THM: A Hindley-Milner Typed Language

Taran Lynn

March 12, 2022

1 Introduction

In this paper I will cover the design and implementation of the THM programming language. THM is a minimalist Hindley-Milner typed programming language with algebraic data types. The goal of designing THM was to create a language and implementation that was simple enough that it could be used as a teaching tool for courses on type inference. To this end the it has a small specification (see the appendix) and a small implementation (less than 1,000 lines of Haskell).

THM is loosely base on the ML family of languages, specifically Standard ML. Besides dropping support for modules, other major changes were made to the syntax and semantics to reduce the size of the specification and implementation. For example, unlike most MLs, THM is lazily evaluated. This is relatively trivial to implement, and avoids the need to implement other control structures. Another major change is that there is no `case` construct. Instead there is a `case` **function**, which acts as the elimination rule for a type.

The THM interpreter is implemented in Haskell. To use it the user simply gives it a list of files with type and function definitions to type check and load, and the interpreter drops the user into a read/eval/print loop (REPL) that process THM expressions. An example of one definition file is

```
type Nat [ Z S(Nat) ]
fix natInd x f = case[Nat] x (\p. f (natInd x f p));

pred = case[Nat] Z (\p. p);
fix add m n = case[Nat] n (\p. S (add p n)) m;
fix mult m n = case[Nat] Z (\p. add n (mult p n)) m;
```

, and a corresponding REPL session is

```
\> Z
Z : Nat
\> S
S : Nat -> Nat
\> mult (S (S Z)) (S (S (S Z)))
S (S (S (S (S (S Z))))) : Nat
\>
```

The following sections of the paper will give an intutionistic description of the background theory, design, and implementation of THM. For a more formal description, please see the appendix.

2 Background and Related Work

Lambda calculus was originally developed by Alonzo Church as a foundational formulation of mathematics and computation. However, because the lambda calculus allowed for non-reducible terms, it was shown to be inconsistent. To get around this Church introduced types to the lambda calculus, creating the simply typed lambda calculus[1]. When lambda calculus began to be as a bases programming languages, computer scientists also started adopting its type theories[1].

A mathematician by the name of J. Roger Hindley later worked on a method for deducing types of SKI combinator expressions[4]. About a decade later Robin Milner worked on ML, the metalanguage for the LCF proof system, based on typed lambda calculus. As part of ML there was an algorithm to infer the types of terms to reduce programmer overhead. This algorithm W for inferring types was later presented in [8], which they later noted corresponded to Hindley's work. Luis Damas then refined the corresponding type system as part of his PhD thesis[2, 3]. The resulting type system is called the Hindley-Milner-Damas or Hindley-Milner typing.

It should be noted that Hindley-Milner type inference relies on a unification algorithm to work. Unification is a process that take multiple equations relating constructed objects that contain variables, and determining what values for the variables could make the equations valid. My work uses the unification algorithm presented by Martelli and Montanari[6]. This unification algorithm was chosen because it is fairly straightforward to implement and is decently efficient.

Several languages later built off of the work of ML. A notable example is Standard ML (SML)[9], which is notable among programming languages for having its specification fully mathematically formalized. More mainstream languages based off

[1]By this time there where many such type theories.

of ML include Haskell[5] and Rust. One important component of the languages is algebraic data types (ADTs). ADTs form a convenient way to describe data, and are important in Hindley-Milner typed languages because the restrictions necessary for decidable inference limits the usability of Church encodings.

3 Design

3.1 Base Language

At the heart of THM is the lambda calculus. Expressions, denoted by e, are formed by several different constructs.

Variables Denoted by v, refer to the parameters of encapsulating lambda expressions.

Lambda Abstractions Denoted by $\lambda v.e$, are anonymous functions that take v as a parameter and return e. Lambda abstractions group to the right, so $\lambda u.\lambda v.e$ is equivalent to $\lambda u.(\lambda v.e)$. Note that the user may use \ instead of λ for ease of use on ASCII keyboards.

