Compiling with Abstract Interpretation

PLDI 2024

Dorian Lesbre and Matthieu Lemerre

Copenhagen – June 26th, 2024

Motivations

Program analysis and transformations are mutually beneficial:

- Transformations can make analysis easier (ex: $e + e \rightarrow 2 * e$)
- Prior analysis can improve transformations
  (ex: dead code elimination)

Motivations

Program analysis and transformations are mutually beneficial:

- Transformations can make analysis easier (ex: $e + e \to 2 * e$)
- Prior analysis can improve transformations (ex: dead code elimination)

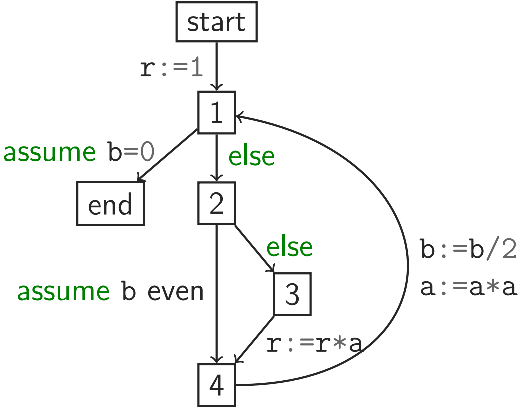
Abstract interpretation allows simultaneous combination of analyses, could it be extended to transformations?

Key ideas

1. **Free algebra** domains generate programs as analysis results, different languages have different domain signature;
2. **Domain functors** perform simultaneous compilation passes, soundness/completeness imply forward/backward simulations;
3. **Online compilation to SSA** improves precision of non-relational domains with constant overhead

1. Free algebra domain

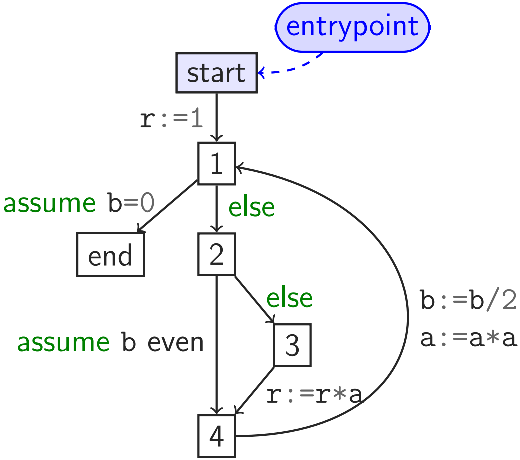
Abstract Interpretation



Domain signature:

```
type state
```

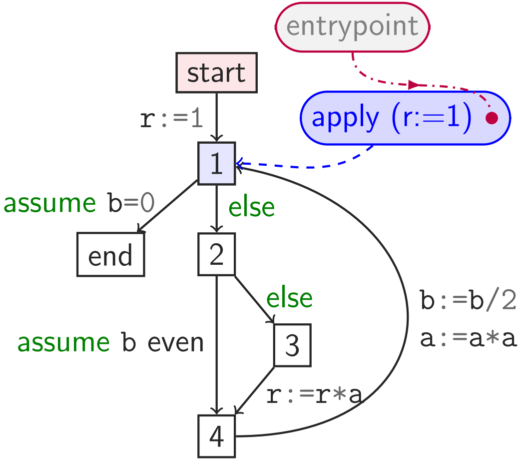
Abstract Interpretation



Domain signature:

```
type state
val entrypoint: state
```

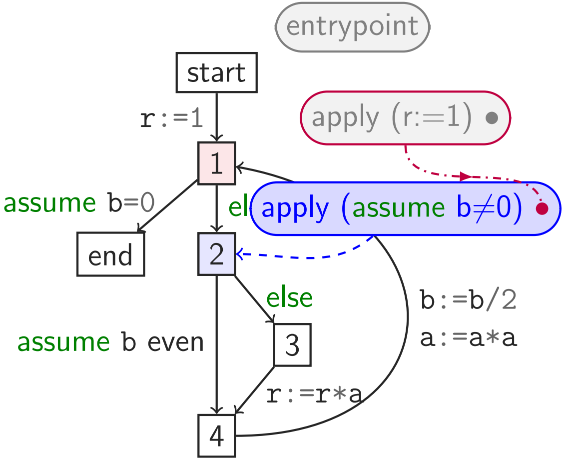
Abstract Interpretation



Domain signature:

```
type state
val entrypoint: state
val apply: rel → state → state
```

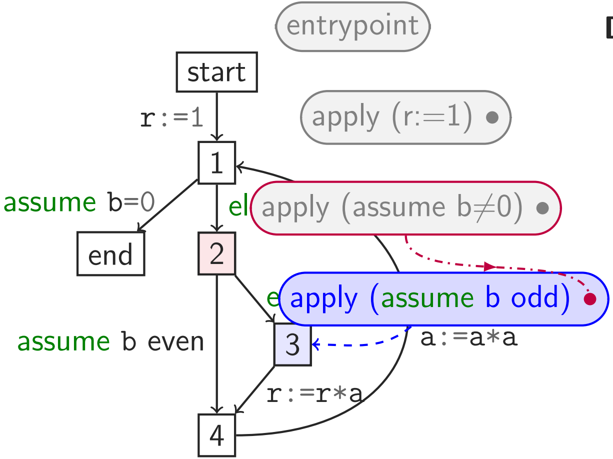
Abstract Interpretation



Domain signature:

```
type state
val entrypoint: state
val apply: rel → state → state
```

