Technical primer

Patrick D. Elliott

October 8, 2022

Contents

1 Prerequisites

I'll be teaching this class presupposing the following background:

- At least one introductory course on compositional semantics, with similar coverage to, e.g., *Semantics in Generative Grammar* (Heim & Kratzer 1998) or *Invitation to Formal Semantics* (`https://eecoppock.info/bootcamp/semantics-boot-camp.pdf`).
- Basic knowledge of:

– Set theory.

– Logic (propositional and first order).

The main formal analytical tool we'll be making use of in this class is the *Simply-Typed Lambda Calculus*. Before we dive into linguistic issues, we'll spend some time going through the basics.

2 Syntax of the Simply-Typed Lambda Calculus (STLC)

The lambda calculus was invented by Alonso Church in the early 20th century as a formal language for talking about functions.

The simply-typed variant has been overwhelmingly adopted in formal semantics as a kind of *lingua france* for reasoning about how complex meanings are composed from simpler meanings.

Note that the discussion here is primarily based on Chapters 2 and 3 of (Carpenter 1998).

2.1 Types

You can think of types as the *syntactic categories* of the STLC - they provide formal constraints on what kind of things can combine.

There are just two kinds of types we'll see in this course:

- Basic types: E, T
- Function types.

The basic types are just that - primitives. E is used to classify expressions of the STLC which denote *individuals*, and T is is used to classify expressions which denote *truth-values*.

We'll exploit a general recipe for talking about *function types*.

Definition 2.1. Function types. If σ and τ are **types**, then $\sigma \to \tau$ is a **function type**.

Note that we can use our function type recipe to generate an *infinite* number of types! Unlike in many grammatical formalisms (with the notable exception of categorial grammars), we have, in essence, an infinite number of syntactic categor

We use function types to classify expressions which denote functions. For example, $T \to T$ is the type of a function from truth-values to truth-values (this might be exploited for something like negation).

N.b. that $\to$ is *right-associative*, which means that, e.g., $E \to E \to T$ is parsed as $E \to (E \to T)$ (not $(E \to E) \to T$!).

2.2 Variables and constants

Expressions of the STLC are built up out of variables and constants, which you can conveniently think of as the 'lexical items'.

Unlike in other logical languages you might be familiar with, here variables and constants are categorized by *type*.

Every type is associated with a (countably infinite!) set of variables.

We'll typically use $x, y, z, \ldots$ for variables of type E, $P, Q, R, \ldots$ for variables of type $E \to T$, etc., but ultimately it doesn't matter much what we use as variable names.

Constants will typically be used to talk about 'lexical' concepts, i.e., **Louise** is a constant of type E, **run** is a constant of type $E \to T$, and **not** is a constant of type $T \to T$.

You can be explicit about the types of constants and variables using type annotations, but these can be omitted when the type is obvious:

- $\textbf{Louise}_E$, $\textbf{run}_{E \to T}$
- x_E, $R_{E \to E \to T}$

It's also common to use a colon when declaring the type of an expression:

- **Louise** : E

2.3 functional applications

This is where the magic starts to happen - we're going to define some recursive syntactic rules for constructing expressions from expressions.

The most fundamental such complex expression is a **functional application**.

Definition 2.2. functional application: If α is an expression of type $\sigma \to \tau$, and, β is an expression of type τ, then $\alpha(\beta)$ is a *functional application* of type τ.

Crucially, the type system restricts what counts as a well-formed application (just like syntactic categories restrict what can merge with what).

For example, $\textbf{not}(\textbf{Josie})$ is an ill-formed application, assuming that **not** : $T \to T$, and **Josie** : E.

Are the following valid functional applications? Make explicit what you're assuming about types:

(1) $\textbf{hugs}(\textbf{Louise})$

(2) **Josie**(**left**)

(3) **not**(**sad**(**Sarah**))

Unlike in predicate logic, we have no way of talking about n-ary predicates in the $STLC$, rather an n-ary predicate is always encoded as a *curried* function; complex expressions are then built up by successive applications.

