Tipos Dependientes: λP

Juan Pablo Yamamoto Zazueta
(jpyamamoto@ciencias.unam.mx)

***Introducción*— Los tipos dependientes son una forma de entender la extensión del cálculo λ que lleva la correspondencia de Curry-Howard en la dirección de la lógica de predicados.**

Los tipos dependientes permiten expresar una relación en donde los objetos determinan al tipo vinculado. Una de las formas de estudiar esto, es a través del cálculo λP.

I. PREÁMBULO

La correspondencia Curry-Howard habla sobre la relación que existe entre lenguajes de programación y lógica. Esto es mejor descrito por la frase: **proposiciones como tipos y pruebas como programas**. Es decir, existe una correspondencia bidireccional entre los lenguajes de programación tipados y los sistemas de deducción natural en ciertas lógicas.

No entraré a revisar en detalle la base de tal correspondencia que remonta a la Lógica Proposicional Intuicionista y el Cálculo λ con tipos simples, pues ya se ha revisado con anterioridad en el curso. De la misma manera, omitiré la introducción a la Lógica de Predicados.

No obstante, lo que compete a este proyecto de investigación es cómo se puede entender la correspondencia *proposiciones como tipos y programas como pruebas* a la luz de tal lógica.

Comencemos por identificar que la particularidad de la Lógica de Predicados está en el uso de predicados para hablar sobre objetos. Visto de otra forma, la fórmula $P(x_1, ..., x_n)$ es una proposición que depende de los x_i particulares que la instancian.

Además, cabe recordar que en lógica de predicados contamos con cuantificadores. Por simplicidad, nos limitaremos a revisar el fragmento implicacional con cuantificador universal en un inicio, y más adelante se verá cómo es posible extender el sistema con el cuantificador existencial.

Siguiendo la correspondencia *proposiciones como tipos*, debemos dar un sistema en donde los tipos dependan de los objetos particulares que componen al tipo.

Conviene dejar de lado la noción intuitiva de los tipos como etiquetas que describen a los términos en un nivel separado de los términos, puesto que ahora uno o más términos pueden afectar un tipo, lo cuál no empata con la idea usual. Además, es necesaria una forma de cuantificar los objetos sobre los tipos.

Existen distintas propuestas de extensiones al cálculo λ que cumplen con estas propiedades. Nos enfocaremos en la conocida como λP.

II. CÁLCULO λP

A continuación presento el sistema λP. Primero reviso la sintaxis, sus reglas semánticas, seguido de algunas propiedades interesantes que presentan.

A. Sintaxis

Context	Γ	$::=$	$\varnothing$
		$\mid$	$\Gamma, (x : \phi)$
		$\mid$	$\Gamma, (\alpha : \kappa)$
Kind	κ	$::=$	$*$
		$\mid$	$(\Pi x : \phi . \kappa)$
Type	ϕ	$::=$	α
		$\mid$	$(\forall x : \phi . \phi)$
		$\mid$	(ϕM)
		$\mid$	$(\lambda x : \phi . \phi)$
Term	M	$::=$	x
		$\mid$	(MM)
		$\mid$	$(\lambda x : \phi . M)$

Es posible identificar varias particularidades en la sintaxis.

Una que resalta es la necesidad de los *kinds*. Los *kinds* usualmente se describen como los "**tipos de los tipos**". Puesto que en este sistema un tipo puede ser operado (nótese la definición de

Reglas para construir *kinds*:

$$\vdash * : \Box \qquad \frac{\Gamma, x : \tau \vdash \kappa : \Box}{\Gamma \vdash \Pi x : \tau . \kappa : \Box}$$

