Typed Lambda Calculus and the Semantics of Programs

Aaron Stump

Dept. of Computer Science
The University of Iowa
Iowa City, Iowa, USA

Programming Languages Forever

- Hundreds of programming languages invented.

  Haskell, Java, C, C++, OCaml, LISP, Scheme, Basic, Pascal, ...

- More born all the time (and few die...).
- Why?
- Aren't Turing machines enough?

Language Variety

- Languages are centered around different organizing ideas.
  - Object-oriented programming (Java, C++, Ruby, Scala)

    ```
    List l = new LinkedList();
    l.add(4); l.add(3); l.add(2); l.add(1);
    Collections.reverse(l);
    ```

  - Functional programming (LISP, OCaml, Haskell)

    ```
    List.rev [ 4; 3 ; 2 ; 1 ]
    ```

  - More: imperative programming, logic programming.
- Tailored to different domains (web, database, scientific, etc.).
- Provide different mechanisms for *abstraction*...
- ...often through *typing*.

Typing Example: List.rev in OCaml

`List.rev` : 'a list -> 'a list

- This type says that `List.rev` is a function where for any type 'a:
  - the argument should be a list of 'a things.
  - the result (if any) will be a list of 'a things.
- Can operate on list with any element type:
  - `List.rev [3;2;1]`
  - `List.rev ["hi";"bye";"why"]`
  - etc.
- `List.rev`'s type abstracts all the rest of its behavior.

How Types Abstract

- Many different functions have type 'a list -> 'a list:
  - `List.rev`.
  - `List.tl` which maps `[4;3;2;1]` to `[3;2;1]`.
  - `id`, the identity function.
  - `loop`, the function which runs forever.
  - `myfunc` with `myfunc [a;b;...] = [b;a;...]`
  - etc.
- Type denotes set of all functions with the specified behavior.
- Informally:

$$[\![T]\!] = \{f \mid f \text{ has behavior described by } T\}$$

Lambda Calculus

- Proposed by Alonzo Church as basis for logic.
  ["The Calculi of Lambda Conversion," 1941]
- Widely adopted as a language for describing functions:
  - in Linguistics: for semantics.
  - in Logic: for higher-order logic, proof theory.
  - in Computer Science: for functional programming languages.
- Comes in untyped and typed varieties.
- Appeal is its simplicity and power.

Untyped Lambda Calculus

Syntax:

$$\textit{terms } t \quad ::= \quad x \mid t\ t' \mid \lambda x.\, t$$

Reduction Semantics:

$$\frac{}{(\lambda x.\, t)\ t' \ \rightsquigarrow\ [t'/x]t} \qquad \frac{t \ \rightsquigarrow\ t'}{\lambda x.\, t \ \rightsquigarrow\ \lambda x.\, t'}$$

$$\frac{t_1 \ \rightsquigarrow\ t_1'}{(t_1\ \ t_2) \ \rightsquigarrow\ (t_1'\ \ t_2)} \qquad \frac{t_2 \ \rightsquigarrow\ t_2'}{(t_1\ \ t_2) \ \rightsquigarrow\ (t_1\ \ t_2')}$$

$[t'/x]t$ denotes result of substituting t' for x in t.

Example:

$$((((\lambda x.\lambda y.\lambda z.x)\ 3)\ 4)\ 5) \ \rightsquigarrow\ (((\lambda y.\lambda z.3)\ 4)\ 5) \ \rightsquigarrow\ ((\lambda z.3)\ 5) \ \rightsquigarrow\ 3$$

The Power of Lambda Calculus

Looping computation:

$$
\begin{array}{rcl}
(\lambda x.(x\ x))\ (\lambda x.(x\ x)) & \rightsquigarrow & [\lambda x.(x\ x)/x](x\ x) \\
& = & (\lambda x.(x\ x))\ (\lambda x.(x\ x)) \\
& \rightsquigarrow & (\lambda x.(x\ x))\ (\lambda x.(x\ x)) \\
& \rightsquigarrow & \cdots
\end{array}
$$

