# Guarded Kleene Algebra with Tests

## Verification of Uninterpreted Programs in Nearly Linear Time

Steffen Smolka[1] Nate Foster[1] Justin Hsu[2]
*Tobias Kappé*[3] Dexter Kozen[1] Alexandra Silva[3]

[1]Cornell University

[2]University of Wisconsin-Madison

[3]University College London

POPL 2020

# Introduction

```
while a and b do
    e;
end
while a do
    f;
    while a and b do
        e;
    end
end
```

- Uninterpreted programs can be thought of as skeletons of programs.
- The structure of the program is there, but not the concrete actions.
- This allows reasoning about refactoring, optimisation, et cetera.

# Introduction

```
while a and b do
  e;
end
while a do
  f;
  while a and b do
    e;
  end
end
```

```
while a do
  if b then
    e;
  else
    f;
  end
end
```

- Uninterpreted programs can be thought of as skeletons of programs.
- The structure of the program is there, but not the concrete actions.
- This allows reasoning about refactoring, optimisation, et cetera.

# Introduction

```
while a and b do
    e;
end
while a do
    f;
    while a and b do
        e;
    end
end
```

$\stackrel{?}{\equiv}$

```
while a do
    if b then
        e;
    else
        f;
    end
end
```

- Uninterpreted programs can be thought of as skeletons of programs.
- The structure of the program is there, but not the concrete actions.
- This allows reasoning about refactoring, optimisation, et cetera.

# Introduction

Contributions:

- Nearly linear time decision procedure for equivalence.[1]

[1] For fixed number of tests.

# Introduction

Contributions:

- Nearly linear time decision procedure for equivalence.[1]
- Axiomatization of uninterpreted program equivalence.

[1] For fixed number of tests.

## Introduction

Contributions:

- Nearly linear time decision procedure for equivalence.[1]
- Axiomatization of uninterpreted program equivalence.
- Kleene theorem for uninterpreted programs.

[1] For fixed number of tests.

## Syntax and semantics

$$\mathbf{a}, \mathbf{b} ::= t \in T \mid \mathbf{a} + \mathbf{b} \mid \mathbf{ab} \mid \overline{\mathbf{a}} \mid 0 \mid 1$$

$$\mathbf{e}, \mathbf{f} ::= \mathbf{a} \mid p \in \Sigma \mid \mathbf{ef} \mid \mathbf{e} +_{\mathbf{a}} \mathbf{f} \mid \mathbf{e}^{(\mathbf{a})}$$



- We will use compact syntax to denote uninterpreted programs.
- Note: overloading conjunction and concatenation.

# Syntax and semantics

$$\mathsf{a}, \mathsf{b} ::= t \in T \mid \mathsf{a} + \mathsf{b} \mid \mathsf{ab} \mid \bar{\mathsf{a}} \mid 0 \mid 1$$

a or b

$$\mathsf{e}, \mathsf{f} ::= \mathsf{a} \mid p \in \Sigma \mid \mathsf{ef} \mid \mathsf{e} +_{\mathsf{a}} \mathsf{f} \mid \mathsf{e}^{(\mathsf{a})}$$



- We will use compact syntax to denote uninterpreted programs.
- Note: overloading conjunction and concatenation.

## Syntax and semantics

$$\mathbf{a}, \mathbf{b} ::= t \in T \mid \mathbf{a} + \mathbf{b} \mid \mathbf{ab} \mid \bar{\mathbf{a}} \mid 0 \mid 1$$

a and b

$$\mathbf{e}, \mathbf{f} ::= \mathbf{a} \mid p \in \Sigma \mid \mathbf{ef} \mid \mathbf{e} +_{\mathbf{a}} \mathbf{f} \mid \mathbf{e}^{(\mathbf{a})}$$



- We will use compact syntax to denote uninterpreted programs.
- Note: overloading conjunction and concatenation.

# Syntax and semantics

$$\mathbf{a}, \mathbf{b} ::= t \in T \mid \mathbf{a} + \mathbf{b} \mid \mathbf{ab} \mid \overline{\mathbf{a}} \mid 0 \mid 1$$

`not` **a**

$$\mathbf{e}, \mathbf{f} ::= \mathbf{a} \mid p \in \Sigma \mid \mathbf{ef} \mid \mathbf{e} +_{\mathbf{a}} \mathbf{f} \mid \mathbf{e}^{(\mathbf{a})}$$



- We will use compact syntax to denote uninterpreted programs.
- Note: overloading conjunction and concatenation.

# Syntax and semantics

$$\mathbf{a}, \mathbf{b} ::= t \in T \mid \mathbf{a} + \mathbf{b} \mid \mathbf{ab} \mid \overline{\mathbf{a}} \mid 0 \mid 1$$

`false`

$$\mathbf{e}, \mathbf{f} ::= \mathbf{a} \mid p \in \Sigma \mid \mathbf{ef} \mid \mathbf{e} +_{\mathbf{a}} \mathbf{f} \mid \mathbf{e}^{(\mathbf{a})}$$



- We will use compact syntax to denote uninterpreted programs.
- Note: overloading conjunction and concatenation.

## Syntax and semantics

$$\mathbf{a}, \mathbf{b} ::= t \in T \mid \mathbf{a} + \mathbf{b} \mid \mathbf{ab} \mid \bar{\mathbf{a}} \mid 0 \mid 1$$

`true`

$$\mathbf{e}, \mathbf{f} ::= \mathbf{a} \mid p \in \Sigma \mid \mathbf{ef} \mid \mathbf{e} +_{\mathbf{a}} \mathbf{f} \mid \mathbf{e}^{(\mathbf{a})}$$



- We will use compact syntax to denote uninterpreted programs.
- Note: overloading conjunction and concatenation.

## Syntax and semantics

$$\mathbf{a}, \mathbf{b} ::= t \in T \mid \mathbf{a} + \mathbf{b} \mid \mathbf{ab} \mid \overline{\mathbf{a}} \mid 0 \mid 1$$

$$\mathbf{e}, \mathbf{f} ::= \mathbf{a} \mid p \in \Sigma \mid \mathbf{ef} \mid \mathbf{e} +_{\mathbf{a}} \mathbf{f} \mid \mathbf{e}^{(\mathbf{a})}$$

`assert` **a**



- We will use compact syntax to denote uninterpreted programs.
- Note: overloading conjunction and concatenation.