Applications Denoted by $e_1\ e_2$, apply the term e_2 to e_1. Since THM is lazy evaluated, we evaluating applications by first evaluating e_1. If e_1 evaluates to $\lambda v.e_3$, then we evaluate e_3 with v bond to the thunk formed from e_2 (see appendix B.3 for a formal description). Additionally application rules also exist, which will be detailed in later parts of the text.

Let Bindings Denoted by (let $v = e_1$; in e_2) binds the thunk formed from e_1 to v and evaluates e_2.

Fix-point Operator Denoted by fix, has a special application rule such that fix f evaluates to f (fix f). This allows for recursion in functions.

Beyond these basic forms some syntactic sugar is also added to the base language (see appendix A.2).

3.2 Type Inference

Given any expression, THM will decide if the expression is typeable, and if so will infer the most general type for the expression. Types, denoted by t, in the base THM have several constructions.

Type Variables Denoted by $`\tau$, type variables represent unspecified types.

Function Types Denoted by $t_1 \to t_2$, represent functions that take arguments of type t_1 and return values of type t_2. Like lambda abstractions these group to the right, so $t_1 \to t_2 \to t_3$ is equivalent to $t_1 \to (t_2 \to t_3)$.

Quantified Types Denoted by $\forall`\alpha.t$, these denote types where the type variable $`\alpha$ can change. If one likens type variables to expression variables, then quantified types can be likened to lambda abstractions. One thing to note is that only let binded variables may have quantified types, and not any other expression or value. This restriction is necessary for type inference to be decidable.

Before I describe how to infer what the most general type of an expression, let me briefly describe type unification. Suppose we have a set of equations between types (i.e. $t_1 = t_2, t_3 = t_4, \ldots$), the goal of type unification is to determine what types can be substituted for the equations type variables to make the equations valid. For example, suppose we have the equations

$$\begin{aligned} `\tau_1 \to `\tau_2 &= `\tau_1 \to (`\tau_1 \to `\tau_3) \\ `\tau_3 &= `\tau_4 \to `\tau_1 \\ `\tau_4 &= `\tau_4 \end{aligned}$$

Applying the unification algorithm would give

$$\begin{aligned} `\tau_2 &= `\tau_1 \to `\tau_4 \to `\tau_1 \\ `\tau_3 &= `\tau_4 \to `\tau_1 \end{aligned}$$

A couple things to note are that

1. Obvious equalities (i.e. $x = x$) are removed.
2. All type variables are fully expanded on right hand side of the equations. This is why $`\tau_2 = `\tau_1 \to `\tau_4 \to `\tau_1$ and not $`\tau_2 = `\tau_1 \to `\tau_3$

For a more detailed explanation of the unification algorithm, please refer to [7].

Now I will describe the type inference algorithm. I will be omitting some details for the sake of clarity, see appendix B.2 for the complete formalization. At all stages of type inference we have a context Γ, which is simply the mapping of variables to the types of the values they are bound to. There are several cases

- If we have a variable v then we take $t = \Gamma(v)$. If t is a quantified type, then we apply it to free type variables.

- If we have a lambda abstraction $\lambda v.e$, then we first declare a new unique type variable $`\tau$; infer the type $t$ of $e$ while setting $\Gamma(v) = `\tau$; and deduce the abstraction type as $`\tau \rightarrow t$.
- If we have an application $e_1\ e_2$ we first infer $e_1 : t_1$ and $e_2 : t_2$; then let $`\tau$ be a new unique type variable; unify $t_1$ with $t_2 \rightarrow `\tau$; and deduce that $e_1\ e_2$ has the type of whatever $`\tau$ unifies as.
- If we have a let binding (let $v := e_1;$ in e_2), then we deduce the type of e_1 and quantify over it to get the type t_1; and infer the type t_2 of e_2 while setting $\Gamma(v) = t_1$; with the inference giving (let $v := e_1;$ in e_2) : t_2. Note that quantifying over a type t means we take all the free type variable $`\tau_1, \ldots, `\tau_n$ in t, and return the type $\forall `\tau_1. \ldots \forall `\tau_n.t$. That is, we convert it into a quantified type.
- For the fix-point operator we simply let $`\tau$ be a new type variable, and infer that $\text{fix} : (`\tau \rightarrow `\tau) \rightarrow `\tau$.

