Static Inference of Regular Grammars for Ad Hoc Parsers

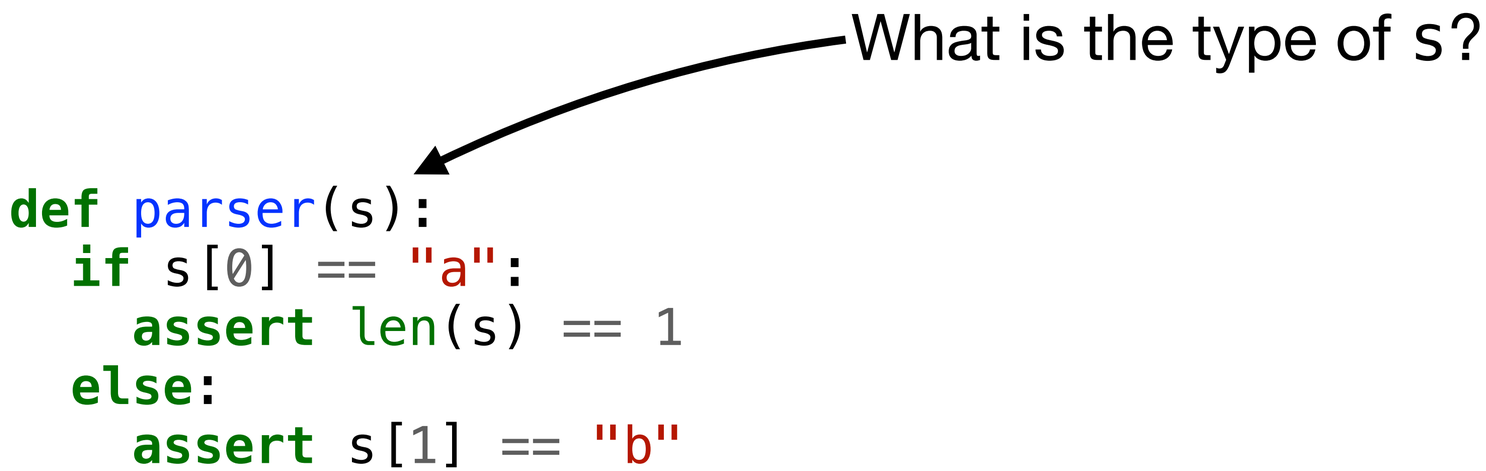
Michael Schröder and **Jürgen Cito**
TU Wien, Vienna, Austria

OOPSLA 2025
Singapore

FWF Österreichischer Wissenschaftsfonds
This research was funded in whole or in part by the Austrian Science Fund (FWF) [10.55776/PIN3275223].

TU WIEN Informatics

```python
def parser(s):
  if s[0] == "a":
    assert len(s) == 1
  else:
    assert s[1] == "b"
```


What is the type of s?
def parser(s):
if s[0] == "a":
assert len(s) == 1
else:
assert s[1] == "b"

```python
def parser(s: str):
  if s[0] == "a":
    assert len(s) == 1
  else:
    assert s[1] == "b"
```

```python
def parser(s: str):
  if s[0] == "a":
    assert len(s) == 1
  else:
    assert s[1] == "b"
```

```
>>> parser("a")
```

```python
def parser(s: str):
  if s[0] == "a":
    assert len(s) == 1
  else:
    assert s[1] == "b"
```

```
>>> parser("a")
>>>
```

```python
def parser(s: str):
  if s[0] == "a":
    assert len(s) == 1
  else:
    assert s[1] == "b"
```

```
>>> parser("a")
>>> parser("b")
```

```python
def parser(s: str):
  if s[0] == "a":
    assert len(s) == 1
  else:
    assert s[1] == "b"
```

```
>>> parser("a")
>>> parser("b")
Traceback (most recent call last):
  File "<python-input-2>", line 1, in <module>
    parser("b")
    ~~~~~~^^^^^
  File "<python-input-0>", line 5, in parser
    assert s[1] == "b"
           ~^^^
IndexError: string index out of range
>>>
```

```python
def parser(s: str):
  if s[0] == "a":
    assert len(s) == 1
  else:
    assert s[1] == "b"
```

```
>>> parser("a")
>>> parser("b")
Traceback (most recent call last):
  File "<python-input-2>", line 1, in <module>
    parser("b")
    ~~~~~~^^^^^
  File "<python-input-0>", line 5, in parser
    assert s[1] == "b"
           ~^^^
IndexError: string index out of range
>>> parser("ab")
```

```python
def parser(s: str):
  if s[0] == "a":
    assert len(s) == 1
  else:
    assert s[1] == "b"
```

