Design of Pointer and Array Memory Objects

1 Motivation

Consider the following simple program involving pointer dereferencing expressions.

```
void fun()
{
  int val, *p, **q;
  p = &val;
  q = &p;
  *p = 10;
  **q = **q + 1;
}
```

Listing 1: Motivating Example

Inorder to correctly determine the value of `*p` and `**q` in the piece of code in Listing 1, the constant propagation analysis needs to implement the transfer function for the = operator (SgAssignOp) in which the dataflow state on rhs is transferred to the lhs. Listing 2 shows a partial implementation of the transfer function. In general the client (constant propagation analysis or any analysis) requests the composer for memory objects corresponding to parts it may encounter in its transfer functions. Section 3 briefly discusses the interactions between client, server and composer. Similar to our example in Listing 1, the dereferencing expression on the lhs and rhs of the = operator in the last two statements can be arbitrarily complex, often involving arrays. We call these dereferencing expressions on lhs, rhs as dereferencing sub-expressions. The lhs of = operator should always be a memory object corresponding to a location (see Section 2 for the semantics of the equals operator). The default implementation (syntax analysis) responds to the query `composer->Expr2MemLoc()` with PointerExprObj and ArrayExprObj for the dereferencing sub-expressions involving pointers and arrays respectively. ExprObj are memory abstractions for temporary memory locations and they do not correspond to a particular memory location. In this example, modifying the state for PointerExprObj on lhs as returned by syntax analysis is incorrect since it does not correspond to the actual memory location. The framework however supports getDereference() method for Pointer/Array based objects. The semantics of the getDereference() method is to return the object pointed to by the corresponding dereferencing sub-expression. The analysis writer is burdened with interpreting these complex dereferencing sub-expressions using only the getDereference() to get to the valid memory object. For Array dereferencing sub-expressions, this is even difficult as it involves index.

```
void visit(SgAssignOp* sgn)
{
  MemLocObjectPtrPair lhsMem = composer->Expr2MemLoc(sgn->lhs_operand());
  MemLocObjectPtrPair rhsMem = composer->Expr2MemLoc(sgn->rhs_operand());
  // Get Lattices for the lhsMem, rhsMem, '=' operator
  // Copy the state from rhs to lhs and to the '=' operator
}
```

Listing 2: Transfer Function for Equals Operator

Instead of the getDereference() method, a cleaner API should treat Pointer/Array dereferencing sub-expressions similar to variable references and the composer/server analysis should be responsible for returning the appropriate memory object for these expressions. The client should be completely ignorant of the complexity of the sub-expression and merely query the composer for the memory object of a given sub-expression. There are two options.

- Interpret dereferencing sub-expressions in Server Analysis (Section 4)
- Interpret dereferencing sub-expressions in Composer (Section 5)

NOTE: How should we implement `Expr2Val(**q)` on the rhs of an expression.

2 Semantics of Equals Operator

The = operator distinguishes mutable and immutable memory regions. In C/C++ the semantics of = operator is an assignment to a mutable memory region similar to the construct `set!` in scheme and `mutable` in OCaml. In scheme, the statement `(set! a 5)` results in the assignment of 5 the to the scalar 'a' and 'a' can be reset to any value later in code. Immutable memory in C/C++ correspond to temporary sub-expressions such as the value of `a+b` similar to `let` and `define` bindings in scheme. Note that 'a' or 'b' might themselves be mutable, however the temporary value of the sub-expression `a+b` is not mutable. In the current framework, we should identify all syntactically valid sub-expressions that may result in an assignment to a mutable memory region (for exampe SgVarRefExp, SgPointerDerefExp when appearing on the lhs) and the framework should make sure that it returns valid memory objects for these sub-expressions and not ExprObjects.

3 Composer

Before discussing the interface it is important to understand the interactions between the client, server and the composer.

3.1 ComposedAnalysis

Every dataflow analysis inherits ComposedAnalysis. A ComposedAnalysis may optionally implement the following functions.

```
ComposedAnalysis::Expr2Val()
ComposedAnalysis::OperandExpr2Val()
ComposedAnalysis::Expr2MemLoc()
ComposedAnalysis::OperandExpr2MemLoc()
ComposedAnalysis::Expr2CodeLoc()
```

Listing 3: Composed Analysis Interface

These functions are useful for the analysis to export its results to the other client analysis interested in its results. Additionally each analysis maintains a pointer to the Composer.

3.2 Composer

Any implementation of a Composer should imeplement the following functions.

```
Composer::Expr2Val()
Composer::OperandExpr2Val()
Composer::Expr2MemLoc()
Composer::OperandExpr2MemLoc()
```

```
Composer::Expr2CodeLoc()
```

Listing 4: Composer Interface

A client analysis in its transfer function invokes `composer->Expr2*()` or `composer->OperandExpr2*()`. Note that all analysis maintain a pointer to the Composer. The Composer maintains the list of analyses that has already finished. The composer also maintains a pointer to each analysis. In its implementation of `Composer::Expr2*()`, the composer calls `analysis->Expr2*()`. If the analysis does not have an implementation for `Expr2*()`, then an exception is thrown which is picked up the composer and the composer tries again with other analyses higher up the chain.

4 Interpret Dereferencing Sub-Expressions in Server Analysis

This approach is intuitive and framework already supports a way to have this functionality.

- Client makes the query `composer->Expr2MemLoc(sub-exp)`
- The composer passes this query to a server analysis by calling `Expr2MemLoc` of the server analysis
- The server analysis should handle dereferencing sub-expressions in its implementation of `Expr2MemLoc`
- Server analysis finds the object for the given derefrencing sub-expression and returns the object to the composer which is then returned to the client.

Consider the following sequence of analyses (i) Points-to Analysis (server) (ii) Constant Propagation Analysis (client). The points-to analysis has the necessary information in the form of points-to graph [Fig 1] for the code in Listing 1. With this information it is possible to infer that the expressions `*p, **q` correspond to memory object for the scalar variable `val`.

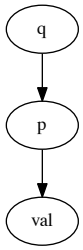

Figure 1: Points-to Graph for Motivating Example

It is reasonable to expect that a points-to analysis will support dereferencing operations in its implementation of `Expr2MemLoc`. Given a query for `**q`, the points-to analysis knows that its a double dereference and it traverses the points-to graph from q twice to return the memory object corresponding to *val*. The analysis has all the necessary information to discover the memory object for arbitrary complex dereferencing sub-expressions and client has no use for the getDereference() method. In the current framework, the syntax analysis should be modified to handle pointer/array dereferencing.

5 Interpret Dereferencing Sub-Expressions in Composer

- Client makes the query `composer->Expr2MemLoc(sub-exp)`

- Instead of asking a server analysis, the composer responds with the appropriate memory object.
- For example to interpret *p, composer calls OperandExpr2MemLoc(*p, p). This function returns the memory object corresponding to 'p' involved in the expression $*p$. Composer then invokes some API functions to access the internal data structure of the analysis inorder to answer the query.

The downside of this approach is that the a common API to access the internal data structure of analysis needs to be established. Note that the analyses can be orthogonal constraining values or memory and the API has to account for different types of analyses.