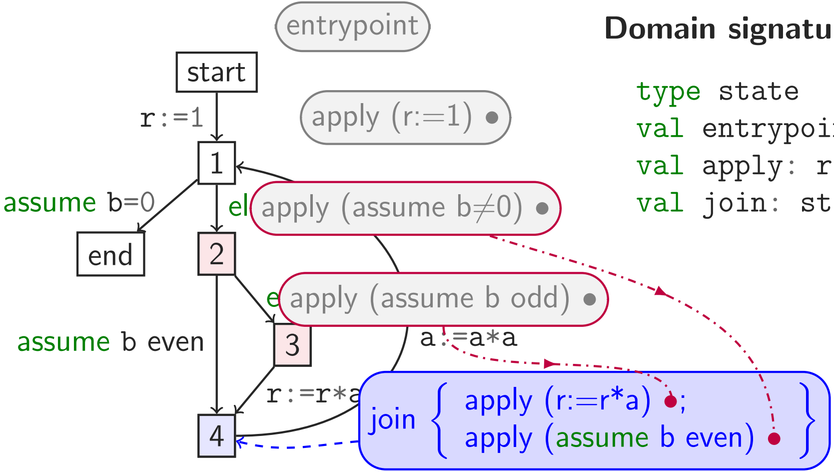
Abstract Interpretation



Domain signature:

```
type state
val entrypoint: state
val apply: rel → state → state
```

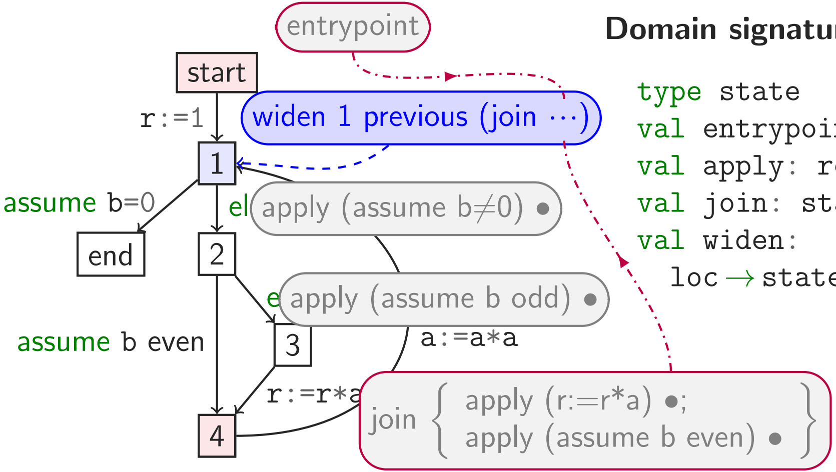
Abstract Interpretation



Domain signature:

```
type state
val entrypoint: state
val apply: rel → state → state
val join: state set → state
```

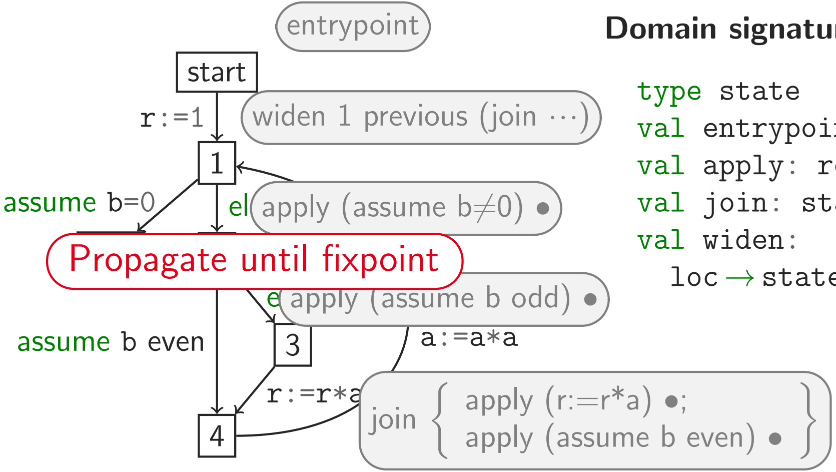
Abstract Interpretation



Domain signature:

```
type state
val entrypoint: state
val apply: rel → state → state
val join: state set → state
val widen:
  loc → state → state → state
```

Abstract Interpretation



Domain signature:

```
type state
val entrypoint: state
val apply: rel → state → state
val join: state set → state
val widen:
  loc → state → state → state
```

Free algebra domain

Implement domain signature:

```
module type DOMAIN = sig
  type state
  val entrypoint: state
  val apply: rel → state → state
  val join: state set → state
  val widen:
    loc → state → state → state
end
```

