Approximation Semantics and Expressive Predicate Assignment for Object-Oriented Programming

R.N.S. Rowe and S.J. van Bakel

Department of Computing

Imperial College London

Summary

We study FJ$^{¢}$, Featherweight Java without casts.

FJ is a restriction of Java defined by removing all but the most essential features of the full language; FJ bears a similar relation to Java as the λ-calculus does to languages such as ML and Haskell.

We study two semantics for FJ:

- An *approximation* model;
- An *(intersection) type-based* model.

We define these because we want to define *semantics-based systems* of *abstract interpretation* for Java.

We will (in future work) extend FJ$^{¢}$ with the usual (imperative) features, working towards Java.

FJ$^{¢}$ Syntax

FJ$^{¢}$ programs P consist of a *class table* $\mathcal{CT}$, comprising the *class declarations*, and an *expression* `e` to be run (cf. the body of the `main` method in a real Java program). They are defined as a variant of Featherweight Java:

$FJ^{¢}$ Syntax

$FJ^{¢}$ programs P consist of a *class table* $\mathcal{CT}$, comprising the *class declarations*, and an *expression* e to be run (cf. the body of the `main` method in a real Java program). They are defined as a variant of Featherweight Java:

$$\mathtt{e} \quad ::= \; x \mid \mathtt{this} \mid \mathtt{new}\ \mathtt{C}(\vec{\mathtt{e}}) \mid \mathtt{e}.f \mid \mathtt{e}.m(\vec{\mathtt{e}})$$

FJ$^{¢}$ Syntax

FJ$^{¢}$ programs P consist of a *class table* $\mathcal{CT}$, comprising the *class declarations*, and an *expression* e to be run (cf. the body of the `main` method in a real Java program). They are defined as a variant of Featherweight Java:

$$\mathsf{e} \quad ::= \; x \mid \texttt{this} \mid \texttt{new}\ \mathsf{C}(\vec{\mathsf{e}}) \mid \mathsf{e}.f \mid \mathsf{e}.m(\vec{\mathsf{e}})$$

$$\mathsf{fd} \quad ::= \; \mathsf{C}\ f\texttt{;}$$

FJ$^{¢}$ Syntax

FJ$^{¢}$ programs P consist of a *class table* $\mathcal{CT}$, comprising the *class declarations*, and an *expression* e to be run (cf. the body of the `main` method in a real Java program). They are defined as a variant of Featherweight Java:

$$
\begin{array}{lll}
\mathsf{e} & ::= & x \mid \mathtt{this} \mid \mathtt{new}\ \mathsf{C}(\vec{\mathsf{e}}) \mid \mathsf{e}.f \mid \mathsf{e}.m(\vec{\mathsf{e}}) \\
\mathsf{fd} & ::= & \mathsf{C}\ f\mathtt{;} \\
\mathsf{md} & ::= & \mathsf{D}\ m(\mathsf{C}_1\ x_1,\ \ldots,\ \mathsf{C}_n\ x_n)\ \{\mathtt{return}\ \mathsf{e};\}
\end{array}
$$

FJ¢ Syntax

FJ¢ programs P consist of a *class table* $\mathcal{CT}$, comprising the *class declarations*, and an *expression* e to be run (cf. the body of the `main` method in a real Java program). They are defined as a variant of Featherweight Java:

$$
\begin{array}{lcl}
\mathsf{e} & ::= & x \mid \mathtt{this} \mid \mathtt{new}\ \mathsf{C}(\vec{\mathsf{e}}) \mid \mathsf{e}.f \mid \mathsf{e}.m(\vec{\mathsf{e}}) \\
\mathsf{fd} & ::= & \mathsf{C}\ f; \\
\mathsf{md} & ::= & \mathsf{D}\ m(\mathsf{C}_1\ x_1,\ \ldots,\ \mathsf{C}_n\ x_n)\ \{\mathtt{return}\ \mathsf{e};\} \\
\mathsf{cd} & ::= & \mathtt{class}\ \mathsf{C}\ \mathtt{extends}\ \mathsf{C}'\ \{\overrightarrow{\mathsf{fd}}\ \overrightarrow{\mathsf{md}}\} \qquad (\mathsf{C} \neq \mathtt{Object}) \\
\mathcal{CT} & ::= & \overrightarrow{\mathsf{cd}} \\
P & ::= & (\mathcal{CT}, \mathsf{e})
\end{array}
$$

FJ$^{¢}$ Syntax

FJ$^{¢}$ programs P consist of a *class table* $\mathcal{CT}$, comprising the *class declarations*, and an *expression* e to be run (cf. the body of the `main` method in a real Java program). They are defined as a variant of Featherweight Java:

$$
\begin{array}{lcl}
\mathsf{e} & ::= & x \mid \mathtt{this} \mid \mathtt{new}\ \mathsf{C}(\vec{\mathsf{e}}) \mid \mathsf{e}.f \mid \mathsf{e}.m(\vec{\mathsf{e}}) \\
\mathsf{fd} & ::= & \mathsf{C}\ f; \\
\mathsf{md} & ::= & \mathsf{D}\ m(\mathsf{C}_1\ x_1, \ldots, \mathsf{C}_n\ x_n)\ \{\mathtt{return}\ \mathsf{e};\} \\
\mathsf{cd} & ::= & \mathtt{class}\ \mathsf{C}\ \mathtt{extends}\ \mathsf{C}'\ \{\overrightarrow{\mathsf{fd}}\ \overrightarrow{\mathsf{md}}\} \qquad (\mathsf{C} \neq \mathtt{Object}) \\
\mathcal{CT} & ::= & \overrightarrow{\mathsf{cd}} \\
P & ::= & (\mathcal{CT}, \mathsf{e})
\end{array}
$$

1. The function $\mathcal{F}(\mathsf{C})$ returns the list of fields $\overrightarrow{f_n}$ belonging to class C (including those it inherits).
2. The function $\mathcal{M}b(\mathsf{C}, m)$ returns a tuple $(\vec{x}, \mathsf{e})$, consisting of a sequence of the method m's formal parameters and its body (as present in the class C).

Reduction

As usual, computation is modelled via a reduction relation on expressions.

Reduction

As usual, computation is modelled via a reduction relation on expressions.

- A *term substitution* $\mathsf{S} = \{\, x_1 \mapsto \mathsf{e}_1, \ldots, x_n \mapsto \mathsf{e}_n \,\}$ is defined as a map on expressions that replaces all occurrences of the variables x_i by the expressions e_i. We write e^{S} for $\mathsf{S}(\mathsf{e})$.

Reduction

As usual, computation is modelled via a reduction relation on expressions.

- A *term substitution* $\mathsf{S} = \{x_1 \mapsto \mathsf{e}_1, \ldots, x_n \mapsto \mathsf{e}_n\}$ is defined as a map on expressions that replaces all occurrences of the variables x_i by the expressions e_i. We write e^{S} for $\mathsf{S}(\mathsf{e})$.
- The reduction relation $\longrightarrow$ is defined as the contextual closure of:

Reduction

As usual, computation is modelled via a reduction relation on expressions.

- A *term substitution* $\mathsf{S} = \{ x_1 \mapsto \mathsf{e}_1, \ldots, x_n \mapsto \mathsf{e}_n \}$ is defined as a map on expressions that replaces all occurrences of the variables x_i by the expressions e_i. We write e^{S} for $\mathsf{S}(\mathsf{e})$.
- The reduction relation $\to$ is defined as the contextual closure of:
  - $\texttt{new}\ \mathsf{C}(\overrightarrow{\mathsf{e}_n}).f_i \to \mathsf{e}_i$, for class name C with $\mathcal{F}(\mathsf{C}) = \overrightarrow{f_n}$ and $i \in \overline{n}$.

Reduction

As usual, computation is modelled via a reduction relation on expressions.

- A *term substitution* $\mathsf{S} = \{\, x_1 \mapsto \mathsf{e}_1, \ldots, x_n \mapsto \mathsf{e}_n \,\}$ is defined as a map on expressions that replaces all occurrences of the variables x_i by the expressions e_i. We write e^{S} for $\mathsf{S}(\mathsf{e})$.
- The reduction relation $\rightarrow$ is defined as the contextual closure of:
  - $\texttt{new C}(\overrightarrow{\mathsf{e}_n}).f_i \rightarrow \mathsf{e}_i$, for class name C with $\mathcal{F}(\mathsf{C}) = \overrightarrow{f_n}$ and $i \in \overline{n}$.
  - $\texttt{new C}(\vec{\mathsf{e}}).m(\overrightarrow{\mathsf{e'}_n}) \rightarrow \mathsf{e}^{\mathsf{S}}$, where $\mathsf{S} = \{\, \texttt{this} \mapsto \texttt{new C}(\vec{\mathsf{e}}),\ x_1 \mapsto \mathsf{e'}_1, \ \ldots,\ x_n \mapsto \mathsf{e'}_n \,\}$, for class name C and method m with $\mathcal{M}b(\mathsf{C}, m) = (\overrightarrow{x_n}, \mathsf{e})$.

This notion of reduction is *confluent*.

Approximants for FJ$^{¢}$

Following the standard concept, we define the notion of approximant for FJ$^{¢}$ by replacing *(possibly) active computations* in expressions with $\bot$.

The set of *approximate normal forms*, $\mathbb{A}$, ranged over by A, is defined by the following grammar (extending expressions with $\bot$):

$$
\begin{array}{rcll}
\mathtt{A} & ::= & x \mid \mathtt{this} \mid \bot \mid \mathtt{new\ C}(\overrightarrow{\mathtt{A}_n}) & (n \geq 0) \\
 & \mid & \mathtt{A}.f \mid \mathtt{A}.m(\overline{\mathtt{A}}) & (\mathtt{A} \neq \bot, \mathtt{A} \neq \mathtt{new\ C}(\overrightarrow{\mathtt{A}_n}))
\end{array}
$$

Notice that all these expressions cannot be reduced - they are in *normal form*.