## Syntax and semantics

$$\mathbf{a}, \mathbf{b} ::= t \in T \mid \mathbf{a} + \mathbf{b} \mid \mathbf{ab} \mid \overline{\mathbf{a}} \mid 0 \mid 1$$

$$\mathbf{e}, \mathbf{f} ::= \mathbf{a} \mid p \in \Sigma \mid \mathbf{ef} \mid \mathbf{e} +_{\mathbf{a}} \mathbf{f} \mid \mathbf{e}^{(\mathbf{a})}$$

$\mathbf{e}; \mathbf{f}$



- We will use compact syntax to denote uninterpreted programs.
- Note: overloading conjunction and concatenation.

## Syntax and semantics

$$\mathbf{a}, \mathbf{b} ::= t \in T \mid \mathbf{a} + \mathbf{b} \mid \mathbf{ab} \mid \overline{\mathbf{a}} \mid 0 \mid 1$$

$$\mathbf{e}, \mathbf{f} ::= \mathbf{a} \mid p \in \Sigma \mid \mathbf{ef} \mid \mathbf{e} +_{\mathbf{a}} \mathbf{f} \mid \mathbf{e}^{(\mathbf{a})}$$

`if` **a** `then` **e** `else` **f**



- We will use compact syntax to denote uninterpreted programs.
- Note: overloading conjunction and concatenation.

## Syntax and semantics

$$\mathbf{a}, \mathbf{b} ::= t \in T \mid \mathbf{a} + \mathbf{b} \mid \mathbf{ab} \mid \bar{\mathbf{a}} \mid 0 \mid 1$$

$$\mathbf{e}, \mathbf{f} ::= \mathbf{a} \mid p \in \Sigma \mid \mathbf{ef} \mid \mathbf{e} +_{\mathbf{a}} \mathbf{f} \mid \mathbf{e}^{(\mathbf{a})}$$

`while` **a** `do` **e**



- We will use compact syntax to denote uninterpreted programs.
- Note: overloading conjunction and concatenation.

## Syntax and semantics

```
while a do
  if b then
    e;
  else
    f;
  end
end
```

$\Downarrow$

$(e +_b f)^{(a)}$

```
while a and b do
  e;
end
while a do
  f;
  while a and b do
    e;
  end
end
```

$\Downarrow$

$e^{(ab)}(fe^{(ab)})^{(a)}$



- The programs from before can now be written down like this.

## Syntax and semantics

$$i = \left(sat : T \to 2^{States}, eval : \Sigma \to 2^{States^2}\right)$$



- We can instantiate tests and actions to obtain a relational semantics.
- We can use sub-Markov kernels to give a probabilistic semantics.
- Equivalence means semantics are the same regardless of interpretation.

## Syntax and semantics

$$i = \left( sat : T \to 2^{States}, eval : \Sigma \to 2^{States^2} \right)$$

| $e$ | $\mathcal{R}_i[\![e]\!]$ |
|---|---|
| $t \in T$ | $\{(s,s) : s \in sat(t)\}$ |
| $\mathbf{a} + \mathbf{b}$ | $\mathcal{R}_i[\![\mathbf{a}]\!] \cup \mathcal{R}_i[\![\mathbf{b}]\!]$ |
| $\mathbf{ab}$ | $\mathcal{R}_i[\![\mathbf{a}]\!] \cap \mathcal{R}_i[\![\mathbf{b}]\!]$ |
| $\overline{\mathbf{a}}$ | $States^2 \setminus \mathcal{R}_i[\![\mathbf{a}]\!]$ |
| $p \in \Sigma$ | $eval(p)$ |
| $\mathbf{e} +_{\mathbf{a}} \mathbf{f}$ | $\mathcal{R}_i[\![\mathbf{a}]\!] \circ \mathcal{R}_i[\![\mathbf{e}]\!] \cup \mathcal{R}_i[\![\overline{\mathbf{a}}]\!] \circ \mathcal{R}_i[\![\mathbf{f}]\!]$ |
| $\mathbf{ef}$ | $\mathcal{R}_i[\![\mathbf{e}]\!] \circ \mathcal{R}_i[\![\mathbf{f}]\!]$ |
| $\mathbf{e}^{(\mathbf{a})}$ | $(\mathcal{R}_i[\![\mathbf{a}]\!] \circ \mathcal{R}_i[\![\mathbf{e}]\!])^* \circ \mathcal{R}_i[\![\overline{\mathbf{a}}]\!]$ |



- We can instantiate tests and actions to obtain a relational semantics.
- We can use sub-Markov kernels to give a probabilistic semantics.
- Equivalence means semantics are the same regardless of interpretation.

# Syntax and semantics

$$Atoms = 2^T$$



- Parameterized semantics is intuitive, but not very easy to handle.
- We can abstract from the interpretation by giving a language semantics.
- The idea behind this semantics is that it gives all possible traces.
- A trace of a program consists of states interleaved with actions.
- Such traces are represented by guarded strings, defined as follows.
- Sets (languages) of guarded strings can be equipped with operators.

# Syntax and semantics

$$Atoms = 2^T$$

$$\frac{\alpha \in Atoms}{\alpha \in \mathrm{GS}(\Sigma, T)} \qquad \frac{\alpha, \beta \in Atoms \qquad p \in \Sigma}{\alpha p \beta \in \mathrm{GS}(\Sigma, T)} \qquad \frac{w\alpha, \alpha x \in \mathrm{GS}(\Sigma, T)}{w \alpha x \in \mathrm{GS}(\Sigma, T)}$$



- Parameterized semantics is intuitive, but not very easy to handle.
- We can abstract from the interpretation by giving a language semantics.
- The idea behind this semantics is that it gives all possible traces.
- A trace of a program consists of states interleaved with actions.
- Such traces are represented by guarded strings, defined as follows.
- Sets (languages) of guarded strings can be equipped with operators.

