Universidad Politécnica de Madrid
Facultad de Informática

Libera Università di Bolzano
Facoltà di Scienze e Tecnologie Informatiche

European Master's Program in Computational Logic

Master's Thesis

A Multi-thread Implementation of Functional Logic Programming

Student: Daniel Guimarães Santos
Supervisor: Julio Mariño Carballo

Madrid, October 2008

A Multi-thread Implementation of Functional Logic Programming

Daniel Guimarães Santos

October 2008

Acknowledgements

My sincere thanks are to Julio Mariño, Emílio Gallego and Lars-Åke Fredlund for the support and guidance along this work. Also to the EMCL commission, IMDEA Software and Babel Group for the support. Thanks to God. Thanks to my beautiful brothers Letícia, Felipe and Eduardo, and to my great friends Hilário, Adilson and Márcio for all the friendship and happy unforgettable moments. My deepest gratitude goes to my parents, William Hassen and Nancy Maria, for guiding me through life, supporting my choices, enduring the distance and loving me unconditionally.

Abstract

Functional Logic Programming has being growing in interest and research. Attempting to be a multi-paradigm declarative language, Curry is the most important representant of the trial of amalgamation of functional and logic programming. Aware of its importance, we propose to add multi-threads into the execution of a Curry program taking advantage of the features of Erlang. Erlang, an important and solid language for process communication and concurency, will complete this attempt for transforming Curry programs as lazy and Erlang-readable functions, and executing them as a multi-thread search, relying on the solid process manipulation that Erlang provides.

Contents

Chapter 1

Introduction

On imperative programming, one can define *how* to obtain the solution over a number of steps. Conversely, in declarative programming, one can define *what* are the problem's properties and lead to the expected solutions. This is possible due to the properties of declarative programming languages on providing higher and abstract level of programming, yielding more reliable and maintainable programs.

Since the beginning of the last decade, many proposals have been made for merging the most important declarative programming paradigms [Han94]. Due to the different characteristics and features between functional programming and (constraint) logic programming, such integrated language comes to amalgamate those paradigms. From many proposals, Curry stand out due to its implementation, dissemination and tools. Its initial semantics [Han] is largely base on the work of Michael Hanus [HIA97, Han97] and report the operational semantic, narrowing steps and definitional trees.

A motivation for this work is a from Tolmach and Antoy. Conversely to incomplete implementations based on back-tracking and breadth-first search, they provide a complete implementation of Curry, from a deterministic big-step operational semantics, including non-determinism, narrowing, and residuation[TA03].

A model for concurrency in Curry was proposed by [Han02], but not totally implemented. Its prototypical implementation is based on Sicstus-Prolog, using a socket library for implement the port communication.

Moreover, despite of preliminary works on concurrent extensions and libraries for Curry, the possibility of implementing definitional trees in parallel motivate us for base research on concurrency integrated on Curry.

1.1 The Problem

Trying to amalgamate Curry and concurrency, on a parallel execution of Curry expressions, our goal is to implement a concurrent model for evaluating and executing Curry within processes and messages communications.

The goal of this work is transforming a Curry program to a semantically equivalent Erlang source code. As known, Curry is a functional logic language, and then combine logic and functional features that must be preserved by this translation. The main challenge is emulate the execution of Curry in Erlang without loses and preserving its characteristics while executing on a multi-thread. It is, from a Curry program, our proposal must achieve the same solutions set as a Curry evaluation: they must be semantically equivalent.

The purposed approach is translate Curry to FlatCurry and then translate it to Erlang in order that functions can be evaluated on demand, and not strictly as Erlang does (Figure 1.1). Once having the Curry functions translated in Erlang, the evaluation process is defined in a way of keeping the semantic of Curry, using narrowing and reducing expressions as Curry.

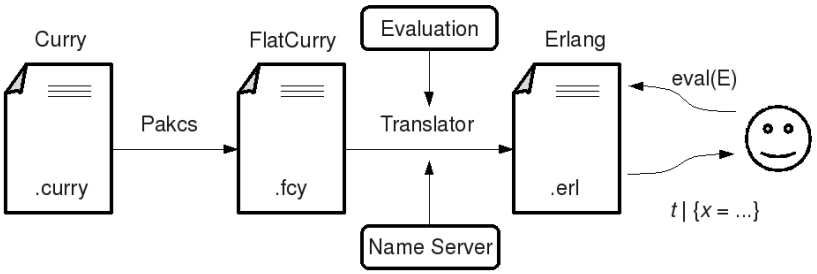

Figure 1.1: The general scheme of the multi-thread implementation.