Approximants for $\mathsf{FJ}^{¢}$

Following the standard concept, we define the notion of approximant for $\mathsf{FJ}^{¢}$ by replacing *(possibly) active computations* in expressions with $\bot$.

The set of *approximate normal forms*, $\mathbb{A}$, ranged over by A, is defined by the following grammar (extending expressions with $\bot$):

$$\begin{array}{rcll} \mathsf{A} & ::= & x \mid \mathtt{this} \mid \bot \mid \mathtt{new}\ \mathsf{C}(\overrightarrow{\mathsf{A}_n}) & (n \geq 0) \\ & & \mid \mathsf{A}.f \mid \mathsf{A}.m(\overrightarrow{\mathsf{A}}) & (\mathsf{A} \neq \bot, \mathsf{A} \neq \mathtt{new}\ \mathsf{C}(\overrightarrow{\mathsf{A}_n})) \end{array}$$

Notice that all these expressions cannot be reduced - they are in *normal form*.

We consider $\bot.f$ not in normal form: it can be that $\bot$ hides an expression that reduces to an object $\mathtt{new}\ \mathsf{C}(\overrightarrow{\mathsf{A}_n})$, in which case the field invocation can run, so disappears.

Approximation Semantics

The *approximation relation* $\sqsubseteq$ is an order on terms defined by taking $\perp$ to be smaller (i.e. containing less information) than any other term, and by extending this to a relation between all terms:

$$\perp \sqsubseteq \mathsf{A}$$

$$\mathsf{A} \sqsubseteq \mathsf{A'} \;\&\; \forall i \leq n\ \mathsf{A}_i \sqsubseteq \mathsf{A'}_i \;\Rightarrow\; \begin{cases} \mathsf{A}.f & \sqsubseteq\ \mathsf{A'}.f \\ \texttt{new C}(\overrightarrow{\mathsf{A}_n}) & \sqsubseteq\ \texttt{new C}(\overrightarrow{\mathsf{A'}_n}) \\ \mathsf{A}.m(\overrightarrow{\mathsf{A}_n}) & \sqsubseteq\ \mathsf{A'}.m(\overrightarrow{\mathsf{A'}_n}) \end{cases}$$

Approximation Semantics

The *approximation relation* $\sqsubseteq$ is an order on terms defined by taking $\bot$ to be smaller (i.e. containing less information) than any other term, and by extending this to a relation between all terms:

$$\bot \sqsubseteq \mathsf{A}$$

$$\mathsf{A} \sqsubseteq \mathsf{A'} \;\&\; \forall i \leq n\ \mathsf{A}_i \sqsubseteq \mathsf{A'}_i \;\Rightarrow\; \begin{cases} \mathsf{A}.f & \sqsubseteq\ \mathsf{A'}.f \\ \texttt{new C}(\overrightarrow{\mathsf{A}_n}) & \sqsubseteq\ \texttt{new C}(\overrightarrow{\mathsf{A'}_n}) \\ \mathsf{A}.m(\overrightarrow{\mathsf{A}_n}) & \sqsubseteq\ \mathsf{A'}.m(\overrightarrow{\mathsf{A'}_n}) \end{cases}$$

The relationship between $\sqsubseteq$ and reduction is: If $\mathsf{A} \sqsubseteq \mathsf{e}$ and $\mathsf{e} \rightarrow^* \mathsf{e'}$, then $\mathsf{A} \sqsubseteq \mathsf{e'}$.

Approximation Semantics

The *approximation relation* $\sqsubseteq$ is an order on terms defined by taking $\perp$ to be smaller (i.e. containing less information) than any other term, and by extending this to a relation between all terms:

$$\perp \sqsubseteq \mathsf{A}$$

$$\mathsf{A} \sqsubseteq \mathsf{A'} \;\&\; \forall i \leq n\; \mathsf{A}_i \sqsubseteq \mathsf{A'}_i \;\Rightarrow\; \begin{cases} \mathsf{A}.f & \sqsubseteq \;\; \mathsf{A'}.f \\ \texttt{new C}(\overrightarrow{\mathsf{A}_n}) & \sqsubseteq \;\; \texttt{new C}(\overrightarrow{\mathsf{A'}_n}) \\ \mathsf{A}.m(\overrightarrow{\mathsf{A}_n}) & \sqsubseteq \;\; \mathsf{A'}.m(\overrightarrow{\mathsf{A'}_n}) \end{cases}$$

The relationship between $\sqsubseteq$ and reduction is: If $\mathsf{A} \sqsubseteq \mathsf{e}$ and $\mathsf{e} \to^* \mathsf{e'}$, then $\mathsf{A} \sqsubseteq \mathsf{e'}$.

We define the set of *approximants* of e as $\mathcal{A}(\mathsf{e}) = \{\, \mathsf{A} \mid \exists\, \mathsf{e'}\ [\mathsf{e} \to^* \mathsf{e'} \;\&\; \mathsf{A} \sqsubseteq \mathsf{e'}]\,\}$.

We can show: $\quad \mathsf{e} \to^* \mathsf{e'} \Rightarrow \mathcal{A}(\mathsf{e}) = \mathcal{A}(\mathsf{e'})$.

This result allows us to define a semantics for $\mathsf{FJ}^{¢}$ by interpreting expressions by the set of their approximants: $[\![\mathsf{e}]\!] = \mathcal{A}(\mathsf{e})$.

Predicates (vs. Java types)

Our predicates describe the capabilities of an expression in terms of

1. the *operations* that may be performed on it (i.e. accessing a field or invoking a method), and
2. the *outcome* of performing those operations.

They express detailed properties about the contexts in which expressions can be safely used.

Predicates (vs. Java types)

Our predicates describe the capabilities of an expression in terms of

1. the *operations* that may be performed on it (i.e. accessing a field or invoking a method), and
2. the *outcome* of performing those operations.

They express detailed properties about the contexts in which expressions can be safely used.

The set of *predicates* is defined by the following grammar:

$$\begin{array}{rcl} \phi,\psi & ::= & \omega \mid \sigma \mid \phi \cap \psi \\ \sigma & ::= & \varphi \mid \mathsf{C} \mid \langle f : \sigma \rangle \mid \langle m : (\phi_1, \ldots, \phi_n) \rightarrow \sigma \rangle \quad (n \geq 0) \end{array}$$

Predicates (vs. Java types)

Our predicates describe the capabilities of an expression in terms of

1. the *operations* that may be performed on it (i.e. accessing a field or invoking a method), and
2. the *outcome* of performing those operations.

They express detailed properties about the contexts in which expressions can be safely used.
The set of *predicates* is defined by the following grammar:

$$\phi,\psi \ ::= \ \omega \mid \sigma \mid \phi \cap \psi$$

$$\sigma \ ::= \ \varphi \mid \mathrm{C} \mid \langle f : \sigma \rangle \mid \langle m : (\phi_1, \ldots, \phi_n) \rightarrow \sigma \rangle \quad (n \geq 0)$$

The *predicate assignment* relation is expressed through the standard notation $\Pi \vdash \mathrm{e} : \phi$, where as usual Π contains type assumptions for the variables in e.

It is defined by ...

The Derivation Rules

$$(\text{VAR}):\ \frac{}{\Pi, x{:}\phi \vdash x : \sigma}\ (\phi \trianglelefteq \sigma) \qquad (\text{FLD}):\ \frac{\Pi \vdash \text{e} : \langle f : \sigma \rangle}{\Pi \vdash \text{e}.f : \sigma}$$

$$(\omega):\ \frac{}{\Pi \vdash \text{e} : \omega} \qquad (\text{JOIN}):\ \frac{\Pi \vdash \text{e} : \sigma_1 \ \ldots\ \Pi \vdash \text{e} : \sigma_n}{\Pi \vdash \text{e} : \sigma_1 \cap \ldots \cap \sigma_n}\ (n \geq 2)$$

$$(\text{INVK}):\ \frac{\Pi \vdash \text{e} : \langle m : (\phi_1, \ldots, \phi_n) \to \sigma \rangle \quad \Pi \vdash \text{e}_1 : \phi_1 \ \ldots\ \Pi \vdash \text{e}_n : \phi_n}{\Pi \vdash \text{e}.m(\overrightarrow{\text{e}_n}) : \sigma}$$

$$(\text{NEWO}):\ \frac{\Pi \vdash \text{e}_1 : \phi_1 \quad \ldots \quad \Pi \vdash \text{e}_n : \phi_n}{\Pi \vdash \texttt{new C}(\overrightarrow{\text{e}_n}) : \text{C}}\ (\mathcal{F}(\text{C}) = \overrightarrow{f_n})$$

$$(\text{NEWF}):\ \frac{\Pi \vdash \text{e}_1 : \phi_1 \quad \ldots \quad \Pi \vdash \text{e}_n : \phi_n}{\Pi \vdash \texttt{new C}(\overrightarrow{\text{e}_n}) : \langle f_i : \sigma \rangle}\ (\mathcal{F}(\text{C}) = \overrightarrow{f_n}, i \in \overline{n}, \phi_i = \sigma)$$

$$(\text{NEWM}):\ \frac{\texttt{this}{:}\psi, \overrightarrow{x{:}\phi_n} \vdash \text{e}_\text{b} : \sigma \quad \Pi \vdash \texttt{new C}(\vec{\text{e}}) : \psi}{\Pi \vdash \texttt{new C}(\vec{\text{e}}) : \langle m : (\overrightarrow{\phi_n}) \to \sigma \rangle}\ (\mathcal{M}b(\text{C}, m) = (\overrightarrow{x_n}, \text{e}_\text{b}))$$

