Incremental Analysis of Logic Programs with Assertions and Open Predicates

Isabel Garcia-Contreras[1,2] Jose F. Morales[1] Manuel V. Hermenegildo[1,2]

[1]IMDEA Software Institute
[2]T. U. Madrid (UPM)

LOPSTR'19 October 8th, 2019

Goal of the paper

We propose an analysis algorithm that reacts incrementally to changes in the program, understanding the type of program edit.

- In particular, it distinguishes between assertion edits and clause edits.

Goal of the paper

We propose an analysis algorithm that reacts incrementally to changes in the program, understanding the type of program edit.

- In particular, it distinguishes between assertion edits and clause edits.

Our contributions are:

- An incremental fixpoint algorithm that reacts to changes in both the program and the assertions.
- An application of this approach to the scalable analysis of generic programming (based on open predicates).

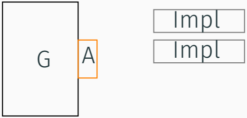
Goal of the paper

We propose an **analysis algorithm** that reacts **incrementally** to changes in the program, understanding the **type of program edit**.

- In particular, it distinguishes between **assertion edits** and **clause edits**.

Our **contributions** are:

- An **incremental fixpoint algorithm** that reacts to changes in both the program **and the assertions**.
- An application of this approach to the scalable analysis of generic programming (based on **open predicates**).



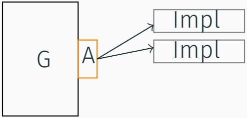
Goal of the paper

We propose an analysis algorithm that reacts incrementally to changes in the program, understanding the type of program edit.

- In particular, it distinguishes between assertion edits and clause edits.

Our contributions are:

- An incremental fixpoint algorithm that reacts to changes in both the program and the assertions.
- An application of this approach to the scalable analysis of generic programming (based on open predicates).



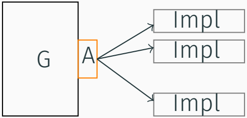
Goal of the paper

We propose an analysis algorithm that reacts incrementally to changes in the program, understanding the type of program edit.

- In particular, it distinguishes between assertion edits and clause edits.

Our contributions are:

- An incremental fixpoint algorithm that reacts to changes in both the program and the assertions.
- An application of this approach to the scalable analysis of generic programming (based on open predicates).



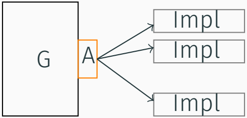
Goal of the paper

We propose an analysis algorithm that reacts incrementally to changes in the program, understanding the type of program edit.

- In particular, it distinguishes between assertion edits and clause edits.

Our contributions are:

- An incremental fixpoint algorithm that reacts to changes in both the program and the assertions.
- An application of this approach to the scalable analysis of generic programming (based on open predicates).



(And we also propose an encoding of generic programming in (Ciao) Prolog.)

Target language

Our analyzer supports several languages by translation to Horn Clauses.

For concreteness we focus on Prolog programs. The concrete semantics is **goal-dependent** and based on generalized AND trees:

- An AND tree represents the **execution of a program**.
- A **node** represents a **call** to a predicate and contains:
  - The program state for that call.
  - The program state at call exit, if the call **succeeds** or $\perp$.

The Ciao assertion language

Assertions express **abstractions of the behavior** of programs.

`pred` assertions (subset)

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

- *Head*: predicate that the assertion applies to.
- *Pre*: properties about how the predicate is used.
- *Post*: properties that hold if Pre holds and the predicate succeeds.

The Ciao assertion language

Assertions express **abstractions of the behavior** of programs.

`pred` assertions (subset)

```
:- pred Head [: Pre] [=> Post].
```

- *Head*: predicate that the assertion applies to.
- *Pre*: properties about how the predicate is used.
- *Post*: properties that hold if Pre holds and the predicate succeeds.

```
:- pred dgst(Word,N) : (str(Word), var(N)) => int(N).
:- pred dgst(Word,N) : (str(Word), int(N)).
```

Meaning of a set of assertions for a predicate

Assertion Conditions

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

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

Meaning of a set of assertions for a predicate

Assertion Conditions

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

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

```
:- pred dgst(Word,N) : (str(Word), var(N)) => int(N).
:- pred dgst(Word,N) : (str(Word), int(N)).
```

Assertion conditions from `dgst/2`:

$$C_i = \left\{ \begin{array}{llll} \text{calls}(& dgst(N,R), & ((\text{str}(Word), \text{var}(N)) \vee (\text{str}(Word), \text{int}(N)))), & \\ \text{success}(& dgst(N,R), & (\text{str}(Word), \text{var}(N)) & , \text{int}(N)), \end{array} \right\}$$

Assertions in action

```
:- pred dgst(Word,N) : (str(Word), var(N)) => int(N).
:- pred dgst(Word,N) : (str(Word), int(N)).
```

```
?- dgst("password", X).
```