Having introduced some paradigms and languages, the connection between them must be explained: the purpose of this work is to use the features of Erlang, e.g. message passing and simple process creation, for parallelize the search of solutions of a Curry program. It is, when Curry have more than one search branch, each one must become a new process and run in parallel[1].

On this work, we try to experiment something new. Instead of implement communications library, we will make use of a declarative language already know by its multi-thread facilities: Erlang. Providing a low cost and simple, but powerful, communication, Erlang will be our target source code. Instead of use the

[1]For this initial attempt, the Peano natural numbers and Booleans will be covered, sufficient to have a taste on how the concurrency will perform.

features to deal with free variables in Prolog, we will try to emulate the presence of unbound variables in Erlang.

1.2 Organization

First we must introduce some technical aspects of the languages we are going to work with on the Chapter 2: Preliminaries. Then introduce the general idea of the execution model on Chapter 3, for a better understanding on the translation on the Chapter 4. Then, on the Chapter 5, the execution model will be defined in order to keep the evaluation of expressions as in a Functional Logic Language.

Further, experimental results and the conclusion are shown on Chapters 6 and 7 respectively. And, for additional support, complete example files are attached in the end of this book (Appendices A, B and C) for consulting and comparing Curry and Erlang code.

Chapter 2

Preliminaries

This chapter introduces the declarative languages used in this work: Curry and Erlnag. The former part is devoted to describe how Curry was developed by the amalgamation of both functional and logic languages, becoming a functional logic language. The main differences on features of functional and logic languages are exposed and how their characteristics are kept in Curry.

Then FlatCurry language, a flat full representation of a Curry program, is introduce as a mechanism for reaching definitional trees. They describe and provide all information need form a Curry program and represent Curry functions as decision trees.

Finally, Erlang is introduced as the target language. Its features, e.g. concurrent and multi-thread execution, will facilitate the final execution of Curry in a multi-thread way, interconnecting the goals proposed.

2.1 Functional Logic Programming

During the last decade, there was many proposals for merging the most important declarative programming paradigms: functional programming and (constraint) logic programing [Han94]. The main characteristics of there paradigms differ on their features and formalisms [Han07]:

- **Logic Languages:** lambda calculus, functions, directed equations and reduction of nested expressions;
- **Functional Languages:** predicate logic, predicates, logical variable, definite clauses and goal solving by resolution;
- **Constraint Languages:** constraint structures, constraints and specific constraint solvers.

Since all these features seems to be useful for application programming and declarative languages lies on common grounds, the proposal of a functional logic language comes to merge them and create a multi-paradigm declarative language.

Curry is the *lingua franca* proposed to be the common platform for integrated functional logic languages and its design and development have been improved since its creation on the last decade. In order to understand the development of Curry, we must before understand the features of both functional and logic language listed in the next subsections. Then Curry is introduced, followed by definitional trees and FlatCurry.

2.1.1 Functional Programming

Functional languages comes from the syntactic sugar over the λ notation invented by Alonzo church on the beginning of the last century. The λ calculus has a highly expressive and minimalist notation by reducing logic to functions and functions to its basic components.

From λ notation

Here we have the abstract syntax for the λ notation:

$$\begin{aligned} (\text{Variables})\ v &::= x \mid y \mid z \mid x_1 \mid x_2 \mid \ldots \\ (\text{Terms})\ e &::= v \mid \lambda v.e \mid e_1 e_2 \end{aligned}$$

To functional languages

Adding syntactic sugar for the λ notation, we can approximate it to the actual programming style. In order to be practical, a set of constants for data and basic operations is needed, yielding the following valid abstractions:

$$\lambda x.x,\ \lambda y.y+y,\ \lambda x.2*x,\ \lambda x.\lambda y.x \wedge \neg y \ldots$$

User defined data can be introduced in the form of data constructors of a given arity. Below some examples using the Haskell syntax as reference:

```
data Bool = True | False
data Nat = Zero | Succ Nat
data Btree a = Leaf | Branch a Btree Btree
```

Those define booleans, *Peano naturals* and a constructor for a binary tree.