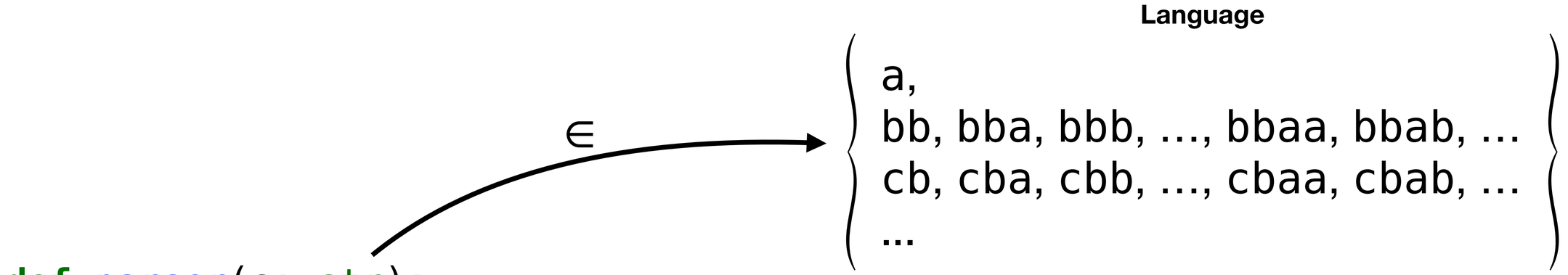
```
>>> parser("a")
>>> parser("b")
Traceback (most recent call last):
  File "<python-input-2>", line 1, in <module>
    parser("b")
    ~~~~~~^^^^^
  File "<python-input-0>", line 5, in parser
    assert s[1] == "b"
           ~^^^
IndexError: string index out of range
>>> parser("ab")
Traceback (most recent call last):
  File "<python-input-3>", line 1, in <module>
    parser("ab")
    ~~~~~~^^^^^^
  File "<python-input-0>", line 3, in parser
    assert len(s) == 1
           ^^^^^^^^^^^
AssertionError
>>>
```

```python
def parser(s: str):
  if s[0] == "a":
    assert len(s) == 1
  else:
    assert s[1] == "b"
```

```
>>> parser("a")
>>> parser("b")
Traceback (most recent call last):
  File "<python-input-2>", line 1, in <module>
    parser("b")
    ~~~~~~^^^^^
  File "<python-input-0>", line 5, in parser
    assert s[1] == "b"
           ~^^^
IndexError: string index out of range
>>> parser("ab")
Traceback (most recent call last):
  File "<python-input-3>", line 1, in <module>
    parser("ab")
    ~~~~~~^^^^^^
  File "<python-input-0>", line 3, in parser
    assert len(s) == 1
           ^^^^^^^^^^^
AssertionError
>>> parser("bb")
```

```python
def parser(s: str):
  if s[0] == "a":
    assert len(s) == 1
  else:
    assert s[1] == "b"
```

```
>>> parser("a")
>>> parser("b")
Traceback (most recent call last):
  File "<python-input-2>", line 1, in <module>
    parser("b")
    ~~~~~~^^^^^
  File "<python-input-0>", line 5, in parser
    assert s[1] == "b"
           ~^^^
IndexError: string index out of range
>>> parser("ab")
Traceback (most recent call last):
  File "<python-input-3>", line 1, in <module>
    parser("ab")
    ~~~~~~^^^^^^
  File "<python-input-0>", line 3, in parser
    assert len(s) == 1
           ^^^^^^^^^^^
AssertionError
>>> parser("bb")
>>>
```


∈
def parser(s: str):
if s[0] == "a":
assert len(s) == 1
else:
assert s[1] == "b"
Language
a,
bb, bba, bbb, ..., bbaa, bbab, ...
cb, cba, cbb, ..., cbaa, cbab, ...
...
Grammar
$a + (\Sigma \backslash a) b \Sigma^*$
Regular Expression
a|[^a]b.*

~~What is the type of s?~~

What is the grammar of `str`?

```python
def parser(s: str):
  if s[0] == "a":
    assert len(s) == 1
  else:
    assert s[1] == "b"
```

Grammar

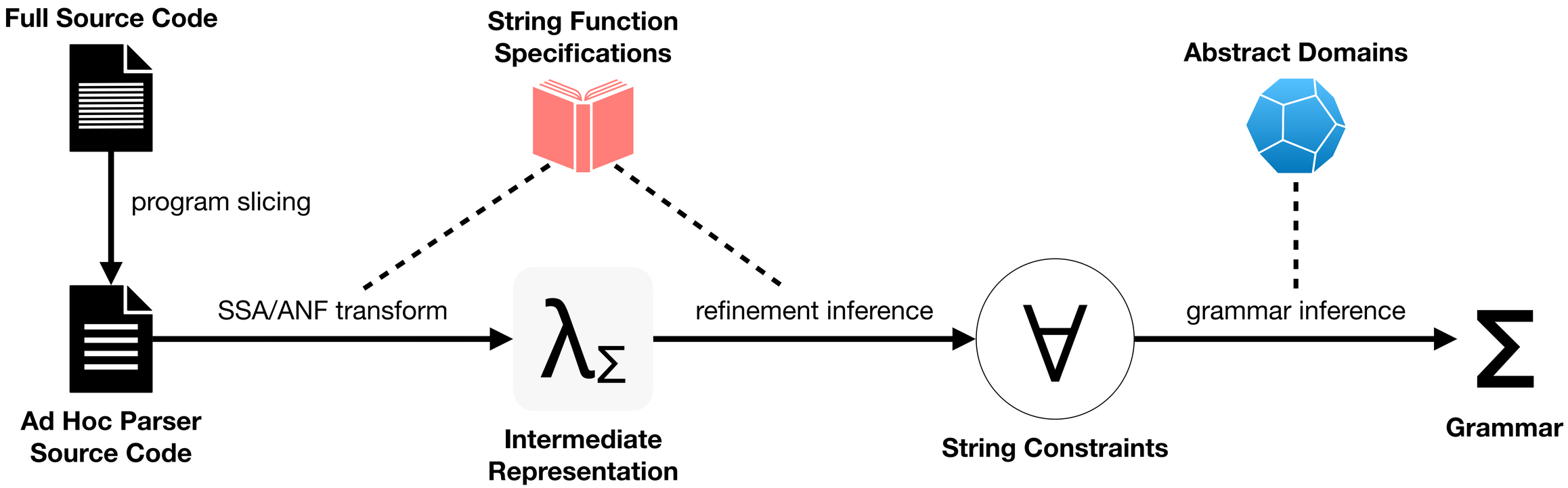
$$a + (\Sigma \backslash a) b \Sigma^*$$

