Simply typed lambda calculus

The **simply typed lambda calculus** ($\lambda^{\to}$), a form of type theory, is a typed interpretation of the lambda calculus with only one type constructor: $\to$ that builds function types. It is the canonical and simplest example of a typed lambda calculus. The simply typed lambda calculus was originally introduced by Alonzo Church in 1940 as an attempt to avoid paradoxical uses of the untyped lambda calculus, and it exhibits many desirable and interesting properties.

The term *simple type* is also used to refer to extensions of the simply typed lambda calculus such as products, coproducts or natural numbers (System T) or even full recursion (like PCF). In contrast, systems which introduce polymorphic types (like System F) or dependent types (like the Logical Framework) are not considered *simply typed*. The former are still considered *simple* because the Church encodings of such structures can be done using only $\to$ and suitable type variables, while polymorphism and dependency cannot.

1 Syntax

In this article, we use σ and τ to range over types. Informally, the *function type* $\sigma \to \tau$ refers to the type of functions that, given an input of type σ , produce an output of type τ . By convention, $\to$ associates to the right: we read $\sigma \to \tau \to \rho$ as $\sigma \to (\tau \to \rho)$.

To define the types, we begin by fixing a set of *base types*, B . These are sometimes called *atomic types* or *type constants*. With this fixed, the syntax of types is:

$$\tau ::= \tau \to \tau \mid T \quad \text{where} \quad T \in B$$

For example, $B = \{a, b\}$, generates an infinite set of types starting with a, b, $a \to a$, $a \to b, b \to b$, $b \to a$, $a \to (a \to a), \ldots, (b \to a) \to (a \to b), \ldots$

We also fix a set of *term constants* for the base types. For example, we might assume a base type nat, and the term constants could be the natural numbers. In the original presentation, Church used only two base types: o for "the type of propositions" and ι for "the type of individuals". The type o has no term constants, whereas ι has one term constant. Frequently the calculus with only one base type, usually o , is considered.

The syntax of the simply typed lambda calculus is essentially that of the lambda calculus itself. We write $x{:}\tau$ to denote that the variable x is of type τ . The term syntax is then:

$$e ::= x \mid \lambda x{:}\tau.e \mid e\,e \mid c$$

where c is a term constant.

That is, *variable reference*, *abstractions*, *application*, and *constant*. A variable reference x is *bound* if it is inside of an abstraction binding x . A term is *closed* if there are no unbound variables.

Compare this to the syntax of untyped lambda calculus:

$$e ::= x \mid \lambda x.e \mid e\,e$$

We see that in typed lambda calculus every function (*abstraction*) must specify the type of its argument.

2 Typing rules

To define the set of well typed lambda terms of a given type, we will define a typing relation between terms and types. First, we introduce *typing contexts* or *typing environments* $\Gamma, \Delta, \ldots$, which are sets of typing assumptions. A *typing assumption* has the form $x{:}\sigma$, meaning x has type σ .

The *typing relation* $\Gamma \vdash e{:}\sigma$ indicates that e is a term of type σ in context Γ . In this case e is said to be *well-typed* (having type σ). Instances of the typing relation are called *typing judgements*. The validity of a typing judgement is shown by providing a *typing derivation*, constructed using typing rules (wherein the premises above the line allow us to derive the conclusion below the line). Simply-typed lambda calculus uses these rules:

In words,

1. If x has type σ in the context, we know that x has type σ .
2. Term constants have the appropriate base types.
3. If, in a certain context with x having type σ , e has type τ , then, in the same context without x , $\lambda x{:}\sigma.\, e$ has type $\sigma \to \tau$.
4. If, in a certain context, e_1 has type $\sigma \to \tau$, and e_2 has type σ , then $e_1\ e_2$ has type τ .

Examples of closed terms, *i.e.* terms typable in the empty context, are:

- For every type τ , a term $\lambda x{:}\tau.x{:}\tau \to \tau$ (identity function/I-combinator),
- For types σ, τ , a term $\lambda x{:}\sigma.\lambda y{:}\tau.x{:}\sigma \to \tau \to \sigma$ (the K-combinator), and
- For types τ, τ', τ'' , a term $\lambda x{:}\tau \to \tau' \to \tau''.\lambda y{:}\tau \to \tau'.\lambda z{:}\tau.xz(yz) : (\tau \to \tau' \to \tau'') \to (\tau \to \tau') \to \tau \to \tau''$ (the S-combinator).

These are the typed lambda calculus representations of the basic combinators of combinatory logic.

