TypOS: An “Operating System” for Typechecking Actors

Guillaume Allais    Malin Altenmüller    Conor McBride
Georgi Nakov       Fredrik Nordvall Forsberg    Craig Roy

University of St Andrews, University of Strathclyde, and Quantinuum

22 June 2022, TYPES, Nantes
A domain-specific language for typecheckers

An experiment in how to write typecheckers that make (more) sense.
A domain-specific language for typecheckers

An experiment in how to write typecheckers that make (more) sense.

Similar endeavours: Andromeda [Bauer, Haselwarter and Petkovic 2020], Redex [Felleisen, Findler and Flatt 2009], Turnstyle+ [Chang, Ballantyne, Turner and Bowman], . . .

However we try to minimise demands on the order in which subproblems are solved.
A domain-specific language for typecheckers

An experiment in how to write typecheckers that make (more) sense.

Similar endeavours: Andromeda [Bauer, Haselwarter and Petkovic 2020], Redex [Felleisen, Findler and Flatt 2009], Turnstyle+ [Chang, Ballantyne, Turner and Bowman], . . .

However we try to minimise demands on the order in which subproblems are solved.

Conor McBride, 20 years ago implementing Epigram 1:

“[redacted] me, I’m implementing an operating system!”
A domain-specific language for typecheckers

An experiment in how to write typecheckers that make (more) sense.

Similar endeavours: Andromeda [Bauer, Haselwarter and Petkovic 2020], Redex [Felleisen, Findler and Flatt 2009], Turnstyle+ [Chang, Ballantyne, Turner and Bowman], . . .

However we try to minimise demands on the order in which subproblems are solved.

Conor McBride, 20 years ago implementing Epigram 1:

“[redacted] me, I’m implementing an operating system!”
A domain-specific language for typecheckers

An experiment in how to write typecheckers that make (more) sense.

Similar endeavours: Andromeda [Bauer, Haselwarter and Petkovic 2020], Redex [Felleisen, Findler and Flatt 2009], Turnstyle+ [Chang, Ballantyne, Turner and Bowman], . . .

However we try to minimise demands on the order in which subproblems are solved.

Conor McBride, 20 years ago implementing Epigram 1:

“[redacted] me, I’m implementing an operating system!”

Concrete motivation: implementing a type theory with rich equational theory for free monoids and free Abelian groups.
Why not just a shallow embedding?
Why not just a shallow embedding?

**Logical Framework aspects:** we implement syntax with binding once, and then it Just Works.
Why not just a shallow embedding?

**Logical Framework aspects:** we implement syntax with binding once, and then it Just Works.

**Resumptions should be updatable:** progress might have happened while a process was asleep.
Why not just a shallow embedding?

**Logical Framework aspects:** we implement syntax with binding once, and then it Just Works.

**Resumptions should be updatable:** progress might have happened while a process was asleep.

**Ruling out design errors by construction:** a first-order representation means we can do static analysis on the typecheckers themselves.
A Tour of TypOS
Syntax descriptions

We support a Lisp-style generic syntax for terms:

- atoms `a`
- cons lists `\[ t_0 t_1 \ldots t_n \]`
- variables `x` and bindings `\ x t`

Simple and uniform to write and parse. Users can restrict the shape of terms using context-free syntax descriptions. We always offer a Wildcard description allowing anything.

There is a syntax description of syntax descriptions, which we use to check syntax descriptions.
Syntax descriptions

We support a Lisp-style generic syntax for terms:

- atoms \( 'a \)
- cons lists \([t_0 \ t_1 \ldots \ t_n]\)
- variables \( x \) and bindings \( \backslash x \text{.} \ t \)
Syntax descriptions

We support a Lisp-style generic syntax for terms:

- atoms ’a
- cons lists \[t_0 \ t_1 \ldots \ t_n\]
- variables \(x\) and bindings \(\backslash x. t\)

Simple and uniform to write and parse.
Syntax descriptions

We support a Lisp-style generic syntax for terms:

- atoms ’a
- cons lists \([t_0 \ t_1 \ldots \ t_n]\)
- variables \(x\) and bindings \(\langle x . t\)

Simple and uniform to write and parse.

