Demand-Driven Execution

Abstract interpretation without widening

Proposer:
Robert Zhang
jzhan239@jhu.edu
robertzhang.vercel.app

Principal Investigator:
Prof. Scott Smith
scott@cs.jhu.edu
cs.jhu.edu/~scott

1 Abstract

Abstract interpretation is a powerful technique to statically derive program properties that are useful for debugging and optimization. Many such systems have been developed over the years, but a compromise has always had to be made between precision and performance. That is, the more accurate the result of a static analysis, the more program paths the analyzer must spend time exploring. For tools that allow some loss of precision in return for a reasonable performance, a widening step is typically required to repeatedly pass over the program to coarsen the derived result until termination.

We propose a novel execution technique - Demand-Driven Execution[1] - that fundamentally eliminates the need for such a process, promising higher precision and performance while guaranteeing termination and soundness . In addition, it inherently supports higher-order programming and has the capacity to generate induction principles for free, making it more akin to a theorem prover, only fully automated. At the heart of our approach is laziness, simulating a call-stack to defer the evaluation of function arguments to only when needed, hence "demand-driven." This work is implemented in OCaml and is being formally verified in Coq.

To our knowledge, no prior work has been done in this direction and it is our hope that our work catalyzes further innovations in program analysis and automated reasoning.

2 Key Components and Progress

This project roughly divides into two parts: a concrete interpreter demonstrating the essence of our approach and an abstract interpreter that is the crux of this work, whose qualities are explained above.

The concrete interpreter was completed during the proposer's collaboration with Prof. Smith while they were an undergraduate. It works atop a functional programming language we designed and implemented in OCaml, extending lambda calculus with integer arithmetic, conditionals, records, and assertions. There is also a test suite containing tests to verify outputs against those of two kinds of classic interpreters - substitution-based and environment-based.

A working implementation of the abstract interpreter was also done before the proposer started their master's. Exactly as the theory indicates, our tool produces the expected results on test programs. A test framework containing unit tests and property-based tests has been established. Benchmarks against classic forward abstract interpreters were also set up and substantial work will soon ensue on this front. In the process of implementing the abstract interpreter, the proposer also uncovered a few critical bugs/inconsistencies in the written theory and helped refine it. Lately, the proposer implemented a translation from the analysis result to constrained Horn clauses (CHCs) so that a CHC solver (e.g., Spacer) can derive a more intuitive representation of the result, improving the user experience. See Section 4 for an example.

Across both parts, the proposer has been working on mechanizing their semantics and formally proving important properties about them in Coq. Once done, such machine-checked correctness proofs will gain us unparalleled confidence in our tool.

[1] https://github.com/JHU-PL-Lab/dde

Over the course of this project starting in October 2022, ~3,900 lines of code have been written (all by the proposer). The Coq verification itself is estimated to be at least 5,000 lines of code when completed.

3 A Clarifying Example

We now consider a simple example written in our OCaml-like language that illustrates the essence of Demand-Driven Execution through our concrete interpreter. Section 4 will go over a more complex example to demonstrate the behavior of our abstract interpreter.

```
1  let a0 =
2    (* stack: [0] *)
3    (fun x -> fun y ->
4      x + y) 10 in
5  let b1 =
6    (* stack: [1; 0] *)
7    a0 20 in
8  b1
```

look up	stack	action
x	[1; 0]	pop
x	[0]	✓
y	[1; 0]	✓

Figure 1: A simple program and its variable lookups

The program to the left of Figure 1 performs two function applications consecutively to compute the sum of 10 and 20. The number attached to the name of these two applications, `a0` and `b1`, indicates the *label* our language parser assigns to their corresponding abstract syntax tree (AST) node. The point of having these labels is to push them onto the front of a list, representing a call stack, in the order of these function calls. As we shall see next, labels in the call stack will be popped off one by one from the front of the list (last-in, first-out) as our concrete interpreter looks up variables in function bodies. The comments on line 2 and 6 show the content of the call stack due to the two function applications.

The chart to the right of the figure shows the steps the concrete interpreter takes to look up `x` and `y` in the body of the function applied in `a0`. For the simpler case of `y`, the label at the top of the stack - 1 - happens to be the application that binds the parameter `y` to the value 20, so we are done. The right-hand side (the argument) of the application `b1` - 20 - is the value for `y`.

To look up the value of `x`, the top of the stack - 1 - does not correspond to an application that binds `x`. So, we pop off label 1 from the front and check the application with the next label - 0. It happens to be what we are looking for, an application that binds `x` to 10, so we are done.

