Mathematics of Program Construction 2019

Porto, Portugal

Verified Self-Explaining Computation

Jan Stolarek[1,2] James Cheney[1,3]

[1]University of Edinburgh, UK

[2]Lodz University of Technology, Poland [3]The Alan Turing Institute, UK

input state: $[x \mapsto 1, y \mapsto 0, z \mapsto 2]$

```
if (y = 1) then { y := x + 1 }
           else { y := y + 1 };
z := z + 1
```

input state: $[x \mapsto 1, y \mapsto 0, z \mapsto 2]$

```
if (y = 1) then { y := x + 1 }
           else { y := y + 1 };
z := z + 1
```

$$\Downarrow$$

$$\boxed{\text{eval}}$$

$$\Downarrow$$

input state: $[x \mapsto 1, y \mapsto 0, z \mapsto 2]$

```
if (y = 1) then { y := x + 1 }
           else { y := y + 1 };
z := z + 1
```

eval

expected: $[x \mapsto 1, y \mapsto 2, z \mapsto 3]$

input state: $[x \mapsto 1, y \mapsto 0, z \mapsto 2]$

```
if (y = 1) then { y := x + 1 }
           else { y := y + 1 };
z := z + 1
```

$$\Downarrow$$

$$\boxed{\text{eval}}$$

$$\Downarrow$$

expected: $[x \mapsto 1, y \mapsto 2, z \mapsto 3]$
actual: $[x \mapsto 1, y \mapsto 1, z \mapsto 3]$

Debugger

Slicing

$$[x \mapsto \square, y \mapsto 1, z \mapsto \square]$$

$$[x \mapsto \Box, y \mapsto 0, z \mapsto \Box]$$

```
if (y = 1) then { □ }
           else { y := y + 1 }; □
```

$$\Uparrow$$

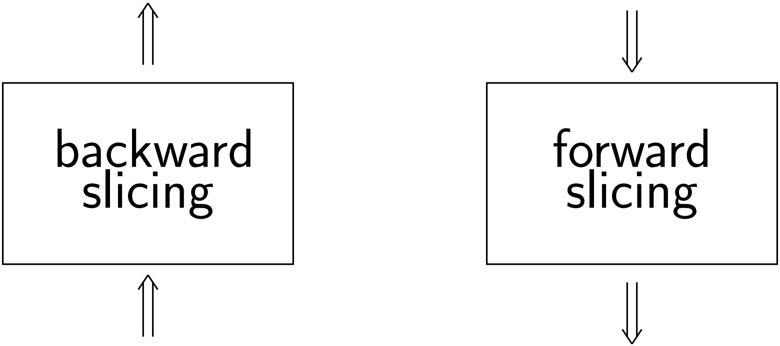
backward slicing

$$\Uparrow$$

$$[x \mapsto \Box, y \mapsto 1, z \mapsto \Box]$$

$[x \mapsto \square, y \mapsto 0, z \mapsto \square]$

```
if (y = 1) then { □ }
            else { y := y + 1 }; □
```



$[x \mapsto \square, y \mapsto 1, z \mapsto \square]$ $\qquad$ $[x \mapsto \square, y \mapsto 1, z \mapsto \square]$

Specifying backward slicing

Consider these two extreme cases of backward slicing behaviour:

Specifying backward slicing

Consider these two extreme cases of backward slicing behaviour:

1. for any slicing criterion backward slicing always returns a $\square$, i.e. it discards all the program code;

Specifying backward slicing

Consider these two extreme cases of backward slicing behaviour:

1. for any slicing criterion backward slicing always returns a $\square$, i.e. it discards all the program code;
2. for any slicing criterion backward slicing always returns a full program with no holes inserted.

Specifying backward slicing

Consider these two extreme cases of backward slicing behaviour:

1. for any slicing criterion backward slicing always returns a $\square$, i.e. it discards all the program code;
2. for any slicing criterion backward slicing always returns a full program with no holes inserted.

- **consistency**: backward slicing retains code required to produce output we are interested in.

Specifying backward slicing

Consider these two extreme cases of backward slicing behaviour:

1. for any slicing criterion backward slicing always returns a $\square$, i.e. it discards all the program code;
2. for any slicing criterion backward slicing always returns a full program with no holes inserted.