That was a fairly brief description of the Hindley-Milner type inference algorithm. For more details or clarifications please see the appendix and Milner and Damas' work[8, 2, 3][2].

3.3 Algebraic Data Types

As note earlier quantified types can not be passed through function application. These means that complex functions that use Church encodings may fail. For example, the common definition of the predecessor function for natural numbers is

$$\begin{aligned} Z\ x\ f &= x; \\ S\ n\ x\ f &= f\ (n\ x\ f); \\ \mathit{pair}\ x\ y\ f &= f\ x\ y; \\ \mathit{pred}\ n &= (n\ (\mathit{pair}\ Z\ Z)\ (\lambda p.\ p\ (\lambda x\ y.\ \mathit{pair}\ y\ (S\ x))))\ (\lambda x\ \_.\ x); \end{aligned}$$

However, running $\mathit{pred}\ (S\ (S\ Z))$ will give a complex type error due to the fact that we swap the pair x and y on each update, giving a circular type.

[2]Milner's work comes from a mathematical perspective, and would be less useful in to learn from in this context

To improve usability the base language is thus extended with algebraic data types (ADTs). These allow us to easily construct new types. The syntax for ADTs is

$$\begin{aligned}&\text{type } T\ `\tau_1\ \ldots\ `\tau_m\ [\\ &\qquad C_1(t_{11}, \ldots, t_{1n_1})\\ &\qquad \vdots\\ &\qquad C_k(t_{k1}, \ldots, t_{kn_k})\\ &]\end{aligned}$$

Where T is the type name, $`\tau_1, \ldots, `\tau_m$ are its type parameters, and $C_1, \ldots, C_k$ are its constructors. Each constructor C_i forms a new function, with

$$C_i : \forall `\tau_1\ \ldots\ `\tau_m.\ t_{i1} \to \cdots \to t_{in_i} \to T\ `\tau_1\ \ldots\ `\tau_m$$

. A new built-in case matching function $\text{case}\,[T]$ is also formed, with type

$$\begin{aligned}\text{case}\,[T] : \forall `\tau_1\ \ldots\ `\tau_m\ `\rho.\quad T\ `\tau_1\ \ldots\ `\tau_m &\to (t_{11} \to \cdots \to t_{1n_1} \to `\rho)\\ &\to \cdots \to (t_{k1} \to \cdots \to t_{kn_k} \to `\rho) \to `\rho\end{aligned}$$

. If we consider a value $z = C_i\, x_1\ \ldots\ x_{n_i}$, then case matching evaluates $(\text{case}\,[T]\ z\ f_1\ \ldots\ f_k)$ to $f_i\ x_1\ \ldots\ x_{n_i}$. Essentially we give a function for each constructor of T, and then apply it the correct function to the arguments of the constructor.

For example, lists can be defined as

$$\begin{aligned}&\text{type List } `a\ [\\ &\qquad \text{nil}\\ &\qquad \text{cons}(`a, \text{List } `a)\\ &]\end{aligned}$$

Here we have

$$\begin{aligned}\text{nil} &: \forall `a.\ \text{List } `a\\ \text{cons} &: \forall `a.\ `a \to \text{List } `a \to \text{List } `a\end{aligned}$$

If $x : t$, then using these constructors we could get $(\text{cons}\ x\ \text{nil}) : \text{List}\ t$. We also have

$$\text{case}\,[\text{List}] : \forall `a `\rho.\ \text{List}\ a \to `\rho \to (`a \to \text{List } `a \to `\rho) \to `\rho$$