On the transformation of λ calculus to functional language, named function constants comes for facilitate recursion[1] and for a clearer syntax:

[1] In fact recursion and conditionals can be encoded in λ calculus

```
add = \n -> \m ->
      if (n==Zero) then m else Succ (add (Pred n) m)
```

Where `Pred` is the (partial) inverse function for the constructor `Succ` of the Peano naturals. [2]

An important step is the introduction of function definitions by means of equations and pattern matching. Thus the last `add` definition can be rewritten as:

```
add Zero m = m
add (Succ n) m = Succ (add n m)
```

With this new syntax, patterns can be more complex and avoid the use of conditional cases inside the function definitions. It is possible through a mechanism called *pattern matching* that access the data and exempt the use of data constructors.

Another important features of functional languages are the higher-order functions, expressive data types, modularity and demand-driven evaluation.

The evaluation on functional languages may be *lazy* or *eager*. On a *eager* evaluation, all the arguments of a function will be evaluated before computing the function. A *lazy* will compute only the necessary arguments for evaluating the function, so some of them may not be computed.

2.1.2 Logic Programming

Also in declarative programming, logic programing combines first-order logic and a deduction algorithm in order to infer facts and properties over the queries. As first-order logic, it is represented as Horn clauses that describes the problem as relations. As the deduction algorithm, a refutation method is used: resolution.

Horn clauses have the following form:

$$H \leftarrow B_1 \wedge \ldots \wedge B_n$$

Here H stands for the predicate head and B the body, composed by a conjunction of predicates B_i. When a clause has no body, it is called fact. A typical example is show, expressed with Prolog syntax:

```
father(terach, abraham).
father(abraham, isaac).

ancestor(Ancestor, Descendant) :-
```

[2] $\backslash v -> e$ is Haskell syntax for $\lambda v.e$

```
    father(Ancestor, Descendant).
ancestor(Ancestor, Descendant) :-
    father(Ancestor, Someone),
    father(Someone, Descendant).
```

As can be seeing, `father/2` is a fact and `Ancestor/2` is a rule, expressing the ancestor relationship. The symbol (`:-`) stands for $\leftarrow$ and the comma for the conjunction. Thus, standing the relations and facts, queries can be done in order to consult the satisfiability of some relation.

This consult is done by the resolution mechanism, i.e. if the negation of the query can be refutable from the set of facts and rules, then it can be deducible from program's clauses. Otherwise, if there is no refutation of the negation of the query, then it is not deducible from the program. In fact, that does not mean that the query is inconsistent with the program, but only that it can not be deducible from the given rules and facts.

An example of queries in Prolog can be:

```
?- ancestor(terach, isaac).
yes

?- ancestor(A, isaac).
A = terach;
A = abraham;
no

?- ancestor(isaac, abraham).
no
```

The second and third queries show cases of failure to find a the refutation of the respective queries. In the second one, an exhaustive search and variable assignment were done and the queries end up with no more solutions. In the third one, no solution was found.

Important features of logic programming are the assignment of values to free variables, the backtracking search in the resolution mechanism, nondeterministic search and unification.

2.1.3 Curry

Many attempts were done trying to merge functional and logic programming. For that, one thought on bringing the features of a functional language to a logic one or, conversely, implement the logic language features on the functional one. The

latter approach was took as more natural and expressive and, as we can see, it is the case of Curry, the first functional logic language designed[HKMN95]. From here, functional logic programming will be referred as FLP.

Back to historical background, Curry was derived of years of research on other FLP languages. It was first designed by Michael Hanus, from the University of Aachen, Germany, in 1996 and it still in design and development. Curry's reports are the main reference to the Curry language, and the document that must be referred when implementing it. Many different proposals have been developed during the last two decades. The main implementations are listed below.

