synrc research center s.r.o.
Roháčova 141/18, Praha 3 13000, Czech Republic

Intermediate Language with Dependent Types for Erlang/OTP applications.

Technical Article

Maxim Sokhatsky, Synrc Research Center

Kyiv 2017

Contents

1 Introduction

LISP. Untyped lambda calculus was discovered as an inner language of the space at origin (Curry, Church, 1932). This language was manifested as LISP (McCarthy, 1958) that was built upon: cons, nil, eq, atom, car, cdr, lambda, apply and id. It was parts of inductive types lately known as inductive type constructors. Still untyped lambda calculus is used as an extraction target for many provers (Idris, F*), and also manifests in different domain languages (JavaScript, Erlang).

ML/LCF. Further teardown of inner space language was ML language, founded merely on algebraic datatypes and algebra on higher terms rather than categorical semantic. Lately it was fixed with categorical methods in CPL (Hagino, 1987) and Charity (Cockett, 1992). Milner, assisted by Morris and Newey designed Meta Language for the purpose of builing LCF in early 70-s. LCF was a predecessor family of automated math provers: HOL88, HOL90, HOL98 and HOL/Isabelle which is now built using Poly/ML.

Fully Automated Provers. In that period during 80-90s other automated math systems were appeared: AUTOMATH (de Bruijn, 1967), Mizar (Trybulec, 1989), PVS (Owre, Rushby, Shankar, 1995), ACL2 (Boyer, Kaufmann, Moore, 1996) and Otter (McCune, 1996).

MLTT. Contemporary provers (built upon consistent Martin-Löf Type Theory, 1972) like Agda, Coq, Lean, F*, Idris are based on Barendregt and Coquand' CoC with different flavours of inifinity universe hierarchies and Calculus of Inductive Constructions. Some of them are automated and some are trying to be and general purpose programming languages with proving facilities.

2 Motivation

From PTS to HTS. We want to have flexible detachable layers on top of PTS core. Then Sigma for proving. Then well-founded trees or polynomial functors as known as data and record. Then higher path types, interval arithmetic, glue and comp for HIT. Each layers is driven by differenth math, the common in only the method – category theory.

Extensible Language Design. Encoding of inductive types is based on categorical semantic of compilation to PTS. All other syntax constructions are inductive definitions, plugged into the stream parser. AST of the PTS language is also defined in terms of inductive constructions and thus allowed in the macros. The language of polynomial functors (data and record) and core language of the process calculus (spawn, receive and send) are just macrosystem over Om language, its syntax extensions.

Changable Encodings. In pure CoC we have only arrows, so all inductive type encodings would be Church-encoding variations. Most extended nowadays is Church-Boehm-Berrarducci encoding, which dedicated to inductive types. Another well known are Scott (lazyness), Parigot (lazyness and constant-time iterators) and CPS (continuations) encodings. However most of them require variations of Fixpoint types.

Proved Categorical Semantic. There was modeled a math model (using higher-order categorical logic) of encoding, which calculates (co)limits in a cathegory of (co)algebras built with given set of (de)constructors. We call such encoding in honour of Lambek lemma that leeds us to the equality of (co)initial object and (co)limit in the categories of (co)algebras. Such encoding works with dependent types and its consistency is proved in Lean model.

3 Intermediate Language Om

The Om language is a dependently typed lambda calculus, an extension of Barendregt' and Coquand Calculus of Constructions with predicative hierarchy of indexed universes. There is no fixpoint axiom needed for the definition of infinity term dependance.

All terms respect ranking *Axioms* inside sequence of universes *Sorts* and complexity of the dependent term is equal maximum complexity of term and its dependency *Rules*. The type system is completely described by the following PTS notation (due to Barendregt):

$$\begin{cases} Sorts = Type.\{i\},\ i : Nat \\ Axioms = Type.\{i\} : Type.\{inc\ i\} \\ Rules = Type.\{i\} \leadsto Type.\{j\} : Type.\{max\ i\ j\} \end{cases}$$

An intermediate Om language is based on Henk [6] languages described first by Erik Meijer and Simon Peyton Jones in 1997. Leter on in 2015 Morte impementation of Henk design appeared in Haskell, using Boem-Berrarducci encoding of non-recursive lambda terms. It is based only on one type constructor Π, its special case λ and theirs eliminators: *apply* and *curry*, infinity number of universes, and one computation rule called β-reduction. The design of Om language resemble Henk and Morte both design and implementation. This language indended to be small, concise, easy provable and able to produce verifiable peace of code that can be distributed over the networks, compiled at target with safe trusted linkage.