, which for example evaluates for each constructor as

$$\text{case}\,[\text{List}]\ \ \text{nil}\ x\ f \Rightarrow x$$
$$\text{case}\,[\text{List}]\ \ (\text{cons}\ y\ l)\ x\ f \Rightarrow f\ y\ l$$

One interesting thing to note about THM's ADT is that they are flexible enough to also allow for bottom types to be defined. A bottom type is defined as

$$\text{type Bottom}\ [\,]$$

. The important thing to note is that there is no way to construct a value with type bottom (aside from infinite recursion). This can make it useful for abstractions where the user wants to state that we should never get a value in some location. This simply demonstrates the flexibility of THM's ADTs.

4 Implementation and Use

THM is implemented as an interpreter written in Haskell using the Stack build tool[10]. Basic usage is to give the interpreter a set of file with function and type definitions to load, after which it will print the types of loaded functions, and then start the REPL to evaluate expressions. An example session is

```
sh > rlwrap stack run lib/prelude.thm
if : Bool -> 'g -> 'g -> 'g
if = <function>

not : Bool -> Bool
not = <function>

and : Bool -> Bool -> Bool
and = <function>

or : Bool -> Bool -> Bool
or = <function>

...

\> and true false
false : Bool
```

```
\> or true false
true : Bool
\> case[Bottom]
case[Bottom] : Bottom -> 'a
```

Note that THM provides a set of library files, with lib/prelude.thm containing basic definitions. Other library files providing natural number, lists, and stateful semantics are also provided. The source code can be found at `https://github.com/lambda-11235/thm`.

5 Conclusion

THM is a Hindley-Milner typed programming language designed for educational use. It's formal specification is small enough to be described in 5 pages (see the appendix), and its source code is composed of 10 files and less than 800 lines of code. In fact, the type checking and evaluation components (the main components of interest in teaching) are less than 400 lines of code. Compared to implementations of SML or Haskell, this makes THM a viable project to study for graduate students trying to learn how to implement type inference, ADTs, and lazy evaluation. If you, the reader, are a professor or teacher who is or will be teaching type theory, consider using this project for instructional use. Pull requests to improve the code's approachability are also welcome[3].

References

[1] Alonzo Church. "A Formulation of the Simple Theory of Types". In: *J. Symb. Log.* 5 (1940), pp. 56–68.

[2] Luis Damas and Robin Milner. "Principal Type-Schemes for Functional Programs". In: *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '82. Albuquerque, New Mexico: Association for Computing Machinery, 1982, pp. 207–212. ISBN: 0897910656. DOI: `10.1145/582153.582176`. URL: `https://doi.org/10.1145/582153.582176`.

[3] Luis Manuel Martins Damas. "Type Assignment in Programming Languages". PhD thesis. University of Edinburgh, 1984.

[3]`https://github.com/lambda-11235/thm/pulls`

[4] R. Hindley. “The Principal Type-Scheme of an Object in Combinatory Logic”. In: *Transactions of the American Mathematical Society* 146 (1969), pp. 29–60. ISSN: 00029947. URL: `http://www.jstor.org/stable/1995158`.

[5] Simon Marlow. “Haskell 2010 Language Report”. In: 2010.

[6] Alberto Martelli and Ugo Montanari. “An Efficient Unification Algorithm”. In: *ACM Trans. Program. Lang. Syst.* 4.2 (Apr. 1982), pp. 258–282. ISSN: 0164-0925. DOI: `10.1145/357162.357169`. URL: `https://doi.org/10.1145/357162.357169`.

[7] Alberto Martelli and Ugo Montanari. “An Efficient Unification Algorithm”. In: *ACM Trans. Program. Lang. Syst.* 4 (1982), pp. 258–282.