- Sloth: Translates Curry to Ciao Prolog, exploiting the functionalities of constraint programming provided by it. Sloth have been developed at the Univesidad Politécnica de Madrid [GM05].
- MCC: Münster Curry Compiler. A mature native code compiler for Curry that includes many features as a declarative debugger of wrong answers and extensions for the language, e.g. disequality constraints, recursive pattern bindings and IO exceptions. The MCC needs no other software than the Gnu C compiler to compile and run[3].
- Curry2Prolog have been developed by the University of Kiel, Germany, and includes many libraries for implementing user interfaces, web programming and application programming, e.g. XML processing, meta-programming, sets. There is also a interactive WWW interface available[4].
- PAKCS: the Portland Aachen Kiel Curry System. Its a joint development by Portland State University, Aachen University of Technology and University of Kiel. It has a common front-end and three different back-ends: compilers into Java, into Prolog and a interpreter in Prolog.

Due to its common front-end, PAKCS was chosen to be the start point of this work. Soon will be shown how it works and how it is used here. From now, it must be shown how Curry designs a FLP, it abstract semantics and a few examples.

CoreCurry

Curry uses both *narrowing* and *residuation* for solving goals. Narrowing applies a substitution of a free variable in order to reduce a expression e. For instance, in

```
f 0 = False
f 1 = True
```

[3] http://danae.uni-muenster.de/~lux/curry/

[4] http://www-ps.informatik.uni-kiel.de/~mh/pakcs/curryinput_c2p.cgi

if we have the goal `f x where x free`, `x` must be *narrow* to `0` or `1` in order to reduce to a solution. It's represented as a set of substitutions or assignments of values to variables: $e \Downarrow \{e'_1|\sigma_1 \ldots e'_n|\sigma_n\}$. In that example, it must be: $(fx) \Downarrow \{False|[x \mapsto 0],\ True|[x \mapsto 1]\}$. In the other hand, *residuation* suspend the computation of a goal until the variable in bound. Functions case that only allow bound values to be computed are called rigid functions. This work is restricted to a CoreCurry, i.e. rigid functions, and then residuation, will not be dealt at this moment. CoreCurry is restricted to flexible case functions that may assignate a value to a unbound variable. Flexible case functions are essential for the multi-thread computation scheme, since permits to a variable assigns to different values.

A very explicative example of narrowing is the less-or-equal predicate(`'<='`) over Peano natural numbers [HP03]:

```
Zero   <= x      -> True
Succ x <= Zero   -> False
Succ x <= Succ y -> x <= y
```

For the goal `(Succ x) <= y where x, y free`, the first narrowing step assigns $\{y \mapsto (Succ y_1)\}$, and applies to the third rule. Then the second narrowing step instantiate `x` to `Zero` and apply the first rule:

$$leq\ (Succ x)\ y \ \rightsquigarrow_{\{y \mapsto (Succ\ y_1)\}}\ leq\ x\ y_1 \ \rightsquigarrow_{\{x \mapsto Zero\}}\ True$$

This narrow symbol $\rightsquigarrow_n$ indicates the reduction of an expression by applying the narrowing n. This demanded substitution, named *Needed narrowing* [AH94], will evaluate only the term needed to compute some result. For instance, evaluating $leqt_1t_2$, t_1 must always be evaluated to some head normal form since the first argument is not a variable, and t_1 will me narrowed to `Zero` or `(Succ x)`. However, t_2 only will be evaluated if $t_1 \neq Zero$.

Thus, the respective execution tree is:

where the underlined variables shows which of them are being narrowed.

2.1.4 The FlatCurry Language

FlatCurry is a intermediate language provided by PAKCS to be a common interface for connecting different tools to a Curry program. A Curry program can be fully represented in FlatCurry, without any lose of information.

In the core of this common language, lies the function representation in a similar way of Definitional Trees, with some renaming: Branches are called Case, and Rules are represented by different Expression types, as will be introduced further.

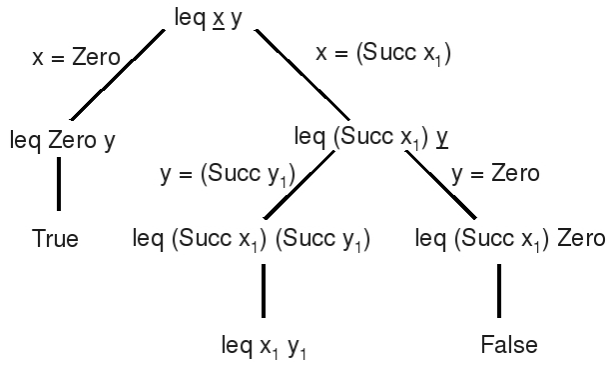

Figure 2.1: The evaluation tree of less-or-equal predicate.