3.1 BNF

Om syntax is compatible with λC Coquand's Calculus of Constructions presented in Morte and Henk languages. However it has extension in a part of specifying universe index as a **Nat** number.

```
<> ::= #option
 I ::= #identifier
 U ::= * < #number >
 O ::= U
     | I | ( O ) | O O | O → O
     | λ ( I : O ) → O
     | ∀ ( I : O ) → O
```

3.2 Hierarchy

```
dep Arg Out impredicative → Out
dep Arg Out predicative   → max Arg Out

h Arg Out → dep Arg Out om:hierarchy(impredicative)
```

3.3 Universes

```
star (:star,N) → N
star _         → (:error, "*")
```

3.4 Functions

```
func ((:forall,),(I,O)) → true
func T                  → (:error,(:forall,T))
```

3.5 Variables

```
var N B              → var N B (proplists:is_defined N B)
var N B true         → true
var N B false        → (:error,("free var",N,proplists:get_keys(B)))
```

3.6 Shift

```
sh (:var,(N,I)),N,P) when I>=P  → (var,(N,I+1))
sh ((:forall,(N,0)),(I,O)),N,P) → ((:forall,(N,0)),sh I N P,sh O N P+1)
sh ((:lambda,(N,0)),(I,O)),N,P) → ((:lambda,(N,0)),sh I N P,sh O N P+1)
sh (Q,(L,R),N,P)                → (Q,sh L N P,sh R N P)
sh (T,N,P)                      → T
```

3.7 Substitution

```
sub Term Name Value                  → sub Term Name Value 0
sub (:arrow,          (I,O)) N V L → (:arrow,        sub I N V L,sub O N V L);
sub ((:forall,(N,0)),(I,O)) N V L → ((:forall,(N,0)),sub I N V L,sub O N(sh V N 0)L+1)
sub ((:forall,(F,X)),(I,O)) N V L → ((:forall,(F,X)),sub I N V L,sub O N(sh V F 0)L)
sub ((:lambda,(N,0)),(I,O)) N V L → ((:lambda,(N,0)),sub I N V L,sub O N(sh V N 0)L+1)
sub ((:lambda,(F,X)),(I,O)) N V L → ((:lambda,(F,X)),sub I N V L,sub O N(sh V F 0)L)
sub (:app,            (F,A)) N V L → (:app,sub F N V L,sub A N V L)
sub (:var,            (N,L)) N V L → V
sub (:var,            (N,I)) N V L when I>L → (:var,(N,I-1))
sub T                        _ _ _ → T.
```

3.8 Normalization

```
norm :none                      → :none
norm :any                       → :any
norm (:app,(F,A))               → case norm F of
                                  ((:lambda,(N,0)),(I,O)) → norm (subst O N A)
                                                       NF → (:app,(NF,norm A)) end
norm (:remote,N)                → cache (norm N [])
norm (:arrow,          (I,O))  → ((:forall,("_",0)),(norm I,norm O))
norm ((:forall,(N,0)),(I,O))   → ((:forall,(N,0)),  (norm I,norm O))
norm ((:lambda,(N,0)),(I,O))   → ((:lambda,(N,0)),  (norm I,norm O))
norm T                          → T
```

3.9 Definitional Equality

```
eq ((:forall,("_",0)), X) (:arrow,Y)     → eq X Y
eq (:app,(F1,A1))         (:app,(F2,A2)) → let true = eq F1 F2 in eq A1 A2
eq (:star,N)              (:star,N)      → true
eq (:var,(N,I))           (:var,(N,I))   → true
eq (:remote,N)            (:remote,N)    → true
eq ((:farall,(N1,0)),(I1,O1))
   ((:forall,(N2,0)),(I2,O2)) →
   let true = eq I1 I2 in eq O1 (subst (shift O2 N1 0) N2 (:var,(N1,0)) 0)
eq ((:lambda,(N1,0)),(I1,O1))
   ((:lambda,(N2,0)),(I2,O2)) →
   let true = eq I1 I2 in eq O1 (subst (shift O2 N1 0) N2 (:var,(N1,0)) 0)
eq (A,B)                        → (:error,(:eq,A,B))
```

3.10 Type Checker

```
type (:star,N)                _ → (:star,N+1)
type (:var,(N,I))             D → let true = var N D in keyget N D I
type (:remote,N)              D → cache type N D
type (:arrow,(I,O))           D → (:star,h(star(type I D)),star(type O D))
type ((:forall,(N,0)),(I,O))  D → (:star,h(star(type I D)),star(type O [(N,norm I)|D]))
type ((:lambda,(N,0)),(I,O))  D → let star (type I D),
                                  NI = norm I in ((:forall,(N,0)),(NI,type(O,[(N,NI)|D])))
type (:app,(F,A))             D → let T = type(F,D),
                                  true = func T,
                                  ((:forall,(N,0)),(I,O)) = T,
                                  Q = type A D,
                                  true = eq I Q in norm (subst O N A)
```

3.11 Target Erlang VM and LLVM platforms

This works expect to compile to limited target platforms. For now Erlang, Haskell and LLVM is awaiting. Erlang version is expected to be useful both on LING and BEAM Erlang virtual machines.