Reglas para asignar *kinds*:

$$\frac{\Gamma \vdash \kappa : \Box}{\Gamma, \alpha : \kappa \vdash \alpha : \kappa} \ (\alpha \notin \text{dom}(\Gamma)) \qquad \frac{\Gamma, x : \tau \vdash \sigma : *}{\Gamma \vdash (\forall x : \tau . \sigma) : *}$$

$$\frac{\Gamma \vdash \varphi : (\Pi x : \tau . \kappa) \quad \Gamma \vdash M : \tau}{\Gamma \vdash (\varphi M) : \kappa[x := M]} \qquad \frac{\Gamma, x : \tau \vdash \varphi : \kappa}{\Gamma \vdash (\lambda x : \tau . \varphi) : (\Pi x : \tau . \kappa)}$$

Reglas de tipado:

$$\frac{\Gamma \vdash \tau : *}{\Gamma, x : \tau \vdash x : \tau} \ (x \notin \text{dom}(\Gamma)) \qquad \frac{\Gamma \vdash M : (\forall x : \tau . \sigma) \quad \Gamma \vdash N : \tau}{\Gamma \vdash (MN) : \sigma[x := N]} \qquad \frac{\Gamma, x : \tau \vdash M : \sigma}{\Gamma \vdash (\lambda x : \tau . M) : (\forall x : \tau . \sigma)}$$

Reglas de debilitamiento ($x \notin \text{dom}(\Gamma)$ a la izquierda, $\alpha \notin \text{dom}(\Gamma)$ a la derecha):

$$\frac{\Gamma \vdash \tau : * \quad \Gamma \vdash \kappa : \Box}{\Gamma, x : \tau \vdash \kappa : \Box} \qquad \frac{\Gamma \vdash \tau : * \quad \Gamma \vdash \varphi : \kappa}{\Gamma, x : \tau \vdash \varphi : \kappa} \qquad \frac{\Gamma \vdash \tau : * \quad \Gamma \vdash M : \sigma}{\Gamma, x : \tau \vdash M : \sigma}$$

$$\frac{\Gamma \vdash \kappa : \Box \quad \Gamma \vdash \kappa' : \Box}{\Gamma, \alpha : \kappa \vdash \kappa' : \Box} \qquad \frac{\Gamma \vdash \kappa : \Box \quad \Gamma \vdash \varphi : \kappa'}{\Gamma, \alpha : \kappa \vdash \varphi : \kappa'} \qquad \frac{\Gamma \vdash \kappa : \Box \quad \Gamma \vdash M : \sigma}{\Gamma, \alpha : \kappa \vdash M : \sigma}$$

Reglas de conversión:

$$\frac{\Gamma \vdash \varphi : \kappa \quad \Gamma \vdash \kappa' : \Box}{\Gamma \vdash \varphi : \kappa'} \ (\kappa =_\beta \kappa') \qquad \frac{\Gamma \vdash M : \sigma \quad \Gamma \vdash \sigma' : *}{\Gamma \vdash M : \sigma'} \ (\sigma =_\beta \sigma')$$

Fig. 1. Reglas de tipado para λP.

la categoría sintáctica Type), se utilizan los *kinds* para garantizar que tales operaciones están aplicadas adecuadamente (de la misma manera que hacen los tipos en el cálculo λ con tipos simples para garantizar que las funciones están aplicadas apropiadamente sobre los términos).

El *kind* $*$ debe entenderse como el *kind* de un tipo simple (funciones y variables con tipos no dependientes). Por otro lado, $(\Pi x : \phi . \kappa_i)$ es dependiente. Es decir, conviene leer la expresión como: *el kind* κ_i *que depende de* x *de tipo* ϕ.

De forma similar se leé $(\forall x : \phi_1 . \phi_2)$ como: *el tipo* ϕ_2 *que depende de* x *de tipo* ϕ_1. La dependencia siempre es de un término con un tipo concreto, no se operan tipos en tipos (eso corresponde a otra extensión del cálculo λ).

Para enfatizar lo que ya ha sido mencionado, nótese que Π y $\forall$ son la misma idea pero en distintos niveles: abstracción de términos en *kinds* y términos en tipos.

B. Semántica

En la figura 1 se encuentran las reglas que rigen el sistema λP.