Free algebra domain

Implement domain signature:

```
module type DOMAIN = sig
  type state
  val entrypoint: state
  val apply: rel → state → state
  val join: state set → state
  val widen:
    loc → state → state → state
end
```

As a free algebra:

```
module FreeAlgrebra = struct
  type state = Entry
    | Apply of rel * state
    | Join of state set
    | Loc of loc

  let entrypoint = Entry
  let apply rel state = Apply(rel, state)
  let join states = Join states
  let widen l _ _ = Loc l
end
```

Generating the program graph

- `Apply` is a labeled edge:

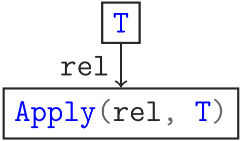
$$\frac{}{\texttt{T} \xrightarrow{\texttt{rel}} \texttt{Apply(rel, T)}}\ \textsc{TApply}$$

$$\begin{tikzcd} \boxed{\texttt{T}} \arrow[d, "\texttt{rel}"'] \\ \boxed{\texttt{Apply(rel, T)}} \end{tikzcd}$$

Generating the program graph

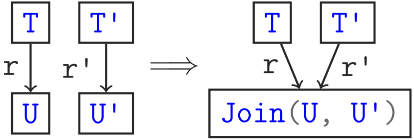
- `Apply` is a labeled edge:

$$\frac{}{\texttt{T} \xrightarrow{\texttt{rel}} \texttt{Apply(rel, T)}}\ \textsc{TApply}$$



- `Join` inherits its elements' edges:

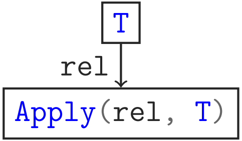
$$\frac{\texttt{T} \xrightarrow{\texttt{rel}} \texttt{U} \quad \wedge \quad \texttt{U} \in \texttt{S}}{\texttt{T} \xrightarrow{\texttt{rel}} \texttt{Join(S)}}\ \textsc{TJoin}$$



Generating the program graph

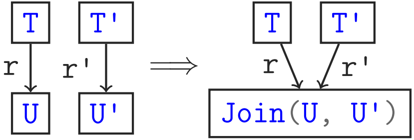
- Apply is a labeled edge:

$$\frac{}{\mathtt{T} \xrightarrow{\mathtt{rel}} \mathtt{Apply(rel,\ T)}}\ \textsc{TApply}$$



- Join inherits its elements' edges:

$$\frac{\mathtt{T} \xrightarrow{\mathtt{rel}} \mathtt{U} \quad \wedge \quad \mathtt{U} \in \mathtt{S}}{\mathtt{T} \xrightarrow{\mathtt{rel}} \mathtt{Join(S)}}\ \textsc{TJoin}$$

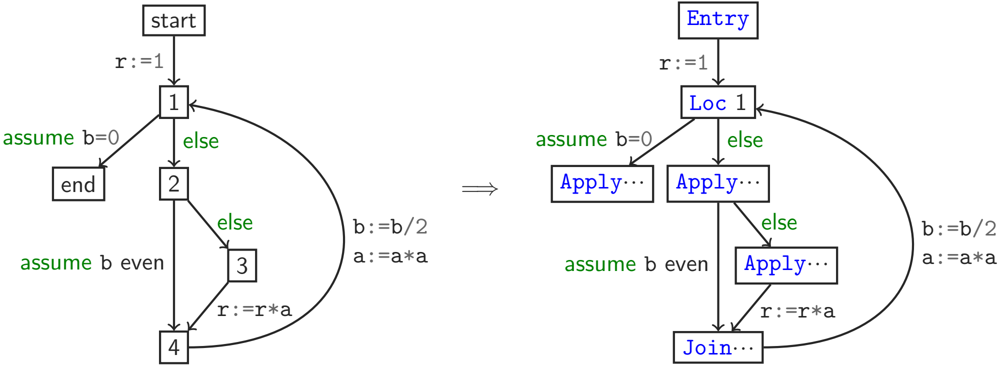


- Loc is unfolded to the pre-widening term

Graph isomorphism

Theorem

Analyzing with the free algebra domain yields a renaming of the initial CFG



2. Functors are compilation passes

Functors are compilation passes

Transformation functors just redefine apply:

```
module F(D : DOMAIN) : DOMAIN = struct
  type state          = D.state
  let entrypoint      = D.entrypoint
  let join set        = D.join set
  let widen loc l r = D.widen loc l r

  let apply rel state =
    composition of state, D.apply, D.join
end
```

