poco: An ML testbed for deductive synthesis tool design

-

Abstract

Experimentation with techniques for deductive program synthesis is limited by the complexity of the necessary analysis, transformation, and search infrastructure. Prior to any experimentation with new ideas to perform code transformation and search, a substantial amount of foundational infrastructure must be created. The poco language is a small extension to the MinCaml [3] subset of Ocaml that adds constructs for specifying code transformations, constraints guiding their application, and strategies for composing them. Its purpose was to serve as this foundational infrastructure for exploring ideas in synthesis methods before applying them to more complex languages. The poco implementation is designed to support backtracking search for enumeration of programs derived from a specification, with static analysis and pattern matching algorithms designed to reduce unnecessary computation wherever possible. This research presentation will detail the notable design decisions in poco, their impact on performance as measured through experimentation, and the use of poco to explore synthesis of algorithms guided by theorems from linear algebra.

1 poco design

poco is a source-to-source transformation tool that is based on term rewriting where rewrite rules are specified within the language via metaprogramming syntax and constrained by proof obligations to be discharged at the program point where a rule is to be applied. poco is both the specification language for defining the high level design of a program as well as the target language that the specification is refined to. A poco program stripped of the extensions related to metaprogramming is a valid Ocaml program and can be compiled with an existing Ocaml compiler. poco shares a number of design choices with theorem provers with regard to the term structure, rewriting process, and internal data structure choices. poco is also similar to past code transformation efforts such as RML [1], with additional attention to the rule specification language, access to static analysis to constrain rule application, and internal design decisions to support not only program optimization but program enumeration as arises in search during program synthesis.

The state maintained by poco during refinement is a purely functional context structure that holds the AST, CFG, and all other auxiliary structures related to the program and derived analyses. When a transformation occurs, a new context is generated that reflects the changes to the state and sharing as much as possible from the parent context. Sharing aids with memory efficiency, and the purely functional nature of the structure makes backtracking and undoing transformations simple.

1.1 Term representation

poco parses programs into a term form that alternates between AST node algebraic data type constructors and pointers into a map of node GUIDs. This design has a number of benefits and tradeoffs. First, we have direct access to any element in the AST without requiring a traversal. This aids in efficiency internally, as well as providing support for binding metavariables to program points. Second, name occurrences that refer to the same binder for a name are represented as references to the same name term. This aids in namespace management and identifying name occurrences that refer to the same binder. Third, the use of identifiers to refer to terms allows derived structures (like a control flow graph) to use the same identifiers. Thus one

can quickly map back and forth between the AST and CFG during analysis and transformation. The most significant tradeoff is that, while `poco` itself is implemented in Ocaml, we are unable to use Ocaml's built in pattern matching facilities to their full extent within the `poco` implementation. We have found that the benefits of the hybrid algebraic data type + GUID scheme outweighs the slight increase in complexity of working with terms internally.

Given this AST and derived CFG, we compute a number of other auxiliary data structures. All terms sharing the same constructor (e.g., BinOp nodes) are stored in a map from constructor tag to the set of term IDs that have that constructor. This aids in performing pattern matching during rewriting by avoiding any traversals of terms that do not occur in the pattern. Additional maps are maintained to map a term to its parent, record the term that defines the value for a name binding, and bookkeeping maps to record pattern match failures such that we can avoid re-attempting a match in the future if the term had not changed.

1.2 Meta-syntax

`poco` metavariables can either bind terms (`$x`) or program points where a term occurs (`@x`). Metavariables referring to program points are important for discharging proof obligations where the context in which a term occurs is important. `poco` has two forms of rules: rewrite rules and fact annotation rules.

A rewrite rule in `poco` is defined as a triple: a pattern, fragment, and proof obligation. A pattern is any syntactically valid expression with metavariables. The rule can be attempted at any program point where the pattern matches and metavariables are bound to program elements. The fragment portion of the rule is similar, except that any bound metavariables are substituted into the fragment term prior to splicing into the AST. The splice into the AST occurs at the location where the pattern matched if the proof obligation can be reduced to `True`. The proof obligation is also term containing metavariables. Metavariables bound in the pattern of the rule are substituted into the proof obligation before reduction. Additional metavariables may be bound during the proof process (e.g., resolving a term that satisfies a specific constraint) and are available for substitution in the fragment. Upon successful reduction of the proof obligation, a new context is generated with the fragment spliced into the AST and all supporting data structures (e.g., the CFG) incrementally updated.

A fact annotation rule is used to decorate terms without directly changing the AST. This allows rule designers to introduce semantics for use by rules via uninterpreted symbols without introducing them into the program code itself. Fact annotation rules are defined as a 4-tuple. As with rewrite rules, the tuple contains a pattern, fragment, and proof obligation. The additional element of the tuple is the program point to associate the fragment with. The pattern, fragment, and obligation are treated the same as rules with respect to metavariable binding, substitution, and term reduction. Instead of splicing the fragment at the location of the pattern match, the fragment term is stored as an annotation at a program point. If the program point is omitted from the rule definition, the program point where the match occurred (`@_`) is used. If a point other than `@_` is to be annotated, the program point provided must be a metavariable (e.g., `@x`) that is bound in the pattern of the rule. The example in Section 2.2 illustrates this.

1.3 Prover

The `poco` prover is very simple. The term corresponding to the proof obligation is transformed using the same matching and rewriting method as applied to the program itself. All rules available to rewrite the program are available for rewriting proof obligations. In addition to rules, a set of primitive functions are available that the prover interprets that gives meta-programs access to internal analyses. These include testing liveness, purity, structural equivalence, name equivalence, and others. The prover can also query program points for the presence of specific fact annotations. The prover operates by exhaustively applying rewrite rules under some strategy until the term reduces to a ground literal `True` or `False`, or the tool runs out of potential rules and sites to apply them. Resolving predicates within the obligation is performed via a backwards traversal of the CFG from the program point where the pattern match occurred.