El símbolo $\Box$ no es propiamente un término en la sintaxis, sino un símbolo en la meta-teoría para afirmar que se tiene una derivación de *kinds* válida.

Se puede entender $\Gamma \vdash (\kappa : \Box)$ mediante la siguiente lectura: κ *es un kind bien formado dado el ambiente* Γ.

Lo mismo sucede al nivel de tipos al entender por $\Gamma \vdash \tau : *$ que se tiene un tipo τ bien formado.

Las reglas para la sustitución se definen de la manera usual, únicamente haciendo la aclaración de que al tener términos en tipos, también deben ser sustituidos. La regla destacada está en la figura 2, donde se debe entender a Λ como alguna de las opciones $\lambda, \forall$ o Π:

$$(\Lambda y : \phi.E)[x := M]$$
$$=(\Lambda y : \phi[x := M].E[x := M])$$

Fig. 2. Regla de sustitución para λP.

Las reglas para la β reducción se definen de la manera usual, con base en las reglas de la figura 3:

$$(\lambda x : \tau.M)N \longrightarrow_\beta M[x := N]$$
$$(\lambda x : \tau.\phi)N \longrightarrow_\beta \phi[x := N]$$

Fig. 3. Reglas de $\longrightarrow_\beta$ para λP.

Entre las particularidades más interesantes del sistema λP se encuentran las reglas de conversión. Esto es debido a que utilizan la operación $=_\beta$, realizando cómputo al nivel de *kinds* y tipos. Esto complica en gran medida el análisis de tipos en este sistema, haciéndolo un problema no trivial.

Algo a notar es la repetición de los mismos patrones en las reglas para los 3 niveles distintos: *términos, tipos* y *kinds*, debido a la muy explícita distinción entre ellos. Ejemplo de ello son las 6 reglas para el *debilitamiento*. Más adelante se verá otra propuesta para eliminar tal distinción.

C. Propiedades

El sistema λP cumple con ciertas propiedades interesantes por las garantías que presentan. A continuación se listan algunas, sin su demostración (las demostraciones se pueden encontrar en las referencias).

Teorema 1 (Preservación):

(i) Si $\Gamma \vdash M : \sigma$ y $M \longrightarrow_\beta M'$, entonces $\Gamma \vdash M' : \sigma$.

(ii) Si $\Gamma \vdash \varphi : \kappa$ y $\varphi \longrightarrow_\beta \varphi'$ entonces $\Gamma \vdash \varphi' : \kappa$.

(iii) Si $\Gamma \vdash \kappa : \Box$ y $\kappa \longrightarrow_\beta \varphi'$, entonces $\Gamma \vdash \varphi' : \kappa$.

Nótese que las 3 proposiciones que conforman al teorema refieren a la misma idea de "preservación del tipado", pero cada una en un nivel distinto.

Teorema 2 (Normalización Fuerte):
Si $\Gamma \vdash M : \tau$ (M es un término bien tipado), entonces toda reducción de M termina. Es decir, existen $M_0, M_1, ..., M_n$ finitos tales que

$$M_0 \longrightarrow_\beta M_1 \longrightarrow_\beta \cdots \longrightarrow_\beta M_n$$

(con $M_0 = M$) y M_n está en forma normal.

Teorema 3 (Propiedad de Church-Rosser):
Sean M_1, M_2, M_3 términos bien tipados. Si $M_1 \twoheadrightarrow_\beta M_2$ y $M_1 \twoheadrightarrow_\beta M_3$, entonces existe M_4 tal que $M_2 \twoheadrightarrow_\beta M_4$ y $M_3 \twoheadrightarrow_\beta M_4$.