[8] Robin Milner. “A theory of type polymorphism in programming”. In: *Journal of Computer and System Sciences* 17.3 (1978), pp. 348–375. ISSN: 0022-0000. DOI: `https://doi.org/10.1016/0022-0000(78)90014-4`. URL: `https://www.sciencedirect.com/science/article/pii/0022000078900144`.

[9] Robin Milner et al. “The Definition of Standard ML (Revised)”. In: 1997.

[10] *The Haskell Tool Stack*. URL: `https://docs.haskellstack.org/`.

[11] Wikipedia contributors. *Hindley–Milner type system — Wikipedia, The Free Encyclopedia*. [Online; accessed 30-January-2022]. 2022. URL: `https://en.wikipedia.org/w/index.php?title=Hindley%E2%80%93Milner_type_system&oldid=1067862972`.

A Syntax

A.1 Abstract Syntax

x indicates a variable, e is an expression, C is a data constructor, $`\tau$ indicates a type variable, $t$ is a type, $T$ is a type constructor, and $`\alpha$ is a quantified type variable.

$$\text{TypeDef} := \text{type } T\ `\tau_1\ \cdots\ `\tau_m \left[C_1\left(t_{11}, \ldots, t_{1n_1}\right) \cdots C_k\left(t_{k1}, \ldots, t_{kn_k}\right)\right] \tag{1}$$

$$t := t_1 \rightarrow t_2 \mid T\ t_1 \ldots t_n \mid `\tau \mid \forall `\alpha.t \tag{2}$$

$$\begin{aligned}
e &:= \text{Var} \mid \text{Cons} \mid \text{Lambda} \mid \text{App} \mid \text{Let} \mid \text{Fix} \mid \text{Case} && (3)\\
\text{Var} &:= x && (4)\\
\text{Cons} &:= C && (5)\\
\text{Lambda} &:= \lambda x.e \mid \lambda\_.e && (6)\\
\text{App} &:= e_1\ e_2 && (7)\\
\text{Let} &:= \text{let } x = e_1;\ \text{ in } e_2 && (8)\\
\text{Fix} &:= \text{fix} && (9)\\
\text{Case} &:= \text{case}\,[T] && (10)
\end{aligned}$$

A.2 EBNF Concrete Syntax

Comments begin with #. Symbols as any non-keyword strings that match the regex `[a-zA-Z][0-9a-zA-Z]*`. A number is `[0-9]+`.

$$\begin{aligned}
\text{tydef} &::= \text{"type"}, \text{symbol}, \{\text{tyvar}\}, \text{"["}, \{\text{condef}\}, \text{"]"}; && (11)\\
\text{condef} &::= \text{symbol}, \left[\text{"("}, \{\text{type}\}, \text{")"}\right]; && (12)
\end{aligned}$$

$$\begin{aligned}
\text{type} &::= \text{tycon}, [\text{"}-> \text{"}, \text{type}]; && (13)\\
\text{tycon} &::= \text{tyatom} \mid \text{symbol}, \{\text{tyatom}\}; && (14)\\
\text{typatom} &::= \text{tyvar} \mid \text{"("}, \text{type}, \text{")"}; && (15)\\
\text{tyvar} &::= \text{"}'\text{"}, \text{symbol}; && (16)
\end{aligned}$$

$$\begin{aligned}
\text{funcdef} &::= [\text{"fix"}]\,, \text{symbol}, \{\text{symbol}\}\,, \text{" = "}, \text{expr}, \text{";"}; && (17)\\
\text{expr} &::= \text{"let"}, \text{funcdef}, \{\text{funcdef}\}\,, \text{"in"}, \text{expr}; && (18)\\
&\mid (\text{"\textbackslash"}|\text{"}\lambda\text{"}), \text{arg}, \{\text{arg}\}\,, \text{"."}, \text{expr}; && (19)\\
&\mid \text{app} && (20)\\
\text{app} &::= \text{atom} \mid \text{app}, \text{atom}; && (21)\\
\text{atom} &::= \text{"("}, \text{expr}, \text{")"} \mid \text{symbol} \mid \text{case}; && (22)\\
\text{arg} &::= \text{symbol} \mid \text{"\_"} && (23)\\
\text{case} &::= \text{"case"}, \text{"["}, \text{symbol}, \text{"]"}; && (24)
\end{aligned}$$