The Basic Structure

In FlatCurry, all functions are defined at the top level and the pattern-matching strategy is made explicit by the use of case expressions. Basically a FlatCurry program consists of:

- A list of data type declarations (defining the data constructors used in the program);
- A list of function definitions.

FlatCurry also contains information about modules and the syntactic representation of source programs, e.g. list of infix operator declarations. Altogether, a FlatCurry program is a tuple:

```
(module,imports,types,functions,operators,translation)
```

with the following components:

- `module`: The name of the module represented by this structure.
- `imports`: The names of the modules imported by this program.
- `types`: The list of data type declarations defined in this module. Each data type declaration consists of a list of type variables as parameters (for polymorphic data structures) and a list of constructor declarations where each constructor declaration contains the name and arity of the data constructor and a list of type expressions (the argument types of this data constructor).

- `functions`: The functions defined in this module. A function declaration consists of the name and arity of the function together with a type expression specifying the function's type and a single rule (where the right-hand side can contain case expressions and disjunctions) specifying the function's meaning. External functions (i.e., primitive functions not defined in this or another module) contain instead of the rule the external name of this function.

- `operators`:The infix operators declared in this module. An operator declaration contains the fixity (or associativity) and the precedence for each operator.

- `translation`: The translation table describes the mapping of identifiers (type constructors, data constructors, defined functions) used in the source program into their corresponding internal names used in the intermediate FlatCurry program.

For a complete description of the structure of a FlatCurry program, the abstract syntax is introduced below.

Abstract Syntax for FlatCurry

The abstract syntax shown below describes the precise structure of FlatCurry programs.

prog	::= module import* type* function* operator* translation*	*(complete program module)*
module	::= ident	*(name of the module)*
import	::= module	*(name of imported module)*
type	::= *t*(typevar*, consdecl*)	*(declaration of data type "t")*
consdecl	::= *c*(arity, typeexpr*)	*(declaration of data constructor "c")*
arity	::= <natural number>	
typeexpr	::= typevar	*(type variable)*
	| typeexpr → typeexpr	*(function type)*
	| `t`(typeexpr*)	*(type constructor application)*
typevar	::= tvar(<integer>)	*(type variables have unique indices)*
function	::= f(arity, typeexpr, rule)	*(declaration of function "f")*
rule	::= lhs → expr	*(rule for user-defined function)*
	| external(<string>)	*(name of external function)*
lhs	::= var*	*(left-hand side is list of variables)*
expr	::= var	*(variable)*
	| <integer> | <float> | char(<integer>)	*(constant literal)*

	| c(expr*)	*(constructor application)*
	| f(expr*)	*(function application)*
	| case expr of pat → expr ;. . . ; pat → expr	*(rigid case expression)*
	| fcase expr of pat → expr ;. . . ; pat → expr	*(flexible case expression)*
	| or(expr, expr)	*(disjunction)*
	| partcall(f, expr*)	*(partial function/constructor application)*
	| apply(expr, expr)	*(application)*
	| constr(var*, expr)	
	(constraint with existentially quantified variables)	
	| guarded(var*, expr, expr)	*(guarded expression)*
	| choice(expr)	*(committed choice)*
	| let(binding*, expr)	*((non-recursive) let binding of variables)*
	| letrec(binding*, expr)	*(recursive let binding of variables)*
var	::= var(<integer>)	*(variables have unique indices)*
pat	::= c(var*)	*(shallow pattern with fresh variables)*
	| <integer> | <float> | char(<integer>)	*(literal constant as pattern)*
binding	::= var = expr	*(local variable binding)*
operator	::= op(fixity, precedence)	*(infix operator declaration)*
fixity	::= infix | infixl | infixr	
precedence	::= 0 | 1 | . . . | 9	
translation	::= (ident, ident)	*(translation of external to internal names)*
ident	::= <string>	

Using FlatCurry

The FlatCurry module is available within the PACKS package and the documentation of the system module can be found at the on line[5]. Examples of FlatCurry translations will be shown on the further chapter on a most complete context.