1.4 Matching

Rules and facts in `poco` are compiled to a trie form to allow a single traversal of the AST to find all possible matches. This is based on the discrimination tree concept introduced by McCune [2]. To avoid excessive work the traversal is limited to those points within the AST that match tags that occur within rules. Furthermore, the trie can be filtered to match only a subset of rules, and the traversal filtered to attempt only a subset of points. These filters are controlled via the `poco` strategy primitives. Upon applying the rule trie to an AST, a set of matches are returned. The order by which these are traversed is also controlled by the strategy primitives. An optional decision tree can be provided as an input to `poco` to associate a priority with each match to control their order of processing. `poco` can optionally emit success/failure statistics for rule application that can be used to perform offline tuning of the decision tree weights. This prioritization scheme is based on a small neighborhood around a term constructor within the AST to take into account a very limited syntactic context.

1.5 Strategy primitives

The `poco` strategy primitives are simple and resemble simple tactic languages present in theorem provers. Rules can be collected into groups. Given a group of rules and a program, we can request that one rule from the group be applied or all rules be applied. This application can be performed once, at most n times, or until the point at which no rules can apply. Rule application can be sequenced (apply group A then group B) or specified as alternatives (try group A, and if that fails, try group B). We are also planning to allow the program points where successful rule application occurred to be passed to subsequent application attempts to constrain where downstream rules are attempted. As in other rewriting systems, the strategy primitives are designed to control an otherwise expensive exhaustive search process.

2 Results

`poco` has been developed against a set of examples. These include generic program transformations that arise in program optimization, those that are relevant to GPU programming (creation of device kernels from composed functions), as well as domain-specific examples that exploit linear algebra properties. Two examples are described below:

2.1 Map-map fusion

The map-map fusion transformation is easy to state in `poco`:

```
rule map_map_fusion { map $f (map $g $y) } => { map (fun x -> $f ($g x)) $y }
  requires (is_pure $g && is_pure $f)
```

The pattern binds `f` and `g` to the functions being mapped and `y` to the collection that `g` is mapped over. If the prover can determine that both are side-effect free, then the pattern can be replaced by a single map in which the composition of `f` and `g` is wrapped within a lambda. The proof obligation relies upon an internal analysis method, `is_pure` that analyzes the given term. In the absence of references, the possibility for side effects in `poco` is restricted to array updates.

2.2 Matrix pseudo-inverse

To demonstrate the utility of the fact annotations, consider a domain-specific transformation that exploits knowledge of linear algebra to substitute a Moore-Penrose pseudoinverse when a non-square matrix is attempted to be inverted. Specifically, we exploit knowledge of the singular value decomposition to define the pseudoinverse as a sequence of simple matrix products. Consider the following program code:

```
let x = matrix_create 4 3 in
inv x
```

with the following collection of rules.

```
fact matrix_size { matrix_create $m $n } => { size ($m,$n) }
fact matrix_real { matrix_create $m $n } => { real }
rule check_square { is_square(@x) }  => { $m=$n } requires (know(@x, size($m,$n)))
rule check_notsquare { not_square(@x) } => { $m<>$n } requires (know(@x, size($m,$n)))
rule pseudoinverse { inv @x } => { let (u,d,v) = svd $x in (inv v)*(inv d)*(inv u) }
    requires (not_square $x)
fact svd1 { let (@u,@d,@v) = svd $a in $b }  => (@u) { size($m, $m) && unitary && real }
    requires ( know(@a, size($m, $n)) )
fact svd2 { let (@u,@d,@v) = svd $a in $b } => (@v) { size($n, $n) && unitary && real }
    requires ( know(@a, size($m, $n)) )
fact svd3 { let (@u,@d,@v) = svd $a in $b }  => (@d) { size($m, $n) && diagonal && real }
    requires ( know(@a, size($m, $n)) )
rule invert_diag { inv @x } => { trans (recip $x) } requires (know(@x, diagonal))
rule invert_unitary { inv @x } => { conj_trans $x } requires (know(@x, unitary))
rule conj_real { conj_trans @x } => { trans $x } requires (know(@x, real))
```

This rule set encodes domain knowledge about linear algebra: the properties of the matrices comprising the singular value decomposition, the implications these properties have regarding shape and invertibility, and so on. The fact annotations associate terms with program points related to properties of the SVD and the matrices it produces. The `know` primitive in rule constraints is used to test whether such annotations exist at specific program points. This allows domain knowledge to be encoded in and exploited by `poco` in a very straightforward way. The result of applying this rule set yields:

```
let x = (matrix_create 4 3) in
let (u, d, v) = (svd x) in
(((trans v) * (trans (recip d))) * (trans u))
```

Subsequent rules can be applied to replace the `*` operator with a matrix multiplication call, and instantiation of the matrix data structures and corresponding implementations of matrix operators (e.g., transpose and element-wise reciprocal).

References

[1] Eelco Visser and Zine-el-Abidine Benaissa and Andrew Tolmach. Building program optimizers with rewriting strategies. In *Proceedings of the 1998 International Conference on Functional Programming*, 1998.

[2] William McCune. Experiments with discrimination-tree indexing and path indexing for term retrieval. *Journal of Automated Reasoning*, 9:147–167, 1992.

[3] Eijiro Sumii. MinCaml: A Simple and Efficient Compiler for a Minimal Functional Language. In *Proceedings of the 2005 workshop on Functional and declarative programming in education*, 2005.