Functor examples

- **Simplifications of conditionals:**

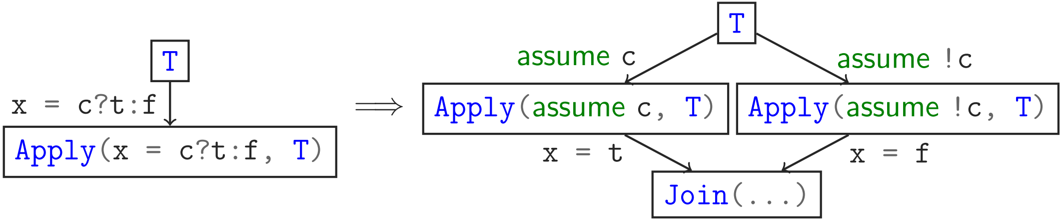
$$
\begin{aligned}
\mathtt{F(D).apply}\ (\mathsf{assume}\ \mathtt{e})\ \mathtt{T} &\triangleq \bot && \text{if } \mathtt{T} \vdash_{\mathtt{D}} \mathtt{e} = 0 \\
&\mid\ \mathtt{T} && \text{if } \mathtt{T} \vdash_{\mathtt{D}} \mathtt{e} \neq 0 \\
&\mid\ \mathtt{D.apply}\ (\mathsf{assume}\ \mathtt{e})\ \mathtt{T} && \text{otherwise} \\
\mathtt{F(D).apply}\ (\mathtt{x:=e})\ \mathtt{T} &\triangleq \mathtt{D.apply}\ (\mathtt{x:=e})\ \mathtt{T}
\end{aligned}
$$

Functor examples

- **Simplifications of conditionals:**

$$
\begin{aligned}
\mathtt{F(D).apply}\ (\mathsf{assume}\ \mathtt{e})\ \mathtt{T} &\triangleq \bot && \text{if } \mathtt{T} \vdash_{\mathtt{D}} \mathtt{e} = 0 \\
&\mid\ \mathtt{T} && \text{if } \mathtt{T} \vdash_{\mathtt{D}} \mathtt{e} \neq 0 \\
&\mid\ \mathtt{D.apply}\ (\mathsf{assume}\ \mathtt{e})\ \mathtt{T} && \text{otherwise} \\
\mathtt{F(D).apply}\ (\mathtt{x{:=}e})\ \mathtt{T} &\triangleq \mathtt{D.apply}\ (\mathtt{x{:=}e})\ \mathtt{T}
\end{aligned}
$$

- **Compilation of ternary expressions:**



Functors modularity

Theorem

- $\mathtt{F}(\mathtt{collecting\ semantics})$ sound $\Rightarrow \forall\ \mathtt{D}$ sound, $\mathtt{F}(\mathtt{D})$ sound
- $\mathtt{F}(\mathtt{collecting\ semantics})$ complete $\Rightarrow \forall\ \mathtt{D}$ complete, $\mathtt{F}(\mathtt{D})$ complete

Corollary:

- $\mathtt{F}$ sound and $\mathtt{G}$ sound $\Rightarrow \mathtt{F} \circ \mathtt{G}$ sound
- $\mathtt{F}$ complete and $\mathtt{G}$ complete $\Rightarrow \mathtt{F} \circ \mathtt{G}$ complete

Generating programs through functors

When applying functors to the **free algebra** domain:

1. **Functor soundness** implies a forward simulation useful for analysis:

$$traces(source) \subseteq traces(target)$$

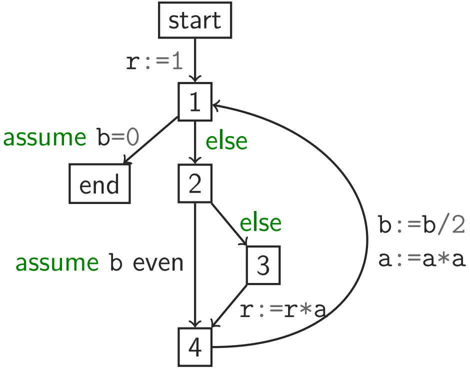
2. **Functor completeness** implies a backward simulation useful for compilation:

$$traces(source) \supseteq traces(target)$$

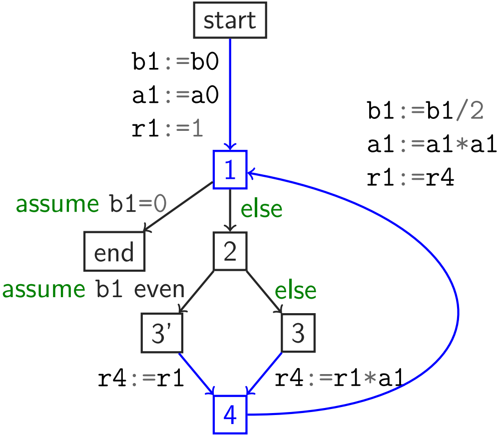
3. Compilation to SSA recovers context

SSA Form

Classical:

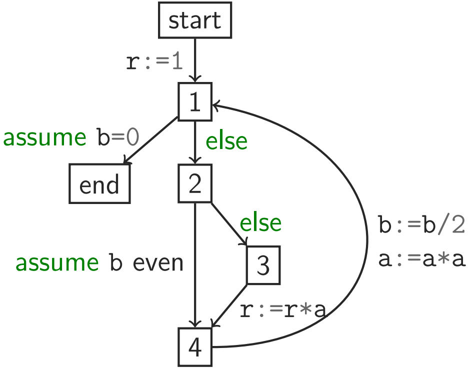


Single static assignment:

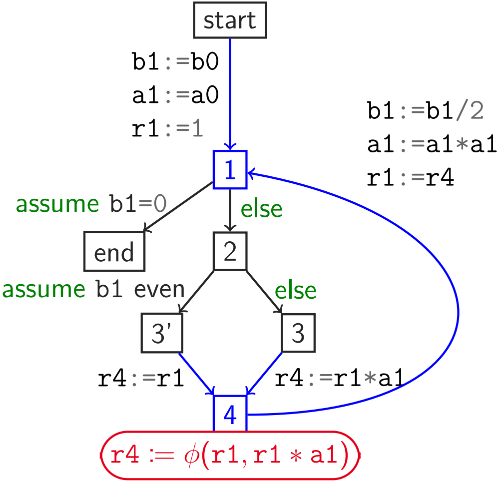


SSA Form

Classical:



Single static assignment:



SSA Domain signature

Classical domain signature:

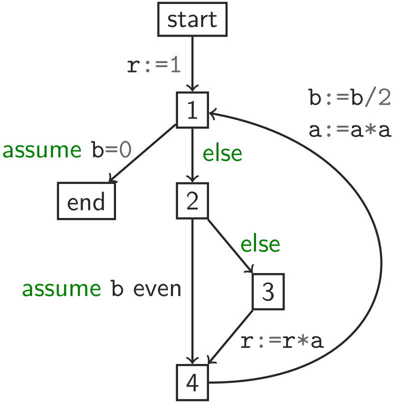
```
module type DOMAIN = sig
  type state
  val entrypoint: state
  val apply:
    rel → state → state
  val join: state set → state

  val widen:
    loc → state → state → state
end
```

SSA domain signature:

```
module type SSA_DOMAIN = sig
  type state
  val entrypoint: state
  val assume:
    expr → state → state
  val join:
    ((var --> expr) * state) set → state
  val widen:
    loc → state → state → state
end
```

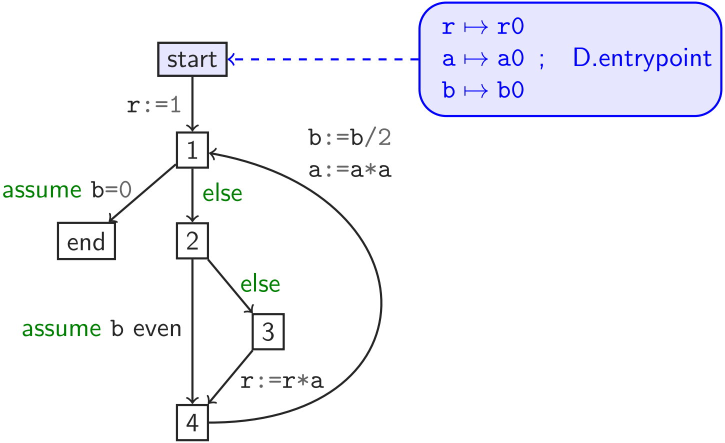
Compilation to SSA: the LiftSSA functor

`LiftSSA(D: SSA_DOMAIN)` $\rightarrow$ `DOMAIN with state` $\triangleq$ `(var --> ssa_expr) * D.state`



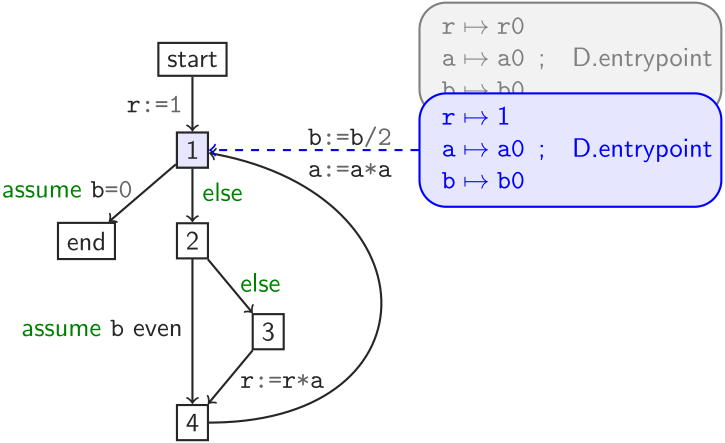
Compilation to SSA: the LiftSSA functor

`LiftSSA(D: SSA_DOMAIN)` $\rightarrow$ `DOMAIN with state` $\triangleq$ `(var --> ssa_expr) * D.state`