Note that the EBNF above has some syntatic sugar to make programming easier. The rules to convert from the main language to the abstract one are

$$\begin{aligned}
\lambda x_1 x_2 \cdots .e &\Rightarrow \lambda x_1.\lambda x_2.\lambda \cdots .e && (25)\\
\text{let } x_1 := e_1;\ x_2 := e_2;\ \ldots\ \text{in } e &\Rightarrow \text{let } x_1 := e_1;\ \text{in let } x_2 := e_2;\ \text{in } \ldots\ \text{in } e && (26)\\
\text{let } f\ x_1\ x_2\ \cdots := e_1;\ \text{in } e_2 &\Rightarrow \text{let } f := \lambda x_1 x_2 \cdots .e_1;\ \text{in } e_2 && (27)\\
\text{let fix } f\ x_1\ x_2\ \cdots := e_1;\ \text{in } e_2 &\Rightarrow \text{let } f := \text{fix }\ (\lambda f x_1 x_2 \cdots .e_1)\,;\ \text{in } e_2 && (28)
\end{aligned}$$

B Semantics

B.1 Type Definitions

Suppose we have a type definition following the pattern given in (1). There are three semantic rules that need to be verified.

1. For any pair of type variables $a, b \in \{`\tau_1, \ldots, `\tau_m\}$ that T parameterizes over $a \neq b$.
2. For any type t used in one of the constructors $C_1, \ldots, C_k$ we require $\text{freevars}\,(t) \subseteq \{`\tau_1, \ldots, `\tau_m\}$.
3. For any type t used in one of the constructors $C_1, \ldots, C_k$ we require that the only type constructors used are T and previously defined type constructors.

B.2 Static

The following is largely adapted from a description of algorithm W from Wikipedia[11]. Notable typing rules that extend those from Wikipedia are those for recursive let bindings and matches.

newtyvar creates a unique type variable. unify (t_1, t_2) denotes the unification substitution of t_1 and t_2. It does by adding $t_1 = t_2$ to the set of unification equations and then simplifying. inst (σ) takes a possibly quantified type (i.e. of the form $\sigma = \forall`\alpha_1, \dots, `\alpha_n.t$) and produces the t_u by replacing the quantified type variables with new unique ones, that is

$$\begin{aligned} `\tau_1, \dots, `\tau_n &= \text{newtyvar} \\ t_u &= [`\tau_1/`\alpha_1, \dots, `\tau_n/`\alpha_n]t \end{aligned}$$

gen (Γ, t) does the opposite of inst (σ), that is it quantifies over all the free type variables in t that not also free in Γ, or

$$\begin{aligned} \{`\alpha_1, \dots, `\alpha_n\} &= \text{freevars}\,(t) - \text{freevars}\,(\Gamma) \\ \text{gen}\,(\Gamma, t) &= \forall`\alpha_1 \ \dots \ `\alpha_n, t \end{aligned}$$

The syntax $\Gamma \vdash e : t$ means that in the context Γ, e has type t.

Var

$$\frac{x : \sigma \in \Gamma \qquad t = \text{inst}\,(\sigma)}{\Gamma \vdash x : t} \tag{29}$$

Cons Suppose $C(t_1, \dots, t_n)$ is a constructor for some type $T\ `\tau_1 \cdots `\tau_m$, then

$$\frac{}{\Gamma \vdash C : \forall`\tau_1 \cdots `\tau_m, t_1 \to \cdots \to t_n \to T\ `\tau_1 \cdots `\tau_m} \tag{30}$$