## Syntax and semantics

$$Atoms = 2^T$$

$$\frac{\alpha \in Atoms}{\alpha \in \mathrm{GS}(\Sigma, T)} \qquad \frac{\alpha, \beta \in Atoms \qquad p \in \Sigma}{\alpha p \beta \in \mathrm{GS}(\Sigma, T)} \qquad \frac{w\alpha, \alpha x \in \mathrm{GS}(\Sigma, T)}{w\alpha x \in \mathrm{GS}(\Sigma, T)}$$

$$L \diamond K = \{w\alpha x : w\alpha \in L, \alpha x \in K\} \qquad L^{(n)} = \underbrace{L \diamond \cdots \diamond L}_{n \text{ times}} \qquad L^{(*)} = \bigcup_{n \in \mathbb{N}} L^{(n)}$$



- Parameterized semantics is intuitive, but not very easy to handle.
- We can abstract from the interpretation by giving a language semantics.
- The idea behind this semantics is that it gives all possible traces.
- A trace of a program consists of states interleaved with actions.
- Such traces are represented by guarded strings, defined as follows.
- Sets (languages) of guarded strings can be equipped with operators.

# Syntax and semantics

| $e$ | $[\![e]\!]$ |
|---|---|
| $t \in T$ | $\{\alpha \in \mathit{Atoms} : t \in \alpha\}$ |
| $\mathbf{a} + \mathbf{b}$ | $[\![\mathbf{a}]\!] \cup [\![\mathbf{b}]\!]$ |
| $\mathbf{ab}$ | $[\![\mathbf{a}]\!] \cap [\![\mathbf{b}]\!]$ |
| $\bar{\mathbf{a}}$ | $\mathit{Atoms} \setminus [\![\mathbf{a}]\!]$ |
| $p \in \Sigma$ | $\{\alpha p \beta : \alpha, \beta \in \mathit{Atoms}\}$ |
| $\mathbf{e} +_{\mathbf{a}} \mathbf{f}$ | $[\![\mathbf{a}]\!] \diamond [\![\mathbf{e}]\!] \cup [\![\bar{\mathbf{a}}]\!] \diamond [\![\mathbf{f}]\!]$ |
| $\mathbf{ef}$ | $[\![\mathbf{e}]\!] \diamond [\![\mathbf{f}]\!]$ |
| $\mathbf{e}^{(\mathbf{a})}$ | $([\![\mathbf{a}]\!] \diamond [\![\mathbf{e}]\!])^{(*)} \diamond [\![\bar{\mathbf{a}}]\!]$ |



- Semantics in terms of guarded strings given as follows.
- Semantics of a test is set of atoms (states) satisfying that test.
- Semantics of an action is an overapproximation — meaning is unknown.
- Inductive cases are as for the relational semantics.
- For example, trace of sequencing finds matching traces and fuses them.

# Syntax and semantics

**Theorem**

$$[\![\mathbf{e}]\!] = [\![\mathbf{f}]\!] \iff \forall i.\ \mathcal{R}_i[\![\mathbf{e}]\!] = \mathcal{R}_i[\![\mathbf{f}]\!]$$



- Parameterized interpretations are related to interpretation in guarded strings.
- We can check equivalence for all interpretations by comparing languages.
- Spoiler: implement languages in automata, compare those automata.
- Note: the conversion from expressions to automata is half a Kleene theorem.
- Complexity of procedure is nearly linear in size of automata.

# Syntax and semantics

**Theorem**

$$[\![\mathbf{e}]\!] = [\![\mathbf{f}]\!] \iff \forall i.\ \mathcal{R}_i[\![\mathbf{e}]\!] = \mathcal{R}_i[\![\mathbf{f}]\!]$$

How to check $[\![\mathbf{e}]\!] = [\![\mathbf{f}]\!]$:

1. Create automata that accept $[\![\mathbf{e}]\!]$ and $[\![\mathbf{f}]\!]$.
2. Check automata for bisimilarity.



- Parameterized interpretations are related to interpretation in guarded strings.
- We can check equivalence for all interpretations by comparing languages.
- Spoiler: implement languages in automata, compare those automata.
- Note: the conversion from expressions to automata is half a Kleene theorem.
- Complexity of procedure is nearly linear in size of automata.

# Axiomatization

$$e +_a e \equiv e$$



- Branching between identical pieces of code can be eliminated.
- Branches can be flipped by negating the condition.
- The guard of a branch holds before the branch starts.
- Contradictory assertions are like asserting false.
- Asserting false means the rest of the code is not executed.
- With these axioms we can already derive some useful things.

# Axiomatization

$$e +_a e \equiv e \qquad e +_a f \equiv f +_{\bar{a}} e$$



- Branching between identical pieces of code can be eliminated.
- Branches can be flipped by negating the condition.
- The guard of a branch holds before the branch starts.
- Contradictory assertions are like asserting false.
- Asserting false means the rest of the code is not executed.
- With these axioms we can already derive some useful things.

## Axiomatization

$$e +_a e \equiv e \qquad e +_a f \equiv f +_{\bar{a}} e \qquad e +_a f \equiv ae +_a f$$

- Branching between identical pieces of code can be eliminated.
- Branches can be flipped by negating the condition.
- The guard of a branch holds before the branch starts.
- Contradictory assertions are like asserting false.
- Asserting false means the rest of the code is not executed.
- With these axioms we can already derive some useful things.

# Axiomatization

$$e +_a e \equiv e \qquad e +_a f \equiv f +_{\bar{a}} e \qquad e +_a f \equiv ae +_a f \qquad \bar{a}a \equiv 0$$



- Branching between identical pieces of code can be eliminated.
- Branches can be flipped by negating the condition.
- The guard of a branch holds before the branch starts.
- Contradictory assertions are like asserting false.
- Asserting false means the rest of the code is not executed.
- With these axioms we can already derive some useful things.

# Axiomatization

$$e +_a e \equiv e \qquad e +_a f \equiv f +_{\bar{a}} e \qquad e +_a f \equiv ae +_a f \qquad \bar{a}a \equiv 0 \qquad 0e \equiv 0$$



- Branching between identical pieces of code can be eliminated.
- Branches can be flipped by negating the condition.
- The guard of a branch holds before the branch starts.
- Contradictory assertions are like asserting false.
- Asserting false means the rest of the code is not executed.
- With these axioms we can already derive some useful things.