Users can restrict the shape of terms using context-free syntax descriptions.
Syntax descriptions

We support a Lisp-style generic syntax for terms:

- atoms ’a
- cons lists [t_0 t_1 \ldots t_n]
- variables x and bindings \(x. t\)

Simple and uniform to write and parse.

Users can restrict the shape of terms using context-free syntax descriptions. We always offer a Wildcard description allowing anything.
Syntax descriptions

We support a Lisp-style generic syntax for terms:

- atoms ’a
- cons lists \[t_0 \ t_1 \ldots \ t_n]\ 
- variables \(x\) and bindings \(\backslash x. \ t\)

Simple and uniform to write and parse.

Users can restrict the shape of terms using context-free syntax descriptions. We always offer a Wildcard description allowing anything.

There is a syntax description of syntax descriptions, which we use to check syntax descriptions.
Judgement forms as interaction protocols

We recast the notion of judgement form as communication protocol:

- What to communicate (of what syntax description)?
- In which direction (input or output)?
Judgement forms as interaction protocols

We recast the notion of judgement form as communication protocol:

▶ What to communicate (of what syntax description)?
▶ In which direction (input or output)?

A basic form of session types [Honda 1993].
Judgement forms as interaction protocols

We recast the notion of judgement form as communication protocol:

▶ What to communicate (of what syntax description)?
▶ In which direction (input or output)?

A basic form of session types [Honda 1993].

For example:

```plaintext
type : ?'Type.
check : ?'Type. ?'Check.
synth : ?'Synth. !'Type.
```
Typing rules as actors

“A rule is a server for its conclusion, and a client for its premises.”
Typing rules as actors

“A rule is a server for its conclusion, and a client for its premises.”

That is: typing rules are implemented by actors, which

- must fulfill their protocol with respect to their parent;
- typically spawns children processes for all its premises.
Typing rules as actors

“A rule is a server for its conclusion, and a client for its premises.”

That is: typing rules are implemented by actors, which
▶ must fulfill their protocol with respect to their parent;
▶ typically spawns children processes for all its premises.

Inspired by the actor model [Hewitt, Bishop and Steiger 1973] of concurrent programming.
Typing rules as actors

“A rule is a server for its conclusion, and a client for its premises.”

That is: typing rules are implemented by actors, which

▶ must fulfill their protocol with respect to their parent;
▶ typically spawns children processes for all its premises.

Inspired by the actor model [Hewitt, Bishop and Steiger 1973] of concurrent programming.

Typechecking process actor with parent channel $p$ is defined by

\[
\text{actor}@p = \ldots
\]
Actor constructs: winning

a successful, finished actor

(Victory is silent.)
Actor constructs: failing

# "error message"

an unsuccessful, finished actor
Actor constructs: printing

PRINTF "message text".

printing a message before continuing
Actor constructs: generating fresh meta variables

\[ \text{sd?X} \]

generate a fresh meta \( X \) of syntax description \( sd \)
Actor constructs: generating fresh meta variables

\[ sd?X. \]

generate a fresh meta \( X \) of syntax description \( sd \)

Meta variables stand for *unknown* terms.
Actor constructs: matching on terms

\[
\text{case } t \{ \ p_1 \rightarrow a_1 \ ; \ \ldots \ \} \\
\text{match term } t \text{ against patterns } p_i; \text{ continue as actor } a_i \text{ when matching}
\]
Actor constructs: matching on terms

```
case t \{ p_1 -> a_1 ; \ldots \}
```

match term $t$ against patterns $p_i$; continue as actor $a_i$ when matching

Blocks if $t$ is a metavariable.
Actor constructs: forking

\[ a \mid b \]

continue as a and b in parallel
Actor constructs: forking

\[
\text{a | b}\]

continue as a and b in parallel

Progress in b might enable further progress in a and vice versa.
Actor constructs: declaring constraints

\[ \sim \]

make \( t_1 \) unify with \( t_2 \)
Actor constructs: spawning children

\[ \text{actor}@p. \]

spawn a new child \textit{actor} on channel \( p \)
Actor constructs: sending and receiving messages

$p!t.$

send term $t$ on channel $p$
Actor constructs: sending and receiving messages

\[ p! t. \]

send term \( t \) on channel \( p \)

\[ p? t. \]

receive term \( t \) on channel \( p \)
Actor constructs: sending and receiving messages

\[ p! t. \]

send term \( t \) on channel \( p \)

\[ p? t. \]

receive term \( t \) on channel \( p \)