- **consistency**: backward slicing retains code required to produce output we are interested in.
- **minimality**: backward slicing produces the smallest partial program and partial input state that suffice to achieve consistency.

Galois connection slicing framework

Slicing based on Galois connections:

Galois connection slicing framework

Slicing based on Galois connections:

- known from literature [15, 17]

[15] Perera, R., Acar, U.A., Cheney, J., Levy, P.B.: *Functional programs that explain their work*. In: ICFP. pp. 365–376. ACM (2012)
[17] Ricciotti, W., Stolarek, J., Perera, R., Cheney, J.: *Imperative functional programs that explain their work*. Proceedings of the ACM on Programming Languages (PACMPL) 1(ICFP) (Sep 2017)

Galois connection slicing framework

Slicing based on Galois connections:

- known from literature [15, 17]
- previous research focused on functional languages

[15] Perera, R., Acar, U.A., Cheney, J., Levy, P.B.: *Functional programs that explain their work*. In: ICFP. pp. 365–376. ACM (2012)
[17] Ricciotti, W., Stolarek, J., Perera, R., Cheney, J.: *Imperative functional programs that explain their work*. Proceedings of the ACM on Programming Languages (PACMPL) 1(ICFP) (Sep 2017)

Galois connection slicing framework

Slicing based on Galois connections:

- known from literature [15, 17]
- previous research focused on functional languages
- pen and paper proofs

[15] Perera, R., Acar, U.A., Cheney, J., Levy, P.B.: *Functional programs that explain their work*. In: ICFP. pp. 365–376. ACM (2012)
[17] Ricciotti, W., Stolarek, J., Perera, R., Cheney, J.: *Imperative functional programs that explain their work*. Proceedings of the ACM on Programming Languages (PACMPL) 1(ICFP) (Sep 2017)

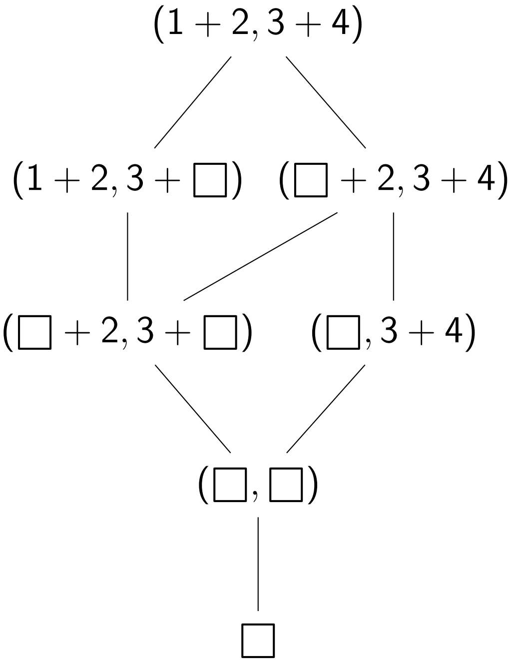
Galois connection slicing framework

Slicing based on Galois connections:

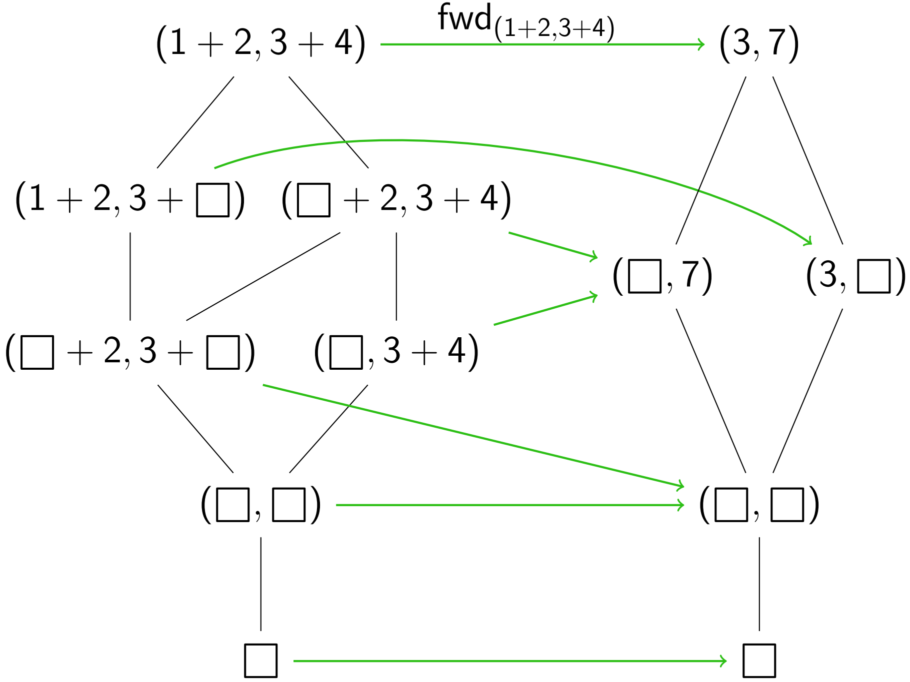
- known from literature [15, 17]
- previous research focused on functional languages
- pen and paper proofs