# Axiomatization

$$e +_a e \equiv e \qquad e +_a f \equiv f +_{\bar{a}} e \qquad e +_a f \equiv ae +_a f \qquad \bar{a}a \equiv 0 \qquad 0e \equiv 0$$

## Example

`if` $a$ `then` $e$ `else assert false` $= e +_a 0$

- Branching between identical pieces of code can be eliminated.
- Branches can be flipped by negating the condition.
- The guard of a branch holds before the branch starts.
- Contradictory assertions are like asserting false.
- Asserting false means the rest of the code is not executed.
- With these axioms we can already derive some useful things.

# Axiomatization

$$e +_a e \equiv e \qquad e +_a f \equiv f +_{\bar{a}} e \qquad \boxed{e +_a f \equiv ae +_a f} \qquad \bar{a}a \equiv 0 \qquad 0e \equiv 0$$

## Example

`if a then e else assert false` $= e +_a 0 \equiv ae +_a 0$



- Branching between identical pieces of code can be eliminated.
- Branches can be flipped by negating the condition.
- The guard of a branch holds before the branch starts.
- Contradictory assertions are like asserting false.
- Asserting false means the rest of the code is not executed.
- With these axioms we can already derive some useful things.

# Axiomatization

$\mathsf{e} +_{\mathsf{a}} \mathsf{e} \equiv \mathsf{e}$ $\qquad$ $\boxed{\mathsf{e} +_{\mathsf{a}} \mathsf{f} \equiv \mathsf{f} +_{\bar{\mathsf{a}}} \mathsf{e}}$ $\qquad$ $\mathsf{e} +_{\mathsf{a}} \mathsf{f} \equiv \mathsf{ae} +_{\mathsf{a}} \mathsf{f}$ $\qquad$ $\bar{\mathsf{a}}\mathsf{a} \equiv 0$ $\qquad$ $0\mathsf{e} \equiv 0$

## Example

$$\texttt{if } \mathsf{a} \texttt{ then } \mathsf{e} \texttt{ else assert false} = \mathsf{e} +_{\mathsf{a}} 0 \equiv \mathsf{ae} +_{\mathsf{a}} 0$$
$$\equiv 0 +_{\bar{\mathsf{a}}} \mathsf{ae}$$



- Branching between identical pieces of code can be eliminated.
- Branches can be flipped by negating the condition.
- The guard of a branch holds before the branch starts.
- Contradictory assertions are like asserting false.
- Asserting false means the rest of the code is not executed.
- With these axioms we can already derive some useful things.

## Axiomatization

$$e +_a e \equiv e \qquad e +_a f \equiv f +_{\bar{a}} e \qquad e +_a f \equiv ae +_a f \qquad \bar{a}a \equiv 0 \qquad \boxed{0e \equiv 0}$$

### Example

$$\begin{aligned} \texttt{if } a \texttt{ then } e \texttt{ else assert false} = e +_a 0 &\equiv ae +_a 0 \\ &\equiv 0 +_{\bar{a}} ae \\ &\equiv 0e +_{\bar{a}} ae \end{aligned}$$



- Branching between identical pieces of code can be eliminated.
- Branches can be flipped by negating the condition.
- The guard of a branch holds before the branch starts.
- Contradictory assertions are like asserting false.
- Asserting false means the rest of the code is not executed.
- With these axioms we can already derive some useful things.

# Axiomatization

$$e +_a e \equiv e \qquad e +_a f \equiv f +_{\bar{a}} e \qquad e +_a f \equiv ae +_a f \qquad \boxed{\bar{a}a \equiv 0} \qquad 0e \equiv 0$$

## Example

$$\begin{aligned}
\texttt{if } a \texttt{ then } e \texttt{ else assert false} = e +_a 0 &\equiv ae +_a 0 \\
&\equiv 0 +_{\bar{a}} ae \\
&\equiv 0e +_{\bar{a}} ae \\
&\equiv \bar{a}ae +_{\bar{a}} ae
\end{aligned}$$



- Branching between identical pieces of code can be eliminated.
- Branches can be flipped by negating the condition.
- The guard of a branch holds before the branch starts.
- Contradictory assertions are like asserting false.
- Asserting false means the rest of the code is not executed.
- With these axioms we can already derive some useful things.

# Axiomatization

$$e +_a e \equiv e \qquad e +_a f \equiv f +_{\bar{a}} e \qquad \boxed{e +_a f \equiv ae +_a f} \qquad \bar{a}a \equiv 0 \qquad 0e \equiv 0$$

## Example

$$\begin{aligned}
\texttt{if } a \texttt{ then } e \texttt{ else assert false} = e +_a 0 &\equiv ae +_a 0 \\
&\equiv 0 +_{\bar{a}} ae \\
&\equiv 0e +_{\bar{a}} ae \\
&\equiv \bar{a}ae +_{\bar{a}} ae \\
&\equiv ae +_{\bar{a}} ae
\end{aligned}$$



- Branching between identical pieces of code can be eliminated.
- Branches can be flipped by negating the condition.
- The guard of a branch holds before the branch starts.
- Contradictory assertions are like asserting false.
- Asserting false means the rest of the code is not executed.
- With these axioms we can already derive some useful things.

## Axiomatization

$\boxed{e +_a e \equiv e}$ $\qquad e +_a f \equiv f +_{\bar{a}} e \qquad e +_a f \equiv ae +_a f \qquad \bar{a}a \equiv 0 \qquad 0e \equiv 0$

### Example

$$
\begin{aligned}
\texttt{if } a \texttt{ then } e \texttt{ else assert false} = e +_a 0 &\equiv ae +_a 0 \\
&\equiv 0 +_{\bar{a}} ae \\
&\equiv 0e +_{\bar{a}} ae \\
&\equiv \bar{a}ae +_{\bar{a}} ae \\
&\equiv ae +_{\bar{a}} ae \\
&\equiv ae \qquad\qquad = \texttt{assert } a; e
\end{aligned}
$$



- Branching between identical pieces of code can be eliminated.
- Branches can be flipped by negating the condition.
- The guard of a branch holds before the branch starts.
- Contradictory assertions are like asserting false.
- Asserting false means the rest of the code is not executed.
- With these axioms we can already derive some useful things.

