Abstract Code Search

Isabel Garcia-Contreras Jose F. Morales Manuel V. Hermenegildo

New York, ICLP 2016, October 18, 2016

IMDEA Software Institute, and

Technical University of Madrid

Introduction

Motivation: Find code to reuse.

Problem: The large amount of code available makes it impossible for a programmer to find useful fragments manually.

Current code finders are mainly based on:

- Documentation keyword indexing (Maarek et al. 1991).
- Signature matching (Rollins and Wing 1991 – λ Prolog).
- Combinations of both (Mitchell 2008 – Hoogle/Haskell).

Introduction

Motivation: Find code to reuse.

Problem: The large amount of code available makes it impossible for a programmer to find useful fragments manually.

Current code finders are mainly based on:

- Documentation keyword indexing (Maarek et al. 1991).
- Signature matching (Rollins and Wing 1991 – λ Prolog).
- Combinations of both (Mitchell 2008 – Hoogle/Haskell).

Our approach:

- Find code based on its *semantic characteristics*.
  (that can be inferred automatically).

(Ciao) Assertions

Assertions are a fundamental component of our approach.

They are **linguistic constructions** for expressing abstractions of the meaning and behavior of programs.

`pred` assertions (subset)

Allow specifying certain conditions on the state (current substitution or constraint store) that must hold at certain points of program execution.

`:- pred` *Head* `:` *Pre* `=>` *Post*.

- *Head*: normalized atom that denotes the predicate that the assertion applies to.
- *Pre* and *Post*: conjunctions of "`prop`" atoms.

(Ciao) Assertions - example

```
:- pred check_length(L,N) : (list(L), int(N)).
check_length(L,N) :- length(L,N).

:- pred gen_list(L,N) : (var(L),  var(N)) => (list(L), int(N)).
gen_list(L,N) :- length(L,N).

% length implementation ...
```

Herein we will use *assertions* for:

- **Programming**: To specify calling modes for predicates.
- **[New!] Searching**: To express properties of the code to be found.

Our approach

Main idea of our method:

1. A **set of modules** is specified within which code is to be found.

Our approach

Main idea of our method:

1. A **set of modules** is specified within which code is to be found.

```
:- module(lengths, [check_length/2, gen_list/2], [assertions]).

:- pred check_length(L,N) : (list(L), int(N)).
check_length(L,N) :- length(L,N).

:- pred gen_list(L,N) : (var(L), var(N)) => (list(L), int(N)).
gen_list(L,N) :- length(L,N).

% length implementation ...
```

Our approach

Main idea of our method:

1. A **set of modules** is specified within which code is to be found.
2. A **static pre-analysis** is made to:
    - Infer semantic properties in one or more abstract domains (e.g.: shapes/types, variable sharing, inst. modes, polyhedra, ...).

Static analysis inference

Shapes and Modes/Sharing analysis:

```
% :- pred check_length(L,N)     : (list(L), int(N)).
:- true pred check_length(L,N)  : (list(L), int(N))
                               => (list(L), int(N)).
:- true pred check_length(L,N)  : (mshare([[L],[L,N],[N]])
                               => (mshare([[L]]), ground([N])).
check_length(L,N) :- length(L,N).

%:- pred gen_list(L,N)     : ((var(L),  var(N) ).
:- true pred gen_list(L,N) : ((term(L), term(N))
                          => ((list(L), int(N)).
:- true pred gen_list(L,N) : (mshare([[L],[L,N],[N]])), var(L), var(N))
                          => (mshare([[L]]), ground([N])).
gen_list(L,N) :- length(L,N).

% length implementation ...
```

Our approach

Main idea of our method:

1. A **set of modules** is specified within which code is to be found.
2. A **static pre-analysis** is made to:
   - Infer semantic properties in one or more abstract domains (e.g.: shapes/types, variable sharing, inst. modes, polyhedra, ...).
3. User specifies **semantic properties** in a query, using a new kind of assertions that we call **query assertions**:
   - Example: `:- pred P(X,Y) : list(X) => sorted(Y).`

   Based on (Stulova et al. 2014).
4. Look within the modules for predicates that meet those properties (by comparing to inferred information).

Inferring semantic properties

Abstract interpretation:

- To simulate the execution using an *abstract domain* D_α.
- It guarantees:
  - Analysis termination, provided that D_α meets some conditions.
  - Results are **safe approximations** of the concrete semantics.

PLAI algorithm (in Ciao)

Input	**P**: Program
	D_α^i: Abstract Domain(s)
	$\mathcal{Q}_\alpha = \boldsymbol{L}\text{:}\lambda$: Initial call pattern
Output	$Analysis(P, L{:}\lambda, D_\alpha) = \{\langle L_1, \lambda_1^c, \lambda_1^s\rangle, \ldots, \langle L_n, \lambda_n^c, \lambda_n^s\rangle\}$, where:
	▪ L_i is an atom
	▪ λ_i^c are abstract call state in D_α
	▪ λ_i^s are abstract success state in D_α.