Our contributions:

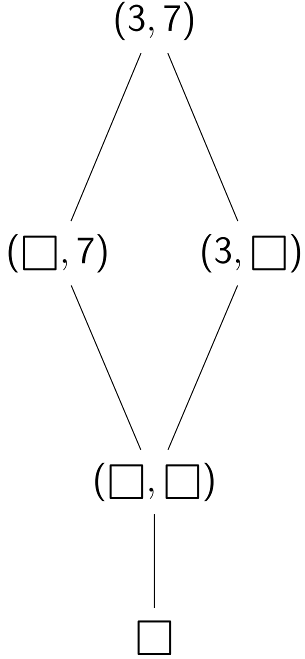
- minimal and consistent slicing for an imperative language
- proofs formalised in Coq
- a different proof strategy



Partial expressions form a *finite lattice*


$\mathrm{fwd}_{(1+2,3+4)}$
$(1+2, 3+4)$
$(3, 7)$
$(1+2, 3+\square)$
$(\square+2, 3+4)$
$(\square, 7)$
$(3, \square)$
$(\square+2, 3+\square)$
$(\square, 3+4)$
$(\square, \square)$
$(\square, \square)$
$\square$
$\square$

Partial values form a *finite lattice*



Galois connections

A pair of forward and backward slicing functions is guaranteed to have both the minimality and consistency properties when they form a Galois connection.

Galois connections

A pair of forward and backward slicing functions is guaranteed to have both the minimality and consistency properties when they form a Galois connection.

$$(P, \sqsubseteq_P), \quad (Q, \sqsubseteq_Q), \quad f : P \to Q, \quad g : Q \to P$$

f and g form a Galois connection (written $f \dashv g$) when

$$f(p) \sqsubseteq_Q q \iff p \sqsubseteq_Q g(q)$$

Galois connections

A pair of forward and backward slicing functions is guaranteed to have both the minimality and consistency properties when they form a Galois connection.

$$\mathit{bwd} : \mathit{values}_{\Box} \to \mathit{expressions}_{\Box}, \quad \mathit{fwd} : \mathit{expressions}_{\Box} \to \mathit{values}_{\Box}$$

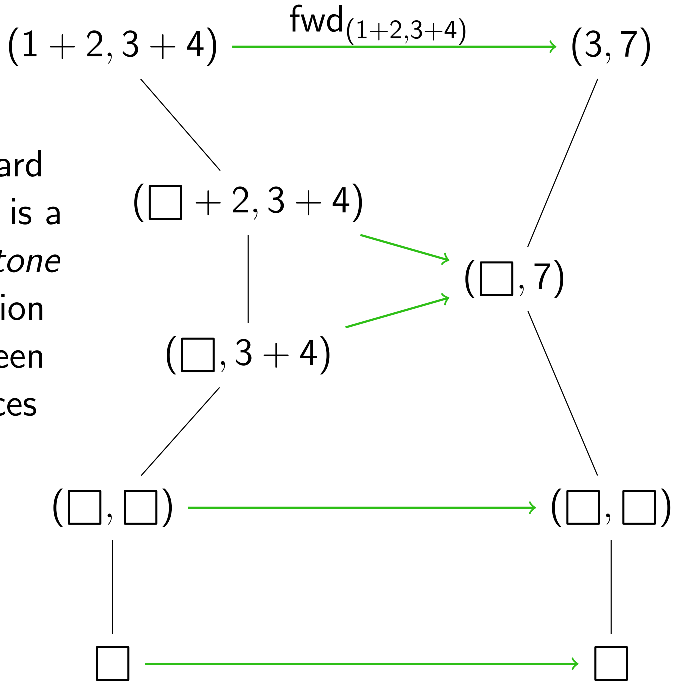
bwd and *fwd* form a Galois connection (written $\mathit{bwd} \dashv \mathit{fwd}$) when