Iterating computation f:

$$
\begin{array}{rcl}
(\lambda x.(f\ (x\ x)))\ (\lambda x.(f\ (x\ x))) & \rightsquigarrow & f\ ((\lambda x.(f\ (x\ x)))\ (\lambda x.(f\ (x\ x)))) \\
& \rightsquigarrow & f\ (f\ (\lambda x.(f\ (x\ x)))\ (\lambda x.(f\ (x\ x)))) \\
& \rightsquigarrow & f\ (f\ (f\ (\lambda x.(f\ (x\ x)))\ (\lambda x.(f\ (x\ x))))) \\
& \rightsquigarrow & \cdots
\end{array}
$$

Lambda-Encoded Data

- Can represent numbers, lists, etc. as lambda terms.
- Church encoding:

$$
\begin{array}{lcl}
0 & := & \lambda s.\lambda z.z \\
1 & := & \lambda s.\lambda z.s\ z \\
2 & := & \lambda s.\lambda z.s\ (s\ z) \\
\ldots & &
\end{array}
$$

- Taking the successor of a Church-encoded number:

$$Succ := \lambda n.(\lambda s.\lambda z.s\ (n\ s\ z))$$

$$Succ\ 1 \ \rightsquigarrow\ \lambda s.\lambda z.s\ (1\ s\ z)\ \rightsquigarrow^*\ \lambda s.\lambda z.s\ (s\ z)\ =\ 2$$

- Addition for Church encoding:

$$plus := \lambda n.\lambda m.n\ Succ\ m$$

Typed Lambda Calculi

- Practical languages have primitive data, operations.
- Types used to enforce safe usage:

$$\begin{array}{lcl} 12 & : & \mathit{int} \\ + & : & \mathit{int} \to \mathit{int} \to \mathit{int} \\ \text{“hi''} & : & \mathit{string} \end{array}$$

- Typed lambda calculi are theoretical basis.
- Many different type systems proposed.
- Goal: prove type system *sound*:

  “Well typed programs do not go wrong.” [Milner]

  - $t : T \implies t\ Ok$
  - $t : T \ \wedge\ t \leadsto t' \implies t' : T \quad$ (e.g., $3 + 3 : \mathit{int} \ \wedge\ 3 + 3 \leadsto 6$)

Example: Simply Typed Lambda Calculus

Syntax of Types:

$$\begin{array}{lll} \textit{base types } b & & \\ \textit{simple types } T & ::= & b \mid T_1 \to T_2 \end{array}$$

Semantics:

$$\begin{array}{lll} [\![b]\!]_\sigma & = & \sigma(b) \\ [\![T_1 \to T_2]\!]_\sigma & = & \{t \in \textit{terms} \mid \forall t' \in [\![T_1]\!]_\sigma.\ (t\ t') \in [\![T_2]\!]_\sigma\} \end{array}$$

Assume $\sigma(b)$ *inversion-reduction closed* (for all b):

$$\frac{t' \rightsquigarrow t \qquad t \in \sigma(b)}{t' \in \sigma(b)}$$

Example of Using the Semantics

$$\lambda x.\lambda y.x \in [\![b_1 \to b_2 \to b_1]\!]_\sigma$$

Proof.

Assume arbitrary $t_1 \in [\![b_1]\!]_\sigma = \sigma(b_1)$.
Show $(\lambda x.\lambda y.x)\ t_1 \in [\![b_2 \to b_1]\!]_\sigma$.

Assume arbitrary $t_2 \in [\![b_2]\!]_\sigma = \sigma(b_2)$.
Show $((\lambda x.\lambda y.x)\ t_1)\ t_2 \in [\![b_1]\!]_\sigma = \sigma(b_1)$.