Each type τ is assigned an order, a number $o(\tau)$. For base types, $o(T) = 0$; for function types, $o(\sigma \to \tau) = \max(o(\sigma) + 1, o(\tau))$. That is, the order of a type measures the depth of the most left-nested arrow. Hence:

$$o(\iota \to \iota \to \iota) = 1$$

$$o((\iota \to \iota) \to \iota) = 2$$

3 Semantics

3.1 Intrinsic vs. extrinsic interpretations

Broadly speaking, there are two different ways of assigning meaning to the simply typed lambda calculus, as to typed languages more generally, sometimes called *intrinsic* vs. *extrinsic*, or *Church-style* vs. *Curry-style*.[1] An intrinsic/Church-style semantics only assigns meaning to well-typed terms, or more precisely, assigns meaning directly to typing derivations. This has the effect that terms differing only by type annotations can nonetheless be assigned different meanings. For example, the identity term $\lambda x{:}\texttt{int}.\ x$ on integers and the identity term $\lambda x{:}\texttt{bool}.\ x$ on booleans may mean different things. (The classic intended interpretations are the identity function on integers and the identity function on boolean values.) In contrast, an extrinsic/Curry-style semantics assigns meaning to terms regardless of typing, as they would be interpreted in an untyped language. In this view, $\lambda x{:}\texttt{int}.\ x$ and $\lambda x{:}\texttt{bool}.\ x$ mean the same thing (*i.e.*, the same thing as $\lambda x.\ x$).

The distinction between intrinsic and extrinsic semantics is sometimes associated with the presence or absence of annotations on lambda abstractions, but strictly speaking this usage is imprecise. It is possible to define a Curry-style semantics on annotated terms simply by ignoring the types (*i.e.*, through type erasure), as it is possible to give a Church-style semantics on unannotated terms when the types can be deduced from context (*i.e.*, through type inference). The essential difference between intrinsic and extrinsic approaches is just whether the typing rules are viewed as defining the language, or as a formalism for verifying properties of a more primitive underlying language. Most of the different semantic interpretations discussed below can be seen through either a Church or Curry perspective.

3.2 Equational theory

The simply typed lambda calculus has the same equational theory of βη-equivalence as untyped lambda calculus, but subject to type restrictions. The equation for beta reduction

$$(\lambda x{:}\sigma.\ t)\, u =_\beta t[x := u]$$

holds in context Γ whenever $\Gamma, x{:}\sigma \vdash t{:}\tau$ and $\Gamma \vdash u{:}\sigma$, while the equation for eta reduction

$$\lambda x{:}\sigma.\ t\, x =_\eta t$$

holds whenever $\Gamma \vdash t\colon \sigma \to \tau$ and x does not appear free in t .

3.3 Operational semantics

Likewise, the operational semantics of simply typed lambda calculus can be fixed as for the untyped lambda calculus, using call by name, call by value, or other evaluation strategies. As for any typed language, type safety is a fundamental property of all of these evaluation strategies. Additionally, the strong normalization property described below implies that any evaluation strategy will terminate on all simply typed terms.

3.4 Categorical semantics

The simply typed lambda calculus (with $\beta\eta$ -equivalence) is the internal language of Cartesian closed categories (CCCs), as was first observed by Lambek. Given any specific CCC, the basic types of the corresponding lambda calculus are just the objects, and the terms are the morphisms. Conversely, every simply typed lambda calculus gives a CCC whose objects are the types, and morphisms are equivalence classes of terms.

To make the correspondence clear, a type constructor for the Cartesian product is typically added to the above. To preserve the categoricity of the Cartesian product, one adds type rules for *pairing*, *projection*, and a *unit term*. Given two terms $s{:}\sigma$ and $t{:}\tau$, the term (s, t) has type $\sigma \times \tau$. Likewise, if one has a term $u{:}\tau_1 \times \tau_2$, then there are terms $\pi_1(u){:}\tau_1$ and $\pi_2(u){:}\tau_2$ where the π_i correspond to the projections of the Cartesian product. The *unit term*, of type 1, is written as $()$ and vocalized as 'nil', is the final object. The equational theory is extended likewise, so that one has

$$\pi_1(s{:}\sigma, t{:}\tau) = s{:}\sigma$$

$$\pi_2(s{:}\sigma, t{:}\tau) = t{:}\tau$$

$$(\pi_1(u{:}\sigma \times \tau), \pi_2(u{:}\sigma \times \tau)) = u{:}\sigma \times \tau$$

$$t{:}1 = ()$$

This last is read as "*if t has type 1, then it reduces to nil*".