## Axiomatization

$$\frac{e \equiv fe +_a g}{e \equiv f^{(a)} g}$$

- First intuition for loop axioms is to characterise it as a fixpoint.
- Need to be careful, otherwise we can prove nonsense.

# Axiomatization

$$\frac{e \equiv fe +_{a} g}{e \equiv f^{(a)} g}$$

Allows to derive $1 \equiv 1^{(1)}$, i.e.,

$\texttt{while true do assert true} \equiv \texttt{assert true}$



- First intuition for loop axioms is to characterise it as a fixpoint.
- Need to be careful, otherwise we can prove nonsense.

# Axiomatization

$$\frac{e \equiv fe +_{a} g \qquad f \text{ is productive}}{e \equiv f^{(a)}g}$$



- Instead we need to put a side-condition on the loop body; see paper for details.
- Loops are themselves a fixpoint, and skips inside loops can be eliminated.
- We can make the body of any loop productive.
- With these axioms, we can now prove useful things about loops.

# Axiomatization

$$\frac{e \equiv fe +_a g \qquad f \text{ is productive}}{e \equiv f^{(a)} g} \qquad\qquad e^{(a)} \equiv ee^{a} +_a 1$$



- Instead we need to put a side-condition on the loop body; see paper for details.
- Loops are themselves a fixpoint, and skips inside loops can be eliminated.
- We can make the body of any loop productive.
- With these axioms, we can now prove useful things about loops.

# Axiomatization

$$\frac{e \equiv fe +_a g \qquad f \text{ is productive}}{e \equiv f^{(a)} g} \qquad e^{(a)} \equiv ee^a +_a 1 \qquad (e +_a 1)^{(b)} \equiv (ae)^{(b)}$$



- Instead we need to put a side-condition on the loop body; see paper for details.
- Loops are themselves a fixpoint, and skips inside loops can be eliminated.
- We can make the body of any loop productive.
- With these axioms, we can now prove useful things about loops.

## Axiomatization

$$\frac{e \equiv fe +_a g \qquad f \text{ is productive}}{e \equiv f^{(a)}g} \qquad e^{(a)} \equiv ee^a +_a 1 \qquad (e +_a 1)^{(b)} \equiv (ae)^{(b)}$$

**Lemma**

*For every* $e$*, there exists a productive* $\hat{e}$ *such that* $e^{(b)} \equiv \hat{e}^{(b)}$.



- Instead we need to put a side-condition on the loop body; see paper for details.
- Loops are themselves a fixpoint, and skips inside loops can be eliminated.
- We can make the body of any loop productive.
- With these axioms, we can now prove useful things about loops.

# Axiomatization

$$\frac{e \equiv fe +_a g \qquad f \text{ is productive}}{e \equiv f^{(a)}g} \qquad e^{(a)} \equiv ee^{a} +_a 1 \qquad (e +_a 1)^{(b)} \equiv (ae)^{(b)}$$

**Lemma**

*For every* $e$*, there exists a productive* $\hat{e}$ *such that* $e^{(b)} \equiv \hat{e}^{(b)}$.

**Lemma**

$$e^{(a)} \equiv e^{(a)}\overline{a} \qquad e^{(a)} \equiv (ae)^{(a)} \qquad e^{(ab)}e^{(b)} \equiv e^{(b)}$$



- Instead we need to put a side-condition on the loop body; see paper for details.
- Loops are themselves a fixpoint, and skips inside loops can be eliminated.
- We can make the body of any loop productive.
- With these axioms, we can now prove useful things about loops.

# Axiomatization

**Theorem (Soundness)**

*If* $\mathbf{e} \equiv \mathbf{f}$*, then* $[\![\mathbf{e}]\!] = [\![\mathbf{f}]\!]$.



- The axioms (minus the naive fixpoint) are sound w.r.t. the semantics.
- We need two ingredients to show the converse, i.e., completeness:
  - An automaton can be converted to an expression.
  - NB: this is the second half of a Kleene theorem.
  - The automaton of an expression yields an equivalent expression.
  - Bisimilar automata have equivalent expressions.
- This is enough to conclude completeness, as follows.
- With some more axioms and a generalized fixpoint, we also have the converse.

# Axiomatization

**Theorem (Soundness)**

*If* $\mathbf{e} \equiv \mathbf{f}$, *then* $[\![\mathbf{e}]\!] = [\![\mathbf{f}]\!]$.

How about the converse?

1. $A \mapsto S(A)$ with $\mathbf{e} \equiv S(A_{\mathbf{e}})$.
2. If $A \sim A'$, then $S(A) \equiv S(A')$.



- The axioms (minus the naive fixpoint) are sound w.r.t. the semantics.
- We need two ingredients to show the converse, i.e., completeness:
  - An automaton can be converted to an expression.
  - NB: this is the second half of a Kleene theorem.
  - The automaton of an expression yields an equivalent expression.
  - Bisimilar automata have equivalent expressions.
- This is enough to conclude completeness, as follows.
- With some more axioms and a generalized fixpoint, we also have the converse.

# Axiomatization

**Theorem (Soundness)**

*If* $\mathbf{e} \equiv \mathbf{f}$*, then* $[\![\mathbf{e}]\!] = [\![\mathbf{f}]\!]$.

How about the converse?

