Guarded Kleene Algebra with Tests
Verification of Uninterpreted Programs in Nearly Linear Time

Steffen Smolka\textsuperscript{1} Nate Foster\textsuperscript{1} Justin Hsu\textsuperscript{2}
Tobias Kappé\textsuperscript{3} Dexter Kozen\textsuperscript{1} Alexandra Silva\textsuperscript{3}

\textsuperscript{1}Cornell University
\textsuperscript{2}University of Wisconsin-Madison
\textsuperscript{3}University College London

POPL 2020
Introduction

- 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.
Uninterpreted programs can be thought of as skeletons of programs. This allows reasoning about refactoring, optimisation, et cetera.
Introduction

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

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

\[
e, f ::= a \mid p \in \Sigma \mid ef \mid e + a f \mid e^{(a)}
\]
Syntax and semantics

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

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

\[
e, f ::= a \mid p \in \Sigma \mid ef \mid e + a f \mid e^{(a)}
\]
Syntax and semantics

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

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

\[ e, f ::= a \mid p \in \Sigma \mid ef \mid e + a f \mid e^{(a)} \]
Syntax and semantics

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

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

\[ e, f ::= a \mid \sigma \in \Sigma \mid ef \mid e + a f \mid e(a) \]
Syntax and semantics

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

\[ e, f ::= a \mid p \in \Sigma \mid ef \mid e + af \mid e^p \]

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

Note: overloading conjunction and concatenation.

\[\begin{align*}
a, b &::= t \in T \mid a + b \mid ab \mid \overline{a} \mid 0 \mid 1 \\
\end{align*}\]

\[\begin{align*}
e, f &::= a \mid p \in \Sigma \mid ef \mid e + af \mid \text{true(a)}
\end{align*}\]
Syntax and semantics

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

\[ a, b ::= t \in T | a + b | ab | \bar{a} | 0 | 1 \]

\[ e, f ::= a | p \in \Sigma | ef | e + a f | e^{(a)} \]

assert a
Syntax and semantics

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

\[ e, f ::= a \mid p \in \Sigma \mid ef \mid e + af \mid e^{(a)} \mid e; f \]

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

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

\[ e, f ::= a \mid \pi \in \Sigma \mid ef \mid \overline{e} + a f \mid e^{(a)} \]

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

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

\[ e, f ::= a \mid p \in \Sigma \mid ef \mid e + a f \mid e(a) \]

while \( a \) do \( e \)
The programs from before can now be written down like this.
Syntax and semantics

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

- 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

\[ i = (\text{sat} : T \to 2^{\text{States}}, \text{eval} : \Sigma \to 2^{\text{States}^2}) \]

\[
e \xrightarrow{R_i[e]}
\]

\[
t \in T \quad \{ (s, s) : s \in \text{sat}(t) \}
\]

\[
a + b \quad R_i[a] \cup R_i[b]
\]

\[
ab \quad R_i[a] \cap R_i[b]
\]

\[
a \quad \text{States}^2 \setminus R_i[a]
\]

\[
p \in \Sigma \quad \text{eval}(p)
\]

\[
e + a f \quad R_i[a] \circ R_i[e] \cup R_i[a] \circ R_i[f]
\]

\[
e f \quad R_i[e] \circ R_i[f]
\]

\[
e^{(a)} \quad (R_i[a] \circ R_i[e])^* \circ R_i[a]
\]

- 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.
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

\[\text{Atoms} = 2^T\]

- 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.

\[
\begin{align*}
\alpha \in \text{Atoms} & \quad \alpha, \beta \in \text{Atoms} & \quad p \in \Sigma & \quad \alpha \rho \beta \in \text{GS}(\Sigma, T) & \quad w\alpha, \alpha x \in \text{GS}(\Sigma, T) \\
\alpha \in \text{GS}(\Sigma, T) & \quad \alpha p \beta \in \text{GS}(\Sigma, T) & \quad w\alpha x \in \text{GS}(\Sigma, T)
\end{align*}
\]
Syntax and semantics

Atoms = $2^T$

\[
\begin{align*}
\alpha &\in \text{Atoms} \\
\alpha &\in \text{GS}(\Sigma, T) \\
\alpha, \beta &\in \text{Atoms} \\
\beta &\in \Sigma \\
\alpha \beta &\in \text{GS}(\Sigma, T) \\
w\alpha &\in \text{GS}(\Sigma, T) \\
w\alpha, \alpha x &\in \text{GS}(\Sigma, T) \\
w\alpha &\in \text{GS}(\Sigma, T) \\
w\alpha x &\in \text{GS}(\Sigma, T) \\
\end{align*}
\]