4 Exe Macrosystem

Exe is a general purpose functional language with functors, lambdas on types, recursive algebraic types, higher order functions, corecursion, free monad for effects encoding. It compilers to a small core of dependent type system without recursion called Om. This language intended to be useful enough to encode KVS (database), N2O (web framework) and BPE (processes) applications.

4.1 Compiler Passes

The underlying OM typechecker and compiler is a target language for EXE general purpose language.

EXPAND	EXE – Macroexpansion
NORMAL	OM – Term normalization and typechecking
ERASE	OM – Delete information about types
COMPACT	OM – Term Compactification
EXTRACT	OM – Extract Erlang Code

4.2 BNF

```
<> ::= #option
[] ::= #list
 I ::= #identifier
 U ::= * < #number >
 O ::= I | ( O ) |
       U | O → O | O O
         | λ ( I : O ) → O
         | ∀ ( I : O ) → O
 L ::= I | L I
 A ::= O | A → A | ( L : O )
 F ::=  | F ( I : O ) | ()
 E ::= O | E data L : A := F
         | E record L : A < extend F > := F
         | E let F in E
         | E case E [ | I O → E ]
         | E receive E [ | I O → E ]
         | E spawn E raise L := E
         | E send E to E
```

4.3 Inductive Types

There is two types of recursion: one is least fixed point (as $F_A\ X = 1 + A \times X$ or $F_A\ X = A + X \times X$), in other words the recursion with a base (terminated with a bounded value), lists are trees are examples of such recursive structures (so we call induction recursive sums); and the second is greatest fixed point or recursion withour base (as $F_A\ X = A \times X$) — such kind of recursion on infinite lists (codata, streams, coinductive types) we can call recursive products.

Natural Numbers: $\mu\ X \to 1 + X$
List A: $\mu\ X \to 1 + A \times X$
Lambda calculus: $\mu\ X \to 1 + X \times X + X$
Stream: $\nu\ X \to A \times X$
Potentialy Infinite List A: $\nu\ X \to 1 + A \times X$
Finite Tree: $\mu\ X \to \mu\ Y \to 1 + X \times Y = \mu\ X = List\ X$

As we know there are several ways to appear for variable in recursive algebraic type. Least fixpoint are known as an recursive expressions that have a base of recursion Both recursive and corecursive datatypes could be encoded using Boem-Berarducci encoding as an non-recursive definitions of folds that include in indentity signature all the constructor components of (co)inductive type.

4.4 Lists

The data type of lists over a given set A can be represented as the initial algebra $(\mu L_A, in)$ of the functor $L_A(X) = 1 + (A \times X)$. Denote $\mu L_A = List(A)$. The constructor functions $nil : 1 \to List(A)$ and $cons : A \times List(A) \to List(A)$ are defined by $nil = in \circ inl$ and $cons = in \circ inr$, so $in = [nil, cons]$. Given any two functions $c : 1 \to C$ and $h : A \times C \to C$, the catamorphism $f = (\![c, h]\!) : List(A) \to C$ is the unique solution of the equation system:

$$\begin{cases} f \circ nil = c \\ f \circ cons = h \circ (id \times f) \end{cases}$$

where $f = foldr(c, h)$. Having this the initial algebra is presented with functor $\mu(1 + A \times X)$ and morphisms sum $[1 \to List(A), A \times List(A) \to List(A)]$ as catamorphism. Using this encdoding the base library of List will have following form:

$$\begin{cases} foldr = (\![f \circ nil, h]\!), f \circ cons = h \circ (id \times f) \\ len = (\![zero, \lambda\ a\ n \to succ\ n]\!) \\ (++) = \lambda\ xs\ ys \to (\![\lambda(x) \to ys, cons]\!)(xs) \\ map = \lambda\ f \to (\![nil, cons \circ (f \times id)]\!) \end{cases}$$

```
data list: (A: *) → * :=
     (nil: list A)
     (cons: A → list A → list A)
```

$$\begin{cases} list = \lambda\ ctor \to \lambda\ cons \to \lambda\ nil \to ctor \\ cons = \lambda\ x\ \to \lambda\ xs \to \lambda\ list \to \lambda\ cons \to\ \lambda\ nil \to cons\ x\ (xs\ list\ cons\ nil) \\ nil = \lambda\ list \to \lambda\ cons \to \lambda\ nil \to nil \end{cases}$$

```
record lists: (A B: *) :=
     (len: list A → integer)
     ((++): list A → list A → list A)
     (map: (A → B) → (list A → list B))
     (filter: (A → bool) → (list A → list A))
```

$$\begin{cases} len = foldr\ (\lambda\ x\ n \to succ\ n)\ 0 \\ (++) = \lambda\ ys \to foldr\ cons\ ys \\ map = \lambda\ f \to foldr\ (\lambda x\ xs \to cons\ (f\ x)\ xs)\ nil \\ filter = \lambda\ p \to foldr\ (\lambda x\ xs \to if\ p\ x\ then\ cons\ x\ xs\ else\ xs)\ nil \\ foldl = \lambda\ f\ v\ xs = foldr\ (\lambda\ xg \to\ (\lambda \to g\ (f\ a\ x)))\ id\ xs\ v \end{cases}$$

4.5 Normal Forms

Lists/Map

```
λ (a: ⋆) → λ (b: ⋆) → λ (f: a → b) → λ (xs: ∀ (List: ⋆) →
∀ (Cons: ∀ (head: a) → ∀ (tail: List) → List) → ∀ (Nil: List)
→ List) → xs (∀ (List: ⋆) → ∀ (Cons: ∀ (head: b) → ∀ (tail: List)
→ List) → ∀ (Nil: List) → List) (λ (head: a) → λ (tail: ∀ (List:
⋆) → ∀ (Cons: ∀ (head: b) → ∀ (tail: List) → List) → ∀ (Nil:
List) → List) → λ (List: ⋆) → λ (Cons: ∀ (head: b) → ∀ (tail:
List) → List) → λ (Nil: List) → Cons (f head) (tail List Cons Nil))
(λ (List: ⋆) → λ (Cons: ∀ (head: b) → ∀ (tail: List) → List)
→ λ (Nil: List) → Nil)
```

4.6 Prelude Base Library

```
  data Nat: Type :=
       (Zero: Unit → Nat)
       (Succ: Nat → Nat)

  data List (A: Type) : Type :=
       (Nil: Unit → List A)
       (Cons: A → List A → List A)

record list: Type :=
       (len: List A → integer)
       ((++): List A → List A → List A)
       (map: (A,B: Type) (A → B) → (List A → List B))
       (filter: (A → bool) → (List A → List A))

record String: List Nat := List.Nil

  data IO: Type :=
       (getLine: (String → IO) → IO)
       (putLint: String → IO)
       (pure: () → IO)

record IO: Type :=
       (data: String)
       ([>>=]: ...)

record Morte: Type :=
       (recursive: IO.replicateM Nat.Five
                   (IO.[>>=] IO.data Unit IO.getLine IO.putLine))
```

References

Category Theory

[1] S.MacLane *Categories for the Working Mathematician* 1972

[2] W.Lawvere *Conceptual Mathematics* 1997

[3] P.Curien *Category theory: a programming language-oriented introduction* 2008

Pure Type Systems

[4] P.Martin-Löf *Intuitionistic Type Theory* 1984

[5] T.Coquand *The Calculus of Constructions.* 1988

[6] E.Meijer *Henk: a typed intermediate language* 1997

[7] H.Barendregt *Lambda Calculus With Types* 2010

Inductive Type Systems

[8] F.Pfenning *Inductively defined types in the Calculus of Constructions* 1989

[9] P.Wadler *Recursive types for free* 1990

[10] N.Gambino *Wellfounded Trees and Dependent Polynomial Functors* 1995

[11] P.Dybjer *Inductive Famalies* 1997

[12] B.Jacobs *(Co)Algebras) and (Co)Induction* 1997

[13] V.Vene *Categorical programming with (co)inductive types* 2000

[14] H.Geuvers *Dependent (Co)Inductive Types are Fibrational Dialgebras* 2015

Homotopy Type Systems

[15] T.Streicher *A groupoid model refutes uniqueness of identity proofs* 1994

[16] T.Streicher *The Groupoid Interpretation of Type Theory* 1996

[17] B.Jacobs *Categorical Logic and Type Theory* 1999

[18] S.Awodey *Homotopy Type Theory and Univalent Foundations* 2013

[19] S.Huber *A Cubical Type Theory* 2015

[20] A.Joyal *What is an elementary higher topos* 2014

[21] A.Mortberg *Cubical Type Theory: a constructive univalence axiom* 2017