2.2 Erlang Programming Language

Erlang is a programming language designed at the Ericsson Computer Science Laboratory and released as open source in 1998. It is a general-purpose programming language which has many features more commonly associated with an operating system than with a programming language: concurrent processes, scheduling, memory management, distribution, networking, etc. The sequential subset of Erlang is a functional language, with strict evaluation, single assignment, and dynamic typing.

[5] Available at http://www.informatik.uni-kiel.de/pakcs/lib/CDOC/Flat.html

Erlang was chosen due to its characteristics (Robustness, Concurrency, Distribution, Soft real-time, Hot code upgrade etc). From those, the most important features for this work are:

- **Concurrency**: Erlang has extremely lightweight processes whose memory requirements can vary dynamically. Processes have no shared memory and communicate by asynchronous message passing. Erlang supports applications with very large numbers of concurrent processes. No requirements for concurrency are placed on the host operating system.

- **Distribution**: Erlang is designed to be run in a distributed environment. An Erlang virtual machine is called an Erlang node. A distributed Erlang system is a network of Erlang nodes (typically one per processor). An Erlang node can create parallel processes running on other nodes, which perhaps use other operating systems. Processes residing on different nodes communicate in exactly the same was as processes residing on the same node.

Concurrency is explicit in Erlang and the processes communicate using message passing. Both process and messages have a low computational cost for being generated and manipulated. Moreover, keep track of the processes and messages are simple and explicit: messages sent by a process always arrive at the same order they were sent by the other process. Below, an example of message passing and process creation is shown:

```
-module(area_server).
-export([start/1, loop/1]).

 start() ->
     spawn(area_server, loop, [0]).

 loop(Tot) ->
     receive
         {Client, {square, X}} ->
             Client ! X*X,
             loop(Tot + X*X);
         {Client, {rectangle,X,Y}} ->
             Client ! X*Y,
             loop(Tot + X*Y);
         {Client, areas} ->
             Client ! Tot,
             loop(Tot)
     end.
```

It is an ”area server” where clients might ask for area of squares or rectangles, and also the total area of the forms already asked (that are kept in the server). The function `start()` spawn the process `loop(0)`, with the accumulated area equal to zero. The primitive Erlang function `spawn(Module,Fun,Args)` creates a parallel process that evaluates `Module:Fun(Args)` and that returns a Pid (Process identifier) that can be used for communicating with the process created.

On the new process created, `loop(Tot)` wait for a message to arrive, asking some area or total. Other messages that do not match with the ones described in `receive` will be kept in the mailbox, the waiting queue of messages of the process. When a new message arrives and matches with the pattern within `receive`, the respective actions are executed. The message sending is represented by `!`. On the example above, when a client asks for something, it sends its own Pid inside the message, matching with `Client`, that is used as the destination for the answer.

Synchronisation in Erlang is done by message passing: sending and receiving message, since sending a message is non-blocking, while receiving suspends until a message matches one of its clauses.

Following with the given example, a client process can communicate with the server with this part of code:

```
...
Server ! {self(), {square, 10}},
receive
    Area ->
        Area
end
...
```

where `self()` returns its own Pid and the area is simply returned by the Area clause on the receive statement (in this case, Area would match with any message that arrives to the client process).

2.3 References

From the many articles and technical reports consulted for this work, we must cite the most important ones. For starting, the Curry Report [Han] introduces and defines Curry and works as the reference for the language. Also the work by Mariño [Car01] on semantics and analysis for references in definitional trees, formalisations, operational semantics and execution trees. For Erlang, the most specific material can be found on its webpage[6].

[6] http://www.erlang.org/doc/

Chapter 3

The Approach

before introduce formally the translation and execution model, we must introduce the general approach for multi-thread execution. in short, we need two things: an evaluation model, distributing the execution of Curry in processes and a transformation of Curry programs to Erlang readable code.

Initially we introduce an informal execution of Curry expressions, then the idea of the translation will comes naturally on the next chapter.

3.1 Name server

A name server is proposed to control the assignation of variables to values. it must work on a similar as Curry: some substitutions are shared between some evaluation branches, however others are only visible for specific branches. The structure of a name server tree provide such organisation for variables substitutions.