Assertions in action

```
:- pred dgst(Word,N) : (str(Word), var(N)) => int(N).
:- pred dgst(Word,N) : (str(Word), int(N)).
```

```
?- dgst("password", X).

X = 42.

yes
```

Assertions in action

```
:- pred dgst(Word,N) : (str(Word), var(N)) => int(N).
:- pred dgst(Word,N) : (str(Word), int(N)).
```

```
?- dgst("password", X).

X = 42.

yes

?- dgst("password", 35).
```

Assertions in action

```
:- pred dgst(Word,N) : (str(Word), var(N)) => int(N).
:- pred dgst(Word,N) : (str(Word), int(N)).
```

```
?- dgst("password", X).

X = 42.

yes

?- dgst("password", 35).

no
```

Assertions in action

```
:- pred dgst(Word,N) : (str(Word), var(N)) => int(N).
:- pred dgst(Word,N) : (str(Word), int(N)).
```

```
?- dgst("password", X).

X = 42.

yes

?- dgst("password", 35).

no

?- dgst(P, 42).
```

Assertions in action

```
:- pred dgst(Word,N) : (str(Word), var(N)) => int(N).
:- pred dgst(Word,N) : (str(Word), int(N)).
```

```
?- dgst("password", X).

X = 42.

yes

?- dgst("password", 35).

no

?- dgst(P, 42).

ERROR
```

Assertions in action

```
:- pred dgst(Word,N) : (str(Word), var(N)) => int(N).
:- pred dgst(Word,N) : (str(Word), int(N)).
```

```
?- dgst("password", X).

X = 42.

yes

?- dgst("password", 35).

no

?- dgst(P, 42).

ERROR
```

The execution is stopped when the assertion conditions are not met.

Abstract Interpretation [Cousot POPL'77]

Simulates the execution of programs using abstract domains. It guarantees:

- Analysis **termination**, provided that the domain meets some conditions.
- Results are **safe approximations** of the concrete semantics.

Abstract Interpretation [Cousot POPL'77]

Simulates the execution of programs using abstract domains. It guarantees:

- Analysis **termination**, provided that the domain meets some conditions.
- Results are **safe approximations** of the concrete semantics.

In our setting: for all the predicate calls we obtain a mapping $\langle Goal, \lambda^c \rangle \mapsto \lambda^s$, where:

- *Goal* is an atom (identifier of the predicate).
- λ^c is a (possible) call pattern to *Goal*.
- λ^s is the answer pattern to *Goal* and λ^c if succeeds.

Abstract Interpretation [Cousot POPL'77]

Simulates the execution of programs using abstract domains. It guarantees:

- Analysis **termination**, provided that the domain meets some conditions.
- Results are **safe approximations** of the concrete semantics.

In our setting: for all the predicate calls we obtain a mapping $\langle Goal, \lambda^c \rangle \mapsto \lambda^s$, where:

- *Goal* is an atom (identifier of the predicate).
- λ^c is a (possible) call pattern to *Goal*.
- λ^s is the answer pattern to *Goal* and λ^c if succeeds.

Example

```
fact(0,1).
fact(N,R) :- N > 0,
             N1 is N - 1,
             fact(N1,R1),
             R is N * R1.
```

Abstract Interpretation [Cousot POPL'77]

Simulates the execution of programs using abstract domains. It guarantees:

- Analysis **termination**, provided that the domain meets some conditions.
- Results are **safe approximations** of the concrete semantics.

In our setting: for all the predicate calls we obtain a mapping $\langle Goal, \lambda^c \rangle \mapsto \lambda^s$, where:

- *Goal* is an atom (identifier of the predicate).
- λ^c is a (possible) call pattern to *Goal*.
- λ^s is the answer pattern to *Goal* and λ^c if succeeds.

Example

```
fact(0,1).
fact(N,R) :- N > 0,
             N1 is N - 1,
             fact(N1,R1),
             R is N * R1.
```

Analysis result (example):

$\{\langle \mathrm{fact}(\mathrm{N},\mathrm{R}), \top \rangle \mapsto R/+$

For any call to `fact` *that succeeds R is positive.*

$\langle \mathrm{fact}(\mathrm{N},\mathrm{R}), N/- \rangle \mapsto \bot$

If `fact` *is called with N a negative number, it fails.* $\}$

Abstract Interpretation [Cousot POPL'77]

Simulates the execution of programs using abstract domains. It guarantees:

- Analysis **termination**, provided that the domain meets some conditions.
- Results are **safe approximations** of the concrete semantics.

In our setting: for all the predicate calls we obtain a mapping $\langle Goal, \lambda^c \rangle \mapsto \lambda^s$, where:

- *Goal* is an atom (identifier of the predicate).
- λ^c is a (possible) call pattern to *Goal*.
- λ^s is the answer pattern to *Goal* and λ^c if succeeds.

Example

```
fact(0,1).
fact(N,R) :- N > 0,
             N1 is N - 1,
             fact(N1,R1),
             R is N * R1.
```

Analysis result (example):

$\{\langle \mathrm{fact(N,R)}, \top \rangle \mapsto R/+$

For any call to `fact` *that succeeds R is positive.*

$\langle \mathrm{fact(N,R)}, N/- \rangle \mapsto \bot$

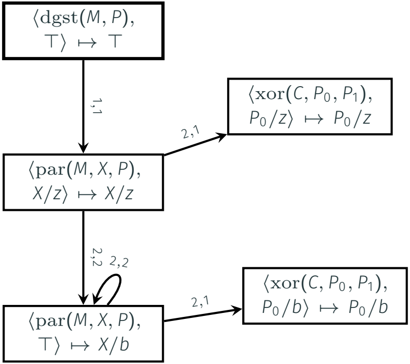
If `fact` *is called with N a negative number, it fails.* $\}$

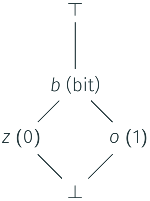
We store dependencies between calls:

$\langle P, \lambda \rangle_{i,j} \xrightarrow[\lambda^r]{\lambda^p} \langle Q, \lambda' \rangle$, *Calling P with λ causes Q to be called with λ'.*

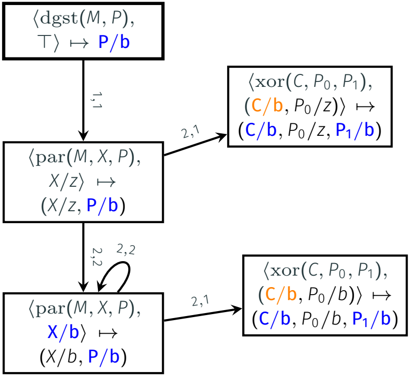
Example – Analysis graph

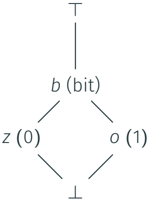
```
dgst(Msg, P) :-
    par(Msg, 0, P).

par([], P, P).
par([C|Cs], P0, P) :-
    xor(C, P0, P1),
    par(Cs, P1, P).

xor(1,1,0).
xor(0,1,1).
xor(B,0,B).
```





```
dgst(Msg, P) :-
    par(Msg, 0, P).

par([], P, P).
par([C|Cs], P0, P) :-
    xor(C, P0, P1),
    par(Cs, P1, P).

:- pred xor(A,B,C) :
     (bit(A), bit(B)).
xor(1,1,0).
xor(0,1,1).
xor(B,0,B).
```





How do assertions affect the analysis result?

Assertions state properties that are guaranteed to hold.

- Call conditions – $\mathrm{calls}(P, \mathit{Cond})$ – are applied when the abstract call is performed. I.e., after parameter passing and renaming.
- Success conditions – $\mathrm{success}(P, \mathit{Call}, \mathit{Succ})$ – are applied when the abstract success is performed. I.e., for each of the clauses, after the last literal is processed.

Baseline: the PLAI incremental analyzer [NACLP'89, TOPLAS'00]

Input	$\mathscr{Q}_\alpha$: initial abstract queries P': target program (changed) ΔP: clauses that changed from P to P' $\mathscr{A}$: (partial) analysis results of P
Output	$\mathscr{A}'$: analysis graph abstracting all the execution AND trees of (concrete) queries for which $\mathscr{Q}_\alpha$ holds.

Proposed algorithm

Input	$\mathscr{Q}_\alpha$: initial abstract queries **P'**: target program (changed) Δ**P**: clauses that changed from **P** to **P'** $\mathscr{A}$: (partial) analysis results of **P**
Output	$\mathscr{A}'$: analysis graph abstracting all the execution AND trees of (concrete) queries for which $\mathscr{Q}_\alpha$ holds.

Proposed algorithm

Input	$\mathscr{Q}_\alpha$: initial abstract queries **P'**: target program (changed) **+ assertions** Δ**P**: **clauses** that changed from **P** to **P'** $\mathscr{A}$: (partial) analysis results of **P**
Output	$\mathscr{A}'$: analysis graph abstracting all the execution AND trees of (concrete) queries for which $\mathscr{Q}_\alpha$ holds.

Proposed algorithm

Input	$\mathscr{Q}_\alpha$: initial abstract queries
	P': target program (changed) + assertions
	ΔP: clauses that changed from P to P'
	$\mathscr{A}$: (partial) analysis results of P
	Δ_{As} : assertions that changed from P to P'
Output	$\mathscr{A}'$: analysis graph abstracting all the execution AND trees of (concrete) queries for which $\mathscr{Q}_\alpha$ holds.

Proposed algorithm

Input	$\mathcal{Q}_\alpha$: initial abstract queries
	P': target program (changed) + assertions
	Δ**P**: clauses that changed from **P** to **P'**
	$\mathcal{A}$: (partial) analysis results of **P**
	Δ_{As} : assertions that changed from **P** to **P'**
Output	$\mathcal{A}'$: analysis graph abstracting all the execution AND trees of (concrete) queries for which $\mathcal{Q}_\alpha$ holds.

Our goal:

- extend the algorithm to react incrementally **changes in the assertions**,
- by preprocessing the previous **analysis results** $\mathcal{A}$ before calling the incremental fixpoint analyzer

(while preserving, of course, *soundness* and *precision*).

Proposed algorithm

Input	$\mathscr{Q}_\alpha$: initial abstract queries
	P': target program (changed) **+ assertions**
	Δ**P**: **clauses** that changed from **P** to **P'**
	$\mathscr{A}$: (partial) analysis results of **P**
	Δ_{As} : **assertions** that changed from **P** to **P'**
Output	$\mathscr{A}'$: analysis graph abstracting all the execution AND trees of (concrete) queries for which $\mathscr{Q}_\alpha$ holds.

Our goal:

- extend the algorithm to react incrementally **changes in the assertions**,
- by preprocessing the previous **analysis results** $\mathscr{A}$ before calling the incremental fixpoint analyzer

(while preserving, of course, *soundness* and *precision*).

The proposed analysis is **interprocedural**, **multivariant**, and **context sensitive**.

Algorithm – high-level idea

IncAnalyze-w/AssrtChanges($Program, \Delta_{Cls}, \Delta_{As}, \mathcal{Q}, \mathcal{A}$)

$R := \emptyset$
for each predicate $p \in Program$ **do**
  if assertions changed **then**
    R.add(`update_calls_pred`(p))
    R.add(`update_success_pred`(p))
$\mathcal{A}' :=$ IncAnalyze($Program, \Delta_{Cls}, \mathcal{Q} \cup R, \mathcal{A}$)
Remove unreachable calls
return $\mathcal{A}'$

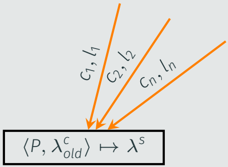
Algorithm – finding changes

`update_calls_pred`(P)

$Q := \emptyset$
for each $N_{c,l} \longrightarrow \langle P, \lambda^c_{old} \rangle \in \mathscr{A}$ **do**

  λ^c get original call substitution
  λ^c_{new} := `apply_call`(P, λ^c)
  $\lambda^{s\prime}$ obtain success substitution
  Q.add(`treat_change`($N_{c,l} \longrightarrow \langle P, \lambda^c{}_{new} \rangle, \lambda^{s\prime}$))

return Q

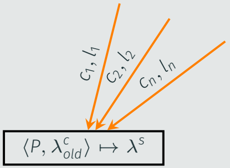


Algorithm – finding changes

`update_calls_pred`(P)

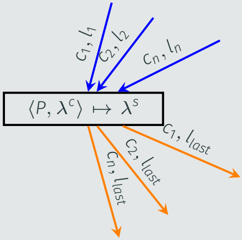
$Q := \emptyset$
for each $N_{c,l} \longrightarrow \langle P, \lambda^c_{old} \rangle \in \mathscr{A}$ **do**
  λ^c get original call substitution
  $\lambda^c_{new} :=$ `apply_call`(P, λ^c)
  $\lambda^{s\prime}$ obtain success substitution
  Q.add(`treat_change`$(N_{c,l} \longrightarrow \langle P, \lambda^c{}_{new} \rangle, \lambda^{s\prime})$)

return Q



`update_successes_pred`(P)

$Q := \emptyset$
for each $\langle P, \lambda^c \rangle \mapsto \lambda^s \in \mathscr{A}$ **do**
  λ get original success (via **`apply_success`**)
  for each $E \in$ incoming edges $\langle P, \lambda^c \rangle \in \mathscr{A}$ **do**
    Q.add(`treat_change` (E, λ))
return Q



Algorithm – amending the analysis result

`treat_change`$(\langle P, \lambda \rangle_{c,l} \xrightarrow[\lambda^r]{\lambda^p} \langle Q, \lambda^c \rangle, \lambda^s)$

$\lambda^{r\prime}$:= Obtain new info at literal return and update edge
if $\lambda^r \sqsubset \lambda^{r\prime}$ **then**
  return $\{\langle P, \lambda \rangle\}$
else if $\lambda^r \not\sqsubseteq \lambda^{r\prime}$ **then**
  Remove potentially imprecise nodes
  return initial queries of deleted nodes
else return $\emptyset$

A use case: Generic programming

We support describing collections of predicate specifications, called traits in Ciao (similar to C++ virtual classes, Rust boxed traits, Go interfaces, etc).

A use case: Generic programming

We support describing collections of predicate specifications, called traits in Ciao (similar to C++ virtual classes, Rust boxed traits, Go interfaces, etc).

```
:- trait hasher {
    :- pred dgst(Str, Digest) : str(Str) => int(Digest).
}.
```

A call to a generic predicate: `(X as T).p(A1,...,An)`, represents the predicate p for X implementing T.

A use case: Generic programming

We support describing collections of predicate specifications, called **traits** in Ciao (similar to C++ virtual classes, Rust boxed traits, Go interfaces, etc).

```
:- trait hasher {
    :- pred dgst(Str, Digest) : str(Str) => int(Digest).
}.
```

A call to a generic predicate: `(X as T).p(A1,...,An)`, represents the predicate p for X implementing T.

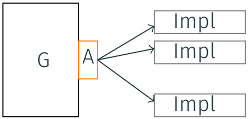
Example

```
check_passwd(User) :-
    get_line(Plain),
    passwd(User,Hasher,Digest,Salt),
    append(Plain,Salt,Salted),
    (Hasher as hasher).dgst(Salted,Digest).
```

Open predicates for generic programming

Open predicates (also referred to as `multifile`): their clauses can be scattered across different modules.

```
:- multifile 'T.p'/(n+1).
:- pred 'T.p'(X, A1, ..., An) : ... => ... .        % A
```



Call to p/n for X implementing T

```
... :- ..., 'T.p'(X, A1, ..., An), % (X as T).p(A1, ..., An)
```

Implementation closed predicate (head renamed)

```
'<f/k as T>.p'(f(...), A1, ..., An) :- ... % (f(...) as T).p(A1, ..., An)        % Impl
```

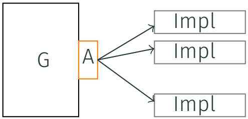
Bridge from interface open predicate to implementation

```
'T.p'(X, A1, ..., An) :- X=f(...), '<f/k as T>.p'(X, A1, ..., An).        % ⟶
```

Open predicates for generic programming

Open predicates (also referred to as `multifile`): their clauses can be scattered across different modules.

```
:- multifile 'T.p'/(n+1).
:- pred 'T.p'(X, A1, ..., An) : ... => ... .          % A
```



Call to p/n for X implementing T

```
... :- ..., 'T.p'(X, A1, ..., An), % (X as T).p(A1, ..., An)
```

Implementation closed predicate (head renamed)

```
'<f/k as T>.p'(f(...), A1, ..., An) :- ... % (f(...) as T).p(A1, ..., An)      % Impl
```

Bridge from interface open predicate to implementation

```
'T.p'(X, A1, ..., An) :- X=f(...), '<f/k as T>.p'(X, A1, ..., An).      % ⟶
```

```
:- impl(hasher, xor8/0).
(xor8 as hasher).dgst(Str, Digest) :- xor8_dgst(Xs, 0, Digest).

xor8_dgst([], D, D).
xor8_dgst([X|Xs], D0, D) :- D1 is D0 # X, xor8_dgst(Xs, D1, D).
```

Open predicates for generic programming

Open predicates (also referred to as `multifile`): their clauses can be scattered across different modules.

```
:- multifile 'hasher.dgst'/3.
:- pred 'hasher.dgst'(X,Str,Digest) : str(Str) => int(Digest).
```

Call to dgst/2 for xor8 implementing hasher

```
... :- ..., 'hasher.dgst'(X, A1, A2), ... % (xor8 as hasher).dgst(A1, A2)
```

Implementation closed predicate (head renamed)

```
'<xor8/0 as hasher>.p'(xor8, A1, A2) :- ... % (xor8 as hasher).dgst(A1, A2)
```

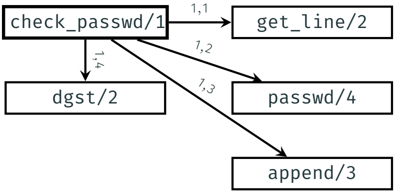
Bridge from interface open predicate to implementation

```
'hasher.dgst'(X, A1, A2) :- X=xor8, '<xor8/0 as hasher>.dgst'(xor8, A1, A2).
```

```
:- impl(hasher, xor8/0).
(xor8 as hasher).dgst(Str, Digest) :- xor8_dgst(Xs, 0, Digest).

xor8_dgst([], D, D).
xor8_dgst([X|Xs], D0, D) :- D1 is D0 # X, xor8_dgst(Xs, D1, D).
```

Example: Adding an implementation

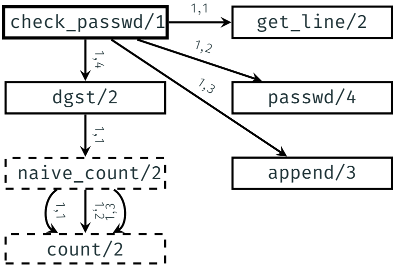
```
:- trait hasher {
    :- pred dgst(Str, Digest)
        : lowercase(Str) => int(Digest).
}.

check_passwd(User) :-
    get_line(Plain),
    passwd(User,Hasher,Digest,Salt),
    append(Plain,Salt,Salted),
    (Hasher as hasher).dgst(Salted,Digest).

passwd(don,xor8,0x6d,"eNfwuBhtN9CUHxg==").
```



Example: Adding an implementation

```
:- trait hasher {
    :- pred dgst(Str, Digest)
        : lowercase(Str) => int(Digest).
}.

check_passwd(User) :-
    get_line(Plain),
    passwd(User,Hasher,Digest,Salt),
    append(Plain,Salt,Salted),
    (Hasher as hasher).dgst(Salted,Digest).

passwd(don,xor8,0x6d,"eNfwuBhtN9CUHxg==").
```

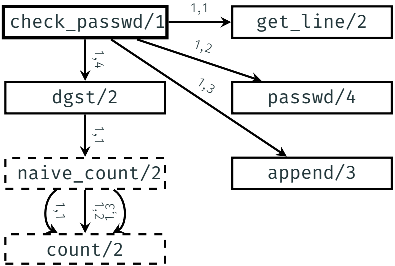
```
:- impl(hasher, naive/0).
(naive as hasher).dgst(Str, Digest) :-
    naive_count(Xs, 0, Digest).

naive_count(L, D0, D) :-
    count(L,'a',Na), D1 is D0 + Na*1,
    count(L,'b',Nb), D2 is D1 + Nb*2,
    count(L,'c',Nc), D3 is D2 + Nc*3,
    %% implementation continues
```



Example: Changing an assertion

```
:- trait hasher {
    :- pred dgst(Str, Digest)
        : lowercase(Str) => int(Digest).
}.

check_passwd(User) :-
    get_line(Plain),
    passwd(User,Hasher,Digest,Salt),
    append(Plain,Salt,Salted),
    (Hasher as hasher).dgst(Salted,Digest).

passwd(don,xor8,0x6d,"eNfwuBhtN9CUHxg==").
```

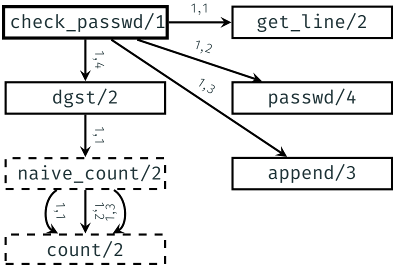
```
:- impl(hasher, naive/0).
(naive as hasher).dgst(Str, Digest) :-
    naive_count(Xs, 0, Digest).

naive_count(L, D0, D) :-
    count(L,'a',Na), D1 is D0 + Na*1,
    count(L,'b',Nb), D2 is D1 + Nb*2,
    count(L,'c',Nc), D3 is D2 + Nc*3,
    %% implementation continues
```



Example: Changing an assertion

```
:- trait hasher {
    :- pred dgst(Str, Digest)
        : str(Str) => int(Digest).
}.

check_passwd(User) :-
    get_line(Plain),
    passwd(User,Hasher,Digest,Salt),
    append(Plain,Salt,Salted),
    (Hasher as hasher).dgst(Salted,Digest).

passwd(don,xor8,0x6d,"eNfwuBhtN9CUHxg==").
```

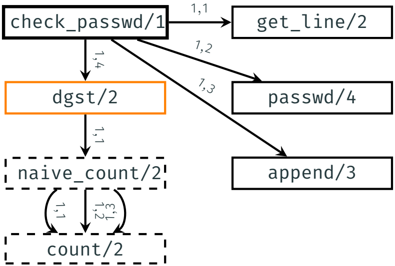
```
:- impl(hasher, naive/0).
(naive as hasher).dgst(Str, Digest) :-
    naive_count(Xs, 0, Digest).

naive_count(L, D0, D) :-
    count(L,'a',Na), D1 is D0 + Na*1,
    count(L,'b',Nb), D2 is D1 + Nb*2,
    count(L,'c',Nc), D3 is D2 + Nc*3,
    %% implementation continues
```



Example: Changing an assertion

```
:- trait hasher {
    :- pred dgst(Str, Digest)
        : str(Str) => int(Digest).
}.

check_passwd(User) :-
    get_line(Plain),
    passwd(User,Hasher,Digest,Salt),
    append(Plain,Salt,Salted),
    (Hasher as hasher).dgst(Salted,Digest).

passwd(don,xor8,0x6d,"eNfwuBhtN9CUHxg==").
```

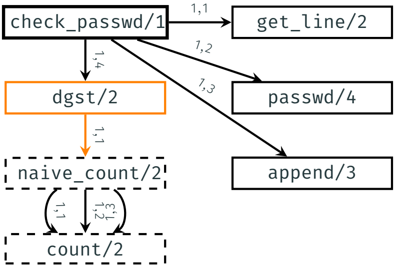
```
:- impl(hasher, naive/0).
(naive as hasher).dgst(Str, Digest) :-
    naive_count(Xs, 0, Digest).

naive_count(L, D0, D) :-
    count(L,'a',Na), D1 is D0 + Na*1,
    count(L,'b',Nb), D2 is D1 + Nb*2,
    count(L,'c',Nc), D3 is D2 + Nc*3,
    %% implementation continues
```



Example: Changing an assertion

```
:- trait hasher {
    :- pred dgst(Str, Digest)
        : str(Str) => int(Digest).
}.

check_passwd(User) :-
    get_line(Plain),
    passwd(User,Hasher,Digest,Salt),
    append(Plain,Salt,Salted),
    (Hasher as hasher).dgst(Salted,Digest).

passwd(don,xor8,0x6d,"eNfwuBhtN9CUHxg==").
```

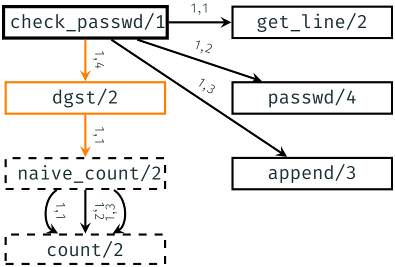
```
:- impl(hasher, naive/0).
(naive as hasher).dgst(Str, Digest) :-
    naive_count(Xs, 0, Digest).

naive_count(L, D0, D) :-
    count(L,'a',Na), D1 is D0 + Na*1,
    count(L,'b',Nb), D2 is D1 + Nb*2,
    count(L,'c',Nc), D3 is D2 + Nc*3,
    %% implementation continues
```



Example: Changing an assertion

```
:- trait hasher {
    :- pred dgst(Str, Digest)
        : str(Str) => int(Digest).
}.

check_passwd(User) :-
    get_line(Plain),
    passwd(User,Hasher,Digest,Salt),
    append(Plain,Salt,Salted),
    (Hasher as hasher).dgst(Salted,Digest).

passwd(don,xor8,0x6d,"eNfwuBhtN9CUHxg==").
```

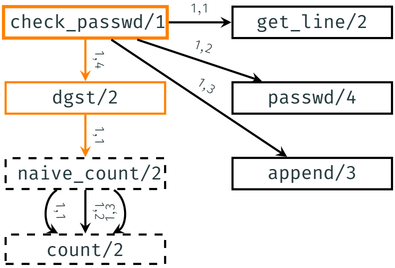
```
:- impl(hasher, naive/0).
(naive as hasher).dgst(Str, Digest) :-
    naive_count(Xs, 0, Digest).

naive_count(L, D0, D) :-
    count(L,'a',Na), D1 is D0 + Na*1,
    count(L,'b',Nb), D2 is D1 + Nb*2,
    count(L,'c',Nc), D3 is D2 + Nc*3,
    %% implementation continues
```



Example: Changing an assertion

```
:- trait hasher {
    :- pred dgst(Str, Digest)
        : str(Str) => int(Digest).
}.

check_passwd(User) :-
    get_line(Plain),
    passwd(User,Hasher,Digest,Salt),
    append(Plain,Salt,Salted),
    (Hasher as hasher).dgst(Salted,Digest).

passwd(don,xor8,0x6d,"eNfwuBhtN9CUHxg==").
```

```
:- impl(hasher, naive/0).
(naive as hasher).dgst(Str, Digest) :-
    naive_count(Xs, 0, Digest).

naive_count(L, D0, D) :-
    count(L,'a',Na), D1 is D0 + Na*1,
    count(L,'b',Nb), D2 is D1 + Nb*2,
    count(L,'c',Nc), D3 is D2 + Nc*3,
    %% implementation continues
```



Experiments

We tested the proposed algorithm with an application, LPdoc, a documentation generator for logic programs which has:

- A generic interface for back ends for different documentation formats.
- Several such back ends.
- 150 files, of mostly (`Ciao`) Prolog code.
- Assertions originally meant for documentation.
- 22K lines of code.

Analysis time adding one backend at a time (time in seconds):

domain	no backend	+ `texinfo`	+ `man`	+ `html`
`reachability`	1.7	2.1	3.4	3.9
`reachability` inc	1.7	1.2	1.0	1.6
`gr`	2.1	2.2	2.3	2.6
`gr` inc	2.1	1.4	0.9	1.8
`def`	6.0	7.1	7.8	9.7
`def` inc	6.0	2.2	1.3	3.5
`sharing`	27.2	28.1	24.2	28.5
`sharing` inc	27.2	3.9	1.4	5.1

Related work

- Mora et al. (ASE 2018) perform modular symbolic execution to prove that some (versions of) libraries are **equivalent with respect to the same client**.
- Chatterjee et al. (POPL 2018) analyze libraries in the presence of **callbacks** incrementally for data dependence analysis.

Conclusions

- We have proposed a context-sensitive program analysis algorithm that (re)computes summaries for predicates, reacting incrementally to fine grain changes in (multivariant) assertions and the program.
- As a specific application of the algorithm we proposed an approach to the analysis of generic code, in a way that we can efficiently specialize the analysis result as implementations become available.
- We have provided a syntax to build generic programs in Prolog using traits.
- We have applied running this algorithm in a fairly large tool (LPdoc), which shows promising results.