The above can then be turned into a category by taking the types as the objects. The morphisms $\sigma \to \tau$ are equivalence classes of pairs $(x{:}\sigma, t{:}\tau)$ where *x* is a variable (of type σ) and *t* is a term (of type τ), having no free variables in it, except for (optionally) *x*. Closure is obtained from currying and application, as usual.

More precisely, there exist functors between the category of Cartesian closed categories, and the category of simply-typed lambda theories.

It is common to extend this case to closed symmetric monoidal categories by using a linear type system. The reason for this is that the CCC is a special case of the closed symmetric monoidal category, which is typically taken to be the category of sets. This is fine for laying the foundations of set theory, but the more general topos seems to provide a superior foundation.

3.5 Proof-theoretic semantics

The simply typed lambda calculus is closely related to the implicational fragment of propositional intuitionistic logic, i.e., minimal logic, via the Curry–Howard isomorphism: terms correspond precisely to proofs in natural deduction, and inhabited types are exactly the tautologies of minimal logic.

4 Alternative syntaxes

The presentation given above is not the only way of defining the syntax of the simply typed lambda calculus. One alternative is to remove type annotations entirely (so that the syntax is identical to the untyped lambda calculus), while ensuring that terms are well-typed via Hindley-Milner type inference. The inference algorithm is terminating, sound, and complete: whenever a term is typable, the algorithm computes its type. More precisely, it computes the term's principal type, since often an unannotated term (such as $\lambda x.\ x$) may have more than one type ($\mathtt{int} \to \mathtt{int}$, $\mathtt{bool} \to \mathtt{bool}$, etc., which are all instances of the principal type $\alpha \to \alpha$).

Another alternative presentation of simply typed lambda calculus is based on **bidirectional type checking**, which requires more type annotations than Hindley-Milner inference but is easier to describe. The type system is divided into two judgments, representing both *checking* and *synthesis*, written $\Gamma \vdash e \Leftarrow \tau$ and $\Gamma \vdash e \Rightarrow \tau$ respectively. Operationally, the three components Γ , e , and τ are all *inputs* to the checking judgment $\Gamma \vdash e \Leftarrow \tau$, whereas the synthesis judgment $\Gamma \vdash e \Rightarrow \tau$ only takes Γ and e as inputs, producing the type τ as output. These judgments are derived via the following rules:

Observe that rules [1]–[4] are nearly identical to rules (1)–(4) above, except for the careful choice of checking or synthesis judgments. These choices can be explained like so:

1. If $x{:}\sigma$ is in the context, we can synthesize type σ for x .
2. The types of term constants are fixed and can be synthesized.
3. To check that $\lambda x.\ e$ has type $\sigma \to \tau$ in some context, we extend the context with $x{:}\sigma$ and check that e has type τ .
4. If e_1 synthesizes type $\sigma \to \tau$ (in some context), and e_2 checks against type σ (in the same context), then $e_1\ e_2$ synthesizes type τ .

Observe that the rules for synthesis are read top-to-bottom, whereas the rules for checking are read bottom-to-top. Note in particular that we do **not** need any annotation on the lambda abstraction in rule [3], because the type of the bound variable can be deduced from the type at which we check the function. Finally, we explain rules [5] and [6] as follows:

1. To check that e has type τ , it suffices to synthesize type τ .
2. If e checks against type τ , then the explicitly annotated term $(e{:}\tau)$ synthesizes τ .

Because of these last two rules coercing between synthesis and checking, it is easy to see that any well-typed but unannotated term can be checked in the bidirectional system, so long as we insert "enough" type annotations. And in fact, annotations are needed only at β-redexes.

5 General observations

Given the standard semantics, the simply typed lambda calculus is strongly normalizing: that is, well-typed terms always reduce to a value, i.e., a λ abstraction. This is because recursion is not allowed by the typing rules: it is impossible to find types for fixed-point combinators and the looping term $\Omega = (\lambda x.\ x\ x)(\lambda x.\ x\ x)$. Recursion can be added to the language by either having a special operator $\texttt{fix}_\alpha$ of type $(\alpha \to \alpha) \to \alpha$ or adding general recursive types, though both eliminate strong normalization.

Since it is strongly normalising, it is decidable whether or not a simply typed lambda calculus program halts: in fact, it *always* halts. We can therefore conclude that the language is *not* Turing complete.

