# $\langle$e$\rangle$ is not a type

Kyle Rawlins  kgr@jhu.edu  11/9/18 (v. 3)

A relatively common mistake I see in formal semantics is writing type $e$ as $\langle e\rangle$ (and $\langle t\rangle$, and $\langle s\rangle$, ...). While there are some kinds of 'notational abuse' I am *totally fine* with (cf. mixing set and function notation), this is a line I won't cross. Here's why.

**Why this is wrong, pt. 1.** A typical recursive definition for the core type system for compositional semantics looks like this (after Heim & Kratzer 1998 p. 28 ex. 5):

(1) a. $e$ and $t$ are types.
  b. If $X$ and $Y$ are types, then $\langle X, Y\rangle$ is a type.
  c. Nothing else is a type.

(2) a. (i) $D_e := D$ (the set of individuals).
   (ii) $D_t := \{0, 1\}$
  b. For any types $X, Y$, $D_{\langle X,Y\rangle}$ is the set of all functions from $D_X$ to $D_y$.

This syntactic definition in (1) corresponds directly to a semantics in (2): the (1)-a rule tells you the *atomic* types, and the (1)-b rule tells you how to write the type for a function. (2) gives a correspondence between types and sets of metalanguage elements: atomic types are interpreted as indicating elements drawn from the sets in (2)-a, and these two sets are used in (2)-b to recursively define sets corresponding to each type generated by (1)-b.

It can readily seen that there is no way to apply the *a,b* rules in (1) to generate a type like $\langle e\rangle$; any type generated using rule *b* will have a ',' in it somewhere (and have at least two elements), and any type generated by rule *a* will lack brackets. Therefore, by rule *c*, $\langle$e$\rangle$ is not a type. Correspondingly, the set $D_{\langle e\rangle}$ is not defined by these rules either.

**Why this is wrong, pt. 2.** It is instructive to see how to modify a definition like (1) to get it to come out so that $\langle e\rangle$ etc. is a type: what is someone who uses $\langle e\rangle$ implicitly presupposing? However, first one would like to settle what the 'etc.' amounts to. Do you want both $\langle e\rangle$ and $e$ to be types (that mean the same thing)? Do you want to allow $\langle\langle e\rangle\rangle$ (with presumably the same meaning again)? Here is a definition that does this by relaxing the correspondence between type strings and the sets they characterize:

(3) Weird type definition 1
  a. $e$ and $t$ are types.
  b. If $X$ and $Y$ are types, then $\langle X, Y\rangle$ is a type.
  c. If $X$ is a type, then $\langle X\rangle$ is a type.
  d. Nothing else is a type.

(4) Weird type interpretation principle: For any type $X$, $D_X = D_{\langle X\rangle}$

This is perfectly formally coherent (though I'll leave it to the reader to convince themselves that sets like $D_{\langle\langle\langle e\rangle\rangle,\langle\langle e,\langle t\rangle\rangle\rangle\rangle}$ can be well-defined with the appropriate meaning), but *why would you do this*?

Weird type def. 1 most naturally leads to a situation where elements in the meta-language characterized by the type system do not have unique types. If you'd prefer a tighter syntax, things get slightly more complicated. Here's another version, that generates $\langle e\rangle, \langle e, t\rangle$, etc., but not just $e$ or any of the extra bracketing in the first weird type definition:

(5) Weird type definition 2
  a. $e$ and $t$ are pre-types.

b. If $X$ is a pre-type, then $\langle X \rangle$ is a type.
c. If $X$ is a pre-type or composite type and $Y$ is a pre-type or composite type, then $\langle X, Y \rangle$ is a composite type.
d. If $X$ is a composite type, then $X$ is a type.
e. Nothing else besides the things strictly declared types in the above rules is a type.

The simplest semantics for this would also use the 'weird types interpretation principle' in (4) to map the generated types onto sets (also mapping the pre-types to sets). This doesn't exhaust the space of possible definitions you could try, and there might be simpler ones out there (e.g. using an obligatory bracket simplification rule, similar to parenthesis conventions in logic), but for all of them, the question arises: what's the point? This is *way* more complicated than (1).

**Why does this matter?** After all, isn't it completely obvious what ⟨e⟩ is supposed to mean? I think that most people make this mistake when learning the simply typed lambda calculus in a semantics class by drawing the inference that ⟨⟩ means, "is a type". This is an understandable mistake – in a typical semantics textbook presentation, and for that matter a lot of published literature, the vast majority of types you see in practice do obey this apparent convention!

The way to interpret ⟨⟩ is as what is called a *type constructor* in the programming languages literature. In particular, ⟨⟩ is a binary type constructor (type constructors have an arity) for functions: any type built using this constructor maps on to a set of functions (not individuals). That is, ⟨⟩ can be thought of as a function from pairs of types to types that characterizes how to build a certain (specific) kind of complex type from its arguments, corresponding to the sets described in (2)-b. Given this, a plausible interpretation for the weird type definitions emerges – ⟨⟩ there is ambiguous or overloaded, used for both the standard binary functional type constructor, and for a unary *vacuous type constructor* – in fact a vacuous type constructor is what the 'weird types interpretation principle' sketched in (4) amounts to. Again, this is all formally coherent, but realizing that unary ⟨⟩ needed to make these systems work is just a vacuous constructor should hopefully illustrate that there's no good reason to write $\langle e \rangle$ in the first place. That is, this system is strictly more complicated than (1) to no effect whatsoever.

It may also be helpful to realize that it is easy, and often linguistically useful, to have more type constructors than just the one. For example, in the Lambda Notebook project (`http://lambdanotebook.com/`), the interpretation system uses () as the type constructor for tuples (at any finite arity), and {} as a unary type constructor for sets. So for example, $\{\langle s, t \rangle\}$ is the type of a set of functions from worlds to truth values in that system (the type of a question on a Hamblin semantics), and $\langle (e, e), t \rangle$ is a function from a pair of entites to a truth value (e.g. a binary predicate, an uncurried version of $\langle e, \langle e, t \rangle\rangle$). Programming languages sometimes used for formal semantics such as Haskell (see e.g. van Eijck & Unger 2010) have even richer systems of type constructors and related notions that can do a lot of interesting work. In addition, the ⟨⟩ convention is arbitrary. As Simon Charlow (p.c.) has pointed out, the Haskell type constructor notion for functions, written as an infix operator `X -> Y`, is not susceptible to this error. Perhaps linguistic semantics should switch.

**In summary**, don't write $\langle e \rangle$ as a type, because while it can be made coherent if you really, really want, (i) it isn't coherent on the usual definitions, and (ii) the work needed to make it coherent just makes things more complicated to no gain, while obscuring the meaning of ⟨⟩ as a binary type constructor.

**Thanks** to Simon Charlow and Peet Klecha (and others) for a twitter discussion about this.

van Eijck, Jan & Christina Unger. 2010. *Computational semantics with functional programming*. Cambridge University Press.
Heim, Irene & Angelika Kratzer. 1998. *Semantics in generative grammar*. Malden: Blackwell.