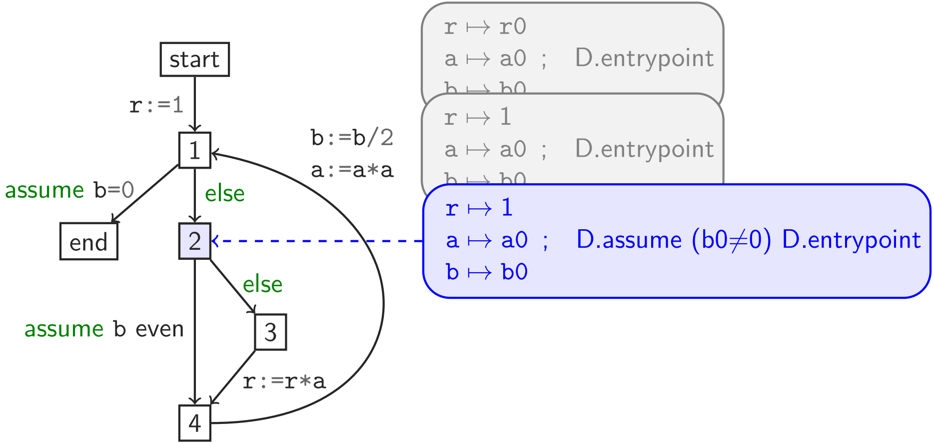
Compilation to SSA: the LiftSSA functor

`LiftSSA(D: SSA_DOMAIN) → DOMAIN with state ≜ (var --> ssa_expr) * D.state`



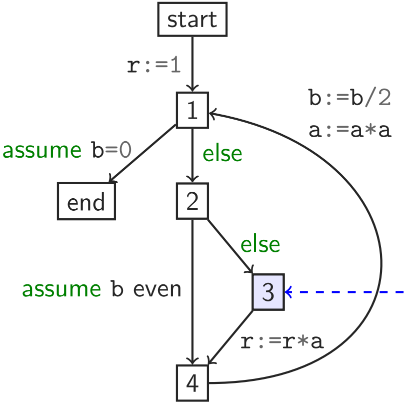
Compilation to SSA: the LiftSSA functor

`LiftSSA(D: SSA_DOMAIN)` $\rightarrow$ `DOMAIN with state` $\triangleq$ `(var --> ssa_expr) * D.state`



Compilation to SSA: the LiftSSA functor

`LiftSSA(D: SSA_DOMAIN)` $\rightarrow$ `DOMAIN with state` $\triangleq$ `(var --> ssa_expr) * D.state`



r $\mapsto$ r0
a $\mapsto$ a0 ; D.entrypoint
b $\mapsto$ b0

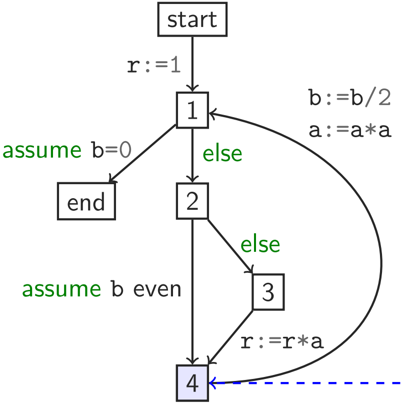
r $\mapsto$ 1
a $\mapsto$ a0 ; D.entrypoint
b $\mapsto$ b0

r $\mapsto$ 1
a $\mapsto$ a0 ; D.assume (b0$\neq$0) D.entrypoint
b $\mapsto$ b0

r $\mapsto$ 1
a $\mapsto$ a0 ; D.assume (b0 odd) (p_2 2)
b $\mapsto$ b0

Compilation to SSA: the LiftSSA functor

`LiftSSA(D: SSA_DOMAIN)` $\rightarrow$ `DOMAIN with state` $\triangleq$ `(var --> ssa_expr) * D.state`



r ↦ r0
a ↦ a0 ; D.entrypoint
b ↦ b0

r ↦ 1
a ↦ a0 ; D.entrypoint
b ↦ b0

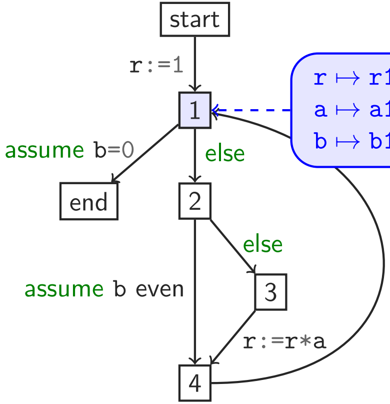
r ↦ 1
a ↦ a0 ; D.assume (b0≠0) D.entrypoint
b ↦ b0

r ↦ 1
a ↦ a0 ; D.assume (b0 odd) (p_2 2)
b ↦ b0

r ↦ r4
a ↦ a0 ; D.join (r4 ↦ 1; p_2 2) (r4 ↦ 1*a0; p_2 3)
b ↦ b0

Compilation to SSA: the LiftSSA functor

`LiftSSA(D: SSA_DOMAIN)` $\rightarrow$ `DOMAIN with state` $\triangleq$ `(var --> ssa_expr) * D.state`



r $\mapsto$ r0
a $\mapsto$ a0 ; D.entrypoint
b $\mapsto$ b0

r $\mapsto$ r1
a $\mapsto$ a1 ; D.join $\begin{pmatrix} \text{r1} \mapsto 1 \\ \text{a1} \mapsto \text{a0} \; ; p_2 \text{ start} \\ \text{b1} \mapsto \text{b0} \end{pmatrix}$ $\begin{pmatrix} \text{r1} \mapsto \text{r4} \\ \text{a1} \mapsto \text{a0} * \text{a0} \; ; p_2 \; 4 \\ \text{b1} \mapsto \text{b0}/2 \end{pmatrix}$
b $\mapsto$ b1