$$\mathit{bwd}(\text{slicing criterion}) \sqsubseteq \text{expr}_{\Box} \iff \text{slicing criterion} \sqsubseteq \mathit{fwd}(\text{expr}_{\Box})$$

Galois connections

Given a monotone forward slicing function fwd there exists a backward slicing function bwd such that bwd $\dashv$ fwd.

Forward slicing is a *monotone* function between lattices



Galois connections

Given a monotone forward slicing function fwd there exists a backward slicing function bwd such that bwd $\dashv$ fwd.

The choice of fwd uniquely determines bwd.

Galois connections

In order to prove that bwd $\dashv$ fwd we need to show that:

1. both fwd and bwd are monotone
2. *deflation* property holds:

$$\forall_{q \in Q} \;\; f(g(q)) \sqsubseteq_Q q$$

3. *inflation* property holds:

$$\forall_{p \in P} \;\; p \sqsubseteq_P g(f(p))$$

In order to prove that $\mathsf{bwd} \dashv \mathsf{fwd}$ we need to show that:

1. both fwd and bwd are monotone
2. *deflation* property holds:

$$\forall\ \mathsf{bwd}(\mathsf{fwd}(\mathsf{expr}_{\Box})) \sqsubseteq \mathsf{expr}_{\Box}$$

3. *inflation* property holds:

$$\forall\ \mathsf{slicing\,criterion} \sqsubseteq \mathsf{fwd}(\mathsf{bwd}(\mathsf{slicing\,criterion}))$$

Our formalisation strategy:

1. prove that any functions fulfilling conditions (1)-(3) form a Galois connection
2. show that our definitions of forward and backward slicing fulfil conditions (1)-(3)

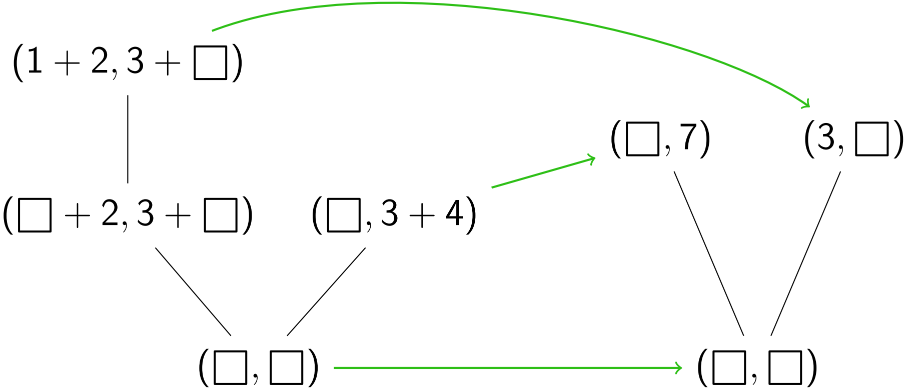
Galois connections

Our formalisation strategy:

1. prove that any functions fulfilling conditions (1)-(3) form a Galois connection
2. show that our definitions of forward and backward slicing fulfil conditions (1)-(3)

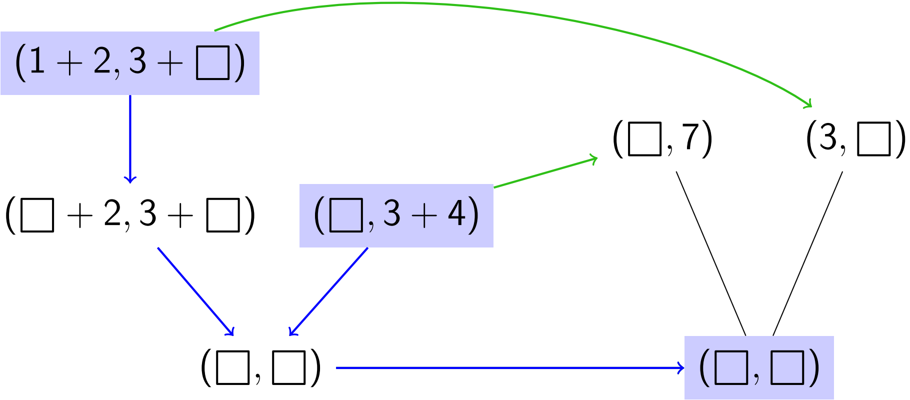
Possible to rely on meet and join preservation instead of monotonicity.