The Derivation Rules

$$(\text{VAR}): \frac{}{\Pi, x{:}\phi \vdash x : \sigma} \ (\phi \trianglelefteq \sigma) \qquad (\text{FLD}): \frac{\Pi \vdash \mathsf{e} : \langle f : \sigma \rangle}{\Pi \vdash \mathsf{e}.f : \sigma}$$

$$(\omega): \frac{}{\Pi \vdash \mathsf{e} : \omega} \qquad (\text{JOIN}): \frac{\Pi \vdash \mathsf{e} : \sigma_1 \ \ldots \ \Pi \vdash \mathsf{e} : \sigma_n}{\Pi \vdash \mathsf{e} : \sigma_1 \cap \ldots \cap \sigma_n} \ (n \geq 2)$$

$$(\text{INVK}): \frac{\Pi \vdash \mathsf{e} : \langle m : (\phi_1, \ldots, \phi_n) \to \sigma \rangle \quad \Pi \vdash \mathsf{e}_1 : \phi_1 \ \ldots \ \Pi \vdash \mathsf{e}_n : \phi_n}{\Pi \vdash \mathsf{e}.m(\overrightarrow{\mathsf{e}_n}) : \sigma}$$

$$(\text{NEWO}): \frac{\Pi \vdash \mathsf{e}_1 : \phi_1 \quad \ldots \quad \Pi \vdash \mathsf{e}_n : \phi_n}{\Pi \vdash \texttt{new C}(\overrightarrow{\mathsf{e}_n}) : \mathsf{C}} \ (\mathcal{F}(\mathsf{C}) = \overrightarrow{f_n})$$

$$(\text{NEWF}): \frac{\Pi \vdash \mathsf{e}_1 : \phi_1 \quad \ldots \quad \Pi \vdash \mathsf{e}_n : \phi_n}{\Pi \vdash \texttt{new C}(\overrightarrow{\mathsf{e}_n}) : \langle f_i : \sigma \rangle} \ (\mathcal{F}(\mathsf{C}) = \overrightarrow{f_n}, i \in \overline{n}, \phi_i = \sigma)$$

$$(\text{NEWM}): \frac{\texttt{this}{:}\psi, \overrightarrow{x{:}\phi_n} \vdash \mathsf{e_b} : \sigma \quad \Pi \vdash \texttt{new C}(\vec{\mathsf{e}}) : \psi}{\Pi \vdash \texttt{new C}(\vec{\mathsf{e}}) : \langle m : (\overrightarrow{\phi_n}) \to \sigma \rangle} \ (\mathcal{M}b(\mathsf{C}, m) = (\overrightarrow{x_n}, \mathsf{e_b}))$$

The Derivation Rules

$$(\textsf{VAR}):\ \frac{}{\Pi, x{:}\phi \vdash x : \sigma}\ (\phi \trianglelefteq \sigma) \qquad (\textsf{FLD}):\ \frac{\Pi \vdash \mathsf{e} : \langle f : \sigma \rangle}{\Pi \vdash \mathsf{e}.f : \sigma}$$

$$(\omega):\ \frac{}{\Pi \vdash \mathsf{e} : \omega} \qquad (\textsf{JOIN}):\ \frac{\Pi \vdash \mathsf{e} : \sigma_1 \ \ldots\ \Pi \vdash \mathsf{e} : \sigma_n}{\Pi \vdash \mathsf{e} : \sigma_1 \cap \ldots \cap \sigma_n}\ (n \geq 2)$$

$$(\textsf{INVK}):\ \frac{\Pi \vdash \mathsf{e} : \langle m : (\phi_1, \ldots, \phi_n) \rightarrow \sigma \rangle \quad \Pi \vdash \mathsf{e}_1 : \phi_1 \ \ldots\ \Pi \vdash \mathsf{e}_n : \phi_n}{\Pi \vdash \mathsf{e}.m(\overrightarrow{\mathsf{e}_n}) : \sigma}$$

$$(\textsf{NEWO}):\ \frac{\Pi \vdash \mathsf{e}_1 : \phi_1 \quad \ldots \quad \Pi \vdash \mathsf{e}_n : \phi_n}{\Pi \vdash \mathtt{new}\ \mathsf{C}(\overrightarrow{\mathsf{e}_n}) : \mathsf{C}}\ (\mathcal{F}(\mathsf{C}) = \overrightarrow{f_n})$$

$$(\textsf{NEWF}):\ \frac{\Pi \vdash \mathsf{e}_1 : \phi_1 \quad \ldots \quad \Pi \vdash \mathsf{e}_n : \phi_n}{\Pi \vdash \mathtt{new}\ \mathsf{C}(\overrightarrow{\mathsf{e}_n}) : \langle f_i : \sigma \rangle}\ (\mathcal{F}(\mathsf{C}) = \overrightarrow{f_n}, i \in \overline{n}, \phi_i = \sigma)$$

$$(\textsf{NEWM}):\ \frac{\mathtt{this}{:}\psi, \overrightarrow{x{:}\phi_n} \vdash \mathsf{e}_b : \sigma \quad \Pi \vdash \mathtt{new}\ \mathsf{C}(\vec{\mathsf{e}}) : \psi}{\Pi \vdash \mathtt{new}\ \mathsf{C}(\vec{\mathsf{e}}) : \langle m : (\overrightarrow{\phi_n}) \rightarrow \sigma \rangle}\ (\mathcal{M}b(\mathsf{C}, m) = (\overrightarrow{x_n}, \mathsf{e}_b))$$

The Derivation Rules

$$(\text{VAR}): \frac{}{\Pi, x{:}\phi \vdash x : \sigma} \; (\phi \trianglelefteq \sigma) \qquad (\text{FLD}): \frac{\Pi \vdash \mathsf{e} : \langle f : \sigma \rangle}{\Pi \vdash \mathsf{e}.f : \sigma}$$

$$(\omega): \frac{}{\Pi \vdash \mathsf{e} : \omega} \qquad (\text{JOIN}): \frac{\Pi \vdash \mathsf{e} : \sigma_1 \; \ldots \; \Pi \vdash \mathsf{e} : \sigma_n}{\Pi \vdash \mathsf{e} : \sigma_1 \cap \ldots \cap \sigma_n} \; (n \geq 2)$$

$$(\text{INVK}): \frac{\Pi \vdash \mathsf{e} : \langle m : (\phi_1, \ldots, \phi_n) \to \sigma \rangle \quad \Pi \vdash \mathsf{e}_1 : \phi_1 \; \ldots \; \Pi \vdash \mathsf{e}_n : \phi_n}{\Pi \vdash \mathsf{e}.m(\overrightarrow{\mathsf{e}_n}) : \sigma}$$

$$(\text{NEWO}): \frac{\Pi \vdash \mathsf{e}_1 : \phi_1 \quad \ldots \quad \Pi \vdash \mathsf{e}_n : \phi_n}{\Pi \vdash \texttt{new } \mathsf{C}(\overrightarrow{\mathsf{e}_n}) : \mathsf{C}} \; (\mathcal{F}(\mathsf{C}) = \overrightarrow{f_n})$$

$$(\text{NEWF}): \frac{\Pi \vdash \mathsf{e}_1 : \phi_1 \quad \ldots \quad \Pi \vdash \mathsf{e}_n : \phi_n}{\Pi \vdash \texttt{new } \mathsf{C}(\overrightarrow{\mathsf{e}_n}) : \langle f_i : \sigma \rangle} \; (\mathcal{F}(\mathsf{C}) = \overrightarrow{f_n}, i \in \overline{n}, \phi_i = \sigma)$$

$$(\text{NEWM}): \frac{\texttt{this}{:}\psi, \overrightarrow{x{:}\phi_n} \vdash \mathsf{e_b} : \sigma \quad \Pi \vdash \texttt{new } \mathsf{C}(\vec{\mathsf{e}}) : \psi}{\Pi \vdash \texttt{new } \mathsf{C}(\vec{\mathsf{e}}) : \langle m : (\overrightarrow{\phi_n}) \to \sigma \rangle} \; (\mathcal{M}b(\mathsf{C}, m) = (\overrightarrow{x_n}, \mathsf{e_b}))$$

The Derivation Rules

$$(\textsc{var}): \frac{}{\Pi, x{:}\phi \vdash x : \sigma} \; (\phi \trianglelefteq \sigma) \qquad (\textsc{fld}): \frac{\Pi \vdash \mathsf{e} : \langle f : \sigma \rangle}{\Pi \vdash \mathsf{e}.f : \sigma}$$

$$(\omega): \frac{}{\Pi \vdash \mathsf{e} : \omega} \qquad (\textsc{join}): \frac{\Pi \vdash \mathsf{e} : \sigma_1 \; \ldots \; \Pi \vdash \mathsf{e} : \sigma_n}{\Pi \vdash \mathsf{e} : \sigma_1 \cap \ldots \cap \sigma_n} \; (n \geq 2)$$

$$(\textsc{invk}): \frac{\Pi \vdash \mathsf{e} : \langle m : (\phi_1, \ldots, \phi_n) \rightarrow \sigma \rangle \quad \Pi \vdash \mathsf{e}_1 : \phi_1 \; \ldots \; \Pi \vdash \mathsf{e}_n : \phi_n}{\Pi \vdash \mathsf{e}.m(\overrightarrow{\mathsf{e}_n}) : \sigma}$$

$$(\textsc{newO}): \frac{\Pi \vdash \mathsf{e}_1 : \phi_1 \quad \ldots \quad \Pi \vdash \mathsf{e}_n : \phi_n}{\Pi \vdash \texttt{new C}(\overrightarrow{\mathsf{e}_n}) : \mathsf{C}} \; (\mathcal{F}(\mathsf{C}) = \overrightarrow{f_n})$$

$$(\textsc{newF}): \frac{\Pi \vdash \mathsf{e}_1 : \phi_1 \quad \ldots \quad \Pi \vdash \mathsf{e}_n : \phi_n}{\Pi \vdash \texttt{new C}(\overrightarrow{\mathsf{e}_n}) : \langle f_i : \sigma \rangle} \; (\mathcal{F}(\mathsf{C}) = \overrightarrow{f}_n, i \in \overline{n}, \phi_i = \sigma)$$

$$(\textsc{newM}): \frac{\texttt{this}{:}\psi, \overrightarrow{x{:}\phi_n} \vdash \mathsf{e_b} : \sigma \quad \Pi \vdash \texttt{new C}(\vec{\mathsf{e}}) : \psi}{\Pi \vdash \texttt{new C}(\vec{\mathsf{e}}) : \langle m : (\overrightarrow{\phi_n}) \rightarrow \sigma \rangle} \; (\mathcal{M}b(\mathsf{C}, m) = (\overrightarrow{x_n}, \mathsf{e_b}))$$

