PoTATo: Points-to Analysis via Domain-Specific MLIR Dialect

Robert Konicar, and Henrich Lauko

Fork me on GitHub
github.com/jezurko/potato

Goals of Points-to Analysis MLIR Dialect

Problem Simplification

PoTATo streamlines the points-to analysis by reducing it to a conversion task into its specialized dialect. The dialect's simplicity facilitates the straightforward conversion of any MLIR program representation into it for subsequent analysis.

Efficiency

Encoding the points-to problem in a dialect enables the application of compiler optimizations to reduce the problem size. Furthermore, by keeping solely on essential information the complexity of the analyzer is also reduced.

Flexibility

PoTATo is built with adaptability in mind, enabling users to conduct a range of points-to analyses. By choosing the points-to lattice representation and adjusting the conversion process, users can customize the analysis.

Integration

PoTATo is a tool for points-to analysis. Using a novel approach, we try to reduce the problem size to provide faster analysis without losing information. We have designed a simple MLIR dialect that models memory effects in the program. To simplify the problem, we apply compiler-style optimizations to the obtained IR. In cooperation with other tools, we then extract the analysis result back to the source IR.

Generic Toolchain

- Source
- MLIR Dialect with Pointers
- Dialect with Points-to Metadata

Leveraging the power of VAST Tower of IRs, we can precisely link the PoTATo IR to the original representation. Using this information in hand, we can proxy the aliasing queries about the program in the original IR and extract the information from the computed analysis. The proxy, after getting the mapping of the IRs, extracts the analysis results using the MLIR dataflow analysis framework and answers the users queries.

VAST Toolchain

- C/C++ Source
- VAST IRs
- LLVM IR dialect

Analysis Procedure

Transform. *The transformation step, provided by the user of the analysis tool, abstracts away point-to-irrelevant information by representing the program in the PoTATo dialect. Here, the user can opt for including field-sensitive operations and types in the resulting IR, toggling this aspect of the analysis. The tool does not mandate the user to lower the control flow operations and function calls, as long as the source dialect implements relevant MLIR interfaces. Users only need to provide abstractions for pointer manipulation, or they can use the conversion from the LLVM.*

Source IR → Transform → PoTATo IR

Simplification. *Leveraging the dialects's simplicity, we can apply canonicalization to reduce the size of the problem, thus accelerating the points-to analysis. A key step is constant folding, which eliminates irrelevant operations in the analyzed IR.*

Points-to Analysis Algorithm

Analysis. *The points-to analysis procedure is implemented using the MLIR dataflow framework, enabling the flow-sensitive analyses. When integrating the analysis into the framework runner, users can select their preferred lattice representation, or provide their own implementation, which determines the analysis algorithm applied.*

Points-to Analysis Result

Result. *Utilizing location metadata or VAST Tower, the domain-specific IR remains linked to its original representation, enabling querying of the analysis result via the MLIR data flow interface. Each IR location is associated with a lattice value representing the analysis result.*

PoTATo Dialect

Field-Insensitive Dialect

Memory allocation *abstracts all location creations for points-to analysis, including both local stack or heap allocations.*

```
%var = pt.alloc : <type>
```

Memory dereference *represents all operations that access memory content, such as the load operation in LLVM.*

```
%val = pt.deref %var : <var-type> → <val-type>
```

Memory assignment *denotes writing to memory content, copying its points-to information into the content. It has similar semantics to the LLVM store operation.*

```
pt.assign %dst = %src : <dst-type>, <src-type>
```

Copy *abstracts all operations that do not alter points-to information. It transfers it from the source to the destination. Such operations are pointer casts and pointer arithmetic in the case of field-insensitive analysis.*

```
%dst = pt.copy %srcs* : <src-types> → <dst-type>
```

Address *operation is used to abstract operations that create references, suiting it to model more high-level manipulations with addresses like* `&val` *in C.*

```
%addr = pt.address %var : <var-type> → <addr-type>
```

Constant *operations models all non-pointer values. The value-less constant enables the efficient elimination of all points-to irrelevant computations. Whereas, a valued constant can be used to obtain values for more sensitive analyses.*

```
%const = pt.const : <type>
%const = pt.valued_const <val> : <type>
```

Dataflow Reduction

Compile

```
1: %one = llvm.mlir.constant(1 : index) : i64
2: %a1  = llvm.alloca %one x i32 : (i64) → !llvm.ptr<i32>
3: %i   = llvm.ptrtoint %a1 : !llvm.ptr<i32> to i64
4: %o   = llvm.add %i, %one : i64
5: %a2  = llvm.inttoptr %o : i64 to !llvm.ptr<i32>
6: %x   = llvm.load %a2 : !llvm.ptr<i32>
```

Transform to PoTATo IR

Describe Source IR memory interactions

```
1: %one = pt.constant : i64
2: %a   = pt.alloc : !llvm.ptr<i32>
3: %i   = pt.copy %a : !llvm.ptr<i32> → i64
4: %o   = pt.copy %i, %one : i64, i64 → i64
5: %a2  = pt.copy %o : i64 → !llvm.ptr<i32>
6: %x   = pt.deref %a2 : !llvm.ptr<i32> → i32
```

Simplify

Reduce PoTATo IR using constant folding

```
1: %a= pt.alloc : !llvm.ptr<i32>
2: %x = pt.deref %a : !llvm.ptr<i32> → i32
```

Default Potato IR corresponds directly to the standard interpretation-based points-to analysis. Each source IR operation has a corresponding PoTATo IR operation, which transforms points-to sets. However, it includes numerous irrelevant operations for points-to analysis, which have no impact on the analysis result. In particular, copies do not modify points-to information in this example.

```
State in: loc("potato.mlir":1)
  var0: %one = pt.constant : i64 → {}
  ...
State in: loc("potato.mlir":3)
  var0: %one = pt.constant : i64 → {}
  var1: %a   = pt.alloc : !llvm.ptr<i32> → {mem_loc1}
  var2: %i   = pt.copy %a : !llvm.ptr<i32> → i64 → {mem_loc1}
  ...
State in: loc("potato.mlir":6)
  ...
  var4: %o  = pt.copy %i, %one : i64, i64 → i64 → {mem_loc1}
  var5: %a2 = pt.copy %o : i64 → !llvm.ptr<i32> → {mem_loc1}
  var6: %x  = pt.deref %a2 : !llvm.ptr<i32> → i32 → {}
```

We can streamline the points-to analysis by simplifying the IR using the common MLIR canonicalization mechanism. For instance, we can fuse (constant-fold) the points-to analysis metadata of all copies into a single state before dereferencing.

```
State in: loc("simple.mlir":2)
  var0: %a  = pt.alloc : !llvm.ptr<i32> → {mem_loc1}
  var1: %x  = pt.deref %a : !llvm.ptr<i32> → i32 → {}
```

The analysis result can be obtained by following the chain of meta-locations. For example, if we have %o in LLVM, it corresponds to %o in the PT dialect. From there, we can trace to the fused location in simplified IR and retrieve the corresponding points-to set from %a.

This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA). The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

MUNI FACULTY OF INFORMATICS