On the Figure 3.1, we relate the processes to a server name that, hierarchically control its children and sharer its information with them. Thus, when a process needs to consult a variable, it will return the search of it on its correspondent branch.

On general scheme represented on the figure 3.1, circles are server name processes, dashed lines corresponds to the link a name server with it respective evaluation processes and the arrows represents the hierarchical link between name servers. It can be seen that the variables bindings do not collapse in any branch, given that a variable cannot have two different bindings on a same environment.

3.2 Processes Execution

The idea of executing expressions lies on a recursive evaluation of the expressions, however, respecting the need of the execution. As we keep a needed narrowing,

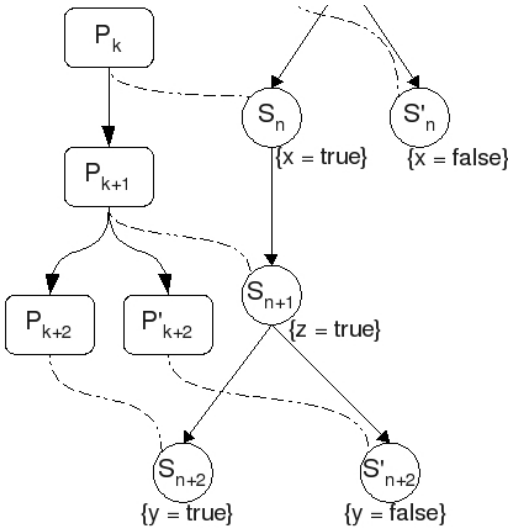

Figure 3.1: A graphical example of name server processes.

only the needed expressions will be evaluated.

The general model consists of a processes creation an communication. Each process executes a expression and communicate with its name server and the parent process, the one which started it. The communication with the name server is restrict to consult variable names, or creating a new name server node with a new name binding.

The general interface of a parent process (Pp) is show on figure 3.2. The processes (as rounded rectangles) have a correspondent name server (S) and may send messages with answers to its parent process, then finalise the computation with a ”died” message.

The communication with its parent processes consists with a set of messages with the answers of the evaluation, if it exists, and them a message indication its death. Semantically, the parent process will know when the computation it ordered to a child process is over, i.e. there is no more answers for the request.

3.2.1 Variables

In Curry, variables are mapped to values if they were already evaluated, i.e. if they were assigned to a value on the respective name server. However, if they do not occur in the name server, the evaluation function returns the same expression, the variable.

Variables are represented on name servers as a tuple of its name and a value.

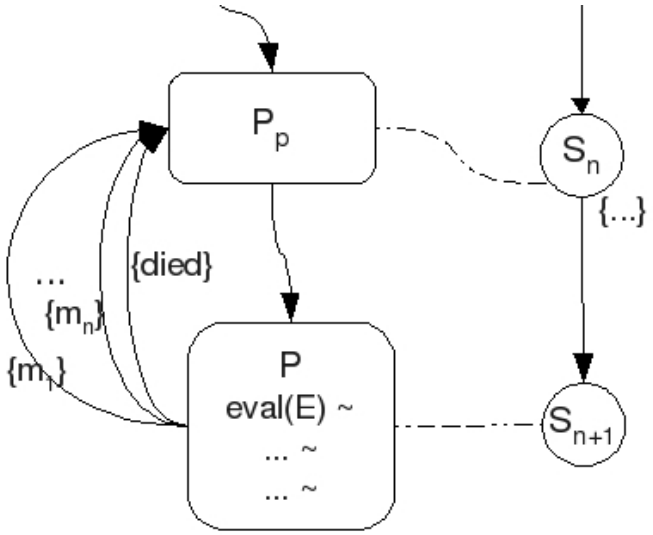

Figure 3.2: A general scheme of evaluation processes.

Syntactically, variables are represented as tuples $\{var, X\}$ where X is the atom that represent the variable.

The Figure 3.3 shows how a process evaluates a variable, asking for possible bindings in the name server.

After the response from the name server, the process sends a message with the data to its parent process and dies.

3.2.2 Peano Natural Numbers

Peano numbers, as constructors, must be represent as nested expressions. In Erlang, a peano number would be represented as $succ, E$, where E is another expression to be evaluated, if needed.

The Figure 3.4 shows the creation of a second process for evaluate the inner expression of the constructor. After receiving the answer, the actual process send the data received nested in the constructor.