The Derivation Rules

$$(\textsf{VAR}): \frac{}{\Pi, x{:}\phi \vdash x : \sigma} \; (\phi \trianglelefteq \sigma) \qquad (\textsf{FLD}): \frac{\Pi \vdash \textsf{e} : \langle f : \sigma \rangle}{\Pi \vdash \textsf{e}.f : \sigma}$$

$$(\omega): \frac{}{\Pi \vdash \textsf{e} : \omega} \qquad (\textsf{JOIN}): \frac{\Pi \vdash \textsf{e} : \sigma_1 \;\ldots\; \Pi \vdash \textsf{e} : \sigma_n}{\Pi \vdash \textsf{e} : \sigma_1 \cap \ldots \cap \sigma_n} \; (n \geq 2)$$

$$(\textsf{INVK}): \frac{\Pi \vdash \textsf{e} : \langle m : (\phi_1, \ldots, \phi_n) \to \sigma \rangle \quad \Pi \vdash \textsf{e}_1 : \phi_1 \;\ldots\; \Pi \vdash \textsf{e}_n : \phi_n}{\Pi \vdash \textsf{e}.m(\overrightarrow{\textsf{e}_n}) : \sigma}$$

$$(\textsf{NEWO}): \frac{\Pi \vdash \textsf{e}_1 : \phi_1 \quad \ldots \quad \Pi \vdash \textsf{e}_n : \phi_n}{\Pi \vdash \texttt{new C}(\overrightarrow{\textsf{e}_n}) : \textsf{C}} \; (\mathcal{F}(\textsf{C}) = \overrightarrow{f_n})$$

$$(\textsf{NEWF}): \frac{\Pi \vdash \textsf{e}_1 : \phi_1 \quad \ldots \quad \Pi \vdash \textsf{e}_n : \phi_n}{\Pi \vdash \texttt{new C}(\overrightarrow{\textsf{e}_n}) : \langle f_i : \sigma \rangle} \; (\mathcal{F}(\textsf{C}) = \overrightarrow{f_n}, i \in \overline{n}, \phi_i = \sigma)$$

$$(\textsf{NEWM}): \frac{\texttt{this}{:}\psi, \overrightarrow{x{:}\phi_n} \vdash \textsf{e}_\textsf{b} : \sigma \quad \Pi \vdash \texttt{new C}(\vec{\textsf{e}}) : \psi}{\Pi \vdash \texttt{new C}(\vec{\textsf{e}}) : \langle m : (\overrightarrow{\phi_n}) \to \sigma \rangle} \; (\mathcal{M}b(\textsf{C}, m) = (\overrightarrow{x_n}, \textsf{e}_\textsf{b}))$$

The Derivation Rules

$$(\text{VAR}):\ \frac{}{\Pi, x{:}\phi \vdash x : \sigma}\ (\phi \trianglelefteq \sigma) \qquad (\text{FLD}):\ \frac{\Pi \vdash \mathsf{e} : \langle f : \sigma \rangle}{\Pi \vdash \mathsf{e}.f : \sigma}$$

$$(\omega):\ \frac{}{\Pi \vdash \mathsf{e} : \omega} \qquad (\text{JOIN}):\ \frac{\Pi \vdash \mathsf{e} : \sigma_1 \ \ \ldots\ \ \Pi \vdash \mathsf{e} : \sigma_n}{\Pi \vdash \mathsf{e} : \sigma_1 \cap \ldots \cap \sigma_n}\ (n \geq 2)$$

$$(\text{INVK}):\ \frac{\Pi \vdash \mathsf{e} : \langle m : (\phi_1, \ldots, \phi_n) \rightarrow \sigma \rangle \quad \Pi \vdash \mathsf{e}_1 : \phi_1 \ \ldots\ \Pi \vdash \mathsf{e}_n : \phi_n}{\Pi \vdash \mathsf{e}.m(\overrightarrow{\mathsf{e}_n}) : \sigma}$$

$$(\text{NEWO}):\ \frac{\Pi \vdash \mathsf{e}_1 : \phi_1 \quad \ldots \quad \Pi \vdash \mathsf{e}_n : \phi_n}{\Pi \vdash \texttt{new}\ \mathsf{C}(\overrightarrow{\mathsf{e}_n}) : \mathsf{C}}\ (\mathcal{F}(\mathsf{C}) = \overrightarrow{f_n})$$

$$(\text{NEWF}):\ \frac{\Pi \vdash \mathsf{e}_1 : \phi_1 \quad \ldots \quad \Pi \vdash \mathsf{e}_n : \phi_n}{\Pi \vdash \texttt{new}\ \mathsf{C}(\overrightarrow{\mathsf{e}_n}) : \langle f_i : \sigma \rangle}\ (\mathcal{F}(\mathsf{C}) = \overrightarrow{f_n}, i \in \overline{n}, \phi_i = \sigma)$$

$$(\text{NEWM}):\ \frac{\texttt{this}{:}\psi, \overrightarrow{x{:}\phi_n} \vdash \mathsf{e_b} : \sigma \quad \Pi \vdash \texttt{new}\ \mathsf{C}(\vec{\mathsf{e}}) : \psi}{\Pi \vdash \texttt{new}\ \mathsf{C}(\vec{\mathsf{e}}) : \langle m : (\overrightarrow{\phi_n}) \rightarrow \sigma \rangle}\ (\mathcal{M}b(\mathsf{C}, m) = (\overrightarrow{x_n}, \mathsf{e_b}))$$

The Derivation Rules

$$(\text{VAR}):\ \frac{}{\Pi, x{:}\phi \vdash x : \sigma}\ (\phi \trianglelefteq \sigma) \qquad (\text{FLD}):\ \frac{\Pi \vdash \mathsf{e} : \langle f : \sigma \rangle}{\Pi \vdash \mathsf{e}.f : \sigma}$$

$$(\omega):\ \frac{}{\Pi \vdash \mathsf{e} : \omega} \qquad (\text{JOIN}):\ \frac{\Pi \vdash \mathsf{e} : \sigma_1 \ \ldots\ \Pi \vdash \mathsf{e} : \sigma_n}{\Pi \vdash \mathsf{e} : \sigma_1 \cap \ldots \cap \sigma_n}\ (n \geq 2)$$

$$(\text{INVK}):\ \frac{\Pi \vdash \mathsf{e} : \langle m : (\phi_1, \ldots, \phi_n) \to \sigma \rangle \quad \Pi \vdash \mathsf{e}_1 : \phi_1 \ \ldots\ \Pi \vdash \mathsf{e}_n : \phi_n}{\Pi \vdash \mathsf{e}.m(\overrightarrow{\mathsf{e}_n}) : \sigma}$$

$$(\text{NEWO}):\ \frac{\Pi \vdash \mathsf{e}_1 : \phi_1 \quad \ldots \quad \Pi \vdash \mathsf{e}_n : \phi_n}{\Pi \vdash \texttt{new C}(\overrightarrow{\mathsf{e}_n}) : \mathsf{C}}\ (\mathcal{F}(\mathsf{C}) = \overrightarrow{f_n})$$

$$(\text{NEWF}):\ \frac{\Pi \vdash \mathsf{e}_1 : \phi_1 \quad \ldots \quad \Pi \vdash \mathsf{e}_n : \phi_n}{\Pi \vdash \texttt{new C}(\overrightarrow{\mathsf{e}_n}) : \langle f_i : \sigma \rangle}\ (\mathcal{F}(\mathsf{C}) = \overrightarrow{f_n}, i \in \overline{n}, \phi_i = \sigma)$$

$$(\text{NEWM}):\ \frac{\texttt{this}{:}\psi, \overrightarrow{x{:}\phi_n} \vdash \mathsf{e}_{\mathsf{b}} : \sigma \quad \Pi \vdash \texttt{new C}(\vec{\mathsf{e}}) : \psi}{\Pi \vdash \texttt{new C}(\vec{\mathsf{e}}) : \langle m : (\overrightarrow{\phi_n}) \to \sigma \rangle}\ (\mathcal{M}b(\mathsf{C}, m) = (\overrightarrow{x_n}, \mathsf{e}_{\mathsf{b}}))$$

