

Does the Lambda Calculus Have Any Applications?

Mikkel Nygaard

Juniorklubben, BRICS
March 7th, 2003

Abstract

Even within theoretical computer science, people tend to disagree on whether or not the lambda calculus and associated theories are of any use. We'll take a brief look at the lambda calculus and caricature the different viewpoints with special emphasis on the "concurrency theory dogma" that lambda calculus is useless because it cannot describe interaction.

1 The lambda calculus

Here's the basic, untyped calculus:

$$t, u ::= x \mid \lambda x.t \mid t u$$

Clearly, without any applications, this calculus is very boring...

To assess its merits as a programming language, we'll follow most modern languages in adding a *type system* which allows many programming errors to be caught by static type-checking, rather than leaving errors to pop up unexpectedly at run-time. Types come at the price of a larger language as well as extra notation:

First, the types:

$$\mathbb{T}, \mathbb{U} ::= \mathbb{T} \rightarrow \mathbb{U} \mid \mathbb{1} \mid \mathbb{T} \times \mathbb{U} \mid \mathbb{T} + \mathbb{U} \mid T \mid \mu T. \mathbb{T} .$$

Intuitively, types are sets. The *arrow type* $\mathbb{T} \rightarrow \mathbb{U}$ is the function space $\mathbb{U}^{\mathbb{T}}$. The *unit type* $\mathbb{1}$ is a singleton. The *product type* $\mathbb{T} \times \mathbb{U}$ is the cartesian product of \mathbb{T} and \mathbb{U} . The *union type* $\mathbb{T} + \mathbb{U}$ is the disjoint union of \mathbb{T} and

\mathbb{U} . T is a type variable, used in the definition of *recursive types*: $\mu T.\mathbb{T}$ is the smallest set T such that $T = \mathbb{T}$; here, \mathbb{T} may contain the variable T .¹

Second, the raw syntax of terms:

$$t, u ::= x \mid \text{rec } x.t \mid \lambda x.t \mid t u \mid \bullet \mid (t, u) \mid \text{fst } t \mid \text{snd } t \mid \\ \text{inl } t \mid \text{inr } t \mid [u > \text{inl } x \Rightarrow t_1, \text{inr } x \Rightarrow t_2] \mid \text{abs } t \mid \text{rep } t$$

The syntax is subject to typing constraints. A syntactic judgement

$$x_1 : \mathbb{T}_1, \dots, x_k : \mathbb{T}_k \vdash t : \mathbb{U}$$

means that the term t has type \mathbb{U} provided the distinct variables x_1, \dots, x_k have types $\mathbb{T}_1, \dots, \mathbb{T}_k$, respectively. We'll let Γ range over type environments $x_1 : \mathbb{T}_1, \dots, x_k : \mathbb{T}_k$ and treat it as a finite function from variables to types.

Here are some of the typing rules:

$$\frac{\Gamma(x) = \mathbb{T}}{\Gamma \vdash x : \mathbb{T}} \quad \frac{\Gamma, x : \mathbb{T} \vdash t : \mathbb{U}}{\Gamma \vdash \lambda x.t : \mathbb{T} \rightarrow \mathbb{U}} \quad \frac{\Gamma \vdash t : \mathbb{T} \rightarrow \mathbb{U} \quad \Gamma \vdash u : \mathbb{T}}{\Gamma \vdash t u : \mathbb{U}}$$

$$\frac{}{\Gamma \vdash \bullet : \mathbb{1}} \quad \frac{\Gamma \vdash t : \mathbb{T} \quad \Gamma \vdash u : \mathbb{U}}{\Gamma \vdash (t, u) : \mathbb{T} \times \mathbb{U}} \quad \frac{\Gamma \vdash t : \mathbb{T} \times \mathbb{U}}{\Gamma \vdash \text{fst } t : \mathbb{T}}$$

$$\frac{\Gamma \vdash t : \mathbb{T}}{\Gamma \vdash \text{inl } t : \mathbb{T} + \mathbb{U}} \quad \frac{\Gamma \vdash u : \mathbb{T} + \mathbb{U} \quad \Gamma, x : \mathbb{T} \vdash t_1 : \mathbb{V} \quad \Gamma, x : \mathbb{U} \vdash t_2 : \mathbb{V}}{\Gamma \vdash [u > \text{inl } x \Rightarrow t_1, \text{inr } x \Rightarrow t_2] : \mathbb{V}}$$

$$\frac{\Gamma \vdash t : \mathbb{T}[\mu T.\mathbb{T}/T]}{\Gamma \vdash \text{abs } t : \mu T.\mathbb{T}} \quad \frac{\Gamma \vdash t : \mu T.\mathbb{T}}{\Gamma \vdash \text{rep } t : \mathbb{T}[\mu T.\mathbb{T}/T]}$$