Lambda

$$\frac{`\tau = \text{newtyvar} \qquad \Gamma, x : `\tau \vdash e : t}{\Gamma \vdash \lambda x.e : `\tau \to t} \tag{31}$$

$$\frac{`\tau = \text{newtyvar} \qquad \Gamma \vdash e : t}{\Gamma \vdash \lambda\_.e : `\tau \to t} \tag{32}$$

App

$$\frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2 \qquad `\tau = \text{newtyvar} \qquad \text{unify}\,(t_1, t_2 \to `\tau)}{\Gamma \vdash e_1\ e_2 : `\tau} \tag{33}$$

Let

$$\frac{\Gamma \vdash e_1 : t \qquad \Gamma, x : \text{gen}\,(\Gamma, t) \vdash e_2 : t`}{\Gamma \vdash \text{let } x = e_1;\ \text{in } e_2 : t`} \tag{34}$$

Fix

$$\frac{`\tau_1, `\tau_2 = \text{newtyvar}}{\Gamma \vdash \text{fix} : (`\tau_1 \to `\tau_2) \to (`\tau_1 \to `\tau_2)} \tag{35}$$

Case Suppose we have a type definition following the pattern given in (1).

$$\frac{`\rho = \text{newtyvar}}{\Gamma \vdash \text{case}\,[T] : T\ `\tau_1\ \cdots\ `\tau_m \to (t_{11} \to \cdots \to t_{1n_1} \to `\rho) \to \cdots \to (t_{k1} \to \cdots \to t_{kn_k} \to `\rho) \to `\rho} \tag{36}$$

B.3 Dynamic

Let σ be the context of variable bindings and their environments. σ thus has the type

$$\Sigma = X \to E \times \Sigma$$

, where X is the set of all variables, and E is the set of all expressions. The notation $\sigma' = [(e, s)/x]\sigma$ means

$$\sigma'(y) = \begin{cases} (e, s) & \text{if } y = x \\ \sigma(y) & \text{if } y \neq x \end{cases}$$

Note that $(e, s) : E \times \Sigma$ is the thunk formed from the expression e in some environment s.

Var

$$\frac{\sigma(x) = (e, s)}{\langle x, \sigma \rangle \to \langle e, s \rangle} \tag{37}$$

Lambda Lambda expressions are fully reduced.

App

$$\frac{\langle e_1, \sigma \rangle \to \langle e_1', \sigma \rangle}{\langle e_1\ e_2, \sigma \rangle \to \langle e_1'\ e_2, \sigma \rangle} \tag{38}$$

$$\frac{}{\langle (\lambda x.e_1)\ e_2, \sigma \rangle \to \langle e_1, [(e_2, \sigma)/x]\sigma \rangle} \tag{39}$$

$$\frac{}{\langle (\lambda \_.e_1)\ e_2, \sigma \rangle \to \langle e_1, \sigma \rangle} \tag{40}$$

Let

$$\frac{\langle e_1, \sigma\rangle \to \langle e_1', \sigma\rangle}{\langle \text{let } x = e_1;\ \text{in } e_2, \sigma\rangle \to \langle \text{let } x = e_1';\ \text{in } e_2, \sigma\rangle} \tag{41}$$

$$\frac{}{\langle \text{let } x = e_1;\ \text{in } e_2, \sigma\rangle \to \langle e_2, [(e_1, \sigma)/x]\sigma\rangle} \tag{42}$$

Fix

$$\frac{}{\langle \text{fix } f, \sigma\rangle \to \langle f\ (\text{fix } f), \sigma\rangle} \tag{43}$$

Case Suppose we have a type definition following the pattern given in (1). Let C_i be the ith listed constructor, then

$$\frac{}{\langle \text{case}\,[T]\ \ (C_i\ x_1\ \ \cdots\ \ x_n)\ f_1\ \ \cdots\ \ f_k, \sigma\rangle \to \langle f_i\ x_1\ \ \cdots\ \ x_n, \sigma\rangle} \tag{44}$$

$$\tag{45}$$