The Derivation Rules

$$(\textsf{VAR}): \frac{}{\Pi, x{:}\phi \vdash x : \sigma} \; (\phi \trianglelefteq \sigma) \qquad (\textsf{FLD}): \frac{\Pi \vdash \mathsf{e} : \langle f : \sigma \rangle}{\Pi \vdash \mathsf{e}.f : \sigma}$$

$$(\omega): \frac{}{\Pi \vdash \mathsf{e} : \omega} \qquad (\textsf{JOIN}): \frac{\Pi \vdash \mathsf{e} : \sigma_1 \;\ldots\; \Pi \vdash \mathsf{e} : \sigma_n}{\Pi \vdash \mathsf{e} : \sigma_1 \cap \ldots \cap \sigma_n} \; (n \geq 2)$$

$$(\textsf{INVK}): \frac{\Pi \vdash \mathsf{e} : \langle m : (\phi_1, \ldots, \phi_n) \rightarrow \sigma \rangle \quad \Pi \vdash \mathsf{e}_1 : \phi_1 \;\ldots\; \Pi \vdash \mathsf{e}_n : \phi_n}{\Pi \vdash \mathsf{e}.m(\overrightarrow{\mathsf{e}_n}) : \sigma}$$

$$(\textsf{NEWO}): \frac{\Pi \vdash \mathsf{e}_1 : \phi_1 \quad \ldots \quad \Pi \vdash \mathsf{e}_n : \phi_n}{\Pi \vdash \texttt{new C}(\overrightarrow{\mathsf{e}_n}) : \mathsf{C}} \; (\mathcal{F}(\mathsf{C}) = \overrightarrow{f_n})$$

$$(\textsf{NEWF}): \frac{\Pi \vdash \mathsf{e}_1 : \phi_1 \quad \ldots \quad \Pi \vdash \mathsf{e}_n : \phi_n}{\Pi \vdash \texttt{new C}(\overrightarrow{\mathsf{e}_n}) : \langle f_i : \sigma \rangle} \; (\mathcal{F}(\mathsf{C}) = \overrightarrow{f_n}, i \in \overline{n}, \phi_i = \sigma)$$

$$(\textsf{NEWM}): \frac{\texttt{this}{:}\psi, \overrightarrow{x{:}\phi_n} \vdash \mathsf{e_b} : \sigma \quad \Pi \vdash \texttt{new C}(\vec{\mathsf{e}}) : \psi}{\Pi \vdash \texttt{new C}(\vec{\mathsf{e}}) : \langle m : (\overrightarrow{\phi_n}) \rightarrow \sigma \rangle} \; (\mathcal{M}b(\mathsf{C}, m) = (\overrightarrow{x_n}, \mathsf{e_b}))$$

We show: if $\mathsf{e} \rightarrow \mathsf{e'}$ then $\Pi \vdash \mathsf{e'} : \phi \Leftrightarrow \Pi \vdash \mathsf{e} : \phi$; this gives a type-based semantics.

The Approximation Result

We now have two approaches to semantics: (1) the approximation model, and (2) a type based semantics. So what is the relation between the two?

The Approximation Result

We now have two approaches to semantics: (1) the approximation model, and (2) a type based semantics. So what is the relation between the two?

We aim to show: $\Pi \vdash \mathsf{e} : \phi \Leftrightarrow \exists \mathsf{A} \in \mathcal{A}(\mathsf{e}) \, [\, \Pi \vdash \mathsf{A} : \phi \,]$.

The Approximation Result

We now have two approaches to semantics: (1) the approximation model, and (2) a type based semantics. So what is the relation between the two?

We aim to show: $\Pi \vdash e : \phi \Leftrightarrow \exists A \in \mathcal{A}(e)\ [\ \Pi \vdash A : \phi\ ]$.

- If an expression e can be given a predicate ϕ, then there exists an approximant A of e that has the same predicate. i.e: if we run e long enough, a stable result will appear (with perhaps computations still going on inside) that has the same predicate ϕ: predicates 'foretell' the shape of the result.

The Approximation Result

We now have two approaches to semantics: (1) the approximation model, and (2) a type based semantics. So what is the relation between the two?

We aim to show: $\Pi \vdash \mathsf{e} : \phi \Leftrightarrow \exists \mathsf{A} \in \mathcal{A}(\mathsf{e}) \, [\, \Pi \vdash \mathsf{A} : \phi \,]$.

- If an expression e can be given a predicate ϕ, then there exists an approximant A of e that has the same predicate. i.e: if we run e long enough, a stable result will appear (with perhaps computations still going on inside) that has the same predicate ϕ: predicates 'foretell' the shape of the result.
- If we can give A a predicate ϕ, then we can give that predicate to *all* expressions that have A as an approximant.

The Approximation Result

We now have two approaches to semantics: (1) the approximation model, and (2) a type based semantics. So what is the relation between the two?

We aim to show: $\Pi \vdash \mathsf{e} : \phi \Leftrightarrow \exists \mathsf{A} \in \mathcal{A}(\mathsf{e}) \, [\, \Pi \vdash \mathsf{A} : \phi \,]$.

- If an expression e can be given a predicate ϕ, then there exists an approximant A of e that has the same predicate. i.e: if we run e long enough, a stable result will appear (with perhaps computations still going on inside) that has the same predicate ϕ: predicates 'foretell' the shape of the result.
- If we can give A a predicate ϕ, then we can give that predicate to *all* expressions that have A as an approximant.

The proof for this property is not straightforward; the main problem is that reduction in $\mathsf{FJ}^{¢}$ is *weak*. To solve it, we use a notion of *derivation reduction* that follows reduction of expressions on, but reduces only those that do not have predicate ω.

Derivation Reduction

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}_1\\ \Pi \vdash \mathtt{e}_1 : \phi_1\end{array} \quad \ldots \quad \begin{array}{c}\mathcal{D}_n\\ \Pi \vdash \mathtt{e}_n : \phi_n\end{array}}{\Pi \vdash \mathtt{new}\ \mathtt{C}(\overrightarrow{\mathtt{e}_n}) : \langle f_i : \sigma \rangle}}{\Pi \vdash \mathtt{new}\ \mathtt{C}(\overrightarrow{\mathtt{e}_n}).f_i : \sigma} \quad \rightarrow_{\mathfrak{D}} \quad \begin{array}{c}\mathcal{D}_i\\ \Pi \vdash \mathtt{e}_i : \sigma\end{array}$$

Notice that $\mathtt{new}\ \mathtt{C}(\overrightarrow{\mathtt{e}_n}).f_i \rightarrow \mathtt{e}_i$. We also contract:

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}_b\\ \mathtt{this}{:}\psi, x_1{:}\phi_1, \ldots, x_n{:}\phi_n \vdash \mathtt{e}_b : \sigma\end{array} \quad \begin{array}{c}\mathcal{D}_{\mathsf{self}}\\ \Pi \vdash \mathtt{new}\ \mathtt{C}(\overrightarrow{\mathtt{e}_n}) : \psi\end{array}}{\Pi \vdash \mathtt{new}\ \mathtt{C}(\overrightarrow{\mathtt{e}_n}) : \langle m : (\overrightarrow{\phi_n}) \rightarrow \sigma \rangle} \quad \begin{array}{c}\mathcal{D}_1\\ \Pi \vdash \mathtt{e}_1 : \phi_1\end{array} \quad \ldots \quad \begin{array}{c}\mathcal{D}_n\\ \Pi \vdash \mathtt{e}_n : \phi_n\end{array}}{\Pi \vdash \mathtt{new}\ \mathtt{C}(\overrightarrow{\mathtt{e}}).m(\overrightarrow{\mathtt{e}_n}) : \sigma} \quad \rightarrow_{\mathfrak{D}} \quad \begin{array}{c}{\mathcal{D}_b}^{\mathcal{S}}\\ \Pi \vdash {\mathtt{e}_b}^{\mathsf{S}} : \sigma\end{array}$$

and $\mathtt{new}\ \mathtt{C}(\overrightarrow{\mathtt{e}}).m(\overrightarrow{\mathtt{e}_n}) \rightarrow \mathtt{e}_b$, where $\mathtt{D}\ m(\overrightarrow{\mathtt{E}\ x_n})\{\mathtt{return}\ \mathtt{e}_b;\}$ is present in $\mathtt{C}$.

Derivation Reduction is Strongly Normalising

First we show that derivation reduction is sound: If $\mathcal{D} :: \Pi \vdash \mathsf{e} : \phi$ and $\mathcal{D} \rightarrow_{\mathfrak{D}} \mathcal{D}'$, then $\mathcal{D}'$ is a well-defined derivation, in that is there exists some e' such that $\mathcal{D}' :: \Pi \vdash \mathsf{e'} : \phi$, and $\mathsf{e} \rightarrow \mathsf{e'}$.

Derivation Reduction is Strongly Normalising

First we show that derivation reduction is sound: If $\mathcal{D} :: \Pi \vdash \mathsf{e} : \phi$ and $\mathcal{D} \rightarrow_{\mathfrak{D}} \mathcal{D}'$, then $\mathcal{D}'$ is a well-defined derivation, in that is there exists some e' such that $\mathcal{D}' :: \Pi \vdash \mathsf{e'} : \phi$, and $\mathsf{e} \rightarrow \mathsf{e'}$.

We show that all reductions on derivations *terminate*.

$$\mathcal{D} :: \Pi \vdash \mathsf{e} : \phi \Rightarrow \mathsf{SN}(\mathcal{D}).$$

This implies that ϕ (or, more precisely, $\mathcal{D}$) has only a *finite amount of information* on the execution of e.

Derivation Reduction is Strongly Normalising

First we show that derivation reduction is sound: If $\mathcal{D} :: \Pi \vdash \mathsf{e} : \phi$ and $\mathcal{D} \rightarrow_{\mathfrak{D}} \mathcal{D}'$, then $\mathcal{D}'$ is a well-defined derivation, in that is there exists some e' such that $\mathcal{D}' :: \Pi \vdash \mathsf{e'} : \phi$, and $\mathsf{e} \rightarrow \mathsf{e'}$.

We show that all reductions on derivations *terminate*.

$$\mathcal{D} :: \Pi \vdash \mathsf{e} : \phi \Rightarrow \mathsf{SN}(\mathcal{D}).$$

This implies that ϕ (or, more precisely, $\mathcal{D}$) has only a *finite amount of information* on the execution of e.