Regular Expression

```
a|[^a]b.*
```

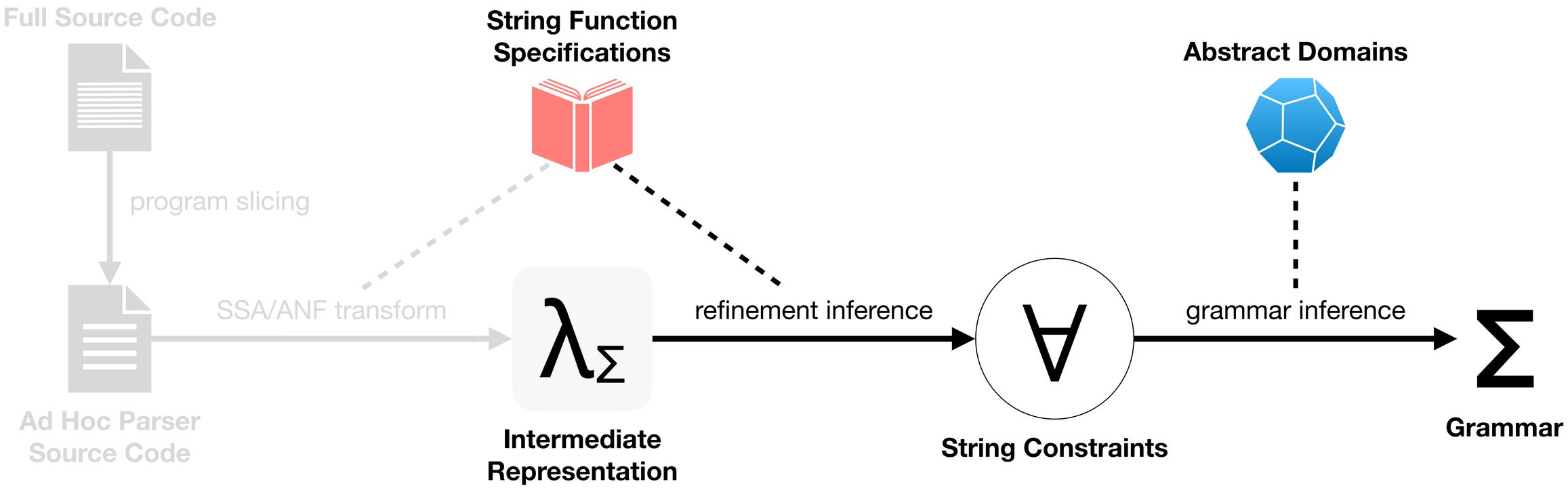
PANINI

"grammars for free"



PANINI

"grammars for free"



Intermediate Representation λ_Σ

- simple λ-calculus in ANF

```python
def parser(s: str):
  if s[0] == "a":
    assert len(s) == 1
  else:
    assert s[1] == "b"
```

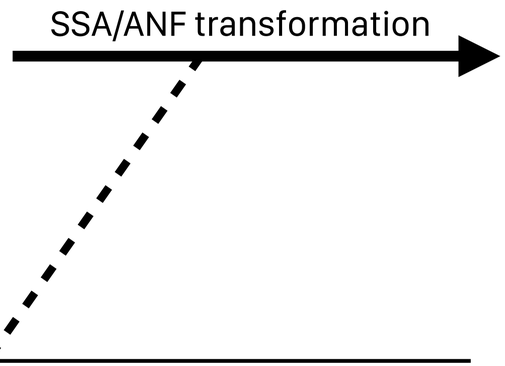
SSA/ANF transformation

```
parser = λs:𝕊.
  let x = charAt s 0 in
  let p1 = eqChar x 'a' in
  if p1 then
    let n = length s in
    let p2 = eq n 1 in
    assert p2
  else
    let y = charAt s 1 in
    let p3 = eqChar y 'b' in
    assert p3
```

Intermediate Representation λ_Σ

- simple λ-calculus in ANF
- common string operations as axiomatic specifications

```python
def parser(s: str):
  if s[0] == "a":
    assert len(s) == 1
  else:
    assert s[1] == "b"
```



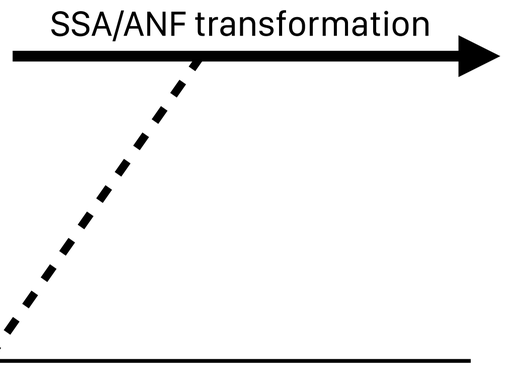

```
parser = λs:𝕊.
  let x = charAt s 0 in
  let p1 = eqChar x 'a' in
  if p1 then
    let n = length s in
    let p2 = eq n 1 in
    assert p2
  else
    let y = charAt s 1 in
    let p3 = eqChar y 'b' in
    assert p3
```

Python	λ_Σ specification
`assert b`	$\text{assert} : \{b : \mathbb{B} \mid b\} \to \mathbb{1}$
`a == b`	$\text{eq} : (a : \mathbb{Z}) \to (b : \mathbb{Z}) \to \{c : \mathbb{B} \mid c \Leftrightarrow a = b\}$
`len(s)`	$\text{length} : (s : \mathbb{S}) \to \{n : \mathbb{N} \mid n = \lvert s \rvert\}$
`s[i]`	$\text{charAt} : (s : \mathbb{S}) \to \{i : \mathbb{N} \mid i < \lvert s \rvert\} \to \{c : \mathbb{Ch} \mid c = s[i]\}$
`s == t`	$\text{eqChar} : (s : \mathbb{Ch}) \to (t : \mathbb{Ch}) \to \{b : \mathbb{B} \mid b \Leftrightarrow s = t\}$