Messages must conform to \( p \)'s protocol.
Actor constructs: binding local variables

\[ \text{x} \]

bring fresh object variable \(x\) into scope
Actor constructs: extending local contexts

$ctx \vdash x \rightarrow t$

extend declared context $ctx$ to map object variable $x$ to term $t$
Actor constructs: querying local contexts

\[
\text{if } x \text{ in } ctx \{ t \rightarrow a \} \text{ else } b
\]

Look up variable \( x \) in declared context \( ctx \);
if found, bind associated value as \( t \) and continue as \( a \),
otherwise continue as \( b \)
Actors for bidirectional type checking of STLC

\[
\text{check}_@p = p?ty. \ p?tm. \ \text{case} \ tm \\
\{ \ [ \text{'}Lam \ \backslash x. \ \text{body} \] \rightarrow \ \text{'}Type?S. \ \text{'}Type?T. \\
( \ \text{ty} \sim [\text{'}Arr \ S \ T] \\
| \ \backslash x. \ \text{ctxt} \ |- \ x \rightarrow S. \ \text{check}_@q. \ q!T. \ q!\text{body}. \\
; \ [\text{'}Emb \ e] \rightarrow \ \text{synth}_@q. \ q!e. \ q?S. \ S \sim \text{ty} \} \\
\]

\[
\text{synth}_@p = p?tm. \ \text{if} \ \text{tm} \ \text{in} \ \text{ctxt} \\
\{ \ S \rightarrow p!S. \} \\
\text{else} \ \text{case} \ \text{tm} \\
\{ \ [\text{'}Ann \ t \ T] \rightarrow ( \ \text{type}_@q. \ q!T. \\
| \ \text{check}_@r. \ r!T. \ r!t. \\
| \ p!T. \) \\
; \ [\text{'}App \ f \ s] \rightarrow \ \text{'}Type?S. \ \text{'}Type?T. \ p!T. \\
( \ \text{synth}_@q. \ q!f. \ q?F. \ F \sim [\text{'}Arr \ S \ T] \\
| \ \text{check}_@r. \ r!S. \ r!s. \) \} \\
\]
Executing actors

We currently run actors on a stack-based virtual machine. We run each actor until it blocks, and then try the next one, until execution stabilises. Metavariables are shared, which is okay, since they are updated monotonically [Kuper 2015]. We can extract a typing derivation from the final configuration of the stack.
Executing actors

We currently run actors on a stack-based virtual machine.
Executing actors

We currently run actors on a stack-based virtual machine.

We run each actor until it blocks, and then try the next one, until execution stabilises.
Executing actors

We currently run actors on a stack-based virtual machine.

We run each actor until it blocks, and then try the next one, until execution stabilises.

Metavariabes are shared, which is okay, since they are updated monotonically [Kuper 2015].
Executing actors

We currently run actors on a stack-based virtual machine.

We run each actor until it blocks, and then try the next one, until execution stabilises.

Metavariabels are shared, which is okay, since they are updated monotonically [Kuper 2015].