$L \circ K = \{ w\alpha x : w\alpha \in L, \alpha x \in K \} \\
L^{(n)} = L \circ \cdots \circ L \quad \text{for } n \in \mathbb{N} \\
L^{(*)} = \bigcup_{n\in\mathbb{N}} L^{(n)}$

- 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

<table>
<thead>
<tr>
<th>$e$</th>
<th>$[e]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t \in T$</td>
<td>{ $\alpha \in \text{Atoms} : t \in \alpha$ }</td>
</tr>
<tr>
<td>$a + b$</td>
<td>$[a] \cup [b]$</td>
</tr>
<tr>
<td>$ab$</td>
<td>$[a] \cap [b]$</td>
</tr>
<tr>
<td>$\overline{a}$</td>
<td>\text{Atoms {} a }}</td>
</tr>
<tr>
<td>$p \in \Sigma$</td>
<td>{ $\alpha \beta : \alpha, \beta \in \text{Atoms}$ }</td>
</tr>
<tr>
<td>$e +_a f$</td>
<td>$[a] \diamond [e] \cup [\overline{a}] \diamond [f]$</td>
</tr>
<tr>
<td>$ef$</td>
<td>$[e] \diamond [f]$</td>
</tr>
<tr>
<td>$e^{(a)}$</td>
<td>$([a] \diamond [e])^{(*)} \diamond [\overline{a}]$</td>
</tr>
</tbody>
</table>

- 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

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

- 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

\[ [e] = [f] \iff \forall i. R_i[e] = R_i[f] \]

How to check \([e] = [f]\):

1. Create automata that accept \([e]\) and \([f]\).
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

\[ e +, a \equiv e \]

- 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

\[ e + a e \equiv e \quad e + a f \equiv f + \pi e \]

- 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

\[ e + a e \equiv e \quad e + a f \equiv f + a e \quad e + a f \equiv a e + a f \]

- 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

- 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.

\[
e + a e \equiv e \quad e + a f \equiv f + e \quad e + a f \equiv a e + a f \quad \exists a \equiv 0
\]
Axiomatization

\[
e + a e \equiv e \\
e + a f \equiv f + e \\
e + a f \equiv a e + a f \\
\exists a \equiv 0 \\
0e \equiv 0
\]

- 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

\[ e + a e = e \quad e + a f = f + e \quad e + a f = ae + af \quad \exists a \equiv 0 \quad 0e \equiv 0 \]

Example

\[ \text{if } a \text{ then } e \text{ else assert false } = e + a 0 \]

- 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

\[ e + a e \equiv e \quad e + a f \equiv f + \pi e \quad [\exists a \equiv 0 \quad 0 e \equiv 0] \]

Example

\[ \text{if } a \text{ then } e \text{ else assert false } = e + a 0 \equiv ae + a 0 \]

- 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

\[
e + a e \equiv e \quad e + a f \equiv f + a e \quad e + a f \equiv a e + a f \quad \exists a \equiv 0 \quad 0e \equiv 0
\]

Example

\[
\text{if } a \text{ then } e \text{ else assert false } = e + a 0 \equiv ae + a 0 \\
\equiv 0 + a ae
\]

- 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

\[ e + a e \equiv e \quad e + a f \equiv f + a e \quad e + a f \equiv ae + a f \quad \exists a \equiv 0 \quad \emptyset e \equiv 0 \]

Example

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

\[ e + a e \equiv e \quad e + a f \equiv f + \pi e \quad e + a f \equiv a e + a f \]

\[ \bar{a} a \equiv 0 \quad 0 e \equiv 0 \]

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

\[ e +_a e = e \quad e +_a f = f +_e e \]

\[ e +_a f = ae + f \quad \exists a = 0 \quad 0e = 0 \]

Example

\[
\text{if } a \text{ then } e \text{ else assert false } = e +_a 0 = ae +_a 0 \\
\equiv 0 +_\pi ae \\
\equiv 0e +_\pi ae \\
\equiv \exists ae +_\pi ae \\
\equiv ae +_\exists ae
\]

- 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

\[
\begin{align*}
    e + a e & \equiv e \\
    e + a f & \equiv f + a e \\
    e + a f & \equiv ae + f \\
    sa & \equiv 0 \\
    0e & \equiv 0
\end{align*}
\]