Intermediate Representation λ_Σ

- simple λ-calculus in ANF
- common string operations as axiomatic specifications
- refinement type system à la *Liquid Types*

```python
def parser(s: str):
  if s[0] == "a":
    assert len(s) == 1
  else:
    assert s[1] == "b"
```




```
parser : {s:𝕊 | ?} → 𝟙
parser = λs:𝕊.
  let x = charAt s 0 in
  let p1 = eqChar x 'a' in
  if p1 then
    let n = length s in
    let p2 = eq n 1 in
    assert p2
  else
    let y = charAt s 1 in
    let p3 = eqChar y 'b' in
    assert p3
```

Python	λ_Σ specification
`assert b`	$\text{assert} : \{b : \mathbb{B} \mid b\} \to \mathbb{1}$
`a == b`	$\text{eq} : (a : \mathbb{Z}) \to (b : \mathbb{Z}) \to \{c : \mathbb{B} \mid c \Leftrightarrow a = b\}$
`len(s)`	$\text{length} : (s : \mathbb{S}) \to \{n : \mathbb{N} \mid n = \lvert s \rvert\}$
`s[i]`	$\text{charAt} : (s : \mathbb{S}) \to \{i : \mathbb{N} \mid i < \lvert s \rvert\} \to \{c : \mathbb{Ch} \mid c = s[i]\}$
`s == t`	$\text{eqChar} : (s : \mathbb{Ch}) \to (t : \mathbb{Ch}) \to \{b : \mathbb{B} \mid b \Leftrightarrow s = t\}$

Refinement Inference

- type system generates *verification conditions* (VCs)
- validity of VC entails correctness of type

verification condition

$$
\begin{aligned}
&\forall s.\ \kappa(s) \Rightarrow \\
&\quad (\forall v_1.\ v_1 = 0 \Rightarrow v_1 \geq 0 \wedge v_1 < |s|) \wedge \\
&\quad (\forall x.\ x = s[0] \Rightarrow \\
&\qquad (\forall p_1.\ p_1 = \text{true} \Leftrightarrow x = \text{`a'} \Rightarrow \\
&\qquad\quad (p_1 = \text{true} \Rightarrow \\
&\qquad\qquad (\forall n.\ n \geq 0 \wedge n = |s| \Rightarrow n = 1)) \wedge \\
&\qquad\quad (p_1 = \text{false} \Rightarrow \\
&\qquad\qquad (\forall v_2.\ v_2 = 1 \Rightarrow v_2 \geq 0 \wedge v_2 < |s|) \wedge \\
&\qquad\qquad (\forall y.\ y = s[1] \Rightarrow y = \text{`b'}))))
\end{aligned}
$$

```
⊨   parser : {s:𝕊 | κ(s)} → 𝟙
    parser = λs:𝕊.
      let x = charAt s 0 in
      let p1 = eqChar x 'a' in
      if p1 then
        let n = length s in
        let p2 = eq n 1 in
        assert p2
      else
        let y = charAt s 1 in
        let p3 = eqChar y 'b' in
        assert p3
```

Refinement Inference

- type system generates *verification conditions* (VCs)
- κ variables represent unknown refinements that still need to be found

verification condition

$$
\begin{aligned}
&\forall s.\ \kappa(s) \Rightarrow \\
&\quad (\forall v_1.\ v_1 = 0 \Rightarrow v_1 \geq 0 \wedge v_1 < |s|) \wedge \\
&\quad (\forall x.\ x = s[0] \Rightarrow \\
&\qquad (\forall p_1.\ p_1 = \text{true} \Leftrightarrow x = \text{`a'} \Rightarrow \\
&\qquad\quad (p_1 = \text{true} \Rightarrow \\
&\qquad\qquad (\forall n.\ n \geq 0 \wedge n = |s| \Rightarrow n = 1)) \wedge \\
&\qquad\quad (p_1 = \text{false} \Rightarrow \\
&\qquad\qquad (\forall v_2.\ v_2 = 1 \Rightarrow v_2 \geq 0 \wedge v_2 < |s|) \wedge \\
&\qquad\qquad (\forall y.\ y = s[1] \Rightarrow y = \text{`b'}))))
\end{aligned}
$$

```
⊨  parser : {s:𝕊 | κ(s)} → 𝟙
   parser = λs:𝕊.
     let x = charAt s 0 in
     let p1 = eqChar x 'a' in
     if p1 then
       let n = length s in
       let p2 = eq n 1 in
       assert p2
     else
       let y = charAt s 1 in
       let p3 = eqChar y 'b' in
       assert p3
```

Grammar Inference

- key insight: the top-level VC of a parser is always $\forall s\,.\,\kappa(s) \Rightarrow \varphi$
- the solution $\kappa(s) \mapsto \varphi$ trivially validates the VC

verification condition

$$
\begin{aligned}
&\forall s.\ \kappa(s) \Rightarrow \\
&\quad (\forall v_1.\ v_1 = 0 \Rightarrow v_1 \geq 0 \wedge v_1 < |s|) \wedge \\
&\quad (\forall x.\ x = s[0] \Rightarrow \\
&\qquad (\forall p_1.\ p_1 = \text{true} \Leftrightarrow x = \text{`a'} \Rightarrow \\
&\qquad\quad (p_1 = \text{true} \Rightarrow \\
&\qquad\qquad (\forall n.\ n \geq 0 \wedge n = |s| \Rightarrow n = 1)) \wedge \\
&\qquad\quad (p_1 = \text{false} \Rightarrow \\
&\qquad\qquad (\forall v_2.\ v_2 = 1 \Rightarrow v_2 \geq 0 \wedge v_2 < |s|) \wedge \\
&\qquad\qquad (\forall y.\ y = s[1] \Rightarrow y = \text{`b'}))))
\end{aligned}
$$

