Musa Al-hassy https://github.com/alhassy/CatsCheatSheet August 8, 2019

Reference Sheet for Simple Type Theory

Abbreviations

The phrase "$\phi_i : s_i \to t_i$ is a family of functions for each i" is abbreviated "$\phi : \forall i \bullet s_i \to t_i$". Whereas the phrase "$\psi$ maps each i of I to a member of t_i" is abbreviated "$\psi : (i : I) \to t_i$".

STT has a simple and highly uniform syntax

STT has to kinds of syntactic objects.

- *Expressions* denote values including truth values `true` and `false`; they do what both terms and formulae do in first-order logic.
- *Types* denote nonempty sets of values; they are used to restrict the scope of variables, control the formation of expressions, and classify expressions by their values.

A 'type' is a string of symbols defined inductively by the following formation rules. The first two being being for the base types of 'individuals' and Booleans, while the third being an infinite hierarchy of 'function' types.

```
𝒯 ::= ι | 𝔹 | (𝒯 → 𝒯)
```

Semantically, a function type denotes a domain of *total functions*: $[\![\alpha \to \beta]\!] = [\![\alpha]\!] \to [\![\beta]\!]$.
A *language* is a pair $L = (C, \tau)$ consisting of a set of *constant symbols* C which are assigned types by the total function $\tau : C \to \mathcal{T}$; i.e., a 'signature' of primitive symbols used to construct expressions of the language.
Expressions E of type α of a language L are strings of symbols defined inductively by the following formation rules.

```
e  ::=  (𝒱 : α)     -- annotation of variable
      | e e        -- application
      | (λ 𝒱 • e)  -- abstraction
      | c          -- constant symbol
      | e = e      -- equality
      | (I 𝒱 • e)  -- definite description
```

Of course not all combinations of symbols are meaningful. A string of symbols `e` is considered an expression only if it can be assigned a type; i.e., we can find some $\gamma \in \mathcal{T}$ so that `e :` γ according to the following rules. That is "_:_" is an inductively defined binary predicate; which is determinstic in its second argument.

```
0. e = (x : α)          ⇒   e : α
1. f : α → β, e : α      ⇒   f e : β
2. e : β                ⇒   (λ x : α • e) : α → β
3. c ∈ C                ⇒   c : τ(c)
4. e : α, d : α          ⇒   (e = d) : 𝔹
5. p : 𝔹                ⇒   (I x : α • p) : α
```

Write $\mathcal{E}\,\alpha$ for the collection of expressions of type α.

- Abstraction defines functions from expressions mentioning variables.
- Definite description builds an expression that denotes the unique value that satisfies a given property.

  $(Ix : \alpha \bullet p)$ denotes the unique value x of type α satisfying p, if it exists and is a canonical "error" value of type α otherwise.
- Annotation constructs a variable by assingin a type to a member of $\mathcal{V}$, an infinite set of symbols.
- Different kinds of expressions are distinuisghed *by type* instead of *by form*.
- Within the body of a qunatification over variable $x : \alpha$, we will write just x rather than $(x : \alpha)$.

The syntax of STT with types in addition to expressions is a bit more complex than the syntax of first-order logic. However, the syntax is also more uniform than the syntax of first-order logic since *expressions serve as both terms and formulae* and *constant include individual constants, function symbols, and predicate symbols as well as constants of many other types.* There are **no propositional or predicate connectives** in STT, but we will see later that these can be easily defined in STT using function application, abstraction, and equality.

The semantics of STT is based on a small collection of well-established ideas

A *standard model* for a language L is a triple $\mathcal{M} = ([\![-]\!]_0, [\![-]\!]_1, err)$ where

1. $[\![-]\!]_0 : \mathcal{T} \to \mathcal{S}et$ assigns each type a non-emppty set.
   - $[\![\mathbb{B}]\!]_0$ is the Booleans.
   - $[\![\alpha \to \beta]\!]_0 = [\![\alpha]\!]_0 \to [\![\beta]\!]_0$ are the total functions between the interpreted sets.
2. $[\![-]\!]_1 : (c : C) \to [\![\tau c]\!]_1$ is a dependent-function, it assigns each constant symbol to a value in the interpreation of the symbol's type.
3. $err : (\alpha : \mathcal{T}) \to [\![\alpha]\!]_0$ is a function associating each type with a 'error' value.

The subscripts of $[\![\text{-}]\!]_i$ will generally be omitted.

- $[\![\iota]\!]$ corresponds to the 'universe of discourse' in a first-order model.
- Note that the function domains of a standard model are determined by the choice of $[\![\iota]\!]$ alone.

1. A *variable assignment* into $\mathcal{M}$ is a variable-to-value assignment $\sigma : \forall \alpha \bullet \mathcal{V} \to [\![\alpha]\!]$.
2. One then defines the *valuation* function $[\![-]\!]_2 = [\![-]\!]_\sigma : \forall \alpha \bullet \mathcal{E}\alpha \to [\![\alpha]\!]$ that takes expressions to values as follows:
   (a) $[\![(x : \alpha)]\!]_2 = \sigma(x)$
   (b) $[\![c]\!]_2 = [\![c]\!]_1$ for $c \in C$
   (c) $[\![fe]\!]_2 = [\![f]\!]_2[\![e]\!]_2$, for function application.