Example

\[
\begin{align*}
    \text{if } a \text{ then } e \text{ else assert false } &= e + a 0 \equiv ae + a 0 \\
    &\equiv 0 + a ae \\
    &\equiv 0e + a ae \\
    &\equiv sae + a ae \\
    &\equiv ae + a ae \\
    &\equiv ae = \text{assert } a; e
\end{align*}
\]

- 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

\[ e \equiv f e + a g \]

\[ e \equiv f^{(a)} g \]

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

\[
e \equiv f\cdot e + a \cdot g \\
e \equiv f^{(a)} \cdot g
\]

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

while true do assert true \(\equiv\) assert true

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

\[ e \equiv f \cdot^a g \quad \text{f is productive} \]
\[ e \equiv f^{e\cdot g} \]

- 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

\[ e \equiv fe + ag \quad \text{f is productive} \]

\[ e \equiv f(a)g \]

\[ e^{(a)} \equiv ee^{a} + a1 \]

- 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.
\[
e \equiv \mathbf{e} \mathbf{f} + \mathbf{a} \mathbf{g} \quad \text{\textit{f}} \text{ is productive} \\
\mathbf{e} \equiv \mathbf{f}^{(a)} \mathbf{g} \\
\mathbf{e}^{(a)} \equiv \mathbf{e} \mathbf{e}^{a} + \mathbf{a} \mathbf{1} \quad (\mathbf{e} + \mathbf{a} \mathbf{1})^{(b)} \equiv (\mathbf{ae})^{(b)}
\]

- 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

\[
e \equiv fe + a g \quad \text{f is productive} \\
e \equiv f^{(a)} g \\
e^{(a)} \equiv e e^a + a 1 \\
(e + a 1)^{(b)} \equiv (ae)^{(b)}
\]

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

\[ e \equiv fe + a g \quad \text{f is productive} \]

\[ e \equiv f^e g \]

\[ e^{(a)} \equiv ee^a + a \]

\[ (e+1)^{(b)} \equiv (ae)^{(b)} \]

Lemma

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

Lemma

\[ e^{(a)} \equiv e^{(a) + a} \]

\[ e^{(a)} \equiv (ae)^{(a)} \]

\[ e^{(ab)} e^{(b)} \equiv e^{(b)} \]

• 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.
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 $e \equiv f$, then $[e] = [f]$.

How about the converse?

1. $A \mapsto S(A)$ with $e \equiv S(A_e)$.
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 \( e \equiv f \), then \( [e] = [f] \).

How about the converse?

1. \( A \mapsto S(A) \) with \( e \equiv S(A_e) \).
2. If \( A \sim A' \), then \( S(A) \equiv S(A') \).

\[
[e] = [f] \implies L(A_e) = L(A_f) \\
\implies A_e \sim A_f \\
\implies S(A_e) \equiv S(A_f) \\
\implies e \equiv f
\]

• 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.
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

- 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

- 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.

\[
\begin{align*}
\alpha & \xleftarrow{\beta/p} S_1 \\
& \xrightarrow{\gamma/q} S_2 \quad \beta p \gamma q \alpha \in \mathcal{L}(s_1) \\
(X, \delta : X \rightarrow (2 + \Sigma \times X)^{\text{Atoms}})
\end{align*}
\]
A Kleene theorem

- 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 $e$. 

![Diagram of automaton]

$e = f + g$
A Kleene theorem

- 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 $e$. 

\[
e = f + a g
\]
A Kleene theorem

- 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 $e$. 

\[ e = f + \beta g \]

\[ e = fg \]
A Kleene theorem

- 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 $e$. 

\[
e = f + \varepsilon g
\]

\[
e = fg
\]
A Kleene theorem

• 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 $e$. 

$e = f + a \cdot g$

$e = fg$

$e = f[a]$
A Kleene theorem

- 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 $e$. 

\[
\begin{align*}
e &= f + a g \\
e &= f g
\end{align*}
\]
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.
• 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 $L \subseteq (\Sigma \cup \text{Atoms})^*$. The following are equivalent:

1. $L = \text{Jek}$ for some $e$.
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
From [Kozen and Tseng 2008]:

\[
\begin{align*}
\alpha_0 + \alpha_3 & \quad \alpha_1 / p_{01} \\
\alpha_0 / p_{10} & \quad \alpha_1 + \alpha_3 \\
\alpha_0 / p_{02} & \quad \alpha_1 / p_{21} \\
\alpha_0 / p_{12} & \quad \alpha_2 + \alpha_3
\end{align*}
\]