We can extract a typing derivation from the final configuration of the stack.
Some examples
\[
\begin{array}{c}
\text{TYPE } \mathbb{N} \rightarrow \mathbb{N} \vdash \mathbb{N} \\
\mathbb{N} \rightarrow \mathbb{N} \ni \lambda \cdot ?u \\
(\lambda \cdot ?u : \mathbb{N} \rightarrow \mathbb{N}) \in \mathbb{N} \rightarrow \mathbb{N} \\
\frac{}{z_0 \in \mathbb{N}^\check{\cdot}}
\end{array}
\]
\[
\begin{align*}
\text{typos --latex=stlc.tex stlc.act completed}\\
\frac{\mathbb{N} \ni \text{Zero}}{
\mathbb{N} \ni \text{Succ Zero}}\
\frac{w_1 : \mathbb{N} \vdash\
\text{TYPE } \mathbb{N} \rightarrow \mathbb{N} \vdash\mathbb{N} \rightarrow \mathbb{N} \ni \lambda_z.\text{Succ Zero}}{\text{TYPE } \mathbb{N} \rightarrow \mathbb{N} \ni \lambda_z.\text{Succ Zero}}\
\frac{(\lambda_z.\text{Succ Zero} : \mathbb{N} \rightarrow \mathbb{N}) \in \mathbb{N} \rightarrow \mathbb{N}}{z_0 \in \mathbb{N}}\
\frac{\mathbb{N} \ni (\lambda_z.\text{Succ Zero} : \mathbb{N} \rightarrow \mathbb{N}) \in \mathbb{N} \rightarrow \mathbb{N}}{z_0 : \mathbb{N} \vdash}\end{align*}
\]
\( \mathbb{N} \to \mathbb{N} \ni \lambda z. (\lambda _. [\text{Succ Zero}] : \mathbb{N} \to \mathbb{N}) z \)
\[ z_0 : \vdash \quad \text{let } f = \lambda z. (\lambda \cdot [\text{Succ Zero}] : \mathbb{N} \to \mathbb{N})z \]
\[
\begin{array}{c}
\mathbb{N} \\
\vdash \mathbb{N} \\
\mathbb{N} \ni \lambda z. (\lambda z. \text{Succ Zero} : \mathbb{N} \rightarrow \mathbb{N})z \\
\end{array}
\]
\[
\begin{align*}
\mathbb{N} & \ni (\lambda_.[\text{Succ Zero}] : \mathbb{N} \to \mathbb{N})z_0 \\
& \quad \quad z_0 : \mathbb{N} \vdash \\
\mathbb{N} \to \mathbb{N} & \ni \lambda z.(\lambda_.[\text{Succ Zero}] : \mathbb{N} \to \mathbb{N})z
\end{align*}
\]
\[ (\lambda \_.[\text{Succ Zero}] : \mathbb{N} \to \mathbb{N})z_0 \in \mathbb{N} \\\exists \ (\lambda \_.[\text{Succ Zero}] : \mathbb{N} \to \mathbb{N})z_0 \]

\[ \mathbb{N} \ni (\lambda \_.[\text{Succ Zero}] : \mathbb{N} \to \mathbb{N})z_0 \]

\[ z_0 : \mathbb{N} \vdash \]

\[ \mathbb{N} \to \mathbb{N} \ni \lambda z. (\lambda \_.[\text{Succ Zero}] : \mathbb{N} \to \mathbb{N})z \]
\(\lambda z.(\lambda_.[\text{Succ Zero}]: \mathbb{N} \to \mathbb{N})z\)
\[
(\lambda_{\_}.[\text{Succ Zero}] : \mathbb{N} \rightarrow \mathbb{N}) \in \\
\frac{}{(\lambda_{\_}.[\text{Succ Zero}] : \mathbb{N} \rightarrow \mathbb{N})z_0 \in ???} \\
\frac{\mathbb{N} \ni (\lambda_{\_}.[\text{Succ Zero}] : \mathbb{N} \rightarrow \mathbb{N})z_0}{z_0 : \mathbb{N} \vdash} \\
\frac{}{\mathbb{N} \rightarrow \mathbb{N} \ni \lambda z.(\lambda_{\_}.[\text{Succ Zero}] : \mathbb{N} \rightarrow \mathbb{N})z}
\]
\[
\begin{align*}
\text{TYPE } \mathbb{N} \rightarrow \mathbb{N} \\
\hline
(\lambda_\_.[\text{Succ Zero}] : \mathbb{N} \rightarrow \mathbb{N}) \in \\
\hline
(\lambda_\_.[\text{Succ Zero}] : \mathbb{N} \rightarrow \mathbb{N})z_0 \in \text{???
}
\hline
\mathbb{N} \ni (\lambda_\_.[\text{Succ Zero}] : \mathbb{N} \rightarrow \mathbb{N})z_0 \\
\hline
z_0 : \mathbb{N} \vdash \\
\hline
\mathbb{N} \rightarrow \mathbb{N} \ni \lambda z.(\lambda_\_.[\text{Succ Zero}] : \mathbb{N} \rightarrow \mathbb{N})z
\end{align*}
\]
\[
\text{TYPE } \mathbb{N} \rightarrow \mathbb{N}
\]