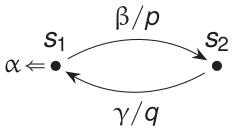
1. $A \mapsto S(A)$ with $\mathbf{e} \equiv S(A_{\mathbf{e}})$.
2. If $A \sim A'$, then $S(A) \equiv S(A')$.

$$\begin{aligned} [\![\mathbf{e}]\!] = [\![\mathbf{f}]\!] &\implies L(A_{\mathbf{e}}) = L(A_{\mathbf{f}}) \\ &\implies A_{\mathbf{e}} \sim A_{\mathbf{f}} \\ &\implies S(A_{\mathbf{e}}) \equiv S(A_{\mathbf{f}}) \\ &\implies \mathbf{e} \equiv \mathbf{f} \end{aligned}$$



- The axioms (minus the naive fixpoint) are sound w.r.t. the semantics.
- We need two ingredients to show the converse, i.e., completeness:
  - An automaton can be converted to an expression.
  - NB: this is the second half of a Kleene theorem.
  - The automaton of an expression yields an equivalent expression.
  - Bisimilar automata have equivalent expressions.
- This is enough to conclude completeness, as follows.
- With some more axioms and a generalized fixpoint, we also have the converse.

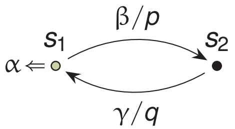
# A Kleene theorem






- A Kleene theorem is the powerhouse of our results.
- Some details about the automata and the conversion operators.
- Given an atom, either accept, reject or transition with an action.
- Word accepted is concatenation of labels, including acceptance.

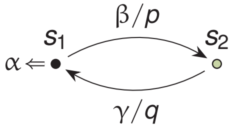
# A Kleene theorem






- A Kleene theorem is the powerhouse of our results.
- Some details about the automata and the conversion operators.
- Given an atom, either accept, reject or transition with an action.
- Word accepted is concatenation of labels, including acceptance.

# A Kleene theorem

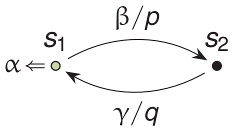



$\beta p$



- A Kleene theorem is the powerhouse of our results.
- Some details about the automata and the conversion operators.
- Given an atom, either accept, reject or transition with an action.
- Word accepted is concatenation of labels, including acceptance.

# A Kleene theorem

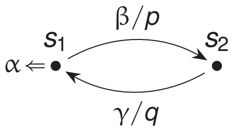



$\beta p \gamma q$



- A Kleene theorem is the powerhouse of our results.
- Some details about the automata and the conversion operators.
- Given an atom, either accept, reject or transition with an action.
- Word accepted is concatenation of labels, including acceptance.

# A Kleene theorem




$\beta p \gamma q \alpha \in \mathcal{L}(s_1)$



- A Kleene theorem is the powerhouse of our results.
- Some details about the automata and the conversion operators.
- Given an atom, either accept, reject or transition with an action.
- Word accepted is concatenation of labels, including acceptance.

# A Kleene theorem

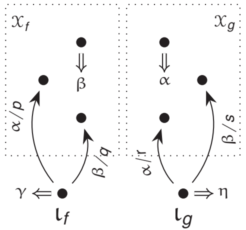
$$\begin{tikzcd} \alpha \Leftarrow s_1 \arrow[r, bend left, "\beta/p"] & s_2 \arrow[l, bend left, "\gamma/q"] \end{tikzcd} \qquad \beta p \gamma q \alpha \in \mathcal{L}(s_1)$$

$$(X, \delta : X \to (2 + \Sigma \times X)^{Atoms})$$



- A Kleene theorem is the powerhouse of our results.
- Some details about the automata and the conversion operators.
- Given an atom, either accept, reject or transition with an action.
- Word accepted is concatenation of labels, including acceptance.

# A Kleene theorem

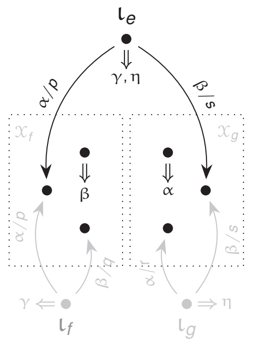



$\mathbf{e} = \mathbf{f} +_{\mathbf{a}} \mathbf{g}$



- Conversion of expression to automaton by induction on structure.
- Inductive cases are shown here.
  - For branching, we juxtapose and make a new initial state based on the guard.
  - For sequencing, we juxtapose and reroute accepting transitions on the left.
  - For loops, we reroute accepting transitions that validate the guard to the initial state.
- This translation is linear in the size of **e**.

# A Kleene theorem

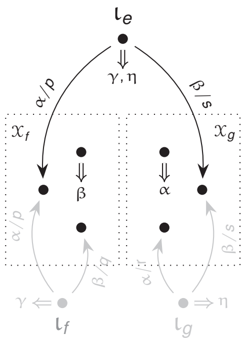



$\mathbf{e} = \mathbf{f} +_{\mathbf{a}} \mathbf{g}$

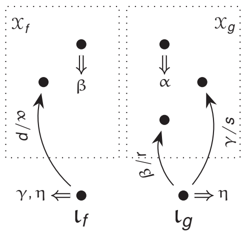


- Conversion of expression to automaton by induction on structure.
- Inductive cases are shown here.
  - For branching, we juxtapose and make a new initial state based on the guard.
  - For sequencing, we juxtapose and reroute accepting transitions on the left.
  - For loops, we reroute accepting transitions that validate the guard to the initial state.
- This translation is linear in the size of **e**.

# A Kleene theorem



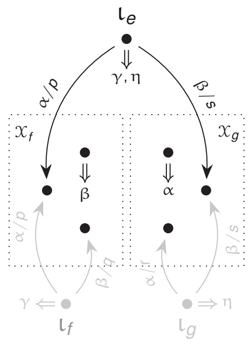
$\mathbf{e} = \mathbf{f} +_{\mathbf{a}} \mathbf{g}$



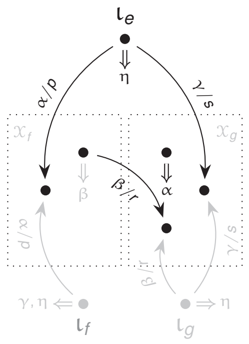
$\mathbf{e} = \mathbf{fg}$

- Conversion of expression to automaton by induction on structure.
- Inductive cases are shown here.
  - For branching, we juxtapose and make a new initial state based on the guard.
  - For sequencing, we juxtapose and reroute accepting transitions on the left.
  - For loops, we reroute accepting transitions that validate the guard to the initial state.
- This translation is linear in the size of **e**.