string constraint that precisely captures parsing operations on s

Grammar Inference

- problem: the string constraint is not a very practical grammar
- it is a first-order formula as big as the original program!

```
parser : {s:𝕊 |
```

$$
\begin{aligned}
&(\forall v_1.\ v_1 = 0 \Rightarrow v_1 \geq 0 \wedge v_1 < |s|) \wedge \\
&(\forall x.\ x = s[0] \Rightarrow \\
&\quad (\forall p_1.\ p_1 = \text{true} \Leftrightarrow x = \text{`a'} \Rightarrow \\
&\qquad (p_1 = \text{true} \Rightarrow \\
&\qquad\quad (\forall n.\ n \geq 0 \wedge n = |s| \Rightarrow n = 1)) \wedge \\
&\qquad (p_1 = \text{false} \Rightarrow \\
&\qquad\quad (\forall v_2.\ v_2 = 1 \Rightarrow v_2 \geq 0 \wedge v_2 < |s|) \wedge \\
&\qquad\quad (\forall y.\ y = s[1] \Rightarrow y = \text{`b'}))))
\end{aligned}
$$

```
} → 𝟙
```

AI

Abstract Interpretation

Abstract Interpretation

of ad hoc parsers represented by first-order formulas

$$\overbrace{\Gamma}^{\text{axioms}} \vdash P \nearrow \overbrace{\{s : \mathbb{S} \mid \kappa(s)\} \to \mathbb{1}}^{\text{incomplete refinement}} \nexists \overbrace{\forall s.\ \kappa(s) \Longrightarrow \varphi}^{\text{verification condition}}$$

Abstract Interpretation

of ad hoc parsers represented by first-order formulas

$$\overbrace{\Gamma}^{\text{axioms}} \vdash P \nearrow \overbrace{\{s : \mathbb{S} \mid \kappa(s)\} \to \mathbb{1}}^{\text{incomplete refinement}} \nexists \overbrace{\forall s.\, \kappa(s) \Longrightarrow \varphi}^{\text{verification condition}}$$

$$\mathfrak{M}, \sigma \models \varphi \quad \Longleftrightarrow \quad \sigma \in [\![\varphi]\!] \quad \Longleftrightarrow \quad \sigma(s) \in L(P)$$

Abstract Interpretation

of ad hoc parsers represented by first-order formulas

$$\overbrace{\Gamma}^{\text{axioms}} \vdash P \nearrow \overbrace{\{s : \mathbb{S} \mid \kappa(s)\} \to \mathbb{1}}^{\text{incomplete refinement}} \nexists \overbrace{\forall s.\, \kappa(s) \Rightarrow \varphi}^{\text{verification condition}}$$

$$\mathfrak{M}, \sigma \models \varphi \iff \sigma \in [\![\varphi]\!] \iff \sigma(s) \in L(P)$$

$$\mathfrak{M}, \hat{\sigma} \models \varphi \impliedby \hat{\sigma} = [\![\varphi]\!]^\sharp \implies \hat{\sigma}(s) \subseteq L(P)$$

Abstract Semantics
of String Constraints

$$[\![\varphi]\!]^\sharp \doteq \left\{ x \mapsto [\![\varphi]\!]{\uparrow_x} \;\middle|\; x \in \mathrm{vars}(\varphi) \right\}$$

$$\begin{aligned}
[\![\varphi_1 \wedge \varphi_2]\!]{\uparrow_x} &\doteq [\![\varphi_1]\!]{\uparrow_x} \sqcap [\![\varphi_2]\!]{\uparrow_x} && \text{if } \varphi_1, \varphi_2 \text{ quantifier-free} \\
[\![\varphi_1 \vee \varphi_2]\!]{\uparrow_x} &\doteq [\![\varphi_1]\!]{\uparrow_x} \sqcup [\![\varphi_2]\!]{\uparrow_x} && \text{if } \varphi_1, \varphi_2 \text{ quantifier-free} \\
[\![\neg\varphi]\!]{\uparrow_x} &\doteq \neg [\![\varphi]\!]{\uparrow_x} && \text{if } \varphi \text{ quantifier-free} \\
[\![\varphi]\!]{\uparrow_x} &\doteq [\![\mathrm{qelim}(\varphi)]\!]{\uparrow_x}
\end{aligned}$$

$$[\![\rho]\!]{\uparrow_x} \doteq \begin{cases} \omega, & \text{as defined by domain;} \\ \langle \rho[x := \bullet] \rangle, & \text{otherwise.} \end{cases}$$

quantifier elimination (excerpt)

$$\begin{aligned}
&\mathrm{qelim1}(x, R) \doteq \hat{R} \wedge R_{\bar{x}} \\
&\quad \text{where} \\
&\quad \hat{R} = \left\{ \omega_1 \between \omega_2 \;\middle|\; (\omega_1, \omega_2) \in \binom{E}{2} \right\} \\
&\quad E = \dot{\bigsqcap} \left\{ [\![\rho]\!]{\uparrow_x} \;\middle|\; \rho \in R_x \right\} \\
&\quad R_x = \{ \rho \in R \mid x \in \mathrm{vars}(\rho) \} \\
&\quad R_{\bar{x}} = \{ \rho \in R \mid x \notin \mathrm{vars}(\rho) \}
\end{aligned}$$

relational string semantics (excerpt)

$$\begin{aligned}
[\![|s| \between \hat{n}]\!]{\uparrow_s} &\doteq \Sigma^{\hat{n}} \\
[\![s[\hat{i}] \between \hat{c}]\!]{\uparrow_s} &\doteq \Sigma^{\hat{i}} \hat{c} \Sigma^* \\
[\![s[|s| - \hat{i}] \between \hat{c}]\!]{\uparrow_s} &\doteq \Sigma^* \hat{c} \Sigma^{\hat{i}-1} \\
[\![s[\hat{i}..\hat{j}] \between \hat{t}]\!]{\uparrow_s} &\doteq \Sigma^{\hat{i}} (\Sigma^{\hat{j}-\hat{i}+1} \sqcap \hat{t}) \Sigma^*
\end{aligned}$$