(d) $[\![\lambda x : \alpha \bullet e]\!]_2$, for $e : \beta$, is the function $f : [\![\alpha]\!] \to [\![\beta]\!]$ such that, for each $a \in [\![\alpha]\!]$, we have $f(a) = [\![e]\!]'_2$ where $[\![-]\!]'_2$ uses $\sigma[(x : \alpha) \mapsto a]$ instead of σ.
The semantics of function abstraction is defined using the same trick, due to Tarski, that is used to define the semantics for universal quantification in first-order logic.

(e) $[\![e = d]\!]_2$ is `true` if $[\![e]\!]_2 = [\![d]\!]_2$ and `false` otherwise.

(f) $[\![Ix : \alpha \bullet p]\!]_2$ is the unique $a \in [\![\alpha]\!]_0$ with $[\![p]\!]'_2 = true$ if it exists, and is err_α otherwise.

Some jargon:

1. A value $a \in [\![\alpha]\!]$ is *nameable* if it is the value of some closed expression `e`; i.e., $[\![e]\!]_2 = a$.

2. *A is valid in* $\mathcal{M}$, written $M \vDash A$, if $[\![A]\!]_\sigma = true$ for all variable assignment σ into $\mathcal{M}$.

3. *A is a semantic consequence* of a set Σ of sentences, written $\Sigma \vDash A$, if for every model $\mathcal{M}$, whenever $\mathcal{M} \vDash B$ for all $B \in \Sigma$ then $\mathcal{M} \vDash A$.

4. A *theory* $T = (L, \Gamma)$ is a language L along with a set of sentences Γ, called *the axioms of T.*

   ◇ Semantic conseqence for theories: $T \vDash A$ means $\Gamma \vDash A$.

   ◇ A *standard model for theory T* is a standard model $\mathcal{M}$ for L such that $M \vDash B$ for all $B \in \Gamma$.

The two semantics of first-order logic and STT are based on essentially the same ideas: Domains of discourse, truth values, and functions; models for languages; variable assignments; and valuation functions defined recursively on the syntax of expressions.

STT is a highly expressive logic

Even though STT is formulated as a 'function theory', it is a form of higher-order predicate logic. Indeed its higher-order quantification lets us define the following connectives.

Operation	Definition
`true`	$(\lambda x : \mathbb{B} \bullet x) = (\lambda x : \mathbb{B} \bullet x)$
`false`	$(\lambda x : \mathbb{B} \bullet true) = (\lambda x : \mathbb{B} \bullet x)$
`p ∧ q`	$(\lambda f : \mathbb{B} \to (\mathbb{B} \to \mathbb{B}) \bullet f\ true\ true) = (\lambda f : \mathbb{B} \to (\mathbb{B} \to \mathbb{B}) \bullet f\ p\ q)$
`∀ x : α • p`	$(\lambda x : \alpha \bullet p) = (\lambda x : \alpha \bullet true)$
$\bot_\alpha$	$Ix : \alpha \bullet x \neq x$

Where the remaining connectives are obtained by duality: $\neg p \equiv (p = false)$, $p \vee q \equiv \neg(\neg p \wedge \neg q)$, $(\exists x : \alpha \bullet p) = \neg(\forall x : \alpha \bullet \neg p)$, and $p \neq q \equiv \neg(p = q)$.
Since STT is equipped with full higher-order quantification and definite description, most mathematical notions can be directly and naturally expressed in STT. None of the following examples can be directly expressed in first-order logic.

1. Function composition is expressed by $\lambda f : \iota \to \iota \bullet \lambda g : \iota \to \iota \bullet \lambda x : \iota \bullet f(g\,x)$.

2. Inverse image is expressed by $\lambda f : \iota \to \iota \bullet \lambda s : \iota \to \mathbb{B} \bullet Is' : \iota \to \mathbb{B} \bullet \forall x : \iota \bullet s'\,x = s\,(f\,x)$.

⟪Its simple semantics makes STT suitable for general purpose logic.⟫

Theoretically speaking the inclusion of definite description adds no additional expressivitiy, however from a practicaly stand-point it makes it easy to actually express ideas. See Russell's "On Denoting". The removal of definite description does simplify the semantics since error values would no longer be needed. However, without definite description it would not be possible to directly express the many mathematical concepts that are defined in informal mathematics using the phrasing "the x that satisfies P" such as the notion of the limit of a function.

STT has a simple, elegant, and powerful proof system

◇ A *proof system* **P** consists of a finite set of axiom schemas and rules of inference.

◇ A *proof* of a formula A from theory $T = (L, \Gamma)$ in **P** is a finite sequence of formulas of L where A is the last member of the sequence and every member of the sequence is an instance of one of the axiom schemas of **P**, a member of Γ, or is inferred from previous members by a rule of inference of **P**.

◇ Let $T \vdash_{\mathbf{P}} A$ mean there is a proof of A from T in **P**.

◇ If for every theory T and formula A, if $T \vdash_{\mathbf{P}} A \;\Rightarrow\; T \vDash A$ then say **P** is *sound.*

◇ If for every theory T and formula A, if $T \vdash_{\mathbf{P}} A \;\Leftarrow\; T \vDash A$ then say **P** is *complete.*

References

The Seven Virtues of Simple Type Theory
by William Farmer

A nifty exposition of type theory and how it compares to first-order logic, with an emphasies on *practicality.*

A beautiful read.