This mechanism forms the foundation for both our concrete and abstract interpreters. Our lazy language semantics is in fact call by name in that a program expression and a call stack uniquely determine the execution. So, the execution can be cached, giving linear/polynomial time complexity for free on programs like computing fibonacci numbers. Alternative, more standard execution models either (evaluate and) substitute the argument for the function parameter at a function application, or save a mapping from the parameter to the argument in an environment/closure for use at variable lookups.

4 A More Interesting Example

To help build intuition for how our abstract interpreter behaves, Figure 2 illustrates how each stage of the project interprets an example recursive program.

The program on the left recursively breaks down the argument into 1s and add them back up. Our concrete interpreter gives a result of 10, just as any sound interpreter would. On the other hand, our abstract interpreter summarizes the program output as a disjunction between two possible values: 0, or 1 more than the previous computation. The $\circ$, a "stub", represents the disjunction itself. Translating such a data structure into CHCs and solving them with Spacer,

```
let id =
  fun self -> fun n ->
    if n = 0 then 0
    else 1 +
      self self (n - 1)
in id id 10
```

Concrete	10
Abstract	0 | (○ + 1)
CHC solving	$\forall x.\, x \geq 0$

Figure 2: A recursive program and results from each stage of the project

we obtain a more intuitive summary indicating all natural numbers, which is more suitable to be presented to users.

There are a few things to note here. One, our abstract intrepretation is capable of reasoning over higher-order programs and statically inferring recursive properties, while traditional program analyses would go straight to widening a range for the output. Two, notice that neither the analysis result nor the solver's solution is as precise as it can be. For instance, it would be nice if the analysis could also infer an *upper bound*, i.e., $\forall x.\, 0 \leq x \leq 10$. This current limitation has to do with the fact that we have not performed call-return alignment to synchronize the reducing of the argument n with the addition of 1s. Without this step, the analysis simply cannot know that these two "loops" exactly correlate with each other. We plan to work on this as future work (see Milestones).

A challenge in working with Spacer has been that it does not produce least solutions (tight bounds) out of the box. Thus, our current approach is to utilize assertions built into our language to help Spacer converge to the desired result. This is not shown in the example in Figure 2, but the syntax of such an assertion is `letassert x = e in x >= 0`. where `e` is a placeholder for the example program. Translated into a logical assertion, `x >= 0` is equivalent to `(assert (forall ((r Int)) (=> (P0 r) (>= r 0))))`, in SMT-LIB. `P0` is the logical predicate corresponding to the analysis result of the program, 0 | (○ + 1). An ambitious next step is to *infer* least solutions instead (see Milestones).

5 Milestones

Based on the current progress, major goals for the next year include but are not limited to the following 5 milestones. More may arise as we go.

- Develop call-return alignment for reasoning over recursive programs so as to statically derive induction principles. Generate constrained Horn clauses to reflect such an alignment. This will allow us to have even richer analysis results and our tool to be utilized as a theorem prover, only fully automated.

- A comprehensive benchmark suite to compare the performance and precision of demand-driven concrete/abstract execution with classic forwardly running systems. These can include those implemented by us or established tools that come with their own benchmarks, perhaps as part of publications. This will help demonstrate the strengths of our approach and identify areas for improvement.

- Leverage solvers like Spacer to infer least solutions on abstract interpretation results (after they are translated into CHCs), obviating the need to manually specify assertions and further streamlining the user experience. The difficulty with this goal lies in how, to our knowledge, no established CHC/SMT solver provides built-in support for computing least solutions. This is most likely due to insufficient demand from the users of these tools, who mostly use them to *verify* program properties, not to *infer* them.

- Formally verify both concrete/abstract interpreters in Coq. Interesting properties include soundness of the latter relative to the former, equivalence of our demand-driven concrete interpreter to classic concrete interpreters, and even the higher precision of our abstract interpreter over classic systems. This is a significant undertaking that builds the kind of trust in our tool we do not get solely with proofs on paper.

- The proposer is driven to broaden the use case of our tool beyond academic research. A long-term goal they have already been steadily working towards is to make the tool practical for general-purpose use through extending our programming language with more realistic features, like records and assertions. In the future, they hope to apply our abstract interpretation technique to popular functional programming languages like OCaml. Going even further, a challenging yet exciting problem to tackle is extending our technique to support imperative languages like C++ and Rust, bringing over the benefits of our abstract interpretation model to an even wider audience.