Abstract Domains

for Grammar Inference

unit	$\wp(\mathbb{1})$	$=$	$\hat{\mathbf{1}}$	the two-element lattice
Booleans	$\wp(\mathbb{B})$	$=$	$\hat{\mathbf{B}}$	Boolean subset lattice
integers	$\wp(\mathbb{Z})$	$\supseteq$	$\hat{\mathbf{Z}}$	open-ended interval lists
characters	$\wp(\mathbb{Ch})$	$=$	$\hat{\mathbf{C}}$	Unicode character sets
strings	$\wp(\mathbb{S})$	$\supseteq$	$\hat{\mathbf{S}}$	regular expressions over $\hat{\mathbf{C}}$

we under-approximate
super-regular languages

Grammar Inference

$$\texttt{parser} : \{s{:}\mathbb{S} \mid [\![(\forall v_1.\ v_1 = 0 \Rightarrow v_1 \geq 0 \wedge v_1 < |s|) \wedge (\forall x.\ x = s[0] \Rightarrow (\forall p_1.\ p_1 = \text{true} \Leftrightarrow x = \text{`a'} \Rightarrow (p_1 = \text{true} \Rightarrow (\forall n.\ n \geq 0 \wedge n = |s| \Rightarrow n = 1)) \wedge (p_1 = \text{false} \Rightarrow (\forall v_2.\ v_2 = 1 \Rightarrow v_2 \geq 0 \wedge v_2 < |s|) \wedge (\forall y.\ y = s[1] \Rightarrow y = \text{`b'}))))]\!]^{\#} \} \rightarrow \mathbb{1}$$

Grammar Inference

```
parser : {s:𝕊 |
```
$$s \in (a + (\Sigma \backslash a) b \Sigma^*)$$
```
} → 𝟙
```

Evaluation

- benchmark dataset of 204 regular ad hoc parsers written in Python
- classified into *straight-line*, *conditional*, and *loop* programs
- manually derived ground truth grammars to compute precision and recall

```
def parser(s):
  if s[0] == "a":
    assert len(s) == 1
  else:
    assert s[1] == "b"
```

```
a|[^a]b.*
```

```
def getAddrSpec(email):
  b1 = email.index('<',0)+1
  b2 = email.index('>',b1)
  return email[b1:b2]
```

```
[^<>]*<[^>]*>.*
```

```
def lsb_check(s):
  i = 0
  while i < len(s)-1:
    assert s[i] == '0'
    i += 1
  assert s[i] == '1'
```

```
0*1
```

Evaluation

- 100% precision: *PANINI never over-approximates!*
- 93% overall recall (100% for non-loop programs)

Type	Parser Example		#	Inferred Language =	⊂	∅	Error	SR	P	R	Time (s)
Straight	getAddrSpec	`[^<>]*<[^>]*>.*`	89	84	0	0	5	.94	1.00	1.00	0.16 ±0.27
+ Cond.	Figure 3	`a|[^a]b.*`	51	50	0	0	1	.98	1.00	1.00	0.09 ±0.05
+ Loops	lsb_check	`0*1`	64	40	7	7	10	.84	1.00	.76	2.32 ±4.77
			204	174	7	7	16	.92	1.00	.93	0.82 ±2.85

SR = success rate, P = precision, R = recall

Evaluation

- 100% precision: *PANINI never over-approximates!*
- 93% overall recall (100% for non-loop programs)

Type	Parser Example		#	Inferred Language =	⊂	∅	Error	SR	P	R	Time (s)
Straight	getAddrSpec	`[^<>]*<[^>]*>.*`	89	84	0	0	5	.94	1.00	1.00	0.16 ±0.27
+ Cond.	Figure 3	`a|[^a]b.*`	51	50	0	0	1	.98	1.00	1.00	0.09 ±0.05
+ Loops	lsb_check	`0*1`	64	40	7	7	10	.84	1.00	.76	2.32 ±4.77
			204	174	7	7	16	.92	1.00	.93	0.82 ±2.85