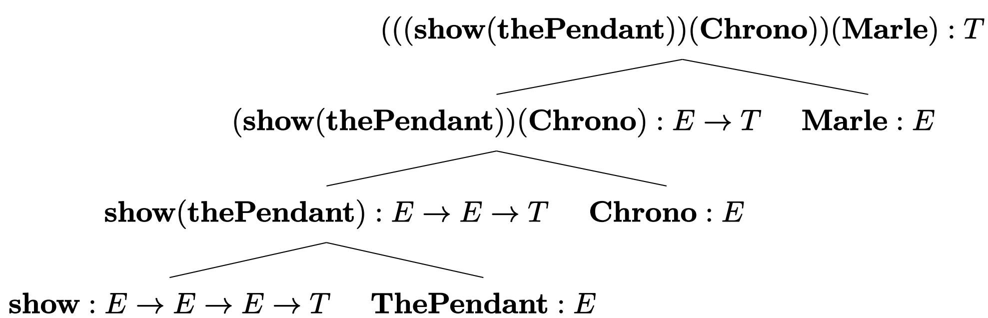
(4) **give** : $E \to E \to E \to T$

(5) **kiss** : $E \to E \to T$

(6) **and** : $T \to T \to T$

One way of visualizing the structure of a complex expression is as a tree diagram, where each non-terminal node represents a functional application.

(7)



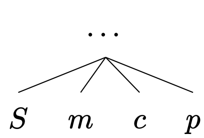
That's a lot of parentheses! We'll typically assume that functional application associates to the left, so we can rewrite the above as:

(8) **show**(**thePendant**)(**Chrono**)(**Marle**)

This brings out what's so compelling about the STLC as a tool for analyzing natural languages - there's a parallelism between the structures implicit in natural language syntax, and the structure of the logical language. This makes it especially easy to translate expressions of natural language into expressions of the STLC.

Compare and contrast the flat structure of a first-order logic expression such as $S(m, c, p)$

(9)



2.4 Functional abstraction

The trademark feature of the lambda calculus (and when its name), is the complex expression known as a *functional abstraction.*

Definition 2.3. Functional abstraction: If x is a variable of type σ, and α is an expression of type τ, then $\lambda x\,.\,\alpha$ is a functional abstraction of type $\sigma \to \tau$.

Abstraction always produces a function type.

Once we come round to the semantics of the STLC, we'll see that there's a special rule for interpreting variables that occur within the body of a functional abstraction.

Are the following all well-formed functional abstractions? Comment on any assumptions we need to make about types.

(10) $\lambda x\,.\,\mathbf{likes}_{E\to E\to T}(x)(x)$

(11) $\lambda y\,.\,\mathbf{hug}_{E\to E\to T}(\mathbf{Louise})(x)$

(12) $\lambda x\,.\,\mathbf{Josie}$

(13) $(\lambda x\,.\,x)(\mathbf{Nathan})$

3 Semantics of the STLC

That's all there is to the syntax of the SLTC, but, since this is a semantics course, we need a way of connecting up these representations with some language-external reality.

3.1 Typed domains

Each type is mapped to a *domain* of values - given that we have a recipe for constructing an infinite number of types, we need a recipe for constructing an infinite number of domains.

We start by specifying the domains of the basic types (in our case, E, T).

- $\mathbf{Dom}_E = \{\, x \mid x \text{ is an individual} \,\}$
- $\mathbf{Dom}_T = \{\, \mathbf{true}, \mathbf{false} \,\}$

Functional types are assigned domains consisting of sets of *functions*; given a functional type $\sigma \to \tau$, $\mathbf{Dom}_\sigma$ gives the *domain* of the function, and $\mathbf{Dom}_\tau$ gives the *codomain* of the function.

Definition 3.1. Domain of a functional type: $\mathbf{Dom}_{\sigma\to\tau} = \{\, f \mid \mathbf{Dom}_\sigma \to \mathbf{Dom}_\tau \,\}$

To give a concrete example, we can in fact enumerate every member in the domain of $T \to T$ (do this!).

3.2 Interpreting the STLC

Given typed domains, the semantics of the STLC is specified by defining the interpretation function $[\![.]\!]$, which maps constants to values, subject to the following constraint (this just makes such, for example, **not** isn't mapped to an individual or something):

Definition 3.2. Type-respecting interpretation: $[\![.]\!]$ is type-respecting if, for any constant $c : \sigma$, $[\![c]\!] \in \mathbf{Dom}_\sigma$.