Now let's see how to do programming in the lambda calculus. Assuming that the reader knows ML, the following example should indicate how recursive types may be used to define any inductive datatype, like numbers, lists or trees.

ML:

datatype N of

Zero

| Succ of N

fun pred(Zero) = Zero

| pred(Succ(t)) = t

: N -> N

Lambda calculus:

$\mathbb{N} \equiv \mu N.\mathbb{1} + N$

$\text{Zero} \equiv \text{abs}(\text{inl } \bullet)$

$\text{Succ}(t) \equiv \text{abs}(\text{inr } t)$

$\lambda t. [\text{rep } t > \text{inl } x \Rightarrow \text{Zero},$

$\text{inr } x \Rightarrow x]$

: $\mathbb{N} \rightarrow \mathbb{N}$

¹Note that we are cheating: there may be no smallest solution to a recursive type definition using set theory. However, using Dana Scott's *domain theory*, such equations do indeed have least solutions, unique up to isomorphism. For most purposes, domains can be regarded as sets, and we'll do so throughout these notes.

This indicates how to give meaning to the typed lambda calculus using ML. But in fact, functional programming languages like ML or Haskell can be defined using the lambda calculus, and so in terms of more primitive notions. Indeed, we can give meaning to the language above from first principle using *operational semantics*. Here, the meaning of any closed, typed term is established by rules saying how to *evaluate* the term to a syntactic *value*:

$$v ::= \lambda x.t \mid (t, u) \mid \text{inl } t \mid \text{inr } t \mid \text{abs } v$$

Here are the rules (we've left out the obvious symmetric rules for projection and union-case):

$$\frac{}{v \rightarrow v} \quad \frac{t[\text{rec } x.t/x] \rightarrow v}{\text{rec } x.t \rightarrow v}$$

$$\frac{t \rightarrow \lambda x.t' \quad t'[u/x] \rightarrow v}{t u \rightarrow v} \quad \frac{t \rightarrow (t_1, t_2) \quad t_1 \rightarrow v}{\text{fst } t \rightarrow v}$$

$$\frac{u \rightarrow \text{inl } u' \quad t_1[u'/x] \rightarrow v}{[u > \text{inl } x \Rightarrow t_1, \text{inr } x \Rightarrow t_2] \rightarrow v} \quad \frac{t \rightarrow \text{abs } v}{\text{rep } t \rightarrow v}$$

An operational semantics specifies how any implementation of the lambda calculus should work in so far as the final outcome of any computation is concerned. This is useful if anyone should want to write a compiler for ML or Haskell, so the lambda calculus seems promising.

But suppose that we want to optimise some *program* $C(t)$, so a closed term of type \mathbb{N} , by replacing the subterm t by a more efficient subterm u . Is this optimisation correct? We cannot simply try to evaluate t and u , because they may contain free variables and so evaluation may not be defined. However, the optimisation is correct if t and u are *contextually equivalent*, meaning that no amount of programming can tell them apart, formally:

Definition 1.1 If C is a term with a hole into which terms $\Gamma \vdash t : \mathbb{T}$ may be put to form a program $C(t)$, we call C a (Γ, \mathbb{T}) -context. Suppose $\Gamma \vdash t : \mathbb{T}$ and $\Gamma \vdash u : \mathbb{T}$. We write $t \preceq u$ iff for all (Γ, \mathbb{T}) -contexts $\vdash C(-) : \mathbb{N}$ we have

$$C(t) \rightarrow \text{Zero} \implies C(u) \rightarrow \text{Zero}$$

The terms t and u are said to be contextually equivalent if $t \preceq u$ and $u \preceq t$. \square

One should note that this definition is robust in that changing \mathbb{N} or Zero to other reasonable types or terms makes no difference. Unfortunately, it can be quite hard to prove contextual equivalence because of the quantification

over all program contexts. So people have considered more abstract ways of giving semantics to the lambda calculus.