In fact, if ω has been used in $\mathcal{D}$, then all executions inside the expression that has this predicate are *invisible* and will not be executed when reducing the derivation.

So in the normal form of a derivation, the expression involved need not be in normal form.

The proof for this property is not straightforward, since typeable terms need not terminate - we use the *computability* technique (Tait).

Main results

From the termination result for derivation reduction, these others follow:

Approximation result: $\Pi \vdash \mathsf{e} : \phi \Leftrightarrow \exists \mathsf{A} \in \mathcal{A}(\mathsf{e})\ [\ \Pi \vdash \mathsf{A} : \phi\ ]$.

Main results

From the termination result for derivation reduction, these others follow:

Approximation result: $\Pi \vdash \mathsf{e} : \phi \Leftrightarrow \exists \mathsf{A} \in \mathcal{A}(\mathsf{e}) \; [\, \Pi \vdash \mathsf{A} : \phi \,]$.

Head Normalisation: $\Pi \vdash \mathsf{e} : \sigma$ if and only if e has a *head-normal form*.

This is very close to the approximation result: the head-normal form H corresponds to the shape of the approximant, which is given by replacing all subexpressions typed with ω in the derivation's *normal form* by $\bot$.

Main results

From the termination result for derivation reduction, these others follow:

Approximation result : $\Pi \vdash \mathsf{e} : \phi \Leftrightarrow \exists\, \mathsf{A} \in \mathcal{A}(\mathsf{e})\ [\, \Pi \vdash \mathsf{A} : \phi \,]$.

Head Normalisation : $\Pi \vdash \mathsf{e} : \sigma$ if and only if e has a *head-normal form*.

This is very close to the approximation result: the head-normal form H corresponds to the shape of the approximant, which is given by replacing all subexpressions typed with ω in the derivation's *normal form* by $\bot$.

Normalisation : $\mathcal{D} :: \Pi \vdash \mathsf{e} : \sigma$ with ω-safe $\mathcal{D}$ and Π only if e has a *normal form*.

If ω is used in $\mathcal{D}$ only for arguments in method invocations, then it is possible to run e to a *result*.

Main results

From the termination result for derivation reduction, these others follow:

Approximation result : $\Pi \vdash \mathsf{e} : \phi \Leftrightarrow \exists \mathsf{A} \in \mathcal{A}(\mathsf{e}) \; [\, \Pi \vdash \mathsf{A} : \phi \,]$.

Head Normalisation : $\Pi \vdash \mathsf{e} : \sigma$ if and only if e has a *head-normal form*.

This is very close to the approximation result: the head-normal form H corresponds to the shape of the approximant, which is given by replacing all subexpressions typed with ω in the derivation's *normal form* by $\bot$.

Normalisation : $\mathcal{D} :: \Pi \vdash \mathsf{e} : \sigma$ with ω-safe $\mathcal{D}$ and Π only if e has a *normal form*.

If ω is used in $\mathcal{D}$ only for arguments in method invocations, then it is possible to run e to a *result*.

Strong Normalisation : $\mathcal{D} :: \Pi \vdash \mathsf{e} : \sigma$ with $\mathcal{D}$ strong if and only if e is *strongly normalisable*.

If ω is *not used at all* in $\mathcal{D}$, then *all* executions of e will produce a result.

Expressivity

We compare our work to previous results (and show that FJ$^{¢}$ is Turing complete) by considering an encoding of the **SK** Combinatory Logic (CL) in FJ$^{¢}$:

$$\begin{array}{lcl} \mathbf{K}\, x\, y & \rightarrow & x \\ \mathbf{S}\, x\, y\, z & \rightarrow & x\, z\, (\, y\, z\,) \end{array}$$

Expressivity

We compare our work to previous results (and show that $FJ^{¢}$ is Turing complete) by considering an encoding of the **SK** Combinatory Logic (CL) in $FJ^{¢}$:

$$\begin{array}{lcl} \mathbf{K}\,x\,y & \to & x \\ \mathbf{S}\,x\,y\,z & \to & x\,z\,(\,y\,z\,) \end{array}$$

The encoding of CL into the $FJ^{¢}$ program OOCL (Object-Oriented CL) is defined using the class table on the next slide and the function $[\![\cdot]\!]$ which translates terms of CL into $FJ^{¢}$ expressions, and is defined as follows:

$$\begin{array}{lcllcl} [\![x]\!] & = & x & [\![t_1 t_2]\!] & = & [\![t_1]\!]\texttt{.app(}[\![t_2]\!]\texttt{)} \\ [\![\mathbf{K}]\!] & = & \texttt{new K}_0\texttt{()} & [\![\mathbf{S}]\!] & = & \texttt{new S}_0\texttt{()} \end{array}$$

If $\mathrm{t}_1 \to \mathrm{t}_2$ in CL, then $[\![\mathrm{t}_1]\!] \to [\![\mathrm{t}_2]\!]$ in $FJ^{¢}$.

OOCL

```
class Combinator extends Object {
      Combinator app(Combinator x) { return this; } }

class K0 extends Combinator {
      Combinator app(Combinator x) { return new K1(x); } }
class K1 extends K0 { Combinator x;
      Combinator app(Combinator y) { return this.x; } }

class S0 extends Combinator {
      Combinator app(Combinator x) { return new S1(x); } }
class S1 extends S0 { Combinator x;
      Combinator app(Combinator y) { return new S2(this.x, y); } }
class S2 extends S1 { Combinator y;
      Combinator app(Combinator z) {
          return this.x.app(z).app(this.y.app(z)); } }
```

Type preservation

The set of *simple types* is defined by $\tau ::= \varphi \mid \tau{\to}\tau$. The *basis* B contains type assumptions for variables. Simple types are assigned to CL-terms using:

$$(\text{VAR}):\ \frac{}{B \vdash_{CL} x{:}\tau}\ (x{:}\tau \in B) \qquad (\to\text{E}):\ \frac{B \vdash_{CL} t_1{:}\tau{\to}\tau' \quad B \vdash_{CL} t_2{:}\tau}{B \vdash_{CL} t_1 t_2{:}\tau'}$$

$$(\mathbf{K}):\ \frac{}{B \vdash_{CL} \mathbf{K}{:}\tau{\to}\tau'{\to}\tau} \qquad (\mathbf{S}):\ \frac{}{B \vdash_{CL} \mathbf{S}{:}(\tau{\to}\tau'{\to}\tau''){\to}(\tau{\to}\tau'){\to}\tau{\to}\tau''}$$

Type preservation

The set of *simple types* is defined by $\tau ::= \varphi \mid \tau{\rightarrow}\tau$. The *basis* B contains type assumptions for variables. Simple types are assigned to CL-terms using:

$$(\text{VAR}): \frac{}{B \vdash_{CL} x{:}\tau}\,(x{:}\tau \in B) \qquad ({\rightarrow}E): \frac{B \vdash_{CL} t_1{:}\tau{\rightarrow}\tau' \quad B \vdash_{CL} t_2{:}\tau}{B \vdash_{CL} t_1 t_2{:}\tau'}$$

$$(\mathbf{K}): \frac{}{B \vdash_{CL} \mathbf{K}{:}\tau{\rightarrow}\tau'{\rightarrow}\tau} \qquad (\mathbf{S}): \frac{}{B \vdash_{CL} \mathbf{S}{:}(\tau{\rightarrow}\tau'{\rightarrow}\tau''){\rightarrow}(\tau{\rightarrow}\tau'){\rightarrow}\tau{\rightarrow}\tau''}$$

We define what the equivalent of Curry's types are in terms of predicates.

$$\begin{aligned} [\![\varphi]\!] &= \varphi \\ [\![\tau{\rightarrow}\tau']\!] &= \langle \mathtt{app}{:}[\![\tau]\!]{\rightarrow}[\![\tau']\!]\rangle \end{aligned}$$

Type preservation

The set of *simple types* is defined by $\tau ::= \varphi \mid \tau{\to}\tau$. The *basis* B contains type assumptions for variables. Simple types are assigned to CL-terms using:

$$(\mathrm{VAR}):\ \frac{}{\mathsf{B} \vdash_{\mathsf{CL}} x{:}\tau}\ (x{:}\tau \in \mathsf{B}) \qquad ({\to}\mathrm{E}):\ \frac{\mathsf{B} \vdash_{\mathsf{CL}} \mathsf{t}_1{:}\tau{\to}\tau' \quad \mathsf{B} \vdash_{\mathsf{CL}} \mathsf{t}_2{:}\tau}{\mathsf{B} \vdash_{\mathsf{CL}} \mathsf{t}_1\mathsf{t}_2{:}\tau'}$$

$$(\mathbf{K}):\ \frac{}{\mathsf{B} \vdash_{\mathsf{CL}} \mathbf{K}{:}\tau{\to}\tau'{\to}\tau} \qquad (\mathbf{S}):\ \frac{}{\mathsf{B} \vdash_{\mathsf{CL}} \mathbf{S}{:}(\tau{\to}\tau'{\to}\tau''){\to}(\tau{\to}\tau'){\to}\tau{\to}\tau''}$$

We define what the equivalent of Curry's types are in terms of predicates.

$$\begin{aligned} [\![\varphi]\!] &= \varphi \\ [\![\tau{\to}\tau']\!] &= \langle \mathsf{app} : [\![\tau]\!]{\to}[\![\tau']\!] \rangle \end{aligned}$$

Then we can show: If $\mathsf{B} \vdash_{\mathsf{CL}} \mathsf{t}{:}\tau$ then $[\![\mathsf{B}]\!] \vdash [\![\mathsf{t}]\!] : [\![\tau]\!]$.

Let $\mathsf{e} = [\![\mathsf{t}]\!]$ for some CL term t; then e has a normal form if and only if there are ω-safe $\mathcal{D}$ and Π and predicate σ such that $\mathcal{D} :: \Pi \vdash \mathsf{e} : \sigma$.