6 Important results

- Tait showed in 1967 that β -reduction is strongly normalizing. As a corollary $\beta\eta$ -equivalence is decidable. Statman showed in 1977 that the normalisation problem is not elementary recursive, a proof which was later simplified by Mairson (1992). A purely semantic normalisation proof (see normalisation by evaluation) was given by Berger and Schwichtenberg in 1991.

- The unification problem for $\beta\eta$ -equivalence is undecidable. Huet showed in 1973 that 3rd order unification is undecidable and this was improved upon by Baxter in 1978 then by Goldfarb in 1981 by showing that 2nd order unification is already undecidable. Whether higher order matching (unification where only one term contains existential variables) is decidable is still open. [2006: Colin Stirling, Edinburgh, has published a proof-sketch in which he claims that the problem is decidable; however, the complete version of the proof is still unpublished]

- We can encode natural numbers by terms of the type $(o \to o) \to (o \to o)$ (Church numerals). Schwichtenberg showed in 1976 that in $\lambda^\to$ exactly the extended polynomials are representable as functions over Church numerals; these are roughly the polynomials closed up under a conditional operator.

- A *full model* of $\lambda^\to$ is given by interpreting base types as sets and function types by the set-theoretic function space. Friedman showed in 1975 that this interpretation is complete for $\beta\eta$ -equivalence, if the base types are interpreted by infinite sets. Statman showed in 1983 that $\beta\eta$ -equivalence is the maximal equivalence which is *typically ambiguous*, i.e. closed under type substitutions (*Statman's Typical Ambiguity Theorem*). A corollary of this is that the *finite model property* holds, i.e. finite sets are sufficient to distinguish terms which are not identified by $\beta\eta$ -equivalence.

- Plotkin introduced logical relations in 1973 to characterize the elements of a model which are definable by lambda terms. In 1993 Jung and Tiuryn showed that a general form of logical relation (Kripke logical relations with varying arity) exactly characterizes lambda definability. Plotkin and Statman conjectured that it is decidable whether a given element of a model generated from finite sets is definable by a lambda term (*Plotkin-Statman-conjecture*). The conjecture was shown to be false by Loader in 1993.

7 Notes

[1] Reynolds, John (1998). *Theories of Programming Languages*. Cambridge, England: Cambridge University Press.

8 See also

- Article Church's Type Theory in the Stanford Encyclopedia of Philosophy.
- Hindley-Milner type inference algorithm

9 References

- A. Church: A Formulation of the Simple Theory of Types, JSL 5, 1940
- W.W.Tait: Intensional Interpretations of Functionals of Finite Type I, JSL 32(2), 1967
- G.D. Plotkin: Lambda-definability and logical relations, Technical report, 1973
- G.P. Huet: The Undecidability of Unification in Third Order Logic Information and Control 22(3): 257-267 (1973)
- H. Friedman: Equality between functionals. LogicColl. '73, pages 22-37, LNM 453, 1975.
- H. Schwichtenberg: Functions definable in the simply-typed lambda calculus, Arch. Math Logik 17 (1976) 113-114.
- R. Statman: The Typed lambda-Calculus Is not Elementary Recursive FOCS 1977: 90-94
- W. D. Goldfarb: The undecidability of the 2nd order unification problem, TCS (1981), no. 13, 225- 230.
- R. Statman. λ -definable functionals and $\beta\eta$ conversion. Arch. Math. Logik, 23:21–26, 1983.
- J. Lambek: Cartesian Closed Categories and Typed Lambda-calculi. Combinators and Functional Programming Languages 1985: 136-175
- U. Berger, H. Schwichtenberg: An Inverse of the Evaluation Functional for Typed lambda-calculus LICS 1991: 203-211
- H. Mairson: A simple proof of a theorem of Statman, TCS 103(2):387-394, 1992.

- Jung, A.,Tiuryn, J.:A New Characterization of Lambda Definability, TLCA 1993
- R. Loader: The Undecidability of λ-definability, appeared in the Church Festschrift, 2001
- H. Barendregt, Lambda Calculi with Types, Handbook of Logic in Computer Science, Volume II, Oxford University Press, 1993. ISBN 0-19-853761-1.
- L. Baxter: The undecidability of the third order dyadic unification problem, Information and Control 38(2), 170-178 (1978)

10 External links

- Loader, Ralph (February 1998). "Notes on Simply Typed Lambda Calculus".

11 Text and image sources, contributors, and licenses

11.1 Text

11.2 Images

11.3 Content license