# A Kleene theorem



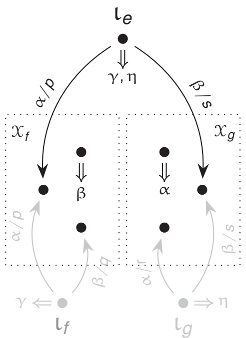
$\mathbf{e} = \mathbf{f} +_{\mathbf{a}} \mathbf{g}$



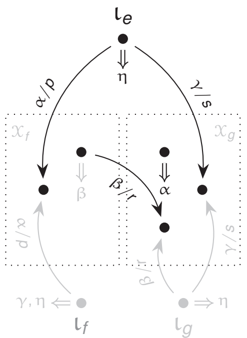
$\mathbf{e} = \mathbf{fg}$

- Conversion of expression to automaton by induction on structure.
- Inductive cases are shown here.
  - For branching, we juxtapose and make a new initial state based on the guard.
  - For sequencing, we juxtapose and reroute accepting transitions on the left.
  - For loops, we reroute accepting transitions that validate the guard to the initial state.
- This translation is linear in the size of **e**.

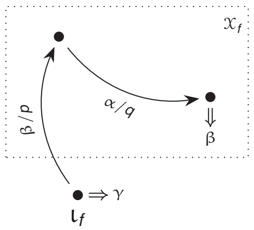
# A Kleene theorem




$\mathbf{e} = \mathbf{f} +_{\mathbf{a}} \mathbf{g}$




$\mathbf{e} = \mathbf{fg}$

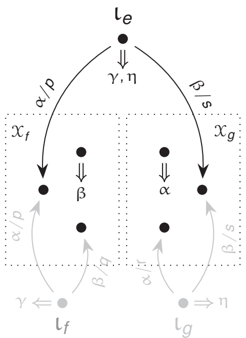



$\mathbf{e} = \mathbf{f}^{(\mathbf{a})}$

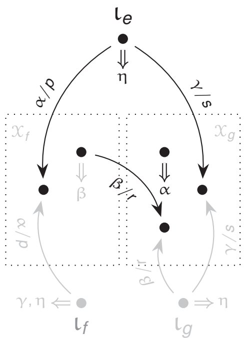


- Conversion of expression to automaton by induction on structure.
- Inductive cases are shown here.
  - For branching, we juxtapose and make a new initial state based on the guard.
  - For sequencing, we juxtapose and reroute accepting transitions on the left.
  - For loops, we reroute accepting transitions that validate the guard to the initial state.
- This translation is linear in the size of **e**.

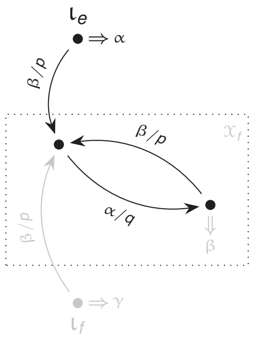
# A Kleene theorem



$e = f +_a g$



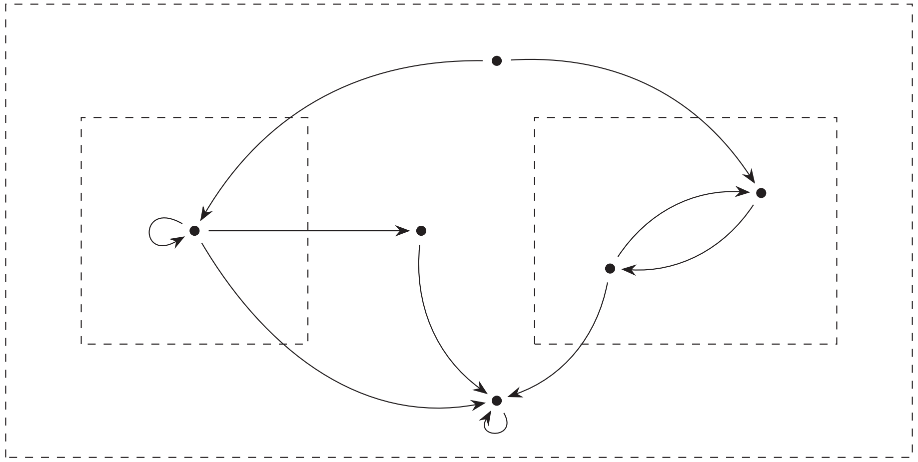
$e = fg$



$e = f^{(a)}$

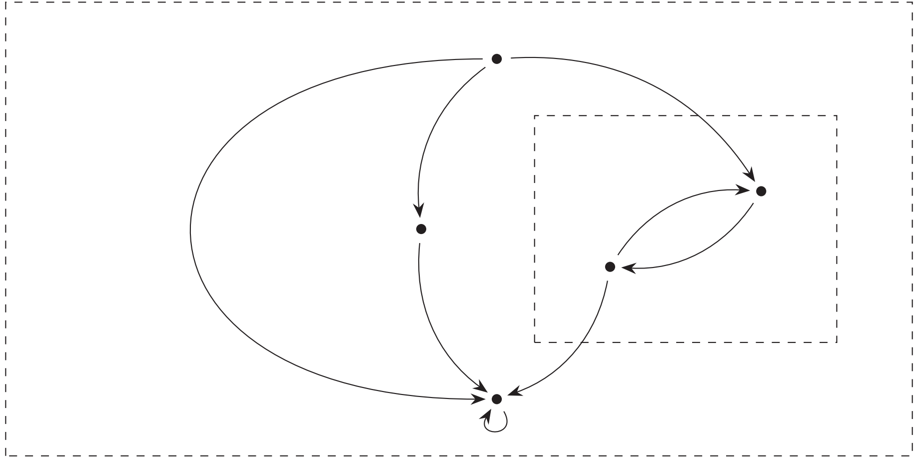
- Conversion of expression to automaton by induction on structure.
- Inductive cases are shown here.
  - For branching, we juxtapose and make a new initial state based on the guard.
  - For sequencing, we juxtapose and reroute accepting transitions on the left.
  - For loops, we reroute accepting transitions that validate the guard to the initial state.
- This translation is linear in the size of e.

