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

Categorical semantic of general purpose functional language that compiles to pure type system minimal core

Technical Article

Maxim Sokhatsky, Synrc Research Center

December 2015

Содержание

1 Introduction

1.1 Verifiable Functional Compiler

This article describes the frontend language **Exe** along with its categorical semantic that compiles to a pure type system minimal core **Om** — the non-recursive subset of dependent type theory from which you can extract: LLVM code, untyped lambda code with erased type information, or even *System* F_ω programs. As a top-level task we expect to produce lean verifiable functional compiler pipeline, but as an milestone task we just produce concise working prototype in Erlang language.

1.2 General Purpose Functional Language Exe

General purpose function language with functors, lambdas on types, recursive algebraic types, higher order functions and embedded process calculus with corecursion, free monad for effects encoding. This language will be called Exe and dedicated to be high level general purpose functional programming language frontend to small core of dependent type system without recursion called Om. This language indended to be useful enough to encode KVS, N2O and BPE applications.

1.3 Intermediate Language Om

An intermediate Om language is base on Henk [1] languages described first by Erik Meyer 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 lamda terms. It is based only on π, λ and *apply* constructions, one axiom and four deduction rules. The design of Exe language resemble Henk and Morte both design and implementation. This language indended to be small, conside, easy provable and clean and produce verifiable peace of code that can be distributed over the networks and compiled at target with safe linkage.

1.4 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.

2 General Purpose Language

2.1 Category Theory, Programs and Functions

Category theory is widely used as an instrument for mathematicians for software analisys. Category theory could be treated as an abstract algebra of functions. Let's define an Category formally: **Category** consists of two lists: the one is morphisms (arrows) and the second is objects (domains and codomains of arrows) along with assoociative operation of composition and unit morphism that exists for all objects in category.

The formation axoims of objects and arrows are not given here and autopostulating yet. Formation axoims will be introduced during exponential definition. Objects A and B of an arrow $f : A \to B$ are called **domain** and **codomain** respectively.

Intro axioms – associativity of composition and left/right unit arrow compisitions show that categories are actually typed monoids, which consist of morphisms and operation of composition. There are many language to show the semantic of categories such as commutative diagrams and string diagrams however here we define here in proof-theoretic manner:

$$\frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash g : B \to C}{\Gamma \vdash g \circ f : A \to C}$$

$$\frac{}{\Gamma \vdash id_A : A \to A}$$

$$\frac{\Gamma \vdash f : B \to A \qquad \Gamma \vdash g : C \to B \qquad \Gamma \vdash h : D \to C}{\Gamma \vdash (f \circ g) \circ h = f \circ (g \circ h) : D \to A}$$

$$\frac{\Gamma \vdash f : A \to B}{\Gamma \vdash f \circ id_A = f : A \to B}$$

$$\frac{\Gamma \vdash f : A \to B}{\Gamma \vdash id_B \circ f = f : A \to B}$$

Composition shows an ability to connect the area of values of the previous evaluation (codoman) and are of defenition the next evaluation (domain). Composition is fundamental property of morphisms that allows us to chain evaluations.

1. $A : *$
2. $A : * \ , \ B : * \implies f : A \to B$
3. $f : B \to C \ , \ g : A \to B \implies f \circ g : A \to C$
4. $(f \circ g) \circ h = f \circ (g \circ h)$
5. $A \implies id : A \to A$
6. $f \circ id = f$
7. $id \circ f = f$

2.2 Algebraic Types and Cartesian Categories

After composition operation of construction of new objects with morphisms we introduce operation of construction cartesian product of two objects A and B of a given category along with morphism product $< f, g >$ with a common domain, that is needed for full definition of cartesian product of $A \times B$.

This is an internal language of cartesian category, in which for all two selected objects there is an object of cartesian product (sum) of two objects along with its $\bot$ terminal (or $\top$ coterminal) type. Exe languages is always equiped with product and sum types.

Product has two eliminators π with an common domain, which are also called projections of an product. The sum has eliminators i with an common codomain. Note that eliminators π and i are isomorphic, that is $\pi \circ \sigma = \sigma \circ \pi = id$.

$$\frac{\Gamma\ x : A \times B}{\Gamma \vdash \pi_1\ : A \times B \to A; \Gamma \vdash \pi_2\ : A \times B \to B}$$

$$\frac{}{\Gamma \vdash\ \top}$$

$$\frac{\Gamma \vdash\ a : A \qquad \Gamma \vdash\ b : B}{\Gamma \vdash\ (a, b) : A \times B}$$

$$\frac{\Gamma \vdash\ a : A \qquad \Gamma \vdash\ b : B}{\Gamma \vdash a \mid b : A \otimes B}$$

$$\frac{}{\Gamma \vdash\ \bot}$$

$$\frac{\Gamma\ x : A \otimes B}{\Gamma \vdash ß_1 : A \to A \otimes B; \Gamma \vdash ß_2 : B \to A \otimes B}$$

Also we include here an axiom of morphism product which are given durin definition of product commutative diagram. This axiom is needed for applicative programming in categorical abstract machine.

$$\frac{\Gamma \vdash\ f : A \to B \qquad \Gamma \vdash\ g : A \to C \qquad \Gamma \vdash\ B \times C}{\Gamma \vdash\ \langle f, g \rangle : A \to B \times C}$$

$$\pi_1 \circ \langle f, g \rangle = f$$
$$\pi_2 \circ \langle f, g \rangle = g$$
$$\langle f \circ \pi_1, f \circ \pi_2 \rangle = f$$
$$\langle f, g \rangle \circ h = \langle f \circ h, g \circ h \rangle$$
$$\langle \pi_1, \pi_2 \rangle = id$$

2.3 Exponential, λ-calculus and Cartesian Closed Categories

Being an internal language of cartesian closed category, lambda calculus except variables and constants provides two operations of abstraction and applications which defines complete evaluation language with higher order functions, recursion and corecursion, etc.

To explain functions from the categorical point of vew we need to define categorica exponential $f : A^B$, which are analogue to functions $f : A \to B$. As we already defined the products and terminals we could define an exponentials with three axioms of function construction, one eliminator of application with apply a function to its argument and axiom of currying the function of two arguments to function of one argument.

$$\frac{\Gamma x : A \vdash M : B}{\Gamma \vdash \lambda\ x\ .\ M : A \to B}$$

$$\frac{\Gamma\ f : A \to B \qquad \Gamma\ a : A}{\Gamma \vdash apply\ f\ a\ : (A \to B) \times A \to B}$$

$$\frac{\Gamma \vdash f : A \times B \to C}{\Gamma \vdash curry\ f : A \to (B \to C)}$$

$$apply \circ \langle (curry\ f) \circ \pi_1, \pi_2 \rangle = f$$
$$curry\ apply \circ \langle g \circ \pi_1, \pi_2 \rangle) = g$$
$$apply \circ \langle curry\ f, g \rangle = f \circ \langle id, g \rangle$$
$$(curry\ f) \circ g = curry\ (f \circ \langle g \circ \pi_1, \pi_2 \rangle)$$
$$curry\ apply = id$$

Об'єкти : $\bot \mid\ \to\ \mid \times$
Морфізми : $id \mid f \circ g \mid \langle f, g \rangle \mid apply \mid \lambda \mid curry$

2.4 Functors

2.5 Recursive Types

As was shown by Wadler in Recursive Types for Free! we could deal with recusrive equations having three axioms: one $fix : (A \to A) \to A$ fixedpoint axiom, and axioms $in : F\ T \to T$ and $out : T \to F\ T$ of recursion direction. We need to define fixed point as axiom because we can't produce recursive axioms. This axioms also needs functor axiom defined earlier.

There is two types of recursion: one is least fixed point (as $rec\ A = 1+(A, rec)$ or $rec\ A = A+(rec, rec)$), 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 $rec\ A = (A, A)$) — such kind of recursion on infinite lists (codata, streams, coinductive types) we can call recursive products.

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 inductive or coinductive type.

$$\frac{\Gamma \vdash M : F\ (\mu\ F)}{\Gamma \vdash in_{\mu F}\ M : \mu\ F} \qquad \frac{\Gamma \vdash M : \mu F}{\Gamma \vdash out_{\mu F}\ M : F\ (\mu\ F)}$$

$$\frac{\Gamma \vdash M : A \to A}{\Gamma \vdash fix\ M : A}$$

Not that there is a class of recursive types such when fixpoint is defined in codomain of morphism, such terms could not be normalized.

2.6 Process Calculus

Теорія π-числення процесів Роберта Мілнера є основним формалізмом обчислювальної теорії розподілених систем та її імплементації. З часів виникнення CSP числення розробленого Хоаром, Мілнеру вдалося значно розширити та адаптувати теорію до сучасних телекомунікаційних вимог, як наприклад хендовери в мобільних мережах. Основні теорми в моделі π-числення стосуються непротиречивості та неблокованості у синхронному виконанні мобільних процесів. Так як сучасний Web можно розглядати як телекомунікаційну систему, тому у розробці додатків можна покладатися у тому числі і на такі моделі як π-числення.

Process and Protocols

Також ми анонсуємо процес як фундаменльний тип даних, подібний до функції але який здатний тримати певний стан у вигляді типа коротежа та є морфізмом-одиницею типу свого стану.

$$\frac{\Gamma \vdash E, \Sigma, X \qquad \Gamma \vdash action : \Sigma \times X \to \Sigma \times X}{\Gamma \vdash spawn\ action : \pi_\Sigma}$$

$$\frac{\Gamma \vdash pid : \pi_\Sigma \qquad \Gamma \vdash msg : \Sigma}{\Gamma \vdash join\ msg\ pid : \Sigma \times \pi_\Sigma \xrightarrow{\bullet} \Sigma;\ \Gamma \vdash send\ msg\ pid : \Sigma \times \pi_\Sigma \to \Sigma}$$

$$\frac{\Gamma \vdash L : A + B, R : X + Y \qquad \Gamma \vdash M : A \to X, N : B \to Y}{\Gamma \vdash receive\ L\ M\ N : L \xrightarrow{\bullet} R}$$

Process Algebra

Алгебра процесів визначає базові операції мультиплексування двох чи декількох протоколів в рамках одного процесу (добуток), а також паралельного та повністю ізольованого запуску включно зі стеком та областю памяті (сума) на віртуальній машині.

$$\begin{aligned} \oplus &: \pi \parallel \pi \\ \otimes &: \pi \mid \pi \end{aligned}$$

2.7 Intuitionistic Type Theory

Розбудовуючи певний фреймворк чи систему конструктивними методами так чи інакше доведеться зробити певний вибір у мові та способі кодування. Так при розробці теорії абстрактної алгебри в Coq були використані поліморфні індуктивні структури [**?**]. Однак Agda та Idris використовують для побудови алгебраїчної теорії типи класів, а у Idris взагалі відсутні поліморфірні індуктивні структури та коіндуктивні структури. В Lean теж відсутні коіндуктивні структури проте повністю реалізована теорія HoTT на нерекурсивних поліморфних структурах що обєднує основні чотири класи математичних теорій: логіка, топологія, теорія множин, теорія типів. Як було показано Стефаном Касом [21], одна з стратегій імплементації типів класів — це використання поліморфних структур. Хоча в Lean також підтримуються типи класів нами була вибрана стратегія імплементації нашої теорії з використаннями нерекурсивних індуктивних структур, що дозволить нам оперувати з персистентними структурами на низькому рівні. Крім того такий спосіб кодування ієрархій повністю відповідає семантиці Erlang, де немає типів класів, а дані передаються запаковані в кортежі-структури.

2.8 Logic and Quantification

Далі йдуть квантори $\forall$ та $\exists$ які теж виражаються як конструкції типів:

$$\frac{\Gamma\ x : A \vdash B \qquad \Gamma\ \vdash A}{\Gamma\ \vdash \Pi(x : A)B} \qquad \frac{\Gamma\ \vdash a : A \qquad \Gamma\ x : A \vdash B \qquad \Gamma\ b : B(x = a)}{\Gamma\ \vdash (a, b) : \Pi(x : A)B}$$

$$\frac{\Gamma\ x : A \vdash B \qquad \Gamma\ \vdash A}{\Gamma\ \vdash \Sigma(x : A)B} \qquad \frac{\Gamma\ \vdash a : A \qquad \Gamma\ x : A \vdash B \qquad \Gamma\ b : B(x = a)}{\Gamma\ \vdash (a, b) : \Sigma(x : A)B}$$

$$\frac{\Gamma\ \vdash x : A \qquad \Gamma\ \vdash x' : A}{\Gamma\ \vdash Id_A(x, x')}$$

рефлексивність	:	$Id_A(a, a)$
підстановка	:	$Id_A(a, a') \to B(x = a) \to B(x = a')$
симетричність	:	$Id_A(a, b) \to Id_A(b, a)$
транзитивність	:	$Id_A(a, b) \to Id_A(b, c) \to Id_A(a, c)$
конгруентність	:	$(f : A \to B) \to Id_A(x, x') \to Id_B(f(x), f(x'))$

3 Intermediate Language

3.1 Om Language Definition

Om resemble both the Henk theory of pure type system and λC calculus of constructions and Morte Core Specification

```
EXPR :=                          EXPR                EXPR
      | "\"    "(" LABEL ":" EXPR ")"  "arrow" EXPR
      | "\/"   "(" LABEL ":" EXPR ")"  "arrow" EXPR
      |                          EXPR      "arrow" EXPR
      |             LABEL
      | "*"
      | "[]"
      |        "("              EXPR ")"
```

During forward pass we stack applications (except typevars), then on reaching close paren ")"we perform backward pass and stack arrows, until neaarest unstacked open paren "("appeared (then we just return control to the forward pass).

We need to preserve applies to typevars as they should be processes lately on rewind pass, so we have just typevars bypassing rule. On the rewind pass we stack lambdas by matching arrow/apply signatures where typevar(x) is an introduction of variable "x"to the Gamma context.

$$apply : (A \to B) \times A \to B$$
$$lambda : arrow\ (app\ (typevar\ x)\ ,\ A)\ B$$

3.2 Abstract Syntax Tree

$$\begin{array}{rcl} E & := & K \\ & | & x \\ & | & EE \\ & | & \lambda(x : E) \to E \\ & | & \Pi(x : E) \to E \end{array}$$

3.3 Encoding Basic Type Constructors

We usually define inductive type in λC theorem provers by specifying types as sum of lambda type constructors with return type as a definition type

```
type List (A: *) : * = Nil: () -> List A
                     | Cons: A -> List A -> List A
```

However we can omit return type from encoding but specify parameters of constructors as named lambdas in Om tree:

```
type Product (A: *) (B: *) : * =          Make (a: A)       (b: B)
type Sum (A: *) (B: *)     : * =          Sum  (a: A)       (b: B)
type Option  (A: *)        : * = None  | Some (value: A)
type List    (A: *)        : * = Nil   | Cons (head: A)    (tail: List)
type Unit                  : * = Make
```

3.4 Samples of Exe programs

```
$ cat List/(++)
'++'[a:*](as1: #List a, as2: #List a): #List a ->
  as1 List Cons (as2 List Cons Nil)
```

Список литературы

[1] E.Meijer, S. Peyton Jones *Henk: a typed intermediate language* 1984

[2] P.Wadler *Recursive types for free!* 2014

[3] P.Wadler *Monads for functional programming*

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

[5] S.Awodey *Category Theory* 2010

[6] A.Baer *Programming with Algebraic Effects and Handlers* 2012

[7] PL. Curien *Category theory: a programming language-oriented introduction* 2008

[8] William Lawvere. *Conceptual Mathematics.* 1997.

[9] Сандерс Мак Лейн. *Категории для работающего математика.* 2004.

[10] И.Бакур, А.Деляну. *Введение в теорию категори и функторов.* 1972.

[11] Robin Milner. *A Calculus of Communicating Systems.* 1986.

[12] Robin Milner. *Communicating and Mobile Systems: The π-calculus.* 1999.

[13] Robin Milner. *The Polyadic π-Calculus: A Tutorial.* 1993.

[14] T.Coquand, G.Huet *The Calculus of Constructions.* 1988

[15] A.Chlipala *Certified Programming with Dependent Types* 2015

[16] E.Brady *Programming in IDRIS: A Tutorial* 2015

[17] C.McBride, R.Patterson *Applicative programming with effects* 2002

[18] S.Andjelkovic *A family of universes for generic programming* 2011

[19] W.Swierstra *Data types a‘ la carte* 2011

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

[21] S.Kaes *http://tuprints.ulb.tu-darmstadt.de/epda/000544/diss.pdf* 2005