\[
(\lambda_.[\text{Succ Zero}] : \mathbb{N} \rightarrow \mathbb{N}) \in \\
(\lambda_.[\text{Succ Zero}] : \mathbb{N} \rightarrow \mathbb{N})z_0 \in ???
\]

\[
\mathbb{N} \ni (\lambda_.[\text{Succ Zero}] : \mathbb{N} \rightarrow \mathbb{N})z_0
\]

\[
\text{z}_0 : \mathbb{N} \vdash \\
\mathbb{N} \rightarrow \mathbb{N} \ni \lambda z_.(\lambda_.[\text{Succ Zero}] : \mathbb{N} \rightarrow \mathbb{N})z
\]
\[
\begin{align*}
\text{TYPE } \mathbb{N} & \rightarrow \mathbb{N} \\
(\lambda_.[\text{Succ Zero}] : \mathbb{N} \rightarrow \mathbb{N}) & \in \\
\mathbb{N} & \ni (\lambda_.[\text{Succ Zero}] : \mathbb{N} \rightarrow \mathbb{N})z_0 \\
\mathbb{N} \ni (\lambda_.[\text{Succ Zero}] : \mathbb{N} \rightarrow \mathbb{N})z_0 \\
\mathbb{N} & \ni (\lambda_.[\text{Succ Zero}] : \mathbb{N} \rightarrow \mathbb{N})z_0 \\
\mathbb{N} & \ni \lambda z.(\lambda_.[\text{Succ Zero}] : \mathbb{N} \rightarrow \mathbb{N})z
\end{align*}
\]
\[ \text{TYPE} \quad \mathbb{N} \rightarrow \mathbb{N}^\checkmark \quad \mathbb{N} \rightarrow \mathbb{N} \ni \]

\[
\frac{\checkmark}{(\lambda_.\text{[Succ Zero]} : \mathbb{N} \rightarrow \mathbb{N}) \in \checkmark}
\]

\[
\frac{(\lambda_.\text{[Succ Zero]} : \mathbb{N} \rightarrow \mathbb{N})_{z_0} \in \checkmark}{\mathbb{N} \ni (\lambda_.\text{[Succ Zero]} : \mathbb{N} \rightarrow \mathbb{N})_{z_0}}
\]

\[
\frac{z_0 : \mathbb{N} \vdash}{\mathbb{N} \rightarrow \mathbb{N} \ni \lambda z. (\lambda_.\text{[Succ Zero]} : \mathbb{N} \rightarrow \mathbb{N})_{z}}
\]
\[
\begin{align*}
\text{TYPE } \mathbb{N} \rightarrow \mathbb{N} & \vdash \mathbb{N} \ni \lambda_.[\text{Succ Zero}] \\
(\lambda_.[\text{Succ Zero}] : \mathbb{N} \rightarrow \mathbb{N}) & \in \\
(\lambda_.[\text{Succ Zero}] : \mathbb{N} \rightarrow \mathbb{N}) z_0 & \in ??? \\
\mathbb{N} \ni (\lambda_.[\text{Succ Zero}] : \mathbb{N} \rightarrow \mathbb{N}) z_0 & \in ??? \\
\vdash z_0 : \mathbb{N} \\
\mathbb{N} \rightarrow \mathbb{N} \ni \lambda z.(\lambda_.[\text{Succ Zero}] : \mathbb{N} \rightarrow \mathbb{N}) z
\end{align*}
\]
\[ \begin{align*} 
\text{TYPE } & \mathbb{N} \rightarrow \mathbb{N}^\check, \quad \mathbb{N} \rightarrow \mathbb{N} \ni \lambda_. \text{[Succ Zero]} \\
& \quad (\lambda_. \text{[Succ Zero]} : \mathbb{N} \rightarrow \mathbb{N}) \in \\
& \quad (\lambda_. \text{[Succ Zero]} : \mathbb{N} \rightarrow \mathbb{N})z_0 \in \text{???} \\
& \quad \mathbb{N} \ni (\lambda_. \text{[Succ Zero]} : \mathbb{N} \rightarrow \mathbb{N})z_0 \\
& \quad z_0 : \mathbb{N} \vdash \\
& \quad \mathbb{N} \rightarrow \mathbb{N} \ni \lambda z.(\lambda_. \text{[Succ Zero]} : \mathbb{N} \rightarrow \mathbb{N})z 
\end{align*} \]
\[
\frac{\mathbb{N} \ni \lambda \cdot [\text{Succ Zero}] \in (\lambda \cdot [\text{Succ Zero}] : \mathbb{N} \rightarrow \mathbb{N})}{(\lambda \cdot [\text{Succ Zero}] : \mathbb{N} \rightarrow \mathbb{N})z_0 \in ???}
\]