Forward slicing *preserves meets*



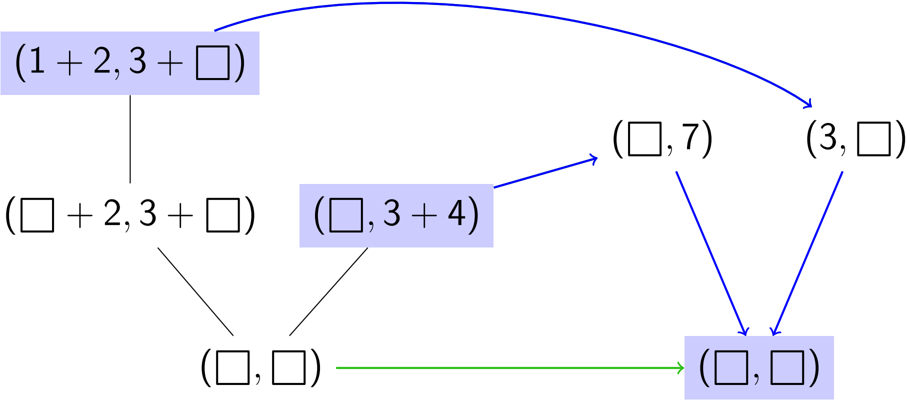
$$fwd(x \sqcap y) \;=\; fwd(x) \sqcap fwd(y)$$

Forward slicing *preserves meets*



$$\mathit{fwd}(x \sqcap y) = \mathit{fwd}(x) \sqcap \mathit{fwd}(y)$$

Forward slicing *preserves meets*



$$\mathit{fwd}(x \sqcap y) = \mathit{fwd}(x) \sqcap \mathit{fwd}(y)$$

Syntax

$$a \quad ::= \quad n \mid x \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \cdot a_2$$

Syntax

$$
\begin{array}{lcl}
a & ::= & n \mid x \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \cdot a_2 \\
b & ::= & \texttt{true} \mid \texttt{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b_2
\end{array}
$$

Syntax

$$
\begin{array}{rcl}
a & ::= & n \mid x \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \cdot a_2 \\
b & ::= & \texttt{true} \mid \texttt{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b_2 \\
c & ::= & \texttt{skip} \mid x := a \mid c_1 \,;\, c_2 \mid \texttt{if}\ b\ \texttt{then}\ \{\ c_1\ \}\ \texttt{else}\ \{\ c_2\ \} \\
 & \mid & \texttt{while}\ b\ \texttt{do}\ \{\ c\ \}
\end{array}
$$

Syntax

$$
\begin{array}{rcl}
a & ::= & n \mid x \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \cdot a_2 \\
b & ::= & \texttt{true} \mid \texttt{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b_2 \\
c & ::= & \texttt{skip} \mid x := a \mid c_1 \,;\, c_2 \mid \texttt{if}\ b\ \texttt{then}\ \{\ c_1\ \}\ \texttt{else}\ \{\ c_2\ \} \\
 & \mid & \texttt{while}\ b\ \texttt{do}\ \{\ c\ \} \\
v & ::= & v_a \mid v_b
\end{array}
$$

Syntax

$$
\begin{array}{rcl}
a & ::= & n \mid x \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \cdot a_2 \\
b & ::= & \texttt{true} \mid \texttt{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b_2 \\
c & ::= & \texttt{skip} \mid x := a \mid c_1 \,;\, c_2 \mid \texttt{if}\ b\ \texttt{then}\ \{\ c_1\ \}\ \texttt{else}\ \{\ c_2\ \} \\
 & \mid & \texttt{while}\ b\ \texttt{do}\ \{\ c\ \} \\
v & ::= & v_a \mid v_b \\
\mu & ::= & \varnothing \mid \mu, x \mapsto v_a \ (x \text{ fresh})
\end{array}
$$