As can be seen, there is no need for creating a new server name in the hierarchy, unless the process evaluating E needs to assign values to possible variables present in E.

3.2.3 Functions

We may say there are two types of functions: the ones that returns a answer (nullary functions), and the flexible ones that may assign values to a variable

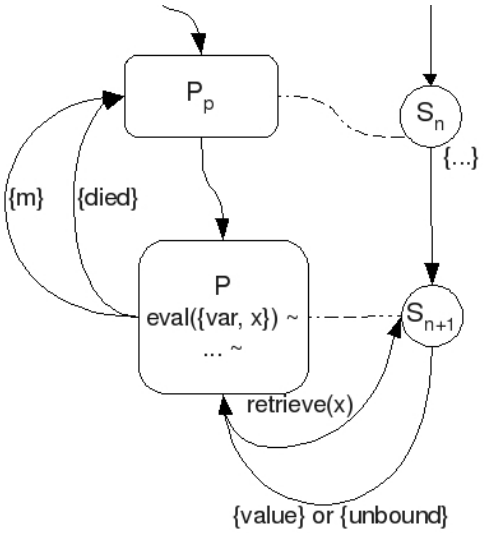

Figure 3.3: Evaluating a variable.

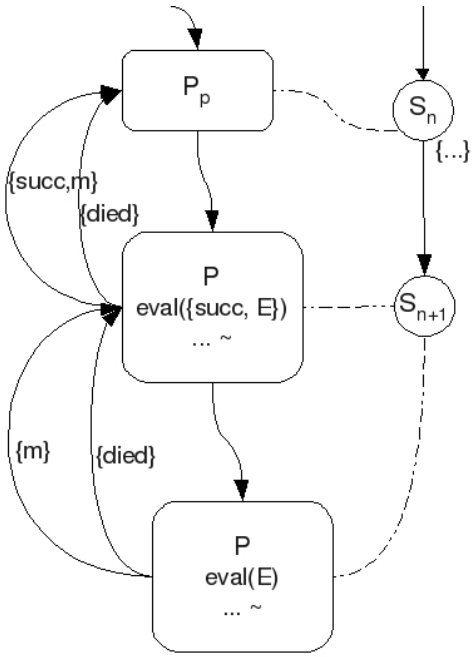

Figure 3.4: Evaluating the Peano's constructor.

and have many branches of computation. We will need that the translation of the Curry functions provides us such differentiation for executing functions and spawn processes when need.

A general scheme of nullary functions executions can be seen in the Figure 3.5. As there is no need for spawning a new process, the evaluation is apply on the current process.

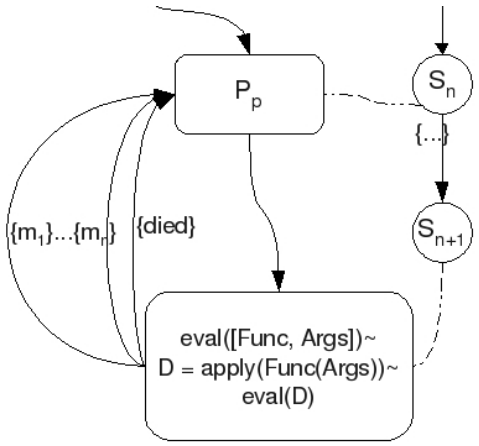

Figure 3.5: The evaluation processes of a nullary function.

For flexible functions, the evaluation is slightly more complicated. We need to verify the pivot of the functions and exchange its values to the previous value. As the pivot may be an expression of many values, we need process to control this answers.

If, among the possible values of a pivot, there is a variable, it will be substituted for each value, creating a new server name branch, and spawning new processes.

The Figure 3.6 shows the state of the process until reading the values of the pivot. After start receiving, new process are spawned, creating new server names under the hierarchy, assigning the substitutions in the tree and sending the answers to the current process.

The loop while_pivot_msg receives all answers from the pivot's evaluation. In case it is a variable, for each possible value, a new server name is created, the variable bound and a new process is spawned for evaluate the next level of the function. If the pivot is a value, it must be directly spawned as argument on the next level of the function.

Finally, the Figure 3.7 shows the messages flow from the many processes created sending the message answers back to its parent. The tuples representing