\[
\frac{\mathbb{N} \ni (\lambda \cdot [\text{Succ Zero}] : \mathbb{N} \rightarrow \mathbb{N})z_0}{z_0 : \mathbb{N} \vdash }
\]

\[
\frac{\mathbb{N} \vdash \lambda z.(\lambda \cdot [\text{Succ Zero}] : \mathbb{N} \rightarrow \mathbb{N})z}{\mathbb{N} \rightarrow \mathbb{N} \ni \lambda z.(\lambda \cdot [\text{Succ Zero}] : \mathbb{N} \rightarrow \mathbb{N})z}
\]
\[\begin{align*}
\mathbb{N} \ni [\text{Succ Zero}] & \quad \text{Type} \quad \mathbb{N} \rightarrow \mathbb{N} \vdash w_1 \colon \mathbb{N} \\
\mathbb{N} \rightarrow \mathbb{N} \ni \lambda_\_.[\text{Succ Zero}] & \quad \text{TYPE} \quad \mathbb{N} \rightarrow \mathbb{N} \vdash (\lambda_\_.[\text{Succ Zero}] : \mathbb{N} \rightarrow \mathbb{N}) \in \\
(\lambda_\_.[\text{Succ Zero}] : \mathbb{N} \rightarrow \mathbb{N})z_0 & \quad \mathbb{N} \ni (\lambda_\_.[\text{Succ Zero}] : \mathbb{N} \rightarrow \mathbb{N})z_0 \\
z_0 & \quad \mathbb{N} \vdash \mathbb{N} \ni \lambda z.(\lambda_\_.[\text{Succ Zero}] : \mathbb{N} \rightarrow \mathbb{N})z
\end{align*}\]
\[
\begin{align*}
    \text{TYPE }& \quad \mathbb{N} \rightarrow \mathbb{N}^\check \\
    \text{ }& \quad \mathbb{N} \rightarrow \mathbb{N} \ni \lambda \cdot \text{[Succ Zero]} \\
    \text{ }& \quad (\lambda \cdot \text{[Succ Zero]} : \mathbb{N} \rightarrow \mathbb{N}) \in \\
    \text{ }& \quad (\lambda \cdot \text{[Succ Zero]} : \mathbb{N} \rightarrow \mathbb{N})_{z_0} \in \text{???} \\
    \text{ }& \quad \mathbb{N} \ni (\lambda \cdot \text{[Succ Zero]} : \mathbb{N} \rightarrow \mathbb{N})_{z_0} \\
    \text{ }& \quad z_0 : \mathbb{N} \vdash \\
    \text{ }& \quad \mathbb{N} \rightarrow \mathbb{N} \ni \lambda z. (\lambda \cdot \text{[Succ Zero]} : \mathbb{N} \rightarrow \mathbb{N})_z
\end{align*}
\]
\[ \begin{align*}
\mathbb{N} \ni \text{Zero} & \quad \mathbb{N} \ni [\text{Succ Zero}] \\
\mathbb{N} \ni \lambda \cdot [\text{Succ Zero}] & \quad \mathbb{N} \rightarrow \mathbb{N} \ni \lambda \cdot [\text{Succ Zero}] \\
(\lambda \cdot [\text{Succ Zero}] : \mathbb{N} \rightarrow \mathbb{N}) \in & \quad (\lambda \cdot [\text{Succ Zero}] : \mathbb{N} \rightarrow \mathbb{N}) \mathcal{z}_0 \in \mathbb{N} \\
\mathbb{N} \ni (\lambda \cdot [\text{Succ Zero}] : \mathbb{N} \rightarrow \mathbb{N}) \mathcal{z}_0 & \quad \mathbb{N} \ni \lambda z. (\lambda \cdot [\text{Succ Zero}] : \mathbb{N} \rightarrow \mathbb{N}) \mathcal{z}
\end{align*} \]
\[
\begin{align*}
\text{N} \ni \text{Zero} & \checkmark \\
\text{N} \ni \text{[Succ Zero]} & \checkmark \\
\mathcal{W}_1 : \text{N} & \vdash \\
\text{TYPE}\ N \to N & \checkmark \\
\text{N} \to N & \ni \lambda_. \text{[Succ Zero]} & \checkmark \\
(\lambda_. \text{[Succ Zero]} : N \to N) & \in N \to N \\
(\lambda_. \text{[Succ Zero]} : N \to N)_{z_0} & \in ??? \\
\text{N} & \ni (\lambda_. \text{[Succ Zero]} : N \to N)_{z_0} \\
\text{z}_0 : N & \vdash \\
\text{N} \to N & \ni \lambda z. (\lambda_. \text{[Succ Zero]} : N \to N)z
\end{align*}
\]
\[
\begin{align*}
\frac{\mathbb{N} \ni \text{Zero}}{\mathbb{N} \ni \text{Succ Zero}} & \\
\frac{\mathbb{N} \ni \text{Succ Zero}}{\nu_1 : \mathbb{N} \vdash} & \\
\frac{\text{TYPE } \mathbb{N} \rightarrow \mathbb{N} \quad \mathbb{N} \rightarrow \mathbb{N} \ni \lambda_.[\text{Succ Zero}]}{\left(\lambda_.[\text{Succ Zero}] : \mathbb{N} \rightarrow \mathbb{N}\right) \in \mathbb{N} \rightarrow \mathbb{N}} \quad \mathbb{N} \ni \\