Querying for predicates

Predicate query

```
?- findp({ As }, M:Pred/A, Residue, Status).
```

Input	***As***: Set of query assertions.
Output	**M:Pred/A**: Module, predicate descriptor and arity. **Residue**: Information of condition matching.
Input or Output	**Status**: Result of the proof for the whole set of conditions. ▪ ***checked*** if all conditions are proved to be checked. ▪ ***false*** if any condition is false. ▪ ***check*** if neither ***checked*** nor ***false*** can be proved.

```
?- findp({ :- pred P(L, S) => (list(L), num(S)). }, M:P/A, Res, St).

 P/A = check_length/2   St = checked
```

Normalizing the query

Assertion Conditions

Given a predicate represented by a normalized atom *Head*, and a corresponding set of assertions $\mathcal{A} = \{A_1 \ldots A_n\}$, with $A_i =$ "`:- pred` *Head* `:` Pre_i `=>` $Post_i$`.`". The set of *assertion conditions* for *Head* determined by $\mathcal{A}$ is $\{C_0, C_1, \ldots, C_n\}$, with:

$$C_i = \begin{cases} \texttt{calls}(Head, \bigvee_{j=1}^{n} Pre_j) & i = 0 \\ \texttt{success}(Head, Pre_i, Post_i) & i = 1..n \end{cases}$$

```
:- pred my_length(L,N) : ((list(L), var(N)) => ((list(L), int(N)).
:- pred my_length(L,N) : ((list(L), int(N)) => ((list(L), int(N)).
my_length(L,N) :- length(L,N).
```

Assertion conditions from `my_length/2`:

$$C_i = \left\{ \begin{array}{llll} \texttt{calls(} & length(L,N), & (list(L) \wedge var(N)) \;\vee\; (list(L) \wedge int(N))), & \\ \texttt{success(} & length(L,N), & (list(L) \wedge var(N)), & (list(L), int(N))), \\ \texttt{success(} & length(L,N), & (list(L) \wedge int(N)), & (list(L), int(N))) \end{array} \right\}$$

Matching *success* conditions

Match $C = \texttt{success}(X(V_1, \dots, V_n), Pre, Post)$ against $analysis(P, Q_\alpha)$

Checked matches

If *Pre* holds at the time of calling the matching predicate and the execution succeeds then the *Post* conditions hold.

C is abstractly 'checked' for predicate $p \in P$ w.r.t. Q_α in D_α iff $\exists L = p(V'_1, \dots, V'_n)$ s.t. $\forall \langle L, \lambda^c, \lambda^s \rangle \in analysis(P, Q_\alpha)$ $s.t.$ $\exists \sigma \in ren,\ L = p(V'_1, \dots, V'_n) = X(V_1, \dots, V_n)\sigma,\ \lambda^c \sqsupseteq \lambda^+_{TS(Pre\ \sigma, P)} \rightarrow \lambda^s \sqsubseteq \lambda^-_{TS(Post\ \sigma, P)}$

False matches

If *Pre* holds at the time of calling the matching predicate and the execution succeeds then its conditions and *Post* are disjoint.

C is abstractly false for $p \in P$ w.r.t. Q_α in D_α iff $\exists L = p(V'_1, \dots, V'_n)$ s.t. $\forall \langle L, \lambda^c, \lambda^s \rangle \in analysis(P, Q_\alpha)$ $s.t.$ $\exists \sigma \in ren,\ L = p(V'_1, \dots, V'_n) = X(V_1, \dots, V_n)\sigma,\ \lambda^c \sqsubseteq \lambda^-_{TS(Pre\ \sigma, P)} \wedge (\lambda^s \sqcap \lambda^+_{TS(Post\ \sigma, P)} = \bot)$

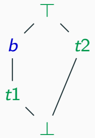
Matching *success* conditions - Example

```

perfect(b1).                                    mixed(b0).
perfect(b0).                                    mixed(b1).
                                                mixed(z).

reduced(b1).
                                                hard(X) :- functor(b1(_), X, _).

outb(z).

:- regtype b/1.
b(b0).
b(b1).
```