Some Observations

Our types extend the standard ‘functional’ view - they give a ‘structural’ one too:

Some Observations

Our types extend the standard ‘functional’ view - they give a ‘structural’ one too:

$$\mathbf{SKK} =_{\mathsf{CL}_{\cap}} \mathbf{SKS} \qquad [\![\mathbf{SKK}]\!] =_{\lambda} [\![\mathbf{SKS}]\!]$$

Some Observations

Our types extend the standard ‘functional’ view - they give a ‘structural’ one too:

$$\mathbf{SKK} =_{\mathsf{CL}_\cap} \mathbf{SKS} \qquad\qquad [\![\mathbf{SKK}]\!] =_\lambda [\![\mathbf{SKS}]\!]$$

$$[\![\mathbf{SKK}]\!] \neq_{\mathsf{OOCL}} [\![\mathbf{SKS}]\!]$$

Some Observations

Our types extend the standard ‘functional’ view - they give a ‘structural’ one too:

$$\mathbf{SKK} =_{\mathsf{CL}_\cap} \mathbf{SKS} \qquad\qquad [\![\mathbf{SKK}]\!] =_\lambda [\![\mathbf{SKS}]\!]$$

$$[\![\mathbf{SKK}]\!] \neq_{\mathsf{OOCL}} [\![\mathbf{SKS}]\!]$$

$$\rightarrow^* \texttt{new } \texttt{S}_2\texttt{(new } \texttt{K}_0\texttt{(),new } \texttt{K}_0\texttt{())} \qquad\qquad \rightarrow^* \texttt{new } \texttt{S}_2\texttt{(new } \texttt{K}_0\texttt{(),new } \texttt{S}_0\texttt{())}$$

Some Observations

Our types extend the standard ‘functional’ view - they give a ‘structural’ one too:

$$\mathbf{SKK} =_{\mathsf{CL}_\cap} \mathbf{SKS} \qquad [\![\mathbf{SKK}]\!] =_\lambda [\![\mathbf{SKS}]\!]$$

$$[\![\mathbf{SKK}]\!] \neq_{\mathsf{OOCL}} [\![\mathbf{SKS}]\!]$$

$$\rightarrow^* \texttt{new S}_2\texttt{(new K}_0\texttt{(),new K}_0\texttt{())} \qquad \rightarrow^* \texttt{new S}_2\texttt{(new K}_0\texttt{(),new S}_0\texttt{())}$$

$$\vdash [\![\mathbf{SKK}]\!] : \langle \texttt{y:K}_0 \rangle \qquad \vdash [\![\mathbf{SKS}]\!] : \langle \texttt{y:S}_0 \rangle$$

Some Observations

Our types extend the standard 'functional' view - they give a 'structural' one too:

$$\mathbf{SKK} =_{\mathsf{CL}_\cap} \mathbf{SKS} \qquad [\![\mathbf{SKK}]\!] =_\lambda [\![\mathbf{SKS}]\!]$$

$$[\![\mathbf{SKK}]\!] \neq_{\mathsf{OOCL}} [\![\mathbf{SKS}]\!]$$

$$\rightarrow^* \texttt{new S}_2\texttt{(new K}_0\texttt{(),new K}_0\texttt{())} \qquad \rightarrow^* \texttt{new S}_2\texttt{(new K}_0\texttt{(),new S}_0\texttt{())}$$

$$\vdash [\![\mathbf{SKK}]\!] : \langle \texttt{y} : \texttt{K}_0 \rangle \qquad \vdash [\![\mathbf{SKS}]\!] : \langle \texttt{y} : \texttt{S}_0 \rangle$$

Not all FJ$^{¢}$ normal forms (or, more generally, approximants) have finite 'principal' types:

```
class C extends Object { C m() { return new C(); } }
```

Some Observations

Our types extend the standard 'functional' view - they give a 'structural' one too:

$$\mathbf{SKK} =_{\mathsf{CL}_\cap} \mathbf{SKS} \qquad [\![\mathbf{SKK}]\!] =_\lambda [\![\mathbf{SKS}]\!]$$

$$[\![\mathbf{SKK}]\!] \neq_{\mathsf{OOCL}} [\![\mathbf{SKS}]\!]$$

$$\rightarrow^* \texttt{new S}_2\texttt{(new K}_0\texttt{(),new K}_0\texttt{())} \qquad \rightarrow^* \texttt{new S}_2\texttt{(new K}_0\texttt{(),new S}_0\texttt{())}$$

$$\vdash [\![\mathbf{SKK}]\!] : \langle \texttt{y} : \texttt{K}_0 \rangle \qquad \vdash [\![\mathbf{SKS}]\!] : \langle \texttt{y} : \texttt{S}_0 \rangle$$

Not all $\mathsf{FJ}^{¢}$ normal forms (or, more generally, approximants) have finite 'principal' types:

```
class C extends Object { C m() { return new C(); } }
```

$\vdash$ `new C():`C

Some Observations

Our types extend the standard ‘functional’ view - they give a ‘structural’ one too:

$$\mathbf{SKK} =_{\mathrm{CL}_\cap} \mathbf{SKS} \qquad [\![\mathbf{SKK}]\!] =_\lambda [\![\mathbf{SKS}]\!]$$

$$[\![\mathbf{SKK}]\!] \neq_{\mathrm{OOCL}} [\![\mathbf{SKS}]\!]$$

$$\rightarrow^* \mathtt{new\ S_2(new\ K_0(),new\ K_0())} \qquad \rightarrow^* \mathtt{new\ S_2(new\ K_0(),new\ S_0())}$$

$$\vdash [\![\mathbf{SKK}]\!] : \langle \mathtt{y} : \mathtt{K_0} \rangle \qquad \vdash [\![\mathbf{SKS}]\!] : \langle \mathtt{y} : \mathtt{S_0} \rangle$$

Not all FJ$^{¢}$ normal forms (or, more generally, approximants) have finite ‘principal’ types:

```
class C extends Object { C m() { return new C(); } }
```

$$\vdash \mathtt{new\ C()} : \mathtt{C}$$

$$\vdash \mathtt{new\ C()} : \langle \mathtt{m} : () \rightarrow \mathtt{C} \rangle$$

Some Observations

Our types extend the standard ‘functional’ view - they give a ‘structural’ one too:

$$\mathbf{SKK} =_{\mathrm{CL}_\cap} \mathbf{SKS} \qquad [\![\mathbf{SKK}]\!] =_\lambda [\![\mathbf{SKS}]\!]$$

$$[\![\mathbf{SKK}]\!] \neq_{\mathrm{OOCL}} [\![\mathbf{SKS}]\!]$$

$$\to^* \texttt{new S}_2\texttt{(new K}_0\texttt{(),new K}_0\texttt{())} \qquad \to^* \texttt{new S}_2\texttt{(new K}_0\texttt{(),new S}_0\texttt{())}$$

$$\vdash [\![\mathbf{SKK}]\!] : \langle \texttt{y} : \texttt{K}_0 \rangle \qquad \vdash [\![\mathbf{SKS}]\!] : \langle \texttt{y} : \texttt{S}_0 \rangle$$

Not all FJ$^{¢}$ normal forms (or, more generally, approximants) have finite ‘principal’ types:

```
class C extends Object { C m() { return new C(); } }
```

$\vdash \texttt{new C()} : \mathrm{C}$

$\vdash \texttt{new C()} : \langle \texttt{m} : () \to \mathrm{C} \rangle$

$\vdash \texttt{new C()} : \langle \texttt{m} : () \to \langle \texttt{m} : () \to \mathrm{C} \rangle \rangle$

Some Observations

Our types extend the standard 'functional' view - they give a 'structural' one too:

$$\mathbf{SKK} =_{\mathsf{CL}_\cap} \mathbf{SKS} \qquad [\![\mathbf{SKK}]\!] =_\lambda [\![\mathbf{SKS}]\!]$$

$$[\![\mathbf{SKK}]\!] \neq_{\mathsf{OOCL}} [\![\mathbf{SKS}]\!]$$

$$\to^* \texttt{new S}_2\texttt{(new K}_0\texttt{(),new K}_0\texttt{())} \qquad \to^* \texttt{new S}_2\texttt{(new K}_0\texttt{(),new S}_0\texttt{())}$$

$$\vdash [\![\mathbf{SKK}]\!] : \langle \texttt{y}:\texttt{K}_0 \rangle \qquad \vdash [\![\mathbf{SKS}]\!] : \langle \texttt{y}:\texttt{S}_0 \rangle$$

Not all $\mathsf{FJ}^{¢}$ normal forms (or, more generally, approximants) have finite 'principal' types:

```
class C extends Object { C m() { return new C(); } }
```

$$\vdash \texttt{new C()}:\mathsf{C}$$

$$\vdash \texttt{new C()}:\langle \texttt{m}:() \to \mathsf{C} \rangle$$

$$\vdash \texttt{new C()}:\langle \texttt{m}:() \to \langle \texttt{m}:() \to \mathsf{C} \rangle \rangle$$

$$\vdash \texttt{new C()}:\langle \texttt{m}:() \to \langle \texttt{m}:() \to \langle \texttt{m}:() \to \mathsf{C} \rangle \rangle \rangle \quad \text{etc...}$$

Conclusions

We have defined two different semantics for a (kernel) class-based object oriented programming language $FJ^{¢}$, and stated how they relate.

We have proven all important properties for predicate assignment. Through our predicate assignment system, we can characterise

- *Head normalisation*.
- *Normalisation*.
- *Strong normalisation*.

We have shown that $FJ^{¢}$ is *fully expressive*, by mapping CL into OOCL (we could even map the λ-calculus).

As a first exercise of how 'handy' our system is, we have shown that all (simply) typeable OOCL programs terminate, and vice-versa.