\frac{\left(\lambda_.[\text{Succ Zero}] : \mathbb{N} \rightarrow \mathbb{N}\right)z_0 \in \mathbb{N}}{\mathbb{N} \ni \left(\lambda_.[\text{Succ Zero}] : \mathbb{N} \rightarrow \mathbb{N}\right)z_0} & \\
\frac{z_0 : \mathbb{N} \vdash}{\mathbb{N} \rightarrow \mathbb{N} \ni \lambda z.\left(\lambda_.[\text{Succ Zero}] : \mathbb{N} \rightarrow \mathbb{N}\right)z}
\end{align*}
\]
\[
\begin{align*}
\mathbb{N} & \ni \text{Zero}^\checkmark \\
\mathbb{N} & \ni [\text{Succ Zero}]^\checkmark \\
\nu_1 : \mathbb{N} & \vdash \\
\text{type} & \quad \mathbb{N} \rightarrow \mathbb{N}^\checkmark \\
\mathbb{N} & \ni \lambda . [\text{Succ Zero}]^\checkmark \\
(\lambda . [\text{Succ Zero}] : \mathbb{N} \rightarrow \mathbb{N}) & \in \mathbb{N} \rightarrow \mathbb{N}^\checkmark \\
\mathbb{N} & \ni z_0 \\
(\lambda . [\text{Succ Zero}] : \mathbb{N} \rightarrow \mathbb{N})z_0 & \in \mathbb{N} \\
\mathbb{N} & \ni (\lambda . [\text{Succ Zero}] : \mathbb{N} \rightarrow \mathbb{N})z_0 \\
z_0 : \mathbb{N} & \vdash \\
\mathbb{N} \rightarrow \mathbb{N} & \ni \lambda z. (\lambda . [\text{Succ Zero}] : \mathbb{N} \rightarrow \mathbb{N})z
\end{align*}
\]
\[
\begin{align*}
\text{N} \ni \text{Zero} & : \checkmark \\
\text{N} \ni \text{[Succ Zero]} & : \checkmark \\
\forall_1 : \text{N} \vdash & \\
\text{TYPE} \quad \text{N} \rightarrow \text{N} & : \checkmark \\
\quad \text{N} \rightarrow \text{N} \ni \lambda_. \text{[Succ Zero]} & : \checkmark \\
(\lambda_. \text{[Succ Zero]} : \text{N} \rightarrow \text{N}) & \in \text{N} \rightarrow \text{N} \checkmark \\
\quad \text{N} \ni z_0 & : \checkmark \\
(\lambda_. \text{[Succ Zero]} : \text{N} \rightarrow \text{N})z_0 & \in \text{N} \\
\quad \text{N} \ni (\lambda_. \text{[Succ Zero]} : \text{N} \rightarrow \text{N})z_0 & : \checkmark \\
\quad z_0 : \text{N} \vdash & \\
\forall_0 : \text{N} \ni \lambda z. (\lambda_. \text{[Succ Zero]} : \text{N} \rightarrow \text{N})z & : \checkmark
\end{align*}
\]
\[
\begin{array}{c}
\text{\(\mathbb{N} \ni \text{Zero}^\checkmark\)} \\
\text{\(\mathbb{N} \ni [\text{Succ Zero}]^\checkmark\)} \\
\text{\(\psi_1 : \mathbb{N} \vdash\)} \\
\text{\(\text{TYPE \(\mathbb{N} \to \mathbb{N}\)}^\checkmark \quad \mathbb{N} \to \mathbb{N} \ni \lambda \_.[\text{Succ Zero}]^\checkmark\)} \\
\text{\((\lambda \_.[\text{Succ Zero}] : \mathbb{N} \to \mathbb{N}) \in \mathbb{N} \to \mathbb{N}^\checkmark\)} \\
\text{\((\lambda \_.[\text{Succ Zero}] : \mathbb{N} \to \mathbb{N})z_0 \in \mathbb{N}\)} \\
\text{\(\mathbb{N} \ni (\lambda \_.[\text{Succ Zero}] : \mathbb{N} \to \mathbb{N})z_0\)} \\
\text{\(z_0 : \mathbb{N} \vdash\)} \\
\text{\(\mathbb{N} \to \mathbb{N} \ni \lambda z. (\lambda \_.[\text{Succ Zero}] : \mathbb{N} \to \mathbb{N})z\)}
\end{array}
\]
\[
\begin{align*}
N \ni \text{Zero} & \checkmark \\
N \ni [\text{Succ Zero}] & \checkmark \\
\vdash w_1 : N \\
\text{TYPE } N \to N & \checkmark \\
N \to N \ni \lambda_.[\text{Succ Zero}] & \checkmark \\
(\lambda_.[\text{Succ Zero}] : N \to N) & \in N \to N \\
(\lambda_.[\text{Succ Zero}] : N \to N)z_0 & \in N \\
N \ni (\lambda_.[\text{Succ Zero}] : N \to N)z_0 & \checkmark \\
\vdash z_0 : N \\
N \to N \ni \lambda z.(\lambda_.[\text{Succ Zero}] : N \to N)z & \checkmark 
\end{align*}
\]
Verification of actors

