describes CTL side-conditions for compiler optimizations[2]. The third language gives a Kleene algebra[3] for specifying optimizations, where each optimization is a graph transformation along with a CTL side-condition allowing the transformation. [1]

Proving Correctness of Compiler Optimizations by Temporal Logic (2002): Lacey et al. use CTL to specify the side conditions which must hold to run an optimization. They use a simple language with assignments, arithmetic expressions, and branches for demonstration. A few optimization side-conditions are expressed (for dead-code-elimination, constant folding, and code motion). In addition, they provide a way to show that an optimization preserves program semantics. [2]

[1] Generalized control flow graphs. Basic blocks are instructions and each instruction has a type which dictates how it may be used in building GCFGs (eg. "an `if` statement must have exactly two outgoing edges").

[2] This is the analog to the program analysis phase of compilers.

[3] This is the analog of the overall optimization of the compiler, where the order of analysis and transformations is specified. Kleene algebras contain regular expressions, giving quite good expressive power.

Imperative Program Transformation by Rewriting (2001): Lacey et al. describe an executable rewriting specification of various compiler optimizations. The transformations are specified as graph-rewrites, with the observation that algebraic matching can be achieved over imperative languages as CTL side conditions. Their executable prototype scales to hundreds of lines of code. [3]

Design

Optimizing compilers require at least two basic functionalities; program analysis and program transformation. For program analysis, we are using CTL side-conditions on the CFG representation of program code. For program transformation, we are using unification and rewriting over the abstract syntax of the target language. We express both the CTL and target language syntax in the K framework, which allows for prototype execution of the rules. Our target language for optimization is the LLVM IR.

Program Analysis

We use CTL to reason about graph rewrite feasibility because CTL has an efficient proof system [4] and provides a natural way to reason about graph structure. To specify a program analysis, one writes down the corresponding CTL formula. For example, the CTL formula
`use(v) /\ def(v,c:Const) <-A not def(v,e) --`
specifies a node that uses program variable `v` *and* has the constant `c` as the dominating definition of variable `v` along all backwards paths. This formula can be used to enable a constant propagation transformation. Most dataflow analysis admit straightforward CTL formulae.

Note that the variables `v`, `c`, and `e` in the formula above range over expressions in the target language. This rule must be instantiated with concrete expressions from a target program, in general not a trivial process. The CTL formula thus not only specifies when a transformation is correct to apply abstractly, but provides concrete substitutions from formula variables to language expressions when a correct instantiation is found.

Program Transformation

For transformations, we use instruction rewrite rules defined over the abstract syntax of the target language coupled with CTL side-conditions on the local CFG. This is done using unification[4] of instruction patterns with concrete instructions in the program.

For example, the full constant propagation optimization is written as follows:
`I => I[v/c] if use(v) /\ def(v,c:Const) <-A not def(v,e) --`
The side condition limits the rule to points in the CFG which are safe to perform constant propagation on. The rewrite `I => I[v/c]` specifies replacement of occurrences of variable `v` with constant `c` at those points.

[4]Unification (or at least matching) is a core component of most rewriting-engines. By using a rewriting framework we can assume a unification algorithm is available for the abstract syntax of a language.

Target Language: LLVM

Our target language is LLVM IR. LLVM is a Static Single Assignment (SSA) based representation that provides type safety, low-level operations, flexibility, and the capability of representing 'all' high-level languages cleanly. It is the common code representation used throughout all phases of the LLVM compilation strategy.[5] LLVM is a significantly more complicated language than many others for which a correct compiler has been attempted.

K Framework

Given these requirements, we need a framework which can express and execute logical side conditions (for the CTL), as well as perform unification over the abstract syntax of a target language. The K semantic framework allows the operational definition of imperative programming languages, which gives a correct[6] interpreter for the language. The syntax and semantics of CTL are defined in K, allowing execution of program analysis, and the syntax and semantics of the target language are also defined in K allowing unification over the abstract syntax of the language. [5]

As unpublished work, the syntax and semantics of LLVM have been defined in the K framework (though not extensively tested). We are using this definition to provide predicates over the LLVM language (such as `use(v)` or `def(v,e)`). The remaining work to be done is to glue the semantics of CTL and LLVM together. This can be quite cumbersome given some non-modularity issues that K has.

Implementation

Prior Analysis Required

As we are building a framework for specifying analysis and optimization, the only prior analysis necessary is that the code have a CFG representation where the basic blocks are instructions. LLVM already has a CFG representation, so this requirement is almost met.

To go from an LLVM CFG to a CFG with instructions as basic-blocks, Liyi had to implement an instruction labeller. After the input program is parsed and loaded into the configuration, the instruction labeller scans the code and assigns each instruction a unique integer. This pass also stores off which instruction is the entry instruction and which is the exit instruction, as well as filling in which other instructions are the predecessors and successors of each instruction.

Major Code Components

LLVM Specification

LLVM has largely been defined in the K framework, though the definition has not been equipped with the necessary tooling to do program analysis and optimization. We've had to modify the LLVM "configuration"

[5] From the LLVM Reference Manual http://llvm.org/docs/LangRef.html

[6] Correct with respect to the defined semantics. There can be bugs in the defined semantics, the advantage of using K instead of English being that the semantics are immediately executable and therefore testable.

(see file `llvm-configuration.k`). The configuration is how K stores the state of an executing LLVM program, and includes information both about the executing program (eg. function definitions and basic blocks), as well as the runtime (eg. what each memory location is storing).

We've added K cells to the configuration which hold information about the optimization process. For example, the patterns describing analysis contain abstract variables, to actually use the analysis we store the concrete substitution for those variables in a map cell: `<substs> .Map </substs>`. Once it's proven that a given CFG models a CTL formula at some instruction, the `<substs>` cell holds the relevant bindings from variables in the CTL formula to language expressions in the program.

CTL Specification

The definition of CTL has been provided by the modules `LLVM-CTL-SYNTAX` and `LLVM-CTL` in the file `llvm-ctl.k`. Formulae are checked for satisfaction incrementally by checking sub-formulae and combining the results. If the base formula were simple atomic propositions, we could label each node with the atomic propositions that are true and in a straightforward manner incrementally calculate the truth of larger formula. However, the atomic predicates are not simple, they contain variables which must be substituted for concrete language sub-expressions (eg. `v` in `use(v)`).

An atomic predicate becomes a map from variables to concrete expressions for which the atomic predicate is satisfied. The standard rules for combining sub-formulae to determine satisfaction of larger formula have to take into account the substitution maps of the sub-formulae. The module `LLVM-CTL` handles this by defining the corresponding set and map operations for the boolean algebraic operations of CTL. Thus, `not` means set complement (look for variables not mentioned by the instruction), `or` means map union, and `and` means map intersection. Additionally, the path predicates of CTL (such as `E-> P` "there exists a succesor where `P` is true") must have the corresponding substitution map algebras defined for them. This is much trickier, and what we are currently working on.

LLVM Integration

Also provided is a simple syntax for connecting CTL atomic predicates to the abstract syntax of the target language.

```
rule use(v) =>  theEq(theAdd(T, hasType(v, var), hasType(b, var))
                             ,valValue(v),valValue(b),normal)
                theEq(theSub(T, hasType(v, var), hasType(b, var))
                             ,valValue(v),valValue(b),normal)
                ... /* many more here */
                theEq(A,false,otherwise)
```

This demonstrates how one would define the atomic predicate `use(v)` for the LLVM language. The right-hand-side of the `use(v)` rule is a list of potential ways an LLVM instruction can use the variable `v`, using the abstract syntax of LLVM (defined in `llvm-abstractsyntax.k`).

The goal of this project is to provide a framework for specifying abstract optimizations over CFGs, only at the very end plugging in the specific language details. This means that a client (some target language) must enumerate all the ways that each atomic predicate can be satisfied by a program instruction. Ideally

this would be given with the language definition (instead of created after the fact), as these descriptions are inherently semantic valuations about the language itself. Additionally, there is no restriction on the atomic predicates that a user may define; they can provide new atomic predicates which they wish to perform dataflow analysis with.

Testing Strategy

So far we have only been hand-testing the semantics of CTL formulae solvers on toy-graphs (where we explicitely provide which atomic predicates are true at each node). We also have tested the instruction labelling to make sure we can have CFGs with instructions as basic blocks (instead of LLVM basic blocks).

Implementation Status

The code is not ready for use on real LLVM programs. We are still working on the CTL satisfaction calculation. The hard part has been determining the correct ways to combine substitution maps given arbitrary CTL formulae.

Our initial CTL satisfaction equations (on the `ctl-optim` branch) were not sufficient in this regard. They did not take into account the fact that atomic predicates would have an implicit substitution associated with whether they were satisfied at a node or not. Care must be taken in combining the substitution maps, as explained above.

Getting the Code

You can find the code online on Liyi's Github page: https://github.com/liyili2/llvm-semantics-1. There you will find the directory `semantics` which contains the definition of the LLVM semantics. The file `llvm-ctl.k` contains the main work towards integrating CTL with the LLVM, with more experimental work happening on the `ctl-optim` branch of the repository. We also have a root-level directory `tests` which contains various tests. We've been workuing mainly with the handwritten tests, as well as the `ctl-optim` tests on the `ctl-optim` branch, which are for testing CTL semantics directly (not the integration with LLVM).

References

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

[2] D. Lacey, N. D. Jones, E. Van Wyk, and C. C. Frederiksen, "Proving Correctness of Compiler Optimizations by Temporal Logic," *SIGPLAN Not. 37 (1)*. 2002.

[3] D. Lacey and O. de Moor, "Imperative Program Transformation by Rewriting," *Lecture Notes in Computer Science*, vol. 2027. 2001.

[4] A. Pnueli and Y. Kesten, "A Deductive Proof System for CTL*," *Lecture Notes in Computer Science*, vol. 2421. 2002.

[5] G. Rosu and T. F. Serbanuta, "An Overview of the K Semantic Framework," *The Journal of Logic and Algebraic Programming*, vol. 79(6). 2010.