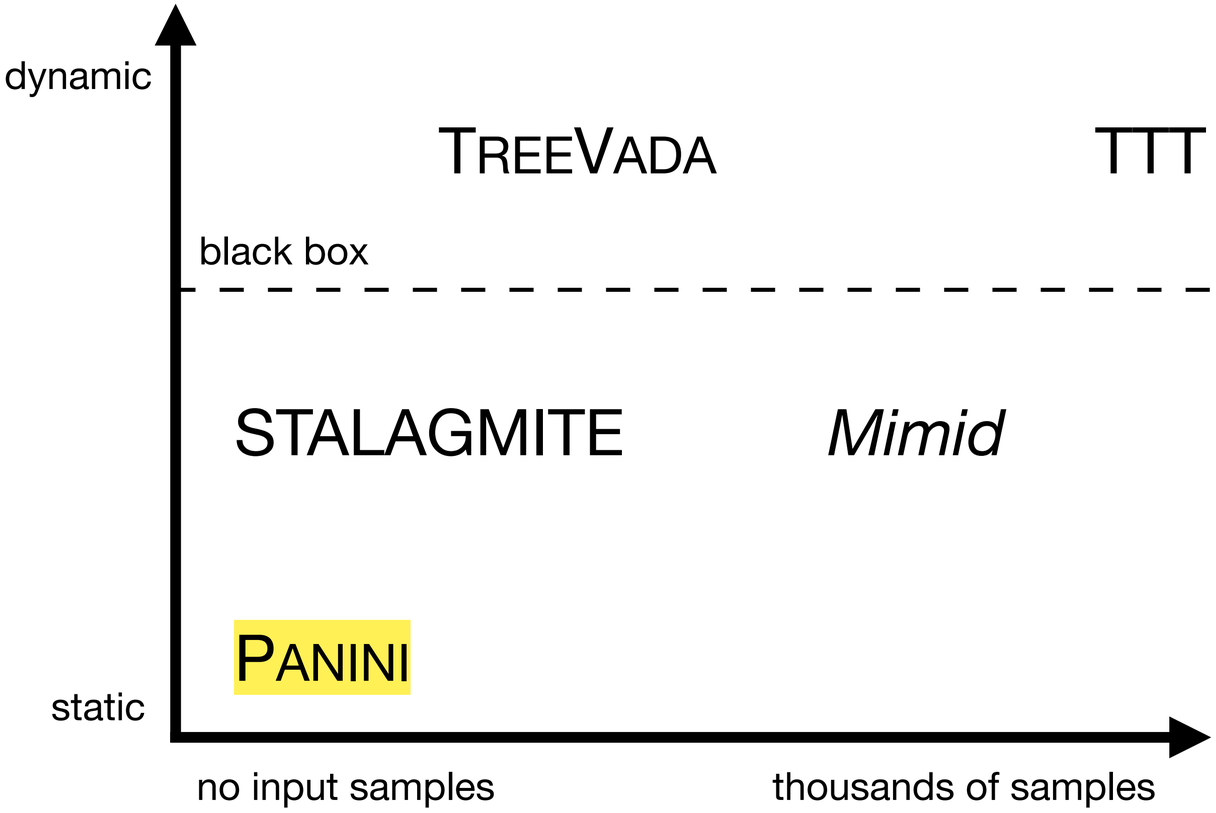
SR = success rate, P = precision, R = recall

insufficient invariants
weak abstract domains

Comparison with Other Approaches

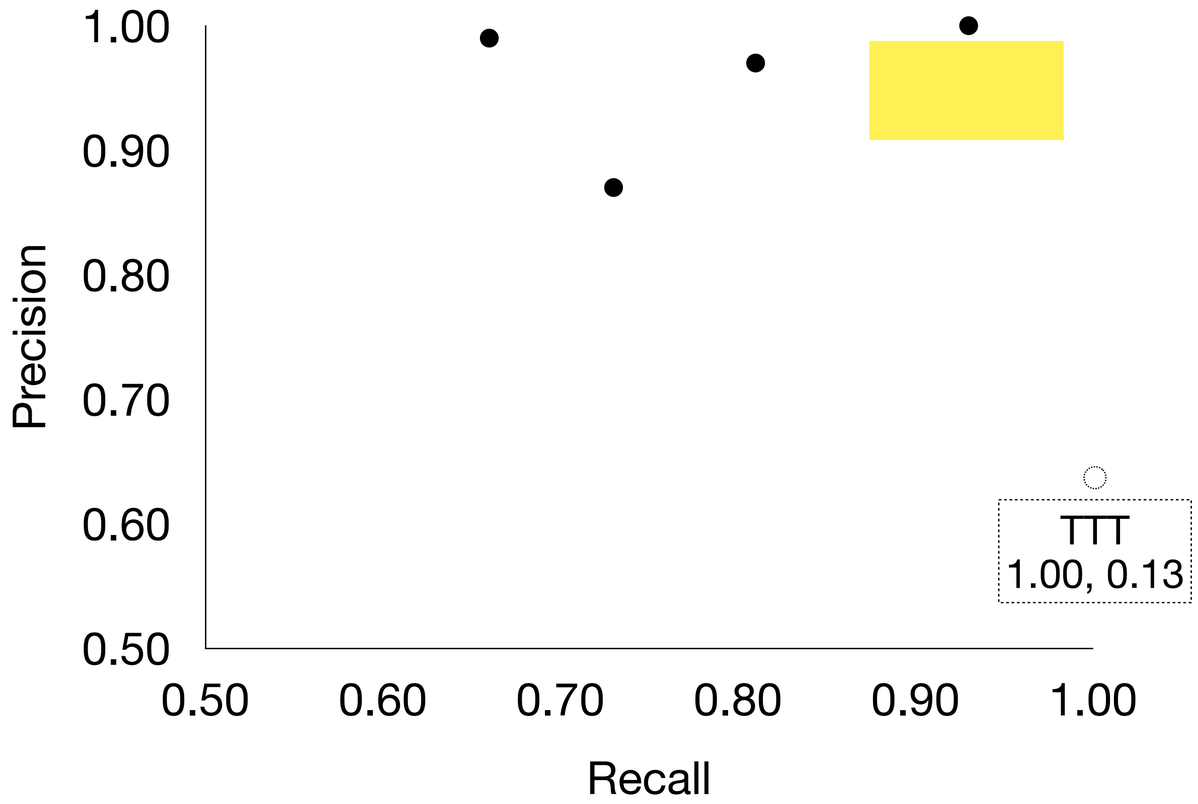
Panini is purely static and requires no pre-existing input samples