Forward slicing: notation

$$T :: \mu, e \nearrow v$$

Notation:

- $T ::$ - execution trace
- μ - partial state
- e - partial program (expression or command)
- v - slicing result (μ' for commands)

Forward slicing: expressions

$$\frac{T_1 :: \mu, a_1 \nearrow v_1 \qquad T_2 :: \mu, a_2 \nearrow v_2}{T_1 + T_2 :: \mu, a_1 + a_2 \nearrow v_1 +_{\mathbb{N}} v_2} \quad v_1, v_2 \neq \square$$

Forward slicing: expressions

$$\frac{T_1 :: \mu, a_1 \nearrow v_1 \qquad T_2 :: \mu, a_2 \nearrow v_2}{T_1 + T_2 :: \mu, a_1 + a_2 \nearrow v_1 +_{\mathbb{N}} v_2} \quad v_1, v_2 \neq \Box$$

$$\frac{T_1 :: \mu, a_1 \nearrow \Box}{T_1 + T_2 :: \mu, a_1 + a_2 \nearrow \Box} \qquad \frac{T_2 :: \mu, a_2 \nearrow \Box}{T_1 + T_2 :: \mu, a_1 + a_2 \nearrow \Box}$$

Forward slicing: expressions

$$\frac{}{x(v_a) :: \mu, x \nearrow \mu(x)}$$

Important that we read x from μ and not v_a from a trace; we can't assume $\mu(x) = v_a$.

Forward slicing: commands

$$\frac{T_a :: \mu, a \nearrow v_a}{x := T_a :: \mu, x := a \nearrow \mu[x \mapsto v_a]}$$

Forward slicing: commands

$$\frac{}{x := T_a :: \mu, \Box \nearrow \mu[x \mapsto \boxed{}]}$$

Forward slicing: commands

$$\frac{}{x := T_a :: \mu, \Box \nearrow \mu[x \mapsto \boxed{}]}$$

$$\frac{T_1 :: \mu, \boxed{} \nearrow \mu' \qquad T_2 :: \mu', \boxed{} \nearrow \mu''}{T_1\,;\ T_2 :: \mu, \boxed{} \nearrow \mu''}$$

Forward slicing: commands

$$\frac{T_1 :: \mu, \Box \nearrow \mu'}{\texttt{if}_{\textbf{true}}\ T_b\ \texttt{then}\ \{\ T_1\ \} :: \mu, \Box \nearrow \mu'}$$

Forward slicing: commands

$$\frac{T_1 :: \mu, \Box \nearrow \mu'}{\mathtt{if}_{\mathbf{true}}\ T_b\ \mathtt{then}\ \{\ T_1\ \} :: \mu, \Box \nearrow \mu'}$$

$$\frac{T_b :: \mu, b \nearrow \Box \qquad T_1 :: \mu, \Box \nearrow \mu'}{\mathtt{if}_{\mathbf{true}}\ T_b\ \mathtt{then}\ \{\ T_1\ \} :: \mu, \mathtt{if}\ b\ \mathtt{then}\ \{\ c_1\ \}\ \mathtt{else}\ \{\ c_2\ \} \nearrow \mu'}$$

Forward slicing: commands

$$\frac{T_1 :: \mu, \Box \nearrow \mu'}{\texttt{if}_{\textbf{true}}\ T_b\ \texttt{then}\ \{\ T_1\ \} :: \mu, \Box \nearrow \mu'}$$

$$\frac{T_b :: \mu, b \nearrow \Box \qquad T_1 :: \mu, \Box \nearrow \mu'}{\texttt{if}_{\textbf{true}}\ T_b\ \texttt{then}\ \{\ T_1\ \} :: \mu, \texttt{if}\ b\ \texttt{then}\ \{\ c_1\ \}\ \texttt{else}\ \{\ c_2\ \} \nearrow \mu'}$$

$$\frac{T_b :: \mu, b \nearrow v_b \qquad T_1 :: \mu, c_1 \nearrow \mu'}{\texttt{if}_{\textbf{true}}\ T_b\ \texttt{then}\ \{\ T_1\ \} :: \mu, \texttt{if}\ b\ \texttt{then}\ \{\ c_1\ \}\ \texttt{else}\ \{\ c_2\ \} \nearrow \mu'}\quad v_b \neq \Box$$