What do we get by construction?
Verification of actors

What do we get by construction?

- Protocols and modes $\implies$ rely/guarantee contracts
Verification of actors

What do we get by construction?

- Protocols and modes $\implies$ rely/guarantee contracts
- Actors only knowing about free variables they themselves create $\implies$ stability under substitution
Verification of actors

What do we get by construction?

- Protocols and modes $\implies$ rely/guarantee contracts
- Actors only knowing about free variables they themselves create $\implies$ stability under substitution
- “Schematic variables” have one explicit binding site $\implies$ scopes are not escaped
Verification of actors

What do we get by construction?

- Protocols and modes $\implies$ rely/guarantee contracts
- Actors only knowing about free variables they themselves create $\implies$ stability under substitution
- “Schematic variables” have one explicit binding site $\implies$ scopes are not escaped
- ...
Summary and future work

TypOS is an domain-specific language for writing typecheckers.

Judgements have modes (input/output protocols), typing rules are actors (spawning and communicating with children).

A wide range of typechecking, evaluation and elaboration processes can be implemented this way.

**In the future:** a truly concurrent runtime.

https://github.com/msp-strath/TypOS
Summary and future work

TypOS is an domain-specific language for writing typecheckers.

Judgements have modes (input/output protocols), typing rules are actors (spawning and communicating with children).

A wide range of typechecking, evaluation and elaboration processes can be implemented this way.

In the future: a truly concurrent runtime.

https://github.com/msp-strath/TypOS
References
In order of appearance


Image credits:
▶ “L’Opéra Graslin (Le Voyage à Nantes)” by Jean-Pierre Dalbra, https://flic.kr/p/f9BB5h, CC BY 2.0