A Schematic Pushdown Reachability Language

Zachary Palmer
Swarthmore College
zachary.palmer@swarthmore.edu

Charlotte Raty
Swarthmore College
ratycharlotte@gmail.com

Abstract

Reachability in pushdown automata has a variety of applications in program analysis. As analyses grow more complex, so do the automata they use. While mathematically elegant, the implementation of reachability for complex automata is an engineering-intensive process. We present a DSL designed to simplify the specification of complex pushdown automata and aid in the development of more sophisticated program analyses.

1. Pushdown Reachability in Program Analysis

In a pushdown automaton, one node is said to *reach* another if there is a path between them with a valid series of stack operations. In program analyses, this notion of pushdown reachability and the equivalent notion of context-free language (CFL) reachability are useful in addressing stack validation problems. Early first-order demand-driven analyses [1, 4, 8, 9] use CFL reachability to model the program's call stack to achieve context sensitivity. More recent higher-order analyses [2, 5] use more sophisticated models of pushdown automata to improve performance through an abstract form of garbage collection. Pushdown reachability has also been applied to achieve structure-transmitted data dependence analysis by modeling data flow as a stack of lookup operations[3, 7].

As these program analyses have become more complex, so have the automata they use. In higher-order analyses like PDCFA [5], it is infeasible for the implementation to conduct reachability over a full automaton; instead, the automaton's states are constructed lazily so only relevant states are explored. The implementations of DDPA and DRSF [3, 7] additionally represent bundles of similar transitions abstractly, only adding to the graph those transitions which affect the analysis's result.

While these techniques make their respective analyses computationally feasible, they require significant engineering effort and the resulting source code bears little resemblance to the original theory. In the implementation of DDPA, for instance, the aforementioned transition bundles would be natural to represent as functions. However, the bundles must be added as edges to the automaton's graph and so must be subject to comparison and deduplication. In practice, this requires a manual defunctionalization [10] of the code, a tedious and error-prone process.

2. A DSL for Schematic Pushdown Reachability

We ease the process of specifying complex automata for pushdown reachability queries by introducing a DSL to represent such automata *schematically*[6]. Given such a schema, the programmer can then compute reachability via existing libraries. This DSL is implemented in the form of a series of OCaml PreProcessing eXtension (PPX) tools which allow the user to define reachability schema in terms of simple nondeterministic operations.

Reachability schema are written as functions accepting a state and producing the legal transitions for that state. Figure 1 gives

```
let%continuation_fn lookup state =
  [%pick_lazy
    (* Record Projection rule *)
    ( let%require Assign((x1:variable),
                         Record(rcd:record)) = state in
      let%require Lookup x1' = [%pop] in
      let%require true = equal_var x1 x1' in
      let%require Project(lbl) = [%pop] in
      let x2 = get_field lbl rcd in
      [ Lookup x2 ]
    );
    (* Capture rule *)
    ( let%require Value(v:value) = [%pop] in
      let%require Capture2 = [%pop] in
      let (se1:stack_element) = [%pop] in
      let (se2:stack_element) = [%pop] in
      [ value; se2; se1 ]
    ); ...
  ];;
```

Figure 1. Schematic Lookup Code

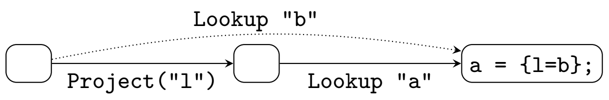


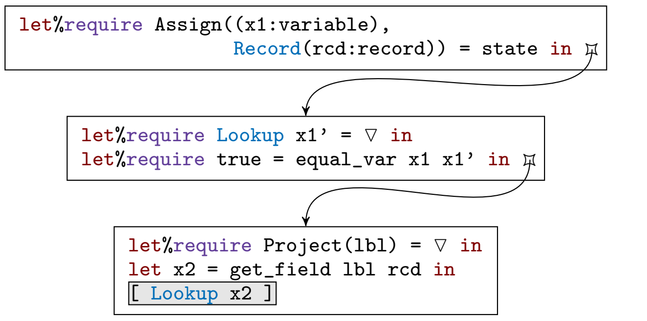
Figure 2. Record Projection: Graph Summarization

a simplified partial schema for the aforementioned DDPA lookup function, which uses states to represent program points and stack elements to represent lookup tasks. Each expression given within the `pick_lazy` extension is executed nondeterministically. Let us consider the Record Projection rule.

The first `let` binding in this rule requires that the state represent a program point which assigns a record to a variable; the `require` extension ensures that computation stops for this rule (and the PDS is unaffected) if its pattern does not match. The rule then pops the topmost element of the stack and verifies that it is a lookup for the variable at this assignment. Finally, we pop the next element of the stack and use it to project the appropriate field from the record. In DDPA, records are shallow; this leaves us with the task of looking up the variable used to define that field.

The code in Figure 1 is written operationally as if we are working with an actual stack. In practice, such an algorithm is not viable: the automata used in program analysis commonly include stack-modifying cycles and so a simple all-paths walk of the graph will not terminate. Pushdown reachability is typically computed by graph summarization as depicted in Figure 2. As per the rule above, we have two pushed stack elements – a lookup and a projection – arriving at a node which defines a record. We can summarize these two edges with another edge which simplifies the task at hand.