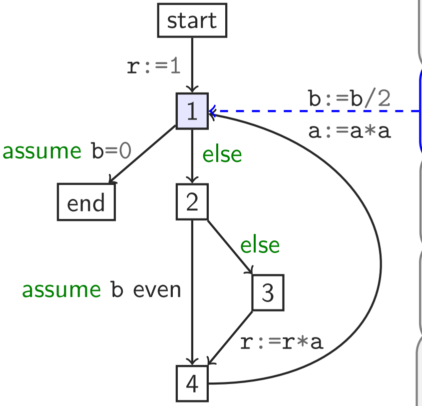
r $\mapsto$ 1
a $\mapsto$ a0 ; D.assume (b0$\neq$0) D.entrypoint
b $\mapsto$ b0

r $\mapsto$ 1
a $\mapsto$ a0 ; D.assume (b0 odd) (p_2 2)
b $\mapsto$ b0

r $\mapsto$ r4
a $\mapsto$ a0 ; D.join (r4 $\mapsto$ 1; p_2 2) (r4 $\mapsto$ 1*a0; p_2 3)
b $\mapsto$ b0

Compilation to SSA: the LiftSSA functor

`LiftSSA(D: SSA_DOMAIN)` $\rightarrow$ `DOMAIN with state` $\triangleq$ `(var --> ssa_expr) * D.state`



r ↦ r0
a ↦ a0 ; D.entrypoint
b ↦ b0

r ↦ r1
a ↦ a1 ; D.widen 1 (p_2 1) (D.join ⋯)
b ↦ b1

r ↦ 1
a ↦ a0 ; D.assume (b0≠0) D.entrypoint
b ↦ b0

r ↦ 1
a ↦ a0 ; D.assume (b0 odd) (p_2 2)
b ↦ b0

r ↦ r4
a ↦ a0 ; D.join (r4 ↦ 1; p_2 2) (r4 ↦ 1*a0; p_2 3)
b ↦ b0

Compilation to SSA: the LiftSSA functor

Compilation to SSA is done via a functor:

- `LiftSSA(D).state` $\triangleq$ `(var --> ssa_expr) * D.state`

Compilation to SSA: the LiftSSA functor

Compilation to SSA is done via a functor:

- `LiftSSA(D).state` $\triangleq$ `(var --> ssa_expr) * D.state`
- `LiftSSA(D).apply(x:=e, (store, state))` $\triangleq$
  `(store[x -> subst store e], state)`

Compilation to SSA: the LiftSSA functor

Compilation to SSA is done via a functor:

- `LiftSSA(D).state` ≜ `(var --> ssa_expr) * D.state`
- `LiftSSA(D).apply(x:=e, (store, state))` ≜
  `(store[x -> subst store e], state)`
- `LiftSSA(D).apply(assume e, (store, state))` ≜
  `(store, D.assume (subst store e) state)`

Compilation to SSA: the LiftSSA functor

Compilation to SSA is done via a functor:

- `LiftSSA(D).state` ≜ `(var --> ssa_expr) * D.state`
- `LiftSSA(D).apply(x:=e, (store, state))` ≜
  `(store[x -> subst store e], state)`
- `LiftSSA(D).apply(assume e, (store, state))` ≜
  `(store, D.assume (subst store e) state)`
- `LiftSSA(D).join` creates new ϕ variables when stores disagree

Results on the LiftSSA functor

Theorem

`LiftSSA` is sound and complete.

Theorem

There is a forward and backward simulation between source code and the code generated by `LiftSSA`(SSA free algebra).

SSA Non-relational analysis

SSA immutability allows storing information about value of expressions.

Bytecode:

```
x := ...;
y := x*x;
z := y+1;
c := x != 3;
d := 1 < y;
e := y <= 25;
f := c&&d&&e;
assume f;
```

SSA Non-relational analysis

SSA immutability allows storing information about value of expressions.

Bytecode:

```
x := ...;
y := x*x;
z := y+1;
c := x != 3;
d := 1 < y;
e := y <= 25;
f := c&&d&&e;
assume f;
```

Store: inlines variables

$\mathtt{x} \mapsto x0$

$\mathtt{y} \mapsto x0 \times x0$

$\mathtt{z} \mapsto x0 \times x0 + 1$

$\mathtt{c} \mapsto x0 \neq 3$

$\mathtt{d} \mapsto 1 < x0 \times x0$

$\mathtt{e} \mapsto x0 \times x0 \leqslant 25$

$\mathtt{f} \mapsto x0 \neq 3 \wedge 1 < x0 \times x0 \wedge x0 \times x0 \leqslant 25$

SSA Non-relational analysis

SSA immutability allows storing information about value of expressions.