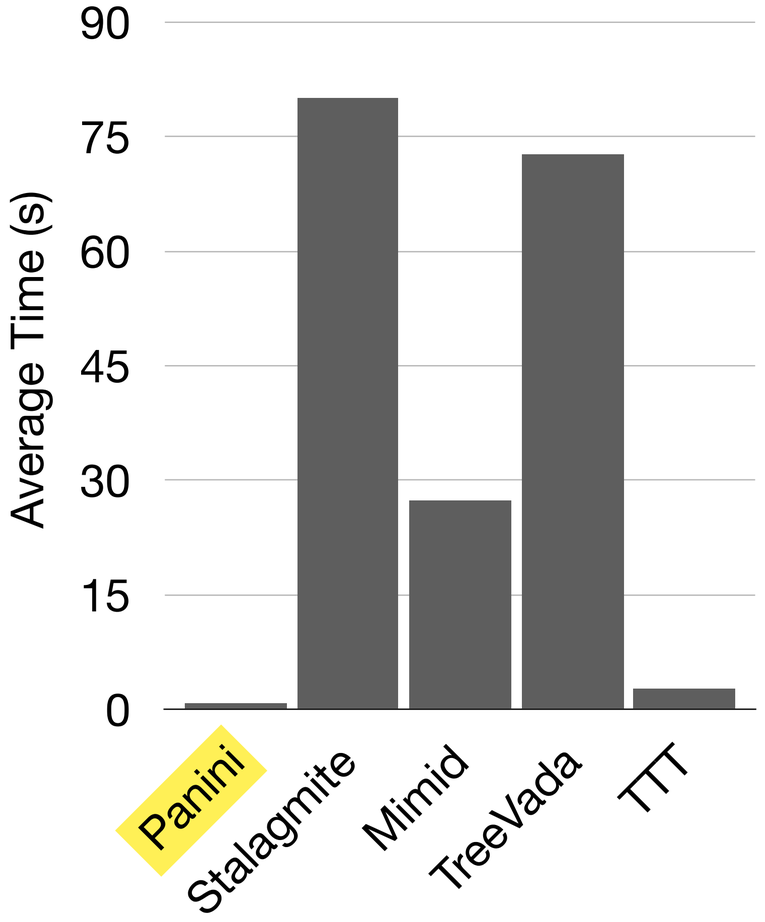
- STALAGMITE
  (Bettscheider and Zeller 2024)
- *Mimid*
  (Gopinath et al. 2020)
- TREEVADA
  (Arefin et al. 2024)
- TTT
  (Isberner 2015; Isberner et al. 2014)



Comparison with Other Approaches

Panini produces better grammars in less time





Future Work

Improved Front End

- Increase support for various Python features
- Add more source languages
- Systematic and trustworthy axiomatization of string functions

Extended IR

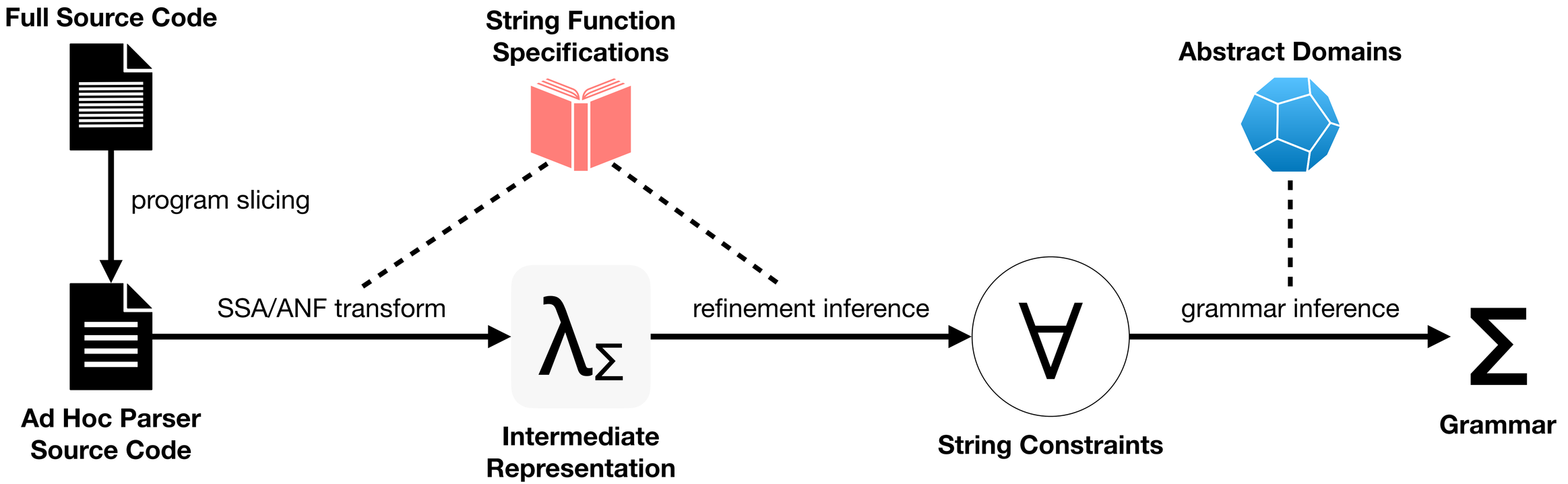
- ~~Add polymorphism and user-defined data types~~ (waiting to be merged)

Better Invariants

- Improve qualifier extraction heuristics
- Integrate external invariant generators
- Solve certain invariants using relational abstraction

Stronger Domains

- Support more complex numerical domains (active work in progress)
- Support super-regular languages




```python
def parser(s: str):
  if s[0] == "a":
    assert len(s) == 1
  else:
    assert s[1] == "b"
```

```
parser : {s:𝕊 | ?} → 𝟙
parser = λs:𝕊.
  let x = charAt s 0 in
  let p1 = eqChar x 'a' in
  if p1 then
    let n = length s in
    let p2 = eq n 1 in
```

$$(\forall v_1.\ v_1 = 0 \Rightarrow v_1 \geq 0 \wedge v_1 < |s|) \wedge$$
$$(\forall x.\ x = s[0] \Rightarrow$$
$$(\forall p_1.\ p_1 = \text{true} \Leftrightarrow x = \text{`a'} \Rightarrow$$
$$(p_1 = \text{true} \Rightarrow$$
$$(\forall n.\ n \geq 0 \wedge n = |s| \Rightarrow n = 1)) \wedge$$

$$a + (\Sigma \backslash a) b \Sigma^*$$

github.com/mcschroeder/panini