J-Calc: A typed λ-calculus for Justification Logic

K. Pouliasis[1], G. Primiero[2]

[1]Department of Computer Science, Graduate Center, CUNY
[2]FWO, Ghent University

IMLA, 2013

Outline

1. Motivation
2. Representing Proofs in T: a simply typed λ-calculus
3. Representing proofs in T': Justification Logic
4. Adjoining judgments of the two theories: $JCalc_1$
5. Generalization and metatheoretic results

Extending Curry-Howard

$$\frac{\text{IPC}}{\textit{Simply Typed } \lambda - \textit{Calculus}}$$

$$\frac{\textit{Intuitionistic } \text{S4}}{\textit{Pfenning and Davies}: \Box^{\rightarrow}; \textit{Bierman and De Paiva}: \lambda_{S_4}, \textit{etc}}$$

$$\frac{\textit{Intuitionistic } \text{K}}{\textit{Bellin}, \textit{Bierman}, \textit{de Paiva}: \textit{IK}}$$

Our Problem

the calculus

$$\frac{\textit{Intuitionistic Justification Logic}}{??}$$

$$\frac{\textit{Intuitionistic Justification Logic}}{\text{JCalc}}$$

Our Problem

the calculus

$$\frac{\textit{Intuitionistic Justification Logic}}{??}$$

$$\frac{\textit{Intuitionistic Justification Logic}}{\text{JCalc}}$$

Our Problem

computational significance

$S_4 \iff$ staged computation

$IK \iff$ explicit substitutions

$JCalc \iff$ separate compilation

Our Problem

computational significance

$S_4 \iff$ staged computation

$IK \iff$ explicit substitutions

$JCalc \iff$ separate compilation

Our Problem
computational significance

$S_4 \iff$ staged computation

$IK \iff$ explicit substitutions

$JCalc \iff$ separate compilation

Our Approach

From a logical point

- We assume two languages: one of the theory T and one of a theory T' that provides the intended semantics of T.
- Prop_0: universe of types of T (intuitionistic)
- A one-to-one and into mapping *Just* from $Prop_0$ into the type universe of T'. We use jtype_0 for the image of this mapping.

Our Approach

From a logical point

- A trivial example:Take T some arithmetic and T' an axiomatic set theory. Then Just $(1 + 1 = 2) \Rightarrow \{\emptyset\} + \{\emptyset\} = \{\emptyset, \{\emptyset\}\}$

Natural Deduction for IPC$^{\rightarrow}$

Logical Rules

$$\frac{\Gamma_0 \vdash_{\mathrm{IPC}} \mathrm{wf} \qquad x : P_i \in \Gamma_0}{\Gamma_0 \vdash_{\mathrm{IPC}} x : P_i}\ \Gamma\text{-}\mathrm{R}{\small\mathrm{EFL}}$$

$$\frac{\Gamma_0, x : \phi_1 \vdash_{\mathrm{IPC}} M : \phi_2}{\Gamma_0 \vdash_{\mathrm{IPC}} \lambda x : \phi_1 .\ M : \phi_1 \rightarrow \phi_2}\ {\rightarrow}\mathrm{I}$$

$$\frac{\Gamma_0 \vdash_{\mathrm{IPC}} M : \phi_1 \rightarrow \phi_2 \qquad \Gamma_0 \vdash_{\mathrm{IPC}} M' : \phi_1}{\Gamma_0 \vdash_{\mathrm{IPC}} (MM') : \phi_2}\ {\rightarrow}\mathrm{E}$$

Basic Idea

- Represent constructive necessity as a proof match between T and T'.
- Have neccesitation as an admissible rule

$$\frac{\vdash \phi \qquad \vdash C :: \phi}{\vdash \Box^{C}\phi}\ \Box\text{-ADM}$$

Basic Idea

- Represent constructive necessity as a proof match between T and T'.
- Have neccesitation as an admissible rule

$$\frac{\vdash \phi \qquad \vdash C :: \phi}{\vdash \Box^{C}\phi}\ \Box\text{-}\textsc{Adm}$$

Type Universe $jtype_0$ and Δ Contexts

- Judgments of the form $\Delta \vdash_{J_0} j :: \phi$. Sugar for $\Delta \vdash_{J_0} j : Just\ \phi$

$$\frac{}{\text{nil} \vdash_{J_0} \text{wf}}\ \text{NIL} \qquad \frac{\Delta_0 \vdash_{J_0} \text{wf} \qquad \Delta_0 \vdash_{J_0} \phi \in \text{Prop}_0}{\Delta_0 \vdash_{J_0} \text{Just}\ \phi \in \text{jtype}_0}\ \text{SIMPLE}$$

$$\frac{\Delta_0 \vdash_{J_0} \text{Just}\ \phi \in \text{jtype}_0 \qquad s \notin \Delta_0}{\Delta_0, s :: \phi \vdash_{J_0} \text{wf}}\ \Delta_0\text{-APP}$$

$$\frac{\Delta_0 \vdash_{J_0} \text{wf} \qquad s :: \phi \in \Delta}{\Delta_0 \vdash_{J_0} s :: \phi}\ \Delta_0\text{-REFL}$$

Constant Specification and Compositionality

- Every $jtype_0$ of the following principal type schemes is inhabited by a proof in T'.

$$\frac{\Delta_0 \vdash_{J_0} \text{Just } \ \phi_1 \rightarrow \phi_2 \rightarrow \phi_1 \in \text{jtype}_0}{\Delta_0 \vdash_{J_0} \text{K}[\phi_1, \phi_2] :: \phi_1 \rightarrow \phi_2 \rightarrow \phi_1} \text{ K}$$

$$\frac{\Delta_0 \vdash_{J_0} \text{Just } \ (\phi_1 \rightarrow \phi_2 \rightarrow \phi_3) \rightarrow (\phi_1 \rightarrow \phi_2) \rightarrow (\phi_1 \rightarrow \phi_3) \in \text{jtype}_0}{\Delta_0 \vdash_{J_0} \text{S}[\phi_1, \phi_2, \phi_3] :: (\phi_1 \rightarrow \phi_2 \rightarrow \phi_3) \rightarrow (\phi_1 \rightarrow \phi_2) \rightarrow (\phi_1 \rightarrow \phi_3)} \text{ S}$$

- Proofs in T' can be composed:

$$\frac{\Delta_0 \vdash_{J_0} j_2 :: \phi_1 \rightarrow \phi_2 \qquad \Delta_0 \vdash_{J_0} j_1 :: \phi_1}{\Delta_0 \vdash_{J_0} j_2 * j_1 :: \phi_2} \text{ TIMES}$$

Constant Specification and Compositionality

- Every $jtype_0$ of the following principal type schemes is inhabited by a proof in T'.

$$\frac{\Delta_0 \vdash_{J_0} \text{Just}\ \ \phi_1 \to \phi_2 \to \phi_1 \in \text{jtype}_0}{\Delta_0 \vdash_{J_0} \text{K}[\phi_1, \phi_2] :: \phi_1 \to \phi_2 \to \phi_1}\ \text{K}$$

$$\frac{\Delta_0 \vdash_{J_0} \text{Just}\ \ (\phi_1 \to \phi_2 \to \phi_3) \to (\phi_1 \to \phi_2) \to (\phi_1 \to \phi_3) \in \text{jtype}_0}{\Delta_0 \vdash_{J_0} \text{S}[\phi_1, \phi_2, \phi_3] :: (\phi_1 \to \phi_2 \to \phi_3) \to (\phi_1 \to \phi_2) \to (\phi_1 \to \phi_3)}\ \text{S}$$

- Proofs in T' can be composed:

$$\frac{\Delta_0 \vdash_{J_0} j_2 :: \phi_1 \to \phi_2 \qquad \Delta_0 \vdash_{J_0} j_1 :: \phi_1}{\Delta_0 \vdash_{J_0} j_2 * j_1 :: \phi_2}\ \text{TIMES}$$

Polymorphic Constant Specification

- Constants reflect the ability of the external theory T' to provide semantics for T.
- Here to include - at least - minimal logic.
- Enriching T (e.g. adding conjunction as pairing) imposes a richer constant specification of T'.

An example

- Assume a signature in an ML-like language:

```
module type INTSTACK =
sig
type intstack
val Empty: intstack
val push : int->intstack->intstack
val pop: int->intstack->intstack
end;;
```

- Assume this code on the client's side:
  $\vdash_{sig} (push\ 2\ Empty) : intstack$
- The computational value of this term is contextual.
- It depends on the implementations to which the signature constants are linked to.

An example

producing generic code for (*push* 2 *Empty*)

$$\dfrac{\begin{array}{llll} \textit{push} & \xrightarrow{\textit{link}} & \texttt{Cons} & :\Box^{\texttt{Cons}}(\texttt{int} \rightarrow \texttt{intstack} \rightarrow \texttt{intstack}) \\ \textit{Empty} & \xrightarrow{\textit{link}} & \texttt{[]} & :\Box^{\texttt{[]}}\texttt{intstack} \end{array}}{\begin{array}{llll} \textit{push}\ 2\ \textit{Empty} & \xrightarrow{\textit{link}} & \texttt{Cons}\ 2\ \texttt{[]} & :\Box^{\texttt{Cons}\star 2\star \texttt{[]}}\texttt{intstack} \end{array}}$$

$$\dfrac{\begin{array}{llll} \textit{push} & \xrightarrow{\textit{link}} & \texttt{Addarr} & :\Box^{\texttt{Addarr}}(\texttt{int} \rightarrow \texttt{intstack} \rightarrow \texttt{intstack}) \\ \textit{Empty} & \xrightarrow{\textit{link}} & \texttt{Void} & :\Box^{\texttt{Void}}\texttt{intstack} \end{array}}{\begin{array}{llll} \textit{push}\ 2\ \textit{Empty} & \xrightarrow{\textit{link}} & \texttt{Addarr}\ 2\ \texttt{Void} & :\Box^{\texttt{Addarr}\star 2\star \texttt{Void}}\texttt{intstack} \end{array}}$$

An example

producing generic code for (*push* 2 *Empty*)

$$\frac{push \xrightarrow{link} \texttt{Cons} : \Box^{\texttt{Cons}}(\texttt{int} \rightarrow \texttt{intstack} \rightarrow \texttt{intstack}) \qquad Empty \xrightarrow{link} \texttt{[]} : \Box^{\texttt{[]}}\texttt{intstack}}{push\ 2\ Empty \xrightarrow{link} \texttt{Cons}\ 2\ \texttt{[]} : \Box^{\texttt{Cons}\star 2\star\texttt{[]}}\texttt{intstack}}$$

$$\frac{push \xrightarrow{link} \texttt{Addarr} : \Box^{\texttt{Addarr}}(\texttt{int} \rightarrow \texttt{intstack} \rightarrow \texttt{intstack}) \qquad Empty \xrightarrow{link} \texttt{Void} : \Box^{\texttt{Void}}\texttt{intstack}}{push\ 2\ Empty \xrightarrow{link} \texttt{Addarr}\ 2\ \texttt{Void} : \Box^{\texttt{Addarr}\star 2\star\texttt{Void}}\texttt{intstack}}$$

An example

producing generic code for (*push* 2 *Empty*)

- To achieve separate compilation of client code and server code we create generic linking processes specialized on the structure of clients source terms.
- First we factorize the usage of the signature. Rewriting the term:

$$\lfloor \Gamma = x_1 : int \rightarrow intstack \rightarrow intstack, x_2 : intstack \vdash (x_1\ 2\ x_2) : intstack$$

$$\frac{\lfloor \Gamma \vdash (x_1\ 2\ x_2) : intstack \qquad \Delta; \Gamma \vdash s1 * 2 * s_2 :: intstack}{\Delta; \Gamma \vdash let^{*}\Gamma\ in\ link(x_1\ 2\ x_2, s_1 * 2 * s_2) : \Box^{s_1 * 2 * s_2} intstack}$$

An example

producing generic code for *(push 2 Empty)*

- To achieve separate compilation of client code and server code we create generic linking processes specialized on the structure of clients source terms.
- First we factorize the usage of the signature. Rewriting the term:

$$\downharpoonleft \Gamma = x_1 : int \rightarrow intstack \rightarrow intstack, x_2 : intstack \vdash (x_1\ 2\ x_2) : intstack$$

$$\frac{\downharpoonleft \Gamma \vdash (x_1\ 2\ x_2) : intstack \qquad \Delta;\Gamma \vdash s1 * 2 * s_2 :: intstack}{\Delta;\Gamma \vdash let^{*}\Gamma\ in\ link(x_1\ 2\ x_2, s_1 * 2 * s_2) : \Box^{s_1 * 2 * s_2} intstack}$$

An example

producing generic code for (*push* 2 *Empty*)

- Assume implementations of "missing" code in the validity context, i.e.

$$\Delta = s_1 :: int \rightarrow intstack \rightarrow intstack, s_2 :: intstack \vdash s1 * 2 * s_2 :: intstack$$

$$\frac{\lfloor \Gamma \vdash (x_1\ 2\ x_2) : intstack \qquad \Delta; \Gamma \vdash s1 * 2 * s_2 :: intstack}{\Delta; \Gamma \vdash let^* \Gamma\ in\ link(x_1\ 2\ x_2, s_1 * 2 * s_2) : \Box^{s_1 * 2 * s_2} intstack}$$

An example

producing generic code for *(push 2 Empty)*

- Lift the judgment to a judgment on links :
  $\Delta; \Gamma = {x_1}' : \Box^{s_1}(int \rightarrow intstack \rightarrow intstack), x_2' : \Box^{s_2} intstack \vdash$
  $let\ link(x_1, s_1) = x_1'\ in$
  $let\ link(x_2, s_2) = x_2'\ in$
  $link(x_1\ 2\ x_2, s_1 * 2 * s_2) : \Box^{s_1 * 2 * s_2} intstack$

$$\frac{\downarrow \Gamma \vdash (x_1\ 2\ x_2) : intstack \qquad \Delta; \Gamma \vdash s1 * 2 * s_2 :: intstack}{\Delta; \Gamma \vdash let^* \Gamma\ in\ link(x_1\ 2\ x_2, s_1 * 2 * s_2) : \Box^{s_1 * 2 * s_2} intstack}$$

An example

producing generic code for (*push* 2 *Empty*)

- Abstracting from Γ and Δ we obtain generic code:

$$\vdash Js_1.Js_2.\ \lambda x_1'.\lambda x_2'.\ let^{*}\Gamma\ in\ link(x_1\ 2\ x_2, s_1 * 2 * s_2)$$

with type:

$$\Pi s_1.\Pi s_2.\Box^{s_1}(int \to intstack \to intstack) \to \Box^{s_2} intstack \to \Box^{s_1 * 2 * s_2} intstack$$

An example

- Closing Δ with implementations e.g.

$$Just[intstack] = \texttt{List}, Just[push] = \texttt{Cons}, Just[Empty] = \texttt{[]}$$

- We obtain the links:

$$link(push, \texttt{Cons}) : \Box^{\texttt{Cons}}(int \to intstack \to intstack)$$

$$link(Empty, \texttt{[]}) : \Box^{\texttt{[]}} intstack$$

- And from the previous generic judgment under standard reduction and let evaluation rules we get

$$link(push\ 2\ Empty, \texttt{Cons 2 []})$$

An example

- Closing Δ with implementations e.g.

$$Just[intstack] = \texttt{List}, Just[push] = \texttt{Cons}, Just[Empty] = \texttt{[]}$$

- We obtain the links:

$$link(push, \texttt{Cons}) : \Box^{\texttt{Cons}}(int \to intstack \to intstack)$$

$$link(Empty, \texttt{[]}) : \Box^{\texttt{[]}} intstack$$

- And from the previous generic judgment under standard reduction and let evaluation rules we get

$$link(push\ 2\ Empty, \texttt{Cons 2 []})$$

An example

- Closing Δ with implementations e.g.

$$Just[intstack] = \texttt{List}, Just[push] = \texttt{Cons}, Just[Empty] = \texttt{[]}$$

- We obtain the links:

$$link(push, \texttt{Cons}) : \Box^{\texttt{Cons}}(int \rightarrow intstack \rightarrow intstack)$$

$$link(Empty, \texttt{[]}) : \Box^{\texttt{[]}} intstack$$

- And from the previous generic judgment under standard reduction and let evaluation rules we get

$$link(push\ 2\ Empty, \texttt{Cons 2 []})$$

An example

- Closing Δ with implementations e.g.

$$Just[intstack] = \texttt{Array}, Just[push] = \texttt{Addarr}, Just[Empty] = \texttt{void}$$

- We obtain the links:

$$link(push, \texttt{Addarr}) : \Box^{\texttt{Addarr}}(int \rightarrow intstack \rightarrow intstack)$$

$$link(Empty, \texttt{void}) : \Box^{\texttt{void}} intstack$$

- And from the previous generic judgment under standard reduction and let evaluation rules we get

$$link(push\ 2\ Empty, \texttt{Addarr 2 void})$$

- Note that the client code does not need to recompile

An example

- Closing Δ with implementations e.g.

$$\mathit{Just}[\mathit{intstack}] = \texttt{Array}, \mathit{Just}[\mathit{push}] = \texttt{Addarr}, \mathit{Just}[\mathit{Empty}] = \texttt{void}$$

- We obtain the links:

$$\mathit{link}(\mathit{push}, \texttt{Addarr}) : \Box^{\texttt{Addarr}}(\mathit{int} \rightarrow \mathit{intstack} \rightarrow \mathit{intstack})$$

$$\mathit{link}(\mathit{Empty}, \texttt{void}) : \Box^{\texttt{void}}\mathit{intstack}$$

- And from the previous generic judgment under standard reduction and let evaluation rules we get

$$\mathit{link}(\mathit{push}\ 2\ \mathit{Empty}, \texttt{Addarr 2 void})$$

- Note that the client code does not need to recompile

An example

- Closing Δ with implementations e.g.

  $$\mathit{Just}[\mathit{intstack}] = \texttt{Array}, \mathit{Just}[\mathit{push}] = \texttt{Addarr}, \mathit{Just}[\mathit{Empty}] = \texttt{void}$$

- We obtain the links:

  $$\mathit{link}(\mathit{push}, \texttt{Addarr}) : \Box^{\texttt{Addarr}}(\mathit{int} \rightarrow \mathit{intstack} \rightarrow \mathit{intstack})$$

  $$\mathit{link}(\mathit{Empty}, \texttt{void}) : \Box^{\texttt{void}}\mathit{intstack}$$

- And from the previous generic judgment under standard reduction and let evaluation rules we get

  $$\mathit{link}(\mathit{push}\ 2\ \mathit{Empty}, \texttt{Addarr 2 void})$$

- Note that the client code does not need to recompile

Zipping the two kinds reasoning

- The $\Box - $ *Intro* rule can be viewed algorithmically as a linking process generator.
- It consumes source code from a client language (T constructs), implementations in a host language (T' constructs) and produces iterative linking processes specialized for compound terms of T.

Zipping the two kinds reasoning

- The $\Box - \textit{Intro}$ rule can be viewed algorithmically as a linking process generator.
- It consumes source code from a client language (T constructs), implementations in a host language (T' constructs) and produces iterative linking processes specialized for compound terms of T.

Abstract Syntax of *JCalc*$_1$

$$
\begin{aligned}
\phi &:= P_i | \bot | \Box^{j} \phi | \phi_1 \to \phi_2 \\
j &:= s_i | C | j_1 * j_2 \\
t &:= x_i | \lambda x_i : \phi . t | Js :: \phi . t \\
C &:= \mathrm{K}[\phi_1, \phi_2] | \mathrm{S}[\phi_1, \phi_2, \phi_3] \\
\pi &:= \Pi s :: \phi_1 .\ \phi_2 | \Pi s :: \phi_1 .\ \pi \\
T &:= \phi | \pi \\
s &:= s_i \\
x &:= x_i
\end{aligned}
$$

$Prop_0$, *$Prop_1$*, and *wf*

$Jcalc_1$ inherits all previous rules of *J_0* and *IPC* in extended type universe as shown below:

$$\frac{\Delta_0 \vdash_{J_0} \mathsf{wf}}{\Delta_0; \mathsf{nil} \vdash_{JC_1} \mathsf{wf}}\ \textsc{ImpWf} \qquad \frac{\Delta_0; \Gamma_1 \vdash_{JC_1} \mathsf{wf} \qquad \Delta_0 \vdash_{J_0} j :: \phi}{\Delta_0; \Gamma_1 \vdash_{JC_1} j :: \phi}\ \textsc{ImpJust}$$

$$\frac{\phi \in \mathsf{Prop}_0 \qquad \Delta_0; \Gamma_1 \vdash_{JC_1} \mathsf{j} :: \phi}{\Delta_0; \Gamma_1 \vdash_{JC_1} \Box^j \phi \in \mathsf{Prop}_1}\ \textsc{Prop}_1\textsc{-Intro}$$

$$\frac{\Delta_0; \Gamma_1 \vdash \phi_1 \in \mathsf{Prop_i} \qquad \Delta_0; \Gamma_1 \vdash_{JC_1} \phi_2 \in \mathsf{Prop_j}}{\Delta_0; \Gamma_1 \vdash_{JC_1} \phi_1 \to \phi_2 \in \mathsf{Prop}_{\mathsf{max\{i,j\}}}}\ \textsc{Prop}_2\textsc{-Intro}$$

$$\frac{\Delta_0; \Gamma_1 \vdash_{JC_1} \phi \in \{\mathsf{Prop}_0, \mathsf{Prop}_1\} \qquad x \notin \Gamma_1}{\Delta_0; \Gamma_1, x : \phi \vdash_{JC_1} \mathsf{wf}}\ \Gamma_1\textsc{-App}$$

Π- Kind

$$\frac{\Delta_0, s :: \phi_1; \vdash_{JC_1} \phi_2 \in \{\text{Prop}_0, \text{Prop}_1\}}{\Delta_0; \vdash_{JC_1} \Pi s :: \phi_1.\phi_2 \in \Pi} \;\Pi\ \text{TYPE}_0$$

$$\frac{\Delta_0, s :: \phi_1; \vdash_{JC_1} \pi \in \Pi}{\Delta_0; \vdash_{JC_1} \Pi s :: \phi_1.\pi \in \Pi} \;\Pi\ \text{TYPE}_1$$

Logical Rules: Propositional Part

$$\frac{\Delta_0; \Gamma_1 \vdash_{JC_1} \text{wf} \qquad x : \phi \in \Gamma_1}{\Delta_0; \Gamma_1 \vdash_{JC_1} x : \phi}\ \Gamma\text{-Refl}$$

$$\frac{\Delta_0; \Gamma_1, x : \phi_1 \vdash_{JC_1} M : \phi_2}{\Delta_0; \Gamma_1 \vdash_{JC_1} \lambda x : \phi_1 .\ M : \phi_1 \to \phi_2}\ \to\text{I}$$

$$\frac{\Delta_0; \Gamma_1 \vdash_{JC_1} M : \phi_1 \to \phi_2 \qquad \Delta_0; \Gamma_1 \vdash_{JC_1} M' : \phi_1}{\Delta_0; \Gamma_1 \vdash_{JC_1} (MM') : \phi_2}\ \to\text{E}$$

Linking: $\Box^j$-Intro

- For relating the two calculi, a lifting rule is formulated for turning strictly $Prop_0$ judgments to judgments on links ($Prop_1$)
- We define an operator on contexts deleting one $\Box$ on the top level of each assumption :

$$\downharpoonleft \Gamma \ := \textbf{match } \Gamma \textbf{ with}$$

$$\text{nil} \Rightarrow \text{nil}$$

$$\mid \Gamma', x_i' : \Box^j \phi_i \Rightarrow \downharpoonleft \Gamma', \ x_i : \phi_i$$

$$\mid \Gamma', \_ \Rightarrow \downharpoonleft \Gamma'$$

Linking: $\Box^j$-Intro

- Analogously, we define iterative let binding generator: *let** Γ.

$$
\begin{aligned}
&\quad let^*\ \Gamma \ := \\
&\textbf{match } \Gamma \textbf{ with} \\
&\qquad\qquad \text{nil} \Rightarrow let\ () = () \\
&\qquad\qquad |\ \Gamma', x_i' : \Box^{j_i}\phi_i \Rightarrow\ (let^*\ \Gamma')\ in\ let\ link(x_i, j_i) = x_i' \\
&\qquad\qquad |\Gamma', \_ \Rightarrow let^*\Gamma'
\end{aligned}
$$

Linking: $\Box^j$-Intro

$$\frac{;\lfloor \Gamma_1 \vdash_{JC_1} M : \phi \qquad \Delta_0; \Gamma_1 \vdash_{JC_1} j :: \phi}{\Delta_0; \Gamma_1 \vdash_{JC_1} let^{*}\Gamma \ in\ link\ (M, j) : \Box^j \phi} \ \Box\text{-INTRO}$$

Π Kind Inhabitation: Linking process generators

$$\frac{\Delta_0, s :: \phi; \vdash_{\mathsf{JC}_1} t : \mathsf{T}}{\Delta_0; \vdash_{\mathsf{JC}_1} Js :: \phi.\ t : \Pi s :: \phi.\mathsf{T}}\ \Pi\text{-INTRO}$$

$$\frac{\Delta_0; \vdash_{\mathsf{JC}_1} t : \Pi s :: \phi.\mathsf{T} \qquad \Delta_0; \vdash_{\mathsf{JC}_1} j :: \phi}{\Delta_0; \vdash_{\mathsf{JC}_1} (t\ j) : \mathsf{T}[\mathsf{s} := \mathsf{j}]}\ \Pi\text{-ELIM}$$

JCalc:Full *K* modality

- We have generalized type construction of $\Box$ types with mutual induction to arbitrary degree.
- With an appropriate extension of the language from justification logic we obtain $IK^{\rightarrow}$ reasoning with justified modal types
- Validity contexts become telescopes. E.g: $s :: \phi, t :: \Box^s\phi, u :: \Box^t\Box^s\phi$
- Exploring computational interpretation as higher-order linking process. I.e. implementations of client code that are themselves clients of some signature.

JCalc:Full *K* modality

- We have generalized type construction of $\Box$ types with mutual induction to arbitrary degree.
- With an appropriate extension of the language from justification logic we obtain $IK^{\rightarrow}$ reasoning with justified modal types
- Validity contexts become telescopes. E.g: $s :: \phi, t :: \Box^s\phi, u :: \Box^t\Box^s\phi$
- Exploring computational interpretation as higher-order linking process. I.e. implementations of client code that are themselves clients of some signature.

JCalc:Full K modality

- We have generalized type construction of $\Box$ types with mutual induction to arbitrary degree.
- With an appropriate extension of the language from justification logic we obtain $IK^{\rightarrow}$ reasoning with justified modal types
- Validity contexts become telescopes. E.g: $s :: \phi, t :: \Box^{s}\phi, u :: \Box^{t}\Box^{s}\phi$
- Exploring computational interpretation as higher-order linking process. I.e. implementations of client code that are themselves clients of some signature.

JCalc:Full K modality

- We have generalized type construction of $\Box$ types with mutual induction to arbitrary degree.
- With an appropriate extension of the language from justification logic we obtain $IK^{\rightarrow}$ reasoning with justified modal types
- Validity contexts become telescopes. E.g: $s :: \phi, t :: \Box^{s}\phi, u :: \Box^{t}\Box^{s}\phi$
- Exploring computational interpretation as higher-order linking process. I.e. implementations of client code that are themselves clients of some signature.

Metatheoretic Results

- We have shown: Weakening, contraction and exchange (in paper).
- We have progress and preservation for call-by-value semantics.
- Currently working big-step semantics that reveal accurately the algorithmic character of K-Intro rule.
- Additionally, working on termination and cut-elimination.

Metatheoretic Results

- We have shown: Weakening, contraction and exchange (in paper).
- We have progress and preservation for call-by-value semantics.
- Currently working big-step semantics that reveal accurately the algorithmic character of K-Intro rule.
- Additionally, working on termination and cut-elimination.

Metatheoretic Results

- We have shown: Weakening, contraction and exchange (in paper).
- We have progress and preservation for call-by-value semantics.
- Currently working big-step semantics that reveal accurately the algorithmic character of K-Intro rule.
- Additionally, working on termination and cut-elimination.

Metatheoretic Results

- We have shown: Weakening, contraction and exchange (in paper).
- We have progress and preservation for call-by-value semantics.
- Currently working big-step semantics that reveal accurately the algorithmic character of K-Intro rule.
- Additionally, working on termination and cut-elimination.

Thanks

- Thank you for your time!