Holds because $((\lambda x.\lambda y.x)\ t_1)\ t_2 \rightsquigarrow^* t_1 \in \sigma(b_1)$ $\square$

Typing Semantics

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T} \quad \frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x.t : T_1 \to T_2} \quad \frac{\Gamma \vdash t_1 : T_2 \to T_1 \quad \Gamma \vdash t_2 : T_2}{\Gamma \vdash t_1\ t_2 : T_1}$$

- Logically less complex notion of typing.
- Basis for actual type-checking algorithms.
- Can be proved sound:

Theorem (Soundness)

If $\Gamma \vdash t : T$*, then* $\gamma t \in [\![T]\!]_\sigma$*, where*

$\gamma(x) \in [\![\Gamma(x)]\!]_\sigma$ *for all* $x \in dom(\sigma)$.

The Curry-Howard Isomorphism

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T} \quad \frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x.t : T_1 \to T_2} \quad \frac{\Gamma \vdash t_1 : T_2 \to T_1 \quad \Gamma \vdash t_2 : T_2}{\Gamma \vdash t_1\ t_2 : T_1}$$

The Curry-Howard Isomorphism

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T} \quad \frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x.t : T_1 \to T_2} \quad \frac{\Gamma \vdash t_1 : T_2 \to T_1 \quad \Gamma \vdash t_2 : T_2}{\Gamma \vdash t_1\ t_2 : T_1}$$

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T} \quad \frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x.t : T_1 \to T_2} \quad \frac{\Gamma \vdash t_1 : T_2 \to T_1 \quad \Gamma \vdash t_2 : T_2}{\Gamma \vdash t_1\ t_2 : T_1}$$

The Curry-Howard Isomorphism

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T} \quad \frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x.t : T_1 \to T_2} \quad \frac{\Gamma \vdash t_1 : T_2 \to T_1 \quad \Gamma \vdash t_2 : T_2}{\Gamma \vdash t_1\ t_2 : T_1}$$

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T} \quad \frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x.t : T_1 \to T_2} \quad \frac{\Gamma \vdash t_1 : T_2 \to T_1 \quad \Gamma \vdash t_2 : T_2}{\Gamma \vdash t_1\ t_2 : T_1}$$

$$\frac{\Gamma \qquad T}{\Gamma \vdash \qquad T} \quad \frac{\Gamma, \quad T_1 \vdash \quad T_2}{\Gamma \vdash \qquad T_1 \to T_2} \quad \frac{\Gamma \vdash \quad T_2 \to T_1 \quad \Gamma \vdash \quad T_2}{\Gamma \vdash \qquad T_1}$$

The Curry-Howard Isomorphism

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T} \quad \frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x.t : T_1 \to T_2} \quad \frac{\Gamma \vdash t_1 : T_2 \to T_1 \quad \Gamma \vdash t_2 : T_2}{\Gamma \vdash t_1\ t_2 : T_1}$$

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T} \quad \frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x.t : T_1 \to T_2} \quad \frac{\Gamma \vdash t_1 : T_2 \to T_1 \quad \Gamma \vdash t_2 : T_2}{\Gamma \vdash t_1\ t_2 : T_1}$$

$$\frac{\Gamma \qquad T}{\Gamma \vdash \qquad T} \quad \frac{\Gamma, \qquad T_1 \vdash \qquad T_2}{\Gamma \vdash \qquad T_1 \to T_2} \quad \frac{\Gamma \vdash \qquad T_2 \to T_1 \quad \Gamma \vdash \qquad T_2}{\Gamma \vdash \qquad T_1}$$

$$\frac{T \in \Gamma}{\Gamma \vdash T} \quad \frac{\Gamma, T_1 \vdash T_2}{\Gamma \vdash T_1 \to T_2} \quad \frac{\Gamma \vdash T_2 \to T_1 \quad \Gamma \vdash T_2}{\Gamma \vdash T_1}$$

Proofs and Programs

- Simply typed lambda terms are notations for proofs.
- The logic is minimal (constructive) propositional logic.
- The semantics of types <–> Kripke semantics for minimal logic.
- Let *Norm* be set of *normalizing* lambda terms.
  - $t \in \mathit{Norm}$ iff $\exists t'.t \leadsto^* t' \not\leadsto$.
- Can prove:

Theorem

Suppose $\sigma(b) \subseteq \mathit{Norm}$ (for all b).