Los 3 teoremas anteriores, garantizan propiedades útiles, especialmente al usar λP como base para la implementación de lenguajes de programación. Por ejemplo, al descartar programas mal tipados se garantiza la correcta ejecución del código. De igual forma, tiene ciertas desventajas, como lo sería el hecho de que un lenguaje con la propiedad de Normalización fuerte, no poseé Turing-completitud.

$$\frac{}{\vdash * : \Box} \text{ (Ax)}$$

$$\frac{\Gamma \vdash A : s \qquad x \notin \text{dom}(\Gamma)}{\Gamma, x : A \vdash x : A} \text{ (Var)}$$

$$\frac{\Gamma \vdash A : * \qquad \Gamma, x : A \vdash B : s}{\Gamma \vdash (\Pi x : A.B) : s} \text{ (Prod)}$$

$$\frac{\Gamma, x : A \vdash B : C \qquad \Gamma \vdash (\Pi x : A.C) : s}{\Gamma \vdash (\lambda x : A.B) : (\Pi x : A.C)} \text{ (Abs)}$$

$$\frac{\Gamma \vdash A : (\Pi x : B.C) \qquad \Gamma \vdash D : B}{\Gamma \vdash (AD) : C[x := D]} \text{ (App)}$$

$$\frac{\Gamma \vdash A : B \qquad \Gamma \vdash C : s \qquad x \notin \text{dom}(\Gamma)}{\Gamma, x : C \vdash A : B} \text{ (Weak)}$$

$$\frac{\Gamma \vdash A : B \qquad \Gamma \vdash B' : s \qquad B =_\beta B'}{\Gamma \vdash A : B'} \text{ (Conv)}$$

Fig. 4. Reglas de tipado para λP compacto.

III. λP COMPACTO

Como vimos anteriormente, el hecho de distinguir los niveles *kinds, tipos* y *términos* agrega complejidad al sistema y resulta en reglas con patrones repetitivos.

Como alternativa a ello, se presenta el siguiente sistema, λP compacto.

Context	Γ	$::=$	$\varnothing$
		$\mid$	$\Gamma, (x : A)$
Sorts	s	$::=$	$*$
		$\mid$	$\Box$
Expr	A	$::=$	x
		$\mid$	s
		$\mid$	(AA)
		$\mid$	$(\lambda x : A.A)$
		$\mid$	$(\Pi x : A.A)$

El sistema en su sintaxis es un poco más liberal, pues permite la generación de términos que en λP no eran válidos. Ejemplo de esto es que $\Box$ ya es un término de la gramática. No obstante, las reglas de tipado van a restringir los programas de tal manera que los programas válidos en λP compacto tengan su análogo en λP y viceversa.

En λP compacto ya no se tiene una sintaxis distinta para cuantificar tipos y *kinds*. Ambos niveles son colapsados en uno mismo, cuantificado con el producto dependiente Π.

Esa modificación resulta en una menor cantidad de reglas necesarias para el análisis sintáctico de las expresiones, como se puede ver en la figura 4.

El sistema preserva todas las propiedades que se enunciaron como teoremas anteriormente.

En particular, es conveniente mencionar cómo rescatar los **tipos función** en este sistema. Es fácil convencerse de la siguiente equivalencia:

$$A \to B \simeq \Pi x : A.B$$

donde B no tiene dependencias.

Este nuevo sistema tiene la ventaja de que puede ser alterado facilmente para permitir otras extensiones al cálculo λ con tipos simples.

Basta modificar la regla $(Prod)$ para permitir distintas dependencias, generando los sistemas $\lambda 2$, $\lambda\omega$, λC, entre otros. Los lenguajes generados y las relaciones entre ellos son los estudiados a través del cubo λ (figura 5).

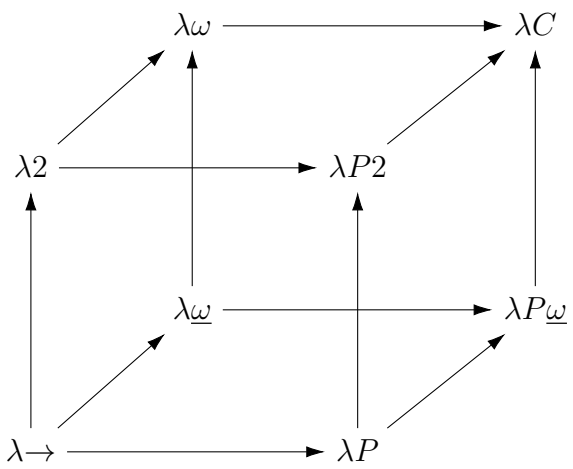

Fig. 5. Cubo λ

Además, puede darse una generalización con un mayor nivel de abstracción permitiendo generar los *Pure Type Systems*. Estos sistemas permiten generar aún más extensiones a las planteadas anteriormente. Por listar algunos:

- λU: órden superior con dominios polimórficos y cuantificación sobre todos los dominios.
- $\lambda CC\omega$: tipos dependientes, órden superior y polimorfismo, con jerarquía de *sorts* infinitos.
- λHOL: corresponde a Lógica de Orden Superior constructiva.

IV. CORRESPONDENCIA CURRY-HOWARD

Si bien al comienzo de este proyecto motivamos el sistema λP como una posible codificación para la lógica de predicados, en realidad el lenguaje es más poderoso. Bajo la interpretación de *proposiciones como tipos* lo que λP nos da es una lógica de predicados multivariada (del inglés *many-sorted logic*), es decir, con varias categorías de objetos.

Sin embargo, es posible dar la siguiente restricción sobre la sintaxis para limitarse a la lógica de predicados usual.

- Existe una única variable de tipo 0.
- Todos los *kinds* son de la forma

  $$\Pi x_1 : 0.(\Pi x_2 : 0.(\cdots(\Pi x_n : 0.*)\cdots))$$

  o como suele denotarse

  $$0 \Rightarrow 0 \Rightarrow \cdots \Rightarrow 0 \Rightarrow *$$

- Hay un número finito de variables constructoras (donde las operaciones son de términos en tipos) con su firma dada correctamente en su *kind*.
- Los símbolos de función son variables de objetos con tipos (acorde a su aridad)

  $$\forall x_1 : 0.(\cdots(\forall x_2 : 0.0)\cdots)$$

  o como suele denotarse

  $$0 \to \cdots \to 0 \to 0$$

- Los símbolos constantes están dados por objetos distinguidos de tipo 0.
- Las declaraciones en el contexto sólo pueden ser de la forma $(x : 0)$ que corresponden a variables individuales.

Como resumen, en la figura 6 se listan las correspondencias.

V. TIPOS SIGMA

Como ya ha sido mencionado, resulta de interés estudiar la correspondencia entre sistemas de tipos y fórmulas lógicas. Hasta ahora no se ha mencionado cómo podría modelarse el cuantificador existencial visto en el lado del cálculo λ, en particular en λP compacto.

En otras extensiones como el sistema F, se utiliza el cuantificador existencial $\exists x\varphi$ al nivel de los tipos, para simular un par que consiste en un par $\langle \tau, M \rangle$ donde τ codifica el tipo del término, y M atestigua que el tipo existe.

fórmulas	~	tipos
pruebas	~	términos
dominio de individuos	~	constante de tipo 0
términos algebráicos	~	términos de tipo 0
relaciones	~	constructores de *kind* $0 \Rightarrow \cdots \Rightarrow 0 \Rightarrow *$
fórmula atómica $r(t_1, ..., t_n)$	~	tipo dependiente $rt_1...t_n$
fórmula universal	~	tipo producto
prueba por generalización	~	abstracción $\lambda x : 0.M^{\varphi}$
prueba por *modus ponens*	~	aplicación $M^{\forall x:0.\varphi} N^0$

Fig. 6. Correspondencia Curry-Howard para λP y lógica de predicados.

$$\frac{\Gamma \vdash A : * \quad \Gamma, x : A \vdash B : s}{\Gamma \vdash \Sigma x : A.B : s} \qquad \frac{\Gamma \vdash M : A \quad \Gamma, x : A \vdash N : B[x := M]}{\Gamma \vdash \langle M, N \rangle : (\Sigma x : A.B)}$$

$$\frac{\Gamma \vdash M : (\Sigma x : A.B)}{\Gamma \vdash \mathtt{fst}(M) : A} \qquad \frac{\Gamma \vdash M : (\Sigma x : A.B)}{\Gamma \vdash \mathtt{snd}(M) : B[x := \mathtt{fst}(M)]}$$

Fig. 7. Reglas de tipado para la extensión con tipos Σ.

Se realiza algo similar para tener tipos existenciales dependientes, mediante el operador de suma dependiente Σ.

La sintaxis se extiende de la siguiente manera:

$$\begin{array}{llll} \textsf{Expr} & A & ::= & ... \\ & & | & \langle A, A \rangle \\ & & | & (\Sigma x : A.A) \\ & & | & \mathtt{fst}(A) \\ & & | & \mathtt{snd}(A) \end{array}$$

Fue añadido $\langle \cdot, \cdot \rangle$ como constructor para los tipos Σ, el término que codifica tipos Σ y las proyecciones para destruir existenciales.

Las reglas para el tipado están en la figura 7.

Las reglas para evaluación son (figura 8):

$$\mathtt{fst}(\langle M, N \rangle) \longrightarrow_\beta M$$
$$\mathtt{snd}(\langle M, N \rangle) \longrightarrow_\beta N$$

Fig. 8. Reglas de $\longrightarrow_\beta$ para la extensión con tipos Σ.

VI. CONCLUSIÓN

En este proyecto se han descrito los sistemas λP y λP compacto: su sintaxis, reglas de tipado y evaluación. También se comprendió mejor el sistema λP a la luz de la correspondencia *proposiciones como tipos y pruebas como términos*. Finalmente se dio una extensión que codifica el cuantificador existencial como un tipo suma dependiente.

Podemos concluir haciendo énfasis en la utilidad de estas extensiones. El caso de uso más popular es **LF Logical Framework** que proveé una manera de definir lógicas. Sobre ese mismo fundamento se desarrolló el lenguaje de programación **Twelf** con paradigma de *programación lógica*, el cuál tiene ciertas utilidades para su uso como asistente de pruebas.

Además, es importante resaltar su utilidad como la base para otros sistemas más elaborados como es el **cálculo de construcciones** λC. Este se ha utilizado como la base para asistentes de prueba como **Coq** y **Lean**.

REFERENCIAS

[1] Henk (Hendrik) Barendregt. An Introduction to Generalized Type Systems. *Journal of Functional Programming*, 1:125–154, April 1991.

[2] Maria Joao Frade. Calculus of Inductive Constructions. 2009.

[3] Herman Geuvers. Pure Type Systems revisited. 2013.

[4] Martin Hofmann. Syntax and Semantics of Dependent Types. In Andrew M. Pitts and P. Dybjer, editors, *Semantics and Logics of Computation*, Publications of the Newton Institute, pages 79–130. Cambridge University Press, Cambridge, 1997.

[5] Jason Koenig. Dependent Types. https://www.cs.cmu.edu/~rwh/courses/hott/notes/notes_week4.pdf, 2013.

[6] nLab authors. Pure type system in nLab. https://ncatlab.org/nlab/show/pure+type+system, 2023.

[7] Cody Roux. Cody Roux - Pure Type Systems - Boston Haskell Meetup. https://www.slideshare.net/slideshow/cody-roux-pure-type-systems-boston-haskell-meetup/46426281, March 2015.

[8] Morten Heine Sørensen and Paweł Urzyczyn. *Lectures on the Curry-Howard Isomorphism*. Number vol. 149 in Studies in Logic and the Foundations of Mathematics. Elsevier, Amsterdam ; Boston, 1st ed edition, 2006.

[9] Matthieu Sozeau. Introduction to Dependent Type Theory (3/4). *TYPES 2018*, 2018.

[10] Petr Štěpánek. Lambda Calculus. https://www.ktiml.mff.cuni.cz/KTIML-13-version1-calculus3.pdf, 2003.

[11] Wikipedia contributors. Lambda cube. *Wikipedia*, February 2024.