Approximants (concept)

An *approximant* is a (finite) description of the result of the running of a program that will not change (the output) while the program still runs. We hide a place where computation takes place with $\bot$; we see $\bot$ as representing "*No information on the computation is available here*".

Approximants (concept)

An *approximant* is a (finite) description of the result of the running of a program that will not change (the output) while the program still runs. We hide a place where computation takes place with $\perp$; we see $\perp$ as representing "*No information on the computation is available here*".

Take the function

$$\begin{aligned} \textit{length list } \; a : b &= S\,(\textit{length list } \; b)\,. \\ \textit{length list } \; [\,] &= 0 \end{aligned}$$

Approximants (concept)

An *approximant* is a (finite) description of the result of the running of a program that will not change (the output) while the program still runs. We hide a place where computation takes place with $\perp$; we see $\perp$ as representing "*No information on the computation is available here*".

Take the function

$$\begin{aligned} \mathit{length\ list}\ \ a:b &= S\,(\mathit{length\ list}\ b)\,. \\ \mathit{length\ list}\ \ [\,] &= 0 \end{aligned}$$

Then

$$\begin{aligned} &\mathit{length\ list}\ 1:2:3 && \to \\ &S\,(\mathit{length\ list}\ 2:3) && \to \\ &S\,(S\,(\mathit{length\ list}\ 3)) && \to \\ &S\,(S\,(S\,(\mathit{length\ list}\ [\,]))) && \to \\ &S\,(S\,(S\,(0))) && \to \end{aligned}$$

Approximants (concept)

An *approximant* is a (finite) description of the result of the running of a program that will not change (the output) while the program still runs. We hide a place where computation takes place with $\perp$; we see $\perp$ as representing "*No information on the computation is available here*".

Take the function

$$\begin{aligned} \textit{length list } a:b &= S\,(\textit{length list } b) \\ \textit{length list } [\,] &= 0 \end{aligned}$$

Then *which has the approximant*

$\textit{length list } 1:2:3$	$\rightarrow$	$\perp$
$S\,(\textit{length list } 2:3)$	$\rightarrow$	
$S\,(S\,(\textit{length list } 3))$	$\rightarrow$	
$S\,(S\,(S\,(\textit{length list } [\,])))$	$\rightarrow$	
$S\,(S\,(S\,(0)))$	$\rightarrow$	

Approximants (concept)

An *approximant* is a (finite) description of the result of the running of a program that will not change (the output) while the program still runs. We hide a place where computation takes place with $\bot$; we see $\bot$ as representing "*No information on the computation is available here*".

Take the function

$$\begin{aligned} \textit{length list } a : b &= S\,(\textit{length list } b) \\ \textit{length list } [\,] &= 0 \end{aligned}$$

Then *which has the approximant*

$\textit{length list } 1 : 2 : 3$	$\to$	$\bot$
$S\,(\textit{length list } 2 : 3)$	$\to$	$S\,(\bot)$
$S\,(S\,(\textit{length list } 3))$	$\to$	
$S\,(S\,(S\,(\textit{length list } [\,])))$	$\to$	
$S\,(S\,(S\,(0)))$	$\to$	

Approximants (concept)

An *approximant* is a (finite) description of the result of the running of a program that will not change (the output) while the program still runs. We hide a place where computation takes place with $\bot$; we see $\bot$ as representing "*No information on the computation is available here*".

Take the function

$$\begin{aligned} \textit{length list} \ \ a:b &= S\,(\textit{length list}\ b)\ . \\ \textit{length list} \ \ [\,] &= 0 \end{aligned}$$

Then — *which has the approximant*

$\textit{length list}\ 1:2:3$	$\rightarrow$	$\bot$
$S\,(\textit{length list}\ 2:3)$	$\rightarrow$	$S\,(\bot)$
$S\,(S\,(\textit{length list}\ 3))$	$\rightarrow$	$S\,(S\,(\bot))$
$S\,(S\,(S\,(\textit{length list}\ [\,])))$	$\rightarrow$	
$S\,(S\,(S\,(0)))$	$\rightarrow$	

Approximants (concept)

An *approximant* is a (finite) description of the result of the running of a program that will not change (the output) while the program still runs. We hide a place where computation takes place with $\perp$; we see $\perp$ as representing "*No information on the computation is available here*".

Take the function

$$\begin{aligned} \textit{length list } a:b &= S\,(\textit{length list } b)\,. \\ \textit{length list } [\,] &= 0 \end{aligned}$$

Then *which has the approximant*

$\textit{length list } 1:2:3$	$\rightarrow$	$\perp$
$S\,(\textit{length list } 2:3)$	$\rightarrow$	$S\,(\perp)$
$S\,(S\,(\textit{length list } 3))$	$\rightarrow$	$S\,(S\,(\perp))$
$S\,(S\,(S\,(\textit{length list } [\,])))$	$\rightarrow$	$S\,(S\,(S\,(\perp)))$
$S\,(S\,(S\,(0)))$	$\rightarrow$	

Approximants (concept)

An *approximant* is a (finite) description of the result of the running of a program that will not change (the output) while the program still runs. We hide a place where computation takes place with $\perp$; we see $\perp$ as representing "*No information on the computation is available here*".

Take the function

$$\begin{array}{lll} \textit{length list } a:b & = & S\ (\textit{length list } b) \\ \textit{length list } [\,] & = & 0 \end{array}.$$

Then *which has the approximant*

$$\begin{array}{lll} \textit{length list } 1:2:3 & \rightarrow & \perp \\ S\ (\textit{length list } 2:3) & \rightarrow & S\ (\perp) \\ S\ (S\ (\textit{length list } 3)) & \rightarrow & S\ (S\ (\perp)) \\ S\ (S\ (S\ (\textit{length list } [\,]))) & \rightarrow & S\ (S\ (S\ (\perp))) \\ S\ (S\ (S\ (0))) & \rightarrow & S\ (S\ (S\ (0))) \end{array}$$

In this case, the output is finite, and the final approximant is the end-result itself.

Sieve of Eratosthenes in Haskell

```
primes        =  sieve [2..]
sieve (p:xs)  =  p :  sieve [x | x <- xs, x `mod` p > 0]
```

Sieve of Eratosthenes in Haskell

```
primes       =  sieve [2..]
sieve (p:xs) =  p :  sieve [x | x <- xs, x `mod` p > 0]
```

Then

$$
\begin{array}{ll}
\texttt{primes} & \to \\
\texttt{sieve}\ [2..] & \to \\
2 : \texttt{sieve}\ [3..] & \to \\
2 : 3 : \texttt{sieve}\ [5..] & \to \\
2 : 3 : 5 : \texttt{sieve}\ [7..] & \to \\
\vdots &
\end{array}
$$

Sieve of Eratosthenes in Haskell

```
primes         =  sieve [2..]
sieve (p:xs)   =  p :  sieve [x | x <- xs, x `mod` p > 0]
```

Then *which has the approximant*

$\texttt{primes} \rightarrow \quad \bot$

$\texttt{sieve}\ [2..] \rightarrow$

$2 : \texttt{sieve}\ [3..] \rightarrow$

$2 : 3 : \texttt{sieve}\ [5..] \rightarrow$

$2 : 3 : 5 : \texttt{sieve}\ [7..] \rightarrow$

$\vdots$

Sieve of Eratosthenes in Haskell

```
primes        =  sieve [2..]
sieve (p:xs)  =  p :  sieve [x | x <- xs, x `mod` p > 0]
```

Then *which has the approximant*

primes	$\rightarrow$	$\bot$
sieve $[2..]$	$\rightarrow$	$\bot$
$2:$ sieve $[3..]$	$\rightarrow$	
$2:3:$ sieve $[5..]$	$\rightarrow$	
$2:3:5:$ sieve $[7..]$	$\rightarrow$	
$\vdots$		

Sieve of Eratosthenes in Haskell

```
primes       =  sieve [2..]
sieve (p:xs) =  p :  sieve [x | x <- xs, x `mod` p > 0]
```

Then | *which has the approximant*
--- | --- | ---
$\mathtt{primes}$ | $\to$ | $\perp$
$\mathtt{sieve}\ [2..]$ | $\to$ | $\perp$
$2 : \mathtt{sieve}\ [3..]$ | $\to$ | $2 : \perp$
$2 : 3 : \mathtt{sieve}\ [5..]$ | $\to$ |
$2 : 3 : 5 : \mathtt{sieve}\ [7..]$ | $\to$ |
$\vdots$ | |

Sieve of Eratosthenes in Haskell

```
primes       =  sieve [2..]
sieve (p:xs) =  p :  sieve [x | x <- xs, x `mod` p > 0]
```

Then *which has the approximant*

primes	$\rightarrow$	$\perp$
sieve $[2..]$	$\rightarrow$	$\perp$
$2:$sieve $[3..]$	$\rightarrow$	$2:\perp$
$2:3:$sieve $[5..]$	$\rightarrow$	$2:3:\perp$
$2:3:5:$sieve $[7..]$	$\rightarrow$	
$\vdots$		

Sieve of Eratosthenes in Haskell

```
primes        =  sieve [2..]
sieve (p:xs)  =  p :  sieve [x | x <- xs, x `mod` p > 0]
```

Then *which has the approximant*

primes	$\rightarrow$	$\perp$
sieve [2..]	$\rightarrow$	$\perp$
2 : sieve [3..]	$\rightarrow$	$2 : \perp$
2 : 3 : sieve [5..]	$\rightarrow$	$2 : 3 : \perp$
2 : 3 : 5 : sieve [7..]	$\rightarrow$	$2 : 3 : 5 : \perp$
$\vdots$		$\vdots$

In this case, the computation is infinite, and so is the output, and there is no final approximant, since the "result" is never reached - $\perp$ is in every approximant.