Bytecode:

```
x := ...;
y := x*x;
z := y+1;
c := x != 3;
d := 1 < y;
e := y <= 25;
f := c&&d&&e;
assume f;
```

Store: inlines variables

$\mathtt{x} \mapsto x0$

$\mathtt{y} \mapsto x0 \times x0$

$\mathtt{z} \mapsto x0 \times x0 + 1$

$\mathtt{c} \mapsto x0 \neq 3$

$\mathtt{d} \mapsto 1 < x0 \times x0$

$\mathtt{e} \mapsto x0 \times x0 \leqslant 25$

$\mathtt{f} \mapsto x0 \neq 3 \wedge 1 < x0 \times x0 \wedge x0 \times x0 \leqslant 25$

Numeric state:

$\mathrm{x0} \in [-5 : 5]$

$\mathrm{x0} \times x0 \in [2 : 25]$

$\mathrm{x0} \times x0 + 1 \in [3 : 26]$

$\mathrm{x0} \neq 3 \in \{1\}$

SSA Non-relational analysis

SSA immutability allows storing information about value of expressions.

Bytecode:

```
x := ...;
y := x*x;
z := y+1;
c := x != 3;
d := 1 < y;
e := y <= 25;
f := c&&d&&e;
assume f;
```

Store: inlines variables

$\mathtt{x} \mapsto x0$

$\mathtt{y} \mapsto x0 \times x0$

$\mathtt{z} \mapsto x0 \times x0 + 1$

$\mathtt{c} \mapsto x0 \neq 3$

$\mathtt{d} \mapsto 1 < x0 \times x0$

$\mathtt{e} \mapsto x0 \times x0 \leqslant 25$

$\mathtt{f} \mapsto x0 \neq 3 \wedge 1 < x0 \times x0 \wedge x0 \times x0 \leqslant 25$

Numeric state:

$\mathtt{x0} \in [-5 : 5]$

$\mathtt{x0} \times x0 \in [2 : 25]$

$\mathtt{x0} \times x0 + 1 \in [3 : 26]$

$\mathtt{x0} \neq 3 \in \{1\}$

Theorem

`LiftSSA(Num)` can analyze bytecode with the same precision as source.

Examples of precision gains

`LiftSSA(Num)` improves precision in a number of cases:

- propagate across statements: `c = y < 0; if (c) ...`
- learn from related variables:
  `y = x+1; z = y*y; if(2 <= y <= 5) ...`
- increase precision of the numeric abstraction:
  `if (x != 0) assert(x != 0)`
- can also perform global value numbering

4. Evaluation

Experiments

RQ1. Precision increase of the SSA non-relational domain?
RQ2. Performance cost of LiftSSA?
RQ3. Performance cost of free algebra domain?

RQ1: Experimental results

Experiment	SSA=Std	SSA$\sqsupset$Std	SSA$\sqsubset$Std	Incomp.
All states, all variables (sum)	2M	10k	0	0
All states, all variables (avg)	12k	52	0	0
All states, successors (sum)	16k	656	0	0
All states, successors (avg)	83	3	0	0

Table: Comparison of SSA and standard non-relational domains precision

Result: no precision loss, gains in 1-10% of cases.

RQ2&3: Experimental results

File	LOC	N	LiftSSA(N)	N×FA	FA	LiftSSA(FA)	LiftSSA(FA×N)
c00.c	237	57	130 (2.3)	66 (1.16)	6 (0.11)	130 (2.3)	136 (2.39)
c02.c	393	87	86 (0.99)	103 (1.18)	17 (0.2)	334 (3.82)	81 (0.93)
c04.c	304	13	39 (3.09)	11 (0.9)	3 (0.25)	45 (3.54)	40 (3.15)
c07.c	397	12	25 (2.09)	12 (1.05)	9 (0.8)	131 (11.1)	27 (2.28)
c18.c	292	84	193 (2.3)	93 (1.11)	8 (0.1)	234 (2.79)	180 (2.15)
c23.c	3174	50	348 (7.02)	52 (1.05)	90 (1.82)	20.7s (418)	346 (6.98)
c24.c	11076	6.2s	20.4s (3.3)	5.3s (0.86)	2s (0.33)	>10min	18.6s (3.01)
c29.c	2347	140	276 (1.98)	119 (0.85)	99 (0.71)	15.1s (108)	588 (4.21)
c30.c	1178	200	355 (1.77)	189 (0.95)	70 (0.35)	8.8s (44.2)	1361 (6.8)

Table: Execution times in milliseconds

Conclusion

Contributions: our method allows us to:

- Generate imperative/SSA programs as abstract interpretation results (free algebra domain)
- Implement and prove compilation passes using abstract interpretation (functors)
- Improve non-relational precision at a constant overhead (`LiftSSA(Num)`)
- Implemented as part of the Codex library (`https://codex.top`)

Limits:

- Focused on forwards analyses, not on backwards ones
- Compilation functors from the CFG signature are local to statements

Going further

https://codex.top/papers/2024-pldi-compiling-with-abstract-interpretation

paper: 10.1145/3656392

appendices: https://hal.science/hal-04535159

artifact: 10.5281/zenodo.10895582

contact: dorian.lesbre@cea.fr and matthieu.lemerre@cea.fr