# A Kleene theorem



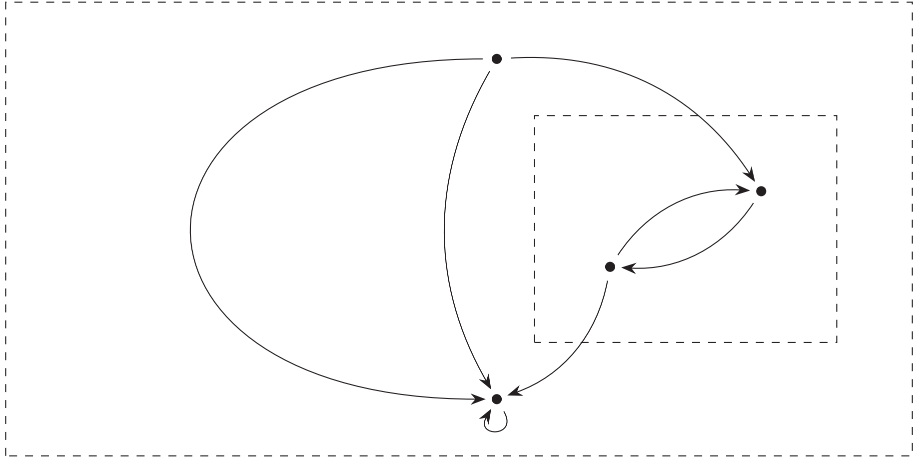
- Conversion of automaton back to expression requires structural restriction.
- This is because constructs like `goto` are absent from our language.
- Well-nested automata are inductively constructed to guarantee this structure.
- We can exploit this inductive structure to craft an expression.

# A Kleene theorem



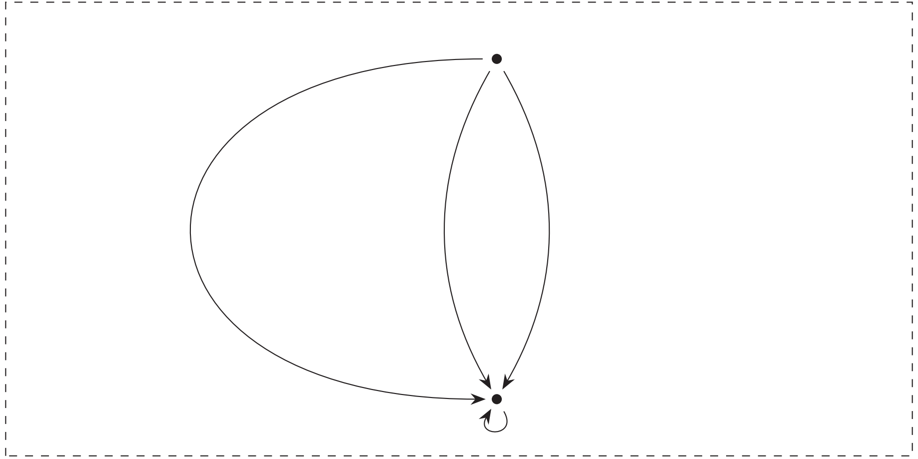
- Conversion of automaton back to expression requires structural restriction.
- This is because constructs like `goto` are absent from our language.
- Well-nested automata are inductively constructed to guarantee this structure.
- We can exploit this inductive structure to craft an expression.

# A Kleene theorem



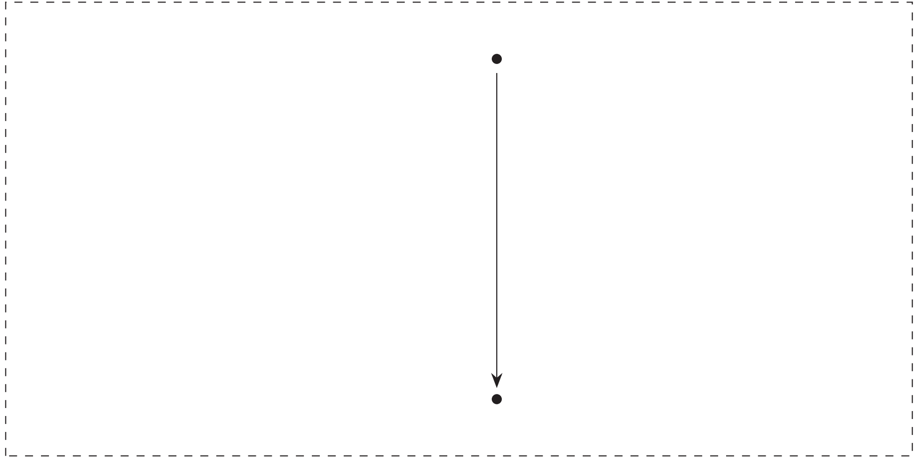
- Conversion of automaton back to expression requires structural restriction.
- This is because constructs like `goto` are absent from our language.
- Well-nested automata are inductively constructed to guarantee this structure.
- We can exploit this inductive structure to craft an expression.

# A Kleene theorem



- Conversion of automaton back to expression requires structural restriction.
- This is because constructs like `goto` are absent from our language.
- Well-nested automata are inductively constructed to guarantee this structure.
- We can exploit this inductive structure to craft an expression.

# A Kleene theorem



- Conversion of automaton back to expression requires structural restriction.
- This is because constructs like `goto` are absent from our language.
- Well-nested automata are inductively constructed to guarantee this structure.
- We can exploit this inductive structure to craft an expression.

# A Kleene theorem

**Theorem**

*Let $L \subseteq (\Sigma \cup Atoms)^*$. The following are equivalent:*

1. $L = [\![\mathbf{e}]\!]$ *for some* $\mathbf{e}$.
2. *L is accepted by a well-nested and finite automaton.*



- This conversion is correct: the automaton created accepts the same language.
- We can go the other way as well for well-structured automata.
- In fact, the automaton created from an expression is well-structured.

## Further work

- Coalgebraic perspective, coequations
- Instantiation framework; hypotheses
- Fully algebraic axiomatization

https://kap.pe/slides

https://arxiv.org/abs/1907.05920

# Bonus: non-well-nested automaton

From [Kozen and Tseng 2008]: