Scheme Workshop 2021 Script

Title:

Davis:

Hi, I'm Davis Ross Silverman, from Syracuse University!

Yihao:

Hi, I'm Yihao Sun, from Syracuse University!!!!!!!!!!!!!!!

Davis:

And our co-authors are Kristopher Micinski from Syracuse University and Thomas Gilray from University of Alabama at Birmingham.

Yihao:

Today we will be talking about our paper, "So You Want To Analyze Scheme Programs With Datalog?".

<CLICK>

Introduction:

Yihao/Kris?:

This presentation covers three main areas:

First, we will be describing the abstract machine and the analysis that we create within it. We use a modified CESK machine and the m-CFA analysis as defined by Might et al. Our machine features flat closures, a direct style interpreter, and a subset of Scheme with interesting features not usually explicitly implemented.

Next, we implement our machine in a modern Datalog dialect, Souffle. Utilizing features such as algebraic data types, we create a program that maps closely to the operational semantics of the abstract mac

Finally, we will evaluate our program in a worst case program designed from Van Horn's work. We find that Souffle has broken parallel scaling, but still performs quite well on a single threaded run.

<CLICK>

ControlFlowProblem:

Davis:

Computing Control Flow is a fundamental problem in optimizing Scheme programs. In the following example, we can not compute a static control flow graph at compile time. To know which function will be called after the conditional, we must first know which closures flow to 'f' and 'g'. We have a chicken and egg problem. To know the control flow, we require knowledge of the data flow. But to compute a data-flow graph, we first need a control flow-graph!

With Abstract Interpretation, Control Flow Analysis, and Abstract Machines, we can get a meaningful solution to this problem.

<CLICK>

AbstractInterpretation:
Yihao:

Abstract Interpretation allows computing program values statically without dynamic runtime information. Unlike a normal interpreter, it will only give an over approximated running result of the target program.

For example, the if expression shown in slides will be evaluated to either 1 or 2 in a normal interpreter. However, in an abstract interpreter, we may only conservatively evaluate the value of guard expression into top value, in that case, the abstract interpreter will evaluate both branches and the result is the set {1,2}.

<CLICK>
AbstractMachines:
DAVIS:

To write our abstract interpreters, we utilize the theory of Abstract Machines. Abstract Machines provide our theoretical background for writing interpreters. With mathematically defined transition rules, we can treat each transition as a discrete time step. This is important during abstract interpretation, as we need to formally separate computations.

Our machine utilizes a modified CESK machine. These letters stand for Control, Environment, Store, and Continuation. We define the control as the syntax we are evaluating, or as the value we are applying. The environment identifies the set of live variables in scope, and will act as the arbiter of precision in our abstract interpreter. The store is where values are stored, and is analogous to the heap. Finally, the continuation is our stack, showing us what to do after we reach an atomic value in our control.

<CLICK>
AbstractAbstractMachines:
Davis:

Our machine utilizes the ‘Abstracting Abstract Machines’ approach. The ‘AAM’ style allows us to easily make abstract interpreters for direct style languages. In the past, writing an abstract interpreter could only be done for simple abstract machines. This meant that a source program had to be translated to a significant subset of the language, such as continuation passing style, before it could be analyzed. AAM gives us all the benefits of abstract interpretation, that is, soundness and decidability, for more complex machines, such as a CESK.

Soundness of an abstract machine lies in the results given. If the concrete result is in the set of results returned by the abstract machine, it is sound. Decidability is the guarantee that the program will terminate. The AAM paper gives proofs for both of these useful features, if you follow their methodology.

To produce this machine, you must remove recursive structures from the state space. Start with concrete semantics, and finitize the environment: adding a store is a common method. Next, store allocate continuations. This removes a key recursive structure from the state space. Finally, finitizing the address domain via the address allocation function ensures that the machine is decidable.

<CLICK>

ContextWords:
Yihao:

The environment of an abstract machine is the set of live variables in the current scope. In an abstracted abstract machine, we also use this as the point of precision. Precision is how much the machine will differentiate states before conflating states for space and time reasons. Limiting precision is necessary to ensure decidability. A machine with infinite precision is a concrete machine. Concrete machines are undecidable of course, thanks to the Halting Problem.

Binding sites are used as context, when new variables are bound, the site is added to the context. We implement a type of analysis called ‘m-CFA’. In a traditional control flow analysis, the context is only added to, but never changed directly. In m-CFA, the context flows with the call stack. When variables are ‘popped’ after a call, the context is popped as well. The context is limited to a small number, which will cause conflation of states in non-trivial programs. This number is generally 0, 1, or 2, but can be any natural number.

<CLICK>
ContextExample:
Davis:

Contexts are used in our machine to control precision and therefore running time. With more context, the machine keeps track of states more thoroughly. When the maximum precision has been reached, the machine may conflate states. In a concrete machine, states are never conflated, only one value must be returned. In an abstract machine, two states may be conflated by the machine to conserve time and space. This may lead to multiple possible values for a given expression or program.

In Shiver’s dissertation, contexts are called ‘contours’, in the AAM paper, they are called timestamps. In more recent work, they are called contexts. We continue to call them contexts.
<CLICK>

In the highlighted section of the example, the context is the two let expressions that form the variables bound.
<CLICK>
When the if expression is evaluated, the context does not change as the variables bound have not changed.
<CLICK>
When the body of the lambda is being executed, the ‘x’ variable is bound, and the context is changed as a result.
<CLICK>
This example shows a machine with at least 3 levels of context-precision: a 3-CFA. To make a machine that more easily conflates states, the maximum number of states can be limited. In a 2-CFA machine, the context in the lambda body would be ‘call0’, ‘let1’, dropping ‘let0’.

<CLICK>
DatalogAndHorn:
Yihao:

Datalog is a declarative language initially designed for deductive relational databases. It is very good at power set operations and computing fixpoints, which is exactly what we do in control flow analysis algorithms.

Its semantics are based on Herbrand Interpretation, meaning it can only start from a relation containing fully grounded facts. Thanks to these special semantics, Datalog is decidable and not Turing complete. Abstract interpreters need to be decidable, so using Datalog forces us to write a decidable interpreter.

For CFA, Datalog semantics are powerful enough, but we also need to point out that Datalog can only manipulate power set lattices, so not all program analyses can be trivially expressed using Datalog.

<CLICK>

OpSem1:

Davis:

In our implementation, we support a significant subset of Scheme. Including let, multi argument lambdas, assignment, and first class continuations. These are all important features of a useful analysis for Scheme programs. This subset is easy to compile to, while still maintaining the main features of Scheme programs. We believe that the key to understandable analyses is to analyze a high level program, and not a low level representation such as continuation passing style.

By partitioning our control structure into two different types, that is syntax evaluation, and value application, we have two main types of rules. This slide features syntax evaluation rules on the right. These rules evaluate syntax to eventually produce a value. Here, we will be focusing on the E-If and E-Let rules, for ‘if’ and ‘let’ expressions respectively.

The E-If rule is the most straightforward. The control transfers to the guard, and a continuation is created to branch after the guard is properly evaluated. The ‘alloc k’ function utilizes Gilray’s ‘Pushdown For Free’ allocator to get pushdown control flow precision for free.

The E-Let rule showcases nondeterminism in Scheme semantics. The bindings in the expression are all evaluated at once, so the rule transitions to multiple output states. Eventually, these states all transition to the same let-body state.

Another interesting feature of these semantics can be found in context creation. Contexts are made in the evaluation rules and stored in continuation frames, but are not transitioned to until an application rule uses it.

<CLICK>

OpSem2:

Davis:

Once the machine transitions to an atomic value, an ‘apply state‘ is created, a state with the control set to a value. Apply states consult the current continuation to determine how to transition, and may not utilize the control value directly.

The two A-If rules transition based on the control value. If the value could be false, it will transition to the else branch, else, the then branch is taken. In a soundly approximated program, a value may be both false and something else. In that case, both rules are applied, and two states are transitioned to.

The A-Let rule is where let bindings coalesce to. Because most of the work is done in the E-Let rule, where the let expression is evaluated, this transition rule is quite simple. The important part of this rule is how it transitions with the context created in the evaluation rule.

The A-Call rule requires context copying. Because our implementation utilizes flat

closures, we must copy variables into new contexts when evaluating a closure's body. We copy the closures environment into the environment we will be using for the lambda's body. In return for this work. Flat closures provide better runtime complexity when compared to traditional linked closures.

<CLICK>
FlattenedFacts:
Yihao:

Implementing CFA using datalog mainly has 2 steps.

First, we need to transform a scheme program into facts so the datalog compiler can take these facts data as input. This step is similar to parsing in a normal compiler, the difference is that a normal compiler converts the original program into an abstract syntax tree, which is an algebraic data structure. However, in most datalog dialects, you can only use a csv-like file as input (which is also called EDB in datalog language), and it has no data structure at all.

We use a racket script parsing input scheme program into flatten csv fact files. For example, this let expression will be converted into one row inside the "let.facts" file. This let expression contains a list of variable binding, since datalog does not support structural data, we need another facts file let list. If you used SQL before, you can think of in the input relation rule, BindId in let rule is a foreign key constraint point to the id, which is the prime key of let_list relation.

<CLICK>
OpSemDlCorrespondence:
Davis:

The Operational Semantics map well to a Datalog implementation.
<CLICK>

Transition rules in operational semantics begin with the input state. This maps to one of the conjunctive clauses of the datalog rule.
<CLICK>

That datalog clause is then joined with another clause specifying the 'if' syntax, a way of descructing the syntax itself.
<CLICK>

Another interesting correspondence is the continuation creation. It is almost a direct translation. Our process for creating the datalog was very straightforward after completing the semantics and the boilerplate datalog code.
<CLICK>

Finally, there is a difference between how stores are handled in the semantics and in the implementation. In the semantics, we use a store that is local to the state. In our full semantics, we globalize the store by combining stores after each state transition. In our Datalog implementation, we place the values and continuations in a global store outright. Because of this, stores are not a part of the state relation in Datalog.

<CLICK>
CFAWorstCase:
Yihao:

In order to evaluate and verify our implementation, we constructed a worst case that can make m-CFA lose precision and run very very slowly.

If we want to beat 1 CFA, we can use the scheme expression shown in the slides, variable f will be bound to lamb_z, and function lamb_z will be applied to different values inside the same let context. During each application, variable z will be bound to a number range from 0

to m. Without lam_x in call_0, 1-CFA will be able to differentiate different call sites in the let binding list, since 1 CFA will track one calling context before.

However if lam_x is added, although in concrete semantics the result of the whole function doesn't change, it will still push a stack frame into context, which causes all 1-CFA conflate the abstract address of variable z, making all z evaluated to a set of value ranging from 1 to m. And then when we evaluate the innermost plus expression we get exponential possible results.

<CLICK>

RunningTime:

Yihao:

The idea of our worst case comes from David Van Horn's PhD dissertation. In his original paper, lam_x is called "padding". If we add more padding to the expression we showed before, we will be able to defeat a CFA with higher precision.

Using a term with 32 let clauses, 4 plus operations, and 1 padding, all analyses with an m- value larger than 1 will finish in almost no time. However for CFAs with not enough precision it will take many minutes to finish.

<CLICK>

ParallelTime:

Yihao:

We use Souffle as our datalog engine, which is supposed to support efficient multi core execution through openmp.

There are a lot of potential areas that can be parallelized in our datalog implementation. For example when our algorithm loses precision, multiple possible states can be computed at the same time, in parallel. When we nondeterministically evaluate different let bindings or function arguments, these computations can again be done in parallel.

During testing, we found that as we added more threads, our analysis ran slower and slower. After searching Souffle's bug tracker, we found that the multi-tasking feature of Souffle has been broken for a long time. Therefore, if we want to scale up our analysis, we need a better Datalog solver.

<CLICK>

Conclusion:

Davis:

In the future, we plan on utilizing a better datalog dialect with better support for structural data types, and better parallelization. We also plan on extending the semantics to add new features such as delimited continuations.

To conclude, we find that abstracting abstract machines gives us the tools to write complex analyses for complex abstract machines. Datalog can be used to implement these semantics and gives us useful guarantees for an analyses.

<CLICK>

Q&A:

Yihao:

Questions or comments?