Conclusions

- We have proposed a context-sensitive program analysis algorithm that (re)computes summaries for predicates, reacting incrementally to fine grain changes in (multivariant) assertions and the program.
- As a specific application of the algorithm we proposed an approach to the analysis of generic code, in a way that we can efficiently specialize the analysis result as implementations become available.
- We have provided a syntax to build generic programs in Prolog using traits.
- We have applied running this algorithm in a fairly large tool (LPdoc), which shows promising results.

Thanks!

Check out the tool: `https://github.com/ciao-lang/ciaopp`

Full version of the algorithm

function $\textsc{IncAnalyze-w/AssrtChanges}((Cls, As), \Delta_{Cls}, \Delta_{As}, \mathcal{Q}, \mathcal{A})$

- $R := \emptyset$
- **for each** $P \in Cls$ **do**
  - **if** $\Delta_{As}[P] \neq \emptyset$ **then**
    - $R := R \cup$ `update_calls_pred`(P)
    - $R := R \cup$ `update_success_pred`(P)
- $\mathcal{A}' := \textsc{IncAnalyze}((Cls, As), \Delta_{Cls}, \mathcal{Q} \cup R, \mathcal{A})$
- del $(\mathcal{A}', \{E \mid E \in \mathcal{A}' \wedge Q \not\rightsquigarrow E \wedge Q \in \mathcal{Q}\})$ ▷ Remove unreachable calls
- **return** $\mathcal{A}'$

Full version of the algorithm

function `update_calls_pred`(P)

$Q := \emptyset$

for each $\langle P', \lambda \rangle_{c,l} \xrightarrow{\lambda^p} \langle P, \lambda^c_{old} \rangle \in \mathscr{A}$ do

$\lambda^c := \sigma(\mathrm{abs\_project}(\lambda^p, vars(P'_{c,l}))$ s.t. $\sigma(P'_{c,l}) = P$ ▷ Original call

$\lambda^c_{new} :=$ `apply_call`(P, λ^c)

if $\exists \langle P', \lambda^c{}_{new} \rangle \mapsto \lambda^s \in \mathscr{A}$ then ▷ A node for that call already exist

$\lambda^{s\prime} := \lambda^s$

else $\lambda^{s\prime} := \bot$

$Q := Q \cup$ `treat_change`($\langle P', \lambda \rangle_{c,l} \xrightarrow[\lambda^r]{\lambda^p} \langle P, \lambda^c{}_{new} \rangle, \lambda^{s\prime}$)

return Q

function `update_successes_pred`(P)

$Q := \emptyset$

for each $\langle P, \lambda^c \rangle \mapsto \lambda^s \in \mathscr{A}$ do

$\lambda := \bot$

for each $\langle P, \lambda^c \rangle_{c,last} \xrightarrow[\lambda^r]{} \langle Q, \lambda \rangle \in \mathscr{A}$ do ▷ Original success

$\lambda := \lambda \sqcup$ `apply_success`($P, \lambda^c, \mathrm{abs\_project}(\lambda^r, vars(P_c))$)

for each $E = N_{\bullet,\bullet} \xrightarrow[\bullet]{\bullet} \langle P, \lambda^c \rangle \in \mathscr{A}$ do ▷ Affected literals

$Q := Q \cup$ `treat_change` (E, λ)

return Q

Full version of the algorithm

function `treat_change`$(\langle P, \lambda\rangle_{c,l} \xrightarrow[\lambda^r]{\lambda^p} \langle Q, \lambda^c\rangle, \lambda^s)$

$\lambda^{r\prime} := \mathrm{abs\_extend}(\lambda^p, \lambda^s)$ ▷ Obtain new info at literal return

del$(\mathscr{A}, \langle P, \lambda\rangle_{c,l} \xrightarrow[\bullet]{\bullet} \bullet)$

add$(\mathscr{A}, \langle P, \lambda\rangle_{c,l} \xrightarrow[\lambda^{r\prime}]{\lambda^p} \langle Q, \lambda^c\rangle)$

if $\lambda^r \sqsubset \lambda^{r\prime}$ **then**

  return $\{\langle P, \lambda\rangle\}$ ▷ Restart the analysis for this predicate and call pattern

else if $\lambda^r \not\sqsubseteq \lambda^{r\prime}$ **then** ▷ Analysis is potentially imprecise

  $Lits := \{E \mid E = \langle P, \lambda\rangle_{c,i} \longrightarrow N \in \mathscr{A} \wedge i > l\}$ ▷ Following literals

  $IN := \{E \mid E \rightsquigarrow L \in \mathscr{A} \wedge L \in Lits\}$ ▷ Potentially imprecise nodes

  $Q = IN \cap \mathscr{Q}$ ▷ Entry point of potentially imprecise nodes

  del$(\mathscr{A}, IN)$

  return Q

else return $\emptyset$