Now, we recursively define the denotation of an expression relative to an assignment function g (a total function from variables to values).

The denotations of variables/constants is easy:

- $[\![x]\!]^g = g(x)$, if x is a variable.
- $[\![c]\!]^g = [\![c]\!]$, if c is a constant.

The denotation of a function application involves... applying a function to an argument!

- $[\![\alpha(\beta)]\!]^g = [\![\alpha]\!]^g\,([\![\beta]\!]^g)$

The denotation of a functional abstraction is a little more involved. It always returns a function:

- $[\![\lambda x\,.\,\alpha]\!]^g = f$ s.t. $f(a) = [\![\alpha]\!]^{g[x \to a]}$

$g[x \to a]$ is the assignment function that is exactly like g, except the variable x is mapped to a (assuming that $x : \sigma$, and $a \in \mathbf{Dom}_\sigma$).

4 Proof theory for the STLC

One of the beautiful features of using the STLC for compositional semantics is that it has a simple and elegant proof theory, stated in terms of a notion of simplification called *reduction*, written as $\Rightarrow$.

We can *simplify* complex expressions using syntactic simplification rules, without having to compute the semantic value of a given expression at every step - thanks to the *soundness* of the STLC, we can be sure that - if we don't make any mistakes in our proof - the resulting expression can be interpreted without having to interpret every intermediate step.

The standard axioms for the STLC are as follows:

4.1 α-reduction

The intuition behind alpha reduction is that we can freely rename bound variables, e.g.:

(14) $\vdash \lambda x \,.\, \lambda y \,.\, \mathbf{likes}(x)(y) \Rightarrow \lambda z \,.\, \lambda y \,.\, \mathbf{likes}(z)(y)$

Th definition of alpha reduction is a little complex, since we don't want to accidentally change the meaning of the lambda term.

(15) $\vdash \lambda x \,.\, \alpha \Rightarrow \lambda y \,.,(\alpha[x \rightarrow y])$, where y isn't a free variable in α, and y is free for x in α.

The first application condition ensures that we don't accidentally bind free variables using alpha reduction, i.e.:

(16) $\nvdash \lambda x \,.\, \mathbf{likes}(x)(y) \Rightarrow \lambda y \,.,\mathbf{likes}(y)(y)$

The second application condition ensures that, e.g., the y substituted for x is not bound in $\alpha[x \rightarrow y]$, since:

(17) $\nvdash \lambda x \,.\, \lambda y \,.\, \mathbf{likes}(x)(y) \Rightarrow \lambda y \,.\, \lambda y \,.\, \mathbf{likes}(y)(y)$

4.2 β-reduction

Beta-reduction is a straightforward simplification scheme for applications; when we beta-reduce, we remove the $\lambda x.$ from the functional expression, and subsistute all occurrences of x in the function body with the argument expression.

(18) $\vdash (\lambda x \,.\, \alpha)(\beta) \Rightarrow \alpha[x \rightarrow \beta]$

Example:

(19) $\vdash (\lambda x \,.\, \mathbf{likes}(x)(x))(\mathbf{Louise}) \Rightarrow \mathbf{likes}(\mathbf{Louise})(\mathbf{Louise})$

4.3 η-reduction

Eta-reduction is another straightforward simplification scheme that we can use to get rid of lambda terms:

(20) $\vdash \lambda x\,.\,(\alpha(x)) \Rightarrow \alpha$

You'll often see semanticists write things like:

(21) $\lambda x\,.\,\mathbf{happy}_{E\rightarrow T}(x)$

Note that, by η-reduction, this is equivalent to just the typed constant **happy**. This is known as the **long form** of an expression of the STLC, and can sometimes be useful for making typing more obvious without explicit annotations.

References

Carpenter, Bob. 1998. *Type-logical semantics* (Language, Speech, and Communication). Cambridge, Mass: MIT Press. 575 pp.

Heim, Irene & Angelika Kratzer. 1998. *Semantics in generative grammar* (Blackwell Textbooks in Linguistics 13). Malden, MA: Blackwell. 324 pp.