Then $[\![T]\!]_\sigma \subseteq \mathit{Norm}$.

Corollary

Simply typable terms are normalizing.

The Tragedy of Programming

- Programs are full of bugs.
  - 1-10 for every 1000 lines of code?
  - Ok for web browser, not for flight control.
- State of the art: testing.
- We are building cathedrals of glass with jack hammers.
- But a new hope dawns...

Programming with Proofs

- Lambda calculus: bridge between programming, proving.
- Simply typed lambda calc. <–> min. prop. logic.
- Fancier type systems <–> more powerful logics.
- New generation of research languages:
  - Coq (INRIA), Agda (Chalmers), Ωmega (Portland), Guru (Iowa).
- Write programs, prove theorems about them.

$$\forall l : list\ a.\ rev\,(rev\ l) = l$$

- Write programs with rich types expressing properties.

$$rev : list\ a\ n\ \rightarrow list\ a\ n$$

- I believe this is a true revolution in programming.

Case Study: versat

- We wrote a verified logic solver in Guru.
  - **Duckki Oe**, Tianyi Liang, Corey Oliver, Kevin Clancy.
  - Guru is our verified-programming language.
- Modern solvers can solve huge logic problems.
  - 100s of thousands of propositional variables.
  - formulas with millions of logical operators.
  - sophisticated heuristics and optimizations.
- We proved (in Guru):
  - if the solver says the formula is unsatisfiable, then
  - one can derive a contradiction from it.
- 10k lines of code, proofs.
- Correct in theory, and in practice (compared to MiniSat).

Richer Type Systems: Levelized

	$\dots$
superkinds :	*kind*
	$\dots$
kinds :	*type*
	type $\rightarrow$ *type*
	$\dots$
types :	*int*
	int $\rightarrow$ *int*
	$\forall X : type.X \rightarrow X$,
	$\lambda X : type.X \rightarrow X$
	$\dots$
terms :	35,
	$\lambda x.x + x$,
	$\dots$

Richer Type Systems: Collapsed

- With levelized systems, each expression is in just one level.
- So cannot reuse that code across levels.
- Can view level structure this way:

$$type_0 \ : \ type_1 \ : \ type_2 \ : \ \cdots$$

- An exciting idea:

$$type : type$$

- Collapses all levels; cannot distinguish terms, types.
- Great reuse: multi-level data structures.

$$\frac{list : type \rightarrow type \qquad type : type}{list\ type : type}$$

- <u>But:</u> compositional semantics is quite challenging.

Types As Abstractions

- Goal: define "simple" semantics for *type*:*type*.
- Idea: view every term as a description of a set of terms.
  - $[\![\, int \,]\!] = \{0, 1, 2, \cdots\}$
  - $[\![0]\!] = \{0, ((\lambda x.x)\ 0), \cdots\}$
  - $[\![\, type \,]\!] = \{type, int, \cdots\}$
- Levelize the semantics: $[\![t]\!]^k_\sigma$.
- Crucial defining clause:

$$t_a \in [\![\lambda x : t_1.t_2]\!]^{k+1}_\sigma \Longrightarrow \forall t_b \in [\![t_1]\!]^{k+1}_\sigma . t_a\ t_b \in [\![t_2]\!]^{k+1}_{\sigma[x \mapsto [\![t_b]\!]^k_\sigma]}$$

- Can interpret this argument t_b at a lower level k.
- Handle case when t_b is a type or a term uniformly.

Conclusion

- Semantics is essential for programming language design.
  - reduction semantics for terms: $(\lambda x.t)\ t' \rightsquigarrow [t'/x]t$
  - compositional semantics of types: $[\![T]\!]_\sigma$
  - typing semantics: $\Gamma \vdash t : T$
- Use semantics to prove type system sound.
- Typed lambda calculus for programs, proofs.
- Prove code is correct!
- Collapse language levels with *type*:*type*.
- Can "types as abstractions" yield compositional semantics?

http://queuea9.wordpress.com

Thanks!