Backward slicing: notation

$$T :: \mu, v \searrow \mu', e$$

Notation:

- $T ::$ - execution trace
- μ, v - slicing criterion (only μ for commands)
- μ', e - reconstructed partial program (state + code)

Backward slicing: notation

$$T :: \mu, v \searrow \mu', e$$

Notation:

- $T ::$ - execution trace
- μ, v - slicing criterion (only μ for commands)
- μ', e - reconstructed partial program (state + code)

Also:

- $\varnothing_\mu$ - all variables map to $\square$, same domain as μ
- $\varnothing$ - empty state (no variables)

Backward slicing: base cases

Expressions:

$$\frac{}{T :: \mu, \Box \searrow \varnothing_\mu, \Box}$$

Backward slicing: base cases

Expressions:

$$\frac{}{T :: \mu, \Box \searrow \varnothing_\mu, \Box}$$

Commands:

$$\frac{}{T :: \varnothing \searrow \varnothing, \Box}$$

Backward slicing: variable assignments

$$\frac{}{x := T_a :: \mu[x \mapsto \square] \searrow \mu[x \mapsto \square], \square}$$

Backward slicing: variable assignments

$$\frac{}{x := T_a :: \mu[x \mapsto \Box] \searrow \mu[x \mapsto \Box], \Box}$$

$$\frac{T_a :: v_a \searrow \mu_a, a}{x := T_a :: \mu[x \mapsto v_a] \searrow \mu_a \sqcup \mu[x \mapsto \Box], x := a} \quad v_a \neq \Box$$

Backward slicing: conditionals

$$\dfrac{T_1 :: \mu \searrow \mu', c_1 \qquad T_b :: \mathtt{true} \searrow \mu_b, b}{\mathtt{if}_{\mathbf{true}}\ T_b\ \mathtt{then}\ \{\ T_1\ \} :: \mu \searrow \mu' \sqcup \mu_b, \mathtt{if}\ b\ \mathtt{then}\ \{\ c_1\ \}\ \mathtt{else}\ \{\ \square\ \}} \quad c_1 \neq \square$$

Backward slicing: conditionals

$$\frac{T_1 :: \mu \searrow \mu', c_1 \qquad T_b :: \texttt{true} \searrow \mu_b, b}{\texttt{if}_{\textbf{true}}\ T_b\ \texttt{then}\ \{\ T_1\ \} :: \mu \searrow \mu' \sqcup \mu_b, \texttt{if}\ b\ \texttt{then}\ \{\ c_1\ \}\ \texttt{else}\ \{\ \Box\ \}} \quad c_1 \neq \Box$$

$$\frac{T_1 :: \mu \searrow \mu', \Box}{\texttt{if}_{\textbf{true}}\ T_b\ \texttt{then}\ \{\ T_1\ \} :: \mu \searrow \mu', \Box}$$

Backward slicing: loops

$$\frac{}{\texttt{while}_{\textsf{false}}\ T_b :: \mu \searrow \mu, \Box}$$

Backward slicing: loops

$$\frac{}{\texttt{while}_{\textbf{false}}\ T_b :: \mu \searrow \mu, \square}$$

$$\frac{T_w :: \mu \searrow \mu_w, \square \qquad T_c :: \mu_w \searrow \mu_c, \square}{\texttt{while}_{\textbf{true}}\ T_b\ \texttt{do}\ \{\ T_c\ \};\ T_w :: \mu \searrow \mu_c, \square}$$

Backward slicing: loops

$$\frac{}{\texttt{while}_{\textbf{false}}\ T_b :: \mu \searrow \mu, \Box}$$

$$\frac{T_w :: \mu \searrow \mu_w, \Box \qquad T_c :: \mu_w \searrow \mu_c, \Box}{\texttt{while}_{\textbf{true}}\ T_b\ \texttt{do}\ \{\ T_c\ \};\ T_w :: \mu \searrow \mu_c, \Box}$$

$$\frac{T_w :: \mu \searrow \mu_w, \Box \qquad T_c :: \mu_w \searrow \mu_c, c \qquad T_b :: \texttt{true} \searrow \mu_b, b}{\texttt{while}_{\textbf{true}}\ T_b\ \texttt{do}\ \{\ T_c\ \};\ T_w :: \mu \searrow \mu_c \sqcup \mu_b, \texttt{while}\ b\ \texttt{do}\ \{\ c\ \}} \quad c \neq \Box$$

Backward slicing: loops

$$\frac{}{\texttt{while}_{\textbf{false}}\ T_b :: \mu \searrow \mu, \square}$$

$$\frac{T_w :: \mu \searrow \mu_w, \square \qquad T_c :: \mu_w \searrow \mu_c, \square}{\texttt{while}_{\textbf{true}}\ T_b\ \mathrm{do}\ \{\ T_c\ \};\ T_w :: \mu \searrow \mu_c, \square}$$

$$\frac{T_w :: \mu \searrow \mu_w, \square \qquad T_c :: \mu_w \searrow \mu_c, c \qquad T_b :: \texttt{true} \searrow \mu_b, b}{\texttt{while}_{\textbf{true}}\ T_b\ \mathrm{do}\ \{\ T_c\ \};\ T_w :: \mu \searrow \mu_c \sqcup \mu_b, \texttt{while}\ b\ \texttt{do}\ \{\ c\ \}}\ c \neq \square$$

$$\frac{T_w :: \mu \searrow \mu_w, c_w \qquad T_c :: \mu_w \searrow \mu_c, c \qquad T_b :: \texttt{true} \searrow \mu_b, b}{\texttt{while}_{\textbf{true}}\ T_b\ \mathrm{do}\ \{\ T_c\ \};\ T_w :: \mu \searrow \mu_c \sqcup \mu_b, c_w \sqcup \texttt{while}\ b\ \texttt{do}\ \{\ c\ \}}\ c_w \neq \square$$

Formalisation

- 5,2k LOC

Formalisation

- 5,2k LOC
- abstract theorems about relations, lattices, and Galois connections

Formalisation

- 5,2k LOC
- abstract theorems about relations, lattices, and Galois connections
- elements of lattice stored with a proof that they belong to a lattice...

Formalisation

- 5,2k LOC
- abstract theorems about relations, lattices, and Galois connections
- elements of lattice stored with a proof that they belong to a lattice...
- ...but implementation of join operations still tricky

Formalisation

- 5,2k LOC
- abstract theorems about relations, lattices, and Galois connections
- elements of lattice stored with a proof that they belong to a lattice...
- ...but implementation of join operations still tricky
- tedious formalisation of state

Formalisation

- 5,2k LOC
- abstract theorems about relations, lattices, and Galois connections
- elements of lattice stored with a proof that they belong to a lattice...
- ...but implementation of join operations still tricky
- tedious formalisation of state
- evaluation defined as inductive data types

Formalisation

- 5,2k LOC
- abstract theorems about relations, lattices, and Galois connections
- elements of lattice stored with a proof that they belong to a lattice...
- ...but implementation of join operations still tricky
- tedious formalisation of state
- evaluation defined as inductive data types
- slicing defined as a (dependent) function; takes evaluation evidence; `refine` tactic used to construct evidence

Formalisation

- 5,2k LOC
- abstract theorems about relations, lattices, and Galois connections
- elements of lattice stored with a proof that they belong to a lattice...
- ...but implementation of join operations still tricky
- tedious formalisation of state
- evaluation defined as inductive data types
- slicing defined as a (dependent) function; takes evaluation evidence; `refine` tactic used to construct evidence
- proofs of monotonicity, inflation, and deflation for defined slicing functions

Summary

We have developed a slicing algorithm for imperative programs based on Galois connections and formalised it in Coq.

More in the paper:

- full rules for forward and backward slicing
- extended example involving loops
- details of Coq formalisation

Future work:

- formalisation of a full-scale language

Implementation available at:

`https://bitbucket.org/jstolarek/gc_imp_slicing`

Mathematics of Program Construction 2019

Porto, Portugal

Verified Self-Explaining Computation

Jan Stolarek[1,2] James Cheney[1,3]

[1]University of Edinburgh, UK

[2]Lodz University of Technology, Poland [3]The Alan Turing Institute, UK

Minimality:

$$\mathsf{bwd}_e(v') = \bigsqcap\{e' \mid v' \sqsubseteq \mathsf{fwd}_e(e')\}$$