Analysis of more complex language features demands summarizations that span many stack elements; for instance, the Capture rule in Figure 1 includes four pops. But the best runtime complex-



```
let%require Assign((x1:variable),
                   Record(rcd:record)) = state in ⌑
```

```
let%require Lookup x1' = ▽ in
let%require true = equal_var x1 x1' in ⌑
```

```
let%require Project(lbl) = ▽ in
let x2 = get_field lbl rcd in
[ Lookup x2 ]
```

Figure 3. Record Projection: Fragment Set

ity is achieved by summarizations which operate on pairs of edges, so rules such as Capture must be completed in multiple steps, with each step recording its progress as a continuation within the graph. The transformation of complex rule into multi-step processes occurs when the DSL's macros are expanded.

3. Macro Expansion

To translate a reachability schema into a series of pairwise operations, we must:

1. Break the schema into a set of fragments around pop operations
2. Define a continuation for each fragment in the schema
3. Defunctionalize the continuations to store them in graph edges
4. Encode nondeterministic semantics into OCaml code

This translation process involves two steps: a *global* step which addresses the first three points above and a *local* step which addresses the fourth. The bulk of theoretical development in the global step centers around the representation of code fragments. A code fragment in this model is an AST decorated with

- A series of indexed *evaluation holes*: result expressions which must be transformed to produce appropriate continuation values
- A series of indexed *extension holes*: points in the AST where a pop is expected and so no value yet exists
- An *input hole*, present on all fragments except the first, which evaluates to the most recently evaluated pop or result value

The global step consists of mapping each traditional AST constructor (e.g. for let expressions) to a corresponding fragment set constructor (e.g. building a let fragments). Figure 3 shows the fragment set for the previous Record Projection rule; ⌑ denotes an extension hole, ▽ denotes an input hole, and a shaded background denotes an evaluation hole. The construction of fragments also produces metadata regarding free and bound variables; this metadata is then used to define the continuations that retain evaluation state between pops.

The continuation type defined in part by the Record Projection rule appears in Figure 4; the remainder of the type and the generated code is omitted for brevity. `Continuation15` carries the variable `x1` and the record `rcd` so that they are available when an element is available to be popped. `Continuation16` only carries `rcd`; `x1` is not used after the second pop. These continuations represent the second and third fragments appearing in Figure 3.

4. Performance and Utility

We experimented with this DSL by reimplementing DDPA's lookup function. DDPA's implementation includes a collection of microbenchmarks used in related higher-order program analyses, so

```
type continuation =
  | ...
  | Continuation15 of variable * record
  | Continuation16 of record
  | ...
```

Figure 4. Continuation Type

we evaluated performance and correctness using those programs. The translation performed by the DSL was correct: it achieved the same result on those benchmark programs as the original analysis. The lookup function written in this DSL incurred an average 9% performance overhead. However, while the original implementation of DDPA's lookup function is ∼1300 lines of code and took multiple weeks to develop from a specification, the reachability DSL implementation of that lookup function is ∼900 lines of code with a development time of one day. This suggests that, at present, a hand-tuned implementation may be desirable for performance purposes but that the DSL is useful for explorative research.

As a next step, an examination of the performance overhead is warranted; we have not yet profiled the resulting code. In principle, the DSL implementation should run *at least as fast* as the hand-written implementation: there is no information hidden from the DSL and the automated defunctionalization makes available a range of optimizations (such as continuation sharing) which may not be evident in or desirable for manually maintained code.

Additionally, a deeper exploration of the tool's utility is warranted. Our experience suggests that the DSL eases pushdown reachability specification, but this evidence is presently anecdotal.

5. Broader Applications

The need for the continuation passing, defunctionalizing approach described above stems from a key property of this problem domain: the continuations must be subject to comparison and deduplication. In particular, the functions represented by continuations are known to be idempotent on the graph to which they add edges and we must take advantage of this idempotency to guarantee termination of the summarization algorithm. It is this constraint that prevents the task from being completed by a simpler tool or even at runtime by e.g. a monad. For this reason, we suspect that this translation process (and the artifact which implements it) may be reapplied in a domain which can make use of a similar idempotency guarantee.

References

[1] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. A practical framework for demand-driven interprocedural data flow analysis. *TOPLAS*, November 1997.

[2] Christopher Earl, Ilya Sergey, Matthew Might, and David Van Horn. Introspective pushdown analysis of higher-order programs. ICFP 2012.

[3] Leandro Facchinetti, Zachary Palmer, and Scott F. Smith. Relative store fragments for singleton abstraction. In *SAS 2017*.

[4] Susan Horwitz, Thomas Reps, and Mooly Sagiv. Demand interprocedural dataflow analysis. SIGSOFT '95.

[5] J. Ian Johnson, Ilya Sergey, Christopher Earl, Matthew Might, and David Van Horn. Pushdown flow analysis with abstract garbage collection. *Journal of Functional Programming*, 24(2-3), 2014.

[6] Zachary Palmer and Charlotte Raty. Pushdown reachability DSL. `https://github.com/JHU-PL-Lab/pdr-programming`, 2018.

[7] Zachary Palmer and Scott F. Smith. Higher-order demand-driven program analysis. In *ECOOP 2016*.

[8] Thomas Reps. Shape analysis as a generalized path problem. PEPM '95.

[9] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. POPL '95.

[10] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference - Volume 2*, ACM '72.