Using *denotational semantics*, we assign to each type and term a denotation in the form of some mathematical object, like a set or a function. Indeed, a type \mathbb{T} may denote a set $\llbracket \mathbb{T} \rrbracket$ of abstract values, while terms $x_1 : \mathbb{T}_1, \dots, x_k : \mathbb{T}_k \vdash t : \mathbb{U}$ may denote partial functions²

$$\llbracket x_1 : \mathbb{T}_1, \dots, x_k : \mathbb{T}_k \vdash t : \mathbb{U} \rrbracket : \mathbb{T}_1 \times \dots \times \mathbb{T}_k \rightarrow \mathbb{U} .$$

We need to use partial functions, because programs may loop, giving no value at all. Note that a closed term of type \mathbb{T} denotes a partial function $\mathbb{1} \rightarrow \mathbb{T}$ and so is either undefined or corresponds to an element of \mathbb{T} , as wanted.

There are many ways to define a denotational semantics, $\llbracket - \rrbracket$, but the important point is that whenever we define the meaning of a term, we do so by induction on the syntactic structure of the term, e.g.

$$\llbracket fst\ t \rrbracket = \pi_1 \circ \llbracket t \rrbracket .$$

This implies that the denotational semantics is *compositional*, so that the denotation $\llbracket C(t) \rrbracket$ of the program $C(t)$ from above is given as a function of $\llbracket t \rrbracket$. The cool thing about this is that it allows easy proofs of contextual equivalence. Of course, this requires some kind of correspondence between the operational and denotational semantics:

- Soundness: if $t \rightarrow v$, then $\llbracket t \rrbracket = \llbracket v \rrbracket$.
- Completeness: if $\llbracket t \rrbracket$ defined, then $\exists v. t \rightarrow v$.

Given this, it is easy to show that

$$\llbracket t \rrbracket \leq \llbracket u \rrbracket \implies t \preceq u$$

—so that $\llbracket t \rrbracket = \llbracket u \rrbracket$ implies contextual equivalence. Showing $\llbracket t \rrbracket = \llbracket u \rrbracket$ is often much easier than proving contextual equivalence directly, because there is no quantification over contexts.

Depending on the exact nature of the denotational semantics, one may also be able to show the converse of the above implication. This is called *full abstraction* and is a kind of holy grail of semantics, because it is normally very difficult to obtain.

All the above is very fine, but it does not by itself show that the lambda calculus is interesting: it all depends upon functional programming being interesting. And here, the views differ.

²Again, sets and functions are not sufficient for interpreting recursion, but can be replaced by domains.

2 Applications?

Consider a functional programmer. He's probably from academia, and he'll sound a bit like Olivier Danvy when saying

- Programs transform input to output, so they are functions.
- Higher-order functions are cool! Loads of examples...
- Witness the elegance of ML, Haskell.

Consider now an imperative programmer. He's probably from industry, and he'll not be impressed:

- Programs have side-effects.
- Higher-order functions are never really needed.
- Witness the dominance of the C family.

So practical programming does not seem to make a particularly convincing case for the utility of the lambda calculus. What about applications to the *foundations* of programming? Perhaps some of the nice results above can be transferred to imperative programming languages.

However, in imperative programming, expressions tend to be simple, parameter passing is first-order, and side-effects are everywhere. The lambda calculus simply gets in the way, and it is much easier to give an operational semantics directly. And axiomatic semantics (like Hoare triples, invariants) can replace the denotational semantics, if need be.

Our functional programmer would respond that this is just because imperative programming is inherently messy, and that functional programming is at least a good intermediate between foundations and practice. But is it? Functional programming may be able to handle "internal" side-effects like assignments and exceptions, but a program that interacts with the user through keyboard or mouse cannot be described adequately as a function from input to output. The pattern of interaction needs to be taken into account.

In its purest form, interaction is the subject of *concurrency theory*, and here, it has been a dogma for some 30 years that

The lambda calculus cannot be used to describe interaction.

3 Interaction

In 1972, Robin Milner tried to apply standard semantic techniques like those we have seen to a concurrent programming language—and failed. In particular, Milner found that viewing a program as a function over memory states, and thus identifying the programs

$$(1) X:=1 \quad \text{and} \quad (2) X:=0; X:=X+1 ,$$

only makes sense if the program has full control over memory. Interference from another program, say $X:=1$, will not change the behavior of program (1), but will make program (2) behave nondeterministically, changing X to either 1 or 2.

Here, the problem is interaction with storage—the same kinds of problems arise with machine-machine interaction, and indeed, human-machine interaction.

This led Milner to search for something more fundamental—a new calculus with *interaction* or *communication* as the central idea. He and Tony Hoare independently came up with the same primitive notion of *indivisible interaction*. We'll look at Milner's calculus, CCS. Like the basic lambda calculus, it is untyped. Here's the syntax:

$$t, u ::= x \mid \text{rec } x.t \mid \sum_{i \in I} t_i \mid \alpha.t \mid t|u \mid t[f] \mid t \setminus L$$

The operational semantics is given as follows:

$$\begin{array}{c} \frac{t[\text{rec } x.t/x] \xrightarrow{\alpha} t'}{\text{rec } x.t \xrightarrow{\alpha} t'} \quad \frac{t_j \xrightarrow{\alpha} t'}{\sum_{i \in I} t_i \xrightarrow{\alpha} t'} \quad j \in I \quad \frac{}{\alpha.t \xrightarrow{\alpha} t} \\ \frac{t \xrightarrow{\alpha} t'}{t|u \xrightarrow{\alpha} t|u} \quad \frac{t \xrightarrow{\alpha} t' \quad u \xrightarrow{\bar{\alpha}} u'}{t|u \xrightarrow{\tau} t'|u'} \quad \frac{u \xrightarrow{\alpha} u'}{t|u \xrightarrow{\alpha} t|u'} \\ \frac{t \xrightarrow{\alpha} t'}{t[f] \xrightarrow{f\alpha} t'[f]} \quad \frac{t \xrightarrow{\alpha} t'}{t \setminus L \xrightarrow{\alpha} t' \setminus L} \quad \alpha \notin L \end{array}$$

Different typed versions of CCS has been proposed over the years. Here, we'll consider a proposal by Glynn Winskel and myself from 2002: HOPLA—a higher order process language.

Types and terms are given as follows

$$\begin{array}{l} \mathbb{T}, \mathbb{U} ::= \mathbb{T} \rightarrow \mathbb{U} \mid \sum_{\alpha \in A} \mathbb{T}_{\alpha} \mid \cdot \mathbb{T} \mid T \mid \mu T. \mathbb{T} \\ t, u ::= x \mid \text{rec } x.t \mid \sum_{i \in I} t_i \mid \lambda x.t \mid t u \mid \alpha t \mid \pi_{\alpha} t \mid .t \mid [u > .x \Rightarrow t] \end{array}$$

The type of a process describes the possible computation paths the process can perform. Computation paths may consist simply of sequences of actions but they may also represent the input-output behaviour of a process. A typing judgement

$$x_1 : \mathbb{T}_1, \dots, x_k : \mathbb{T}_k \vdash t : \mathbb{U}$$

means that a process t yields computation paths in \mathbb{U} once processes with computation paths in $\mathbb{T}_1, \dots, \mathbb{T}_k$ are assigned to the variables x_1, \dots, x_k respectively. Incidentally, because of nondeterministic sum, products and unions coincide at the type level to form a “sum-type” $\Sigma_{\alpha \in A} \mathbb{T}_\alpha$ with associated injections $\alpha(-) : \mathbb{T}_\alpha \rightarrow \Sigma_{\alpha \in A} \mathbb{T}_\alpha$ and projections $\pi_\alpha(-) : \Sigma_{\alpha \in A} \mathbb{T}_\alpha \rightarrow \mathbb{T}_\alpha$. Together with the anonymous prefix type $\cdot\mathbb{T}$, we can give typing to CCS-like prefixing as follows:

$$\mathbb{P} \equiv \mu P. \Sigma_\alpha \alpha. P .$$

So all CCS processes have type \mathbb{P} , in accordance with CCS being an untyped calculus. We may then translate CCS into HOPLA as follows:

$$\begin{aligned} \llbracket x \rrbracket &\equiv x & \llbracket P|Q \rrbracket &\equiv \text{Par } \llbracket P \rrbracket \llbracket Q \rrbracket \\ \llbracket \text{rec } x. P \rrbracket &\equiv \text{rec } x. \llbracket P \rrbracket & \llbracket P \setminus S \rrbracket &\equiv \text{Res}_S \llbracket P \rrbracket \\ \llbracket \Sigma_{i \in I} P_i \rrbracket &\equiv \Sigma_{i \in I} \llbracket P_i \rrbracket & \llbracket P[f] \rrbracket &\equiv \text{Rel}_f \llbracket P \rrbracket \\ \llbracket \alpha. P \rrbracket &\equiv \alpha. \llbracket P \rrbracket \end{aligned}$$

$$\text{Par} \equiv \text{rec } p. \lambda x. \lambda y.$$

$$\begin{aligned} &\Sigma_\alpha [\pi_\alpha x > .x' \Rightarrow \alpha.(p \ x' \ y)] + \\ &\Sigma_\alpha [\pi_\alpha y > .y' \Rightarrow \alpha.(p \ x \ y')] + \\ &\Sigma_{\alpha \neq \tau} [\pi_\alpha x > .x' \Rightarrow [\pi_{\bar{\alpha}} y > .y' \Rightarrow \tau.(p \ x' \ y)]] \end{aligned}$$

—with restriction and relabelling handled in a similar way.

HOPLA has an operational semantics with *actions* given by

$$a ::= u \mapsto a \mid \alpha a \mid .$$

The rules are:

$$\begin{array}{c} \frac{t[\text{rec } x.t/x] \xrightarrow{a} t'}{\text{rec } x.t \xrightarrow{a} t'} \quad \frac{t_j \xrightarrow{a} t'}{\Sigma_{i \in I} t_i \xrightarrow{a} t'} \quad j \in I \\ \\ \frac{t[u/x] \xrightarrow{a} t'}{\lambda x.t \xrightarrow{u \mapsto a} t'} \quad \frac{t \xrightarrow{w \mapsto a} t'}{t \ u \xrightarrow{a} t'} \\ \\ \frac{t \xrightarrow{a} t'}{\alpha t \xrightarrow{\alpha a} t'} \quad \frac{t \xrightarrow{\alpha a} t'}{\pi_\alpha t \xrightarrow{a} t'} \\ \\ \frac{}{.t \xrightarrow{\cdot} t} \quad \frac{u \xrightarrow{\cdot} u' \quad t[u'/x] \xrightarrow{a} t'}{[u > .x \Rightarrow t] \xrightarrow{a} t'} \end{array}$$

The operational semantics induced for CCS by the translation above coincides with Milner’s rules.

In fact, HOPLA also has a denotational semantics:

- Types denote sets of “paths”.
- Terms denote functions,

$$\llbracket x_1 : \mathbb{T}_1, \dots, x_k : \mathbb{T}_k \vdash t : \mathbb{U} \rrbracket : \mathcal{P}\mathbb{T}_1 \times \dots \times \mathcal{P}\mathbb{T}_k \rightarrow \mathcal{P}\mathbb{U}$$

- Soundness: if $t \xrightarrow{a} t'$, then $\llbracket t' \rrbracket \subseteq a^* \llbracket t \rrbracket$.
- Completeness: if $a^* \llbracket t \rrbracket \neq \emptyset$, then $\exists t'. t \xrightarrow{a} t'$.
- Full abstraction: $t \preceq u \iff \llbracket t \rrbracket \subseteq \llbracket u \rrbracket$.

But wait a minute... These are standard semantic results about—a lambda calculus! What happened? The lambda calculus itself does not describe interaction, but it is still useful for defining and applying operators on processes, starting from very basic primitives. In fact in my view,

- The lambda calculus remains *the* fundamental language for defining and applying operators.
- Given base types and primitive operations on them, the lambda calculus provides all the expressive power you need.
- Base types can be missing (ML), commands (Forsythe), processes (HOPLA), objects, components,...
- Programming is the activity of defining and applying operators to computational objects.
- Programming *is* the use of the lambda calculus.