

Questions: Semantic and Computational Issues

Veneeta Dayal and Chung-chieh Shan

November 1, 2006

(1) The woman that {every man/#he}_i loves is his_i mother.

Use properties as types to be unified? Something like:

(2)	the woman that	:	$((\text{man} \rightarrow \text{woman}) \rightarrow t) \rightarrow \text{man} \rightarrow \text{woman}$
	every man	:	$(\text{man} \rightarrow t) \rightarrow t$
	loves	:	$\text{woman} \rightarrow \text{man} \rightarrow t$
	is	:	$(\text{man} \rightarrow \text{woman}) \rightarrow (\text{man} \rightarrow \text{woman}) \rightarrow t$
	his	:	$\text{man} \rightarrow \text{man}$
	mother	:	$\text{man} \rightarrow \text{woman}$

Types need to depend on terms (for individuals and worlds): *the woman that, every man, his (man)*.

(3) Every woman_j knows that the gift from her_j that every son_i of hers_j loves is the dictionary she_j gave him_i.

We need notation (a type system) beyond \rightarrow .

1. Dependent product

(4) Types may now depend on terms.

$\text{st} : (e \rightarrow t) \rightarrow \text{type}$ (“such that”; “satisfying”)

$\llbracket \text{every} \rrbracket : \prod_{p:e \rightarrow t} (\text{st } p \rightarrow t) \rightarrow t$

(5) By analogy with product notation

$$6! = \prod_{x=1}^6 x$$

(6) By analogy with ordinary functions

$$5^7 = 7 \rightarrow 5 = \prod_{x=1}^7 5$$

(7) Subsumes ordinary functions

$$\begin{aligned} \text{st} : \prod_{p:\prod_{x:e} t} \text{type} &= (e \rightarrow t) \rightarrow \text{type} \\ \llbracket \text{every} \rrbracket : \prod_{p:\prod_{x:e} t} \prod_{q:\prod_{x:\text{st } p} t} t &= \prod p:(e \rightarrow t). (\text{st } p \rightarrow t) \rightarrow t \\ \llbracket \text{the} \rrbracket : \prod_{p:\prod_{x:e} t} \text{st } p &= \prod p:(e \rightarrow t). \text{st } p \\ \llbracket \text{that} \rrbracket : \prod_{p:\prod_{x:e} t} \prod_{q:\prod_{x:\text{st } p} t} \prod_{x:e} t &= \prod p:(e \rightarrow t). (\text{st } p \rightarrow t) \rightarrow e \rightarrow t \end{aligned}$$

(8) Typing rules for application and abstraction (cf. \forall)

$$\frac{f : \Pi x : \tau. \tau' \quad x : \tau}{f(x) : (\tau' \{x \mapsto \tau\})} \quad \frac{\tau : \text{type} \quad \begin{array}{c} [x : \tau] \\ \vdots \\ E : \tau' \end{array}}{\lambda x. E : (\Pi x : \tau. \tau')}$$

The context is not just a set of variable-type mappings anymore, but a sequence that keeps track of what we know about each variable!

Dependent sum generalizes ordinary products.

2. Deconstructing st

(9) $t : \text{type}$

(10) $\text{true} : t$

(11) $\text{false} : t$

(12) $\text{is-true} : t \rightarrow \text{type}$

(13) $\text{is-true-indeed} : \text{is-true true}$

(14) $\text{st} : (e \rightarrow t) \rightarrow \text{type}$

(15) $\text{st-indeed} : \Pi p : (e \rightarrow t). \Pi x : e. \text{is-true}(p(x)) \rightarrow \text{st } p$

3. A few references

Tim Fernando. 2001. [A type reduction from proof-conditional to dynamic semantics](#). *Journal of Philosophical Logic* 30(2):121–153.

Aarne Ranta. 1994. *Type-theoretical grammar*. New York: Oxford University Press.

Aarne Ranta. 1998. [Syntactic calculus with dependent types](#). *Journal of Logic, Language and Information* 7(4):413–431.

Florian Rabe. 2006. [First-order logic with dependent types](#). In *Automated reasoning: 3rd international joint conference*, ed. N. Shankar and U. Furbach, 377–391. Lecture Notes in Computer Science 4130, Berlin: Springer-Verlag.