Matching *success* conditions - Example

```
:- true pred perfect(A) => b(A).          :- true pred mixed(X) => top(X).
perfect(b1).                              mixed(b0).
perfect(b0).                              mixed(b1).
                                          mixed(z).
:- true pred reduced(A) => t1(A).
reduced(b1).                              :- true pred hard(X) => top(X).
                                          hard(X) :- functor(b1(_), X, _).
:- true pred outb(A) => t2(A).
outb(z).

:- regtype b/1.       :- regtype t1/1     :- regtype t2/1.
b(b0).                t1(b1).             t2(z).
b(b1).
```

$$\mathrm{D} = \{b0, b1, z\}$$
$$D_\alpha = \{\top, b, t1, t2, \bot\}$$



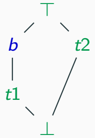
Matching *success* conditions - Example

```
:- true pred perfect(A) => b(A).          :- true pred mixed(X) => top(X).
perfect(b1).                              mixed(b0).
perfect(b0).                              mixed(b1).
                                          mixed(z).
:- true pred reduced(A) => t1(A).
reduced(b1).                              :- true pred hard(X) => top(X).
                                          hard(X) :- functor(b1(_), X, _).
:- true pred outb(A) => t2(A).
outb(z).

:- regtype b/1.      :- regtype t1/1      :- regtype t2/1.
b(b0).               t1(b1).              t2(z).
b(b1).
```

```
?- findp({ :- pred P(V) => b(V). }, M:P/A, Res, St).
  P/A = perfect/1    St = checked
  P/A = reduced/1    St = checked
  P/A = outb/1       St = false
  P/A = mixed/1      St = check
  P/A = hard/1       St = check
```

$$D = \{b0, b1, z\}$$

$$D_\alpha = \{\top, b, t1, t2, \bot\}$$



Matching *calls* conditions - Example

```
:- pred check_length(L,N) : ((list(L), int(N)).
check_length(L,N) :- length(L,N).

:- pred gen_list(L,N) : ((var(L), var(N)).
gen_list(L,N) :- length(L,N).

:- pred get_length(L,N) : var(N).
get_length(L,N) :- length(L,N).
```

```
?- findp({ :- pred P(L, Size) : var(L), var(Size)). }, M:P/A, Res, St).

 P/A = check_length/2    St = false
 P/A = gen_list/2        St = checked
 P/A = get_length/2      St = check
```

Combining abstract domains

```
:- true pred check_length(L,N) : (list(L), int(N)) => (list(L), int(N)).
:- true pred check_length(L,N) : (mshare([[L],[L,N],[N]])
                               => (mshare([[L]]), ground([N])).
check_length(L,N) :- length(L,N).

:- true pred gen_list(L,N) : ((term(L), term(N)) => ((list(L), int(N)).
:- true pred gen_list(L,N) : (mshare([[L],[L,N],[N]]), var(L), var(N))
                           => (mshare([[L]]), ground([N])).
gen_list(L,N) :- length(L,N).
```

```
?- findp({ :- pred P(L, Size) : list(L), num(Size)). }, M:P/A, Res, St).
```

PredName/A	regtypes proof	shfr proof	combined proof
check_length/2	checked	check	checked
gen_list/2	check	false	false

Demo

Conclusions

Finding code by its semantic characteristics:

- Ensures that the found code behaves correctly.
- Reasons with relations between properties (implication, abstraction).
- Is independent from the documentation.
- Implemented in Ciao, in combination with other types of search (which it complements).

Future work:

- Use more domains.
- Extend to other programming languages and combinations.
- Other uses: finding duplicated code.

Thanks!

Matching *calls* conditions

$$C = \texttt{calls}(X(V_1, \ldots, V_n), Pre)$$

Checked matches

The admissible calls of the matching predicate are within the set of *Pre* conditions.

C is abstractly 'checked' for a $p \in P$ w.r.t. Q_α in D_α iff $\forall \langle L, \lambda^c, \lambda^s \rangle \in \mathit{analysis}(P, D_\alpha, \mathcal{Q}_\alpha)$ *s.t.* $\exists \sigma \in \mathit{ren},\ L = p(V_1', \ldots, V_n') = X(V_1, \ldots, V_n)\sigma, \lambda^c \sqsubseteq \lambda^-_{TS(Pre\ \sigma, P)}$.

False matches

The admissible calls of the matching predicate and the set *Pre* conditions are disjoint.

C is abstractly 'false' for a $p \in P$ w.r.t. Q_α in D_α iff $\forall \langle L, \lambda^c, \lambda^s \rangle \in \mathit{analysis}(P, D_\alpha, \mathcal{Q}_\alpha)$ *s.t.* $\exists \sigma \in \mathit{ren},\ L = p(V_1', \ldots, V_n') = X(V_1, \ldots, V_n)\sigma, \lambda^c \sqcap \lambda^+_{TS(Pre\ \sigma, P)} = \bot$.

Performance

Search times in μs.

Ar\Cnds	1	1 (AVG)	2	2 (AVG)	3	3 (AVG)	4	4 (AVG)
1 (85 pr)	19,064	224	53,530	630	180,246	2,121	298,292	3,509
2 (74 pr)	110,092	1,488	207,871	2,809	221,061	2,987	477,440	6,452
3 (47 pr)	294,962	6,276	3,757,208	79,941	3,806,917	80,998	6,127,015	130,362
4 (12 pr)	5,116	426	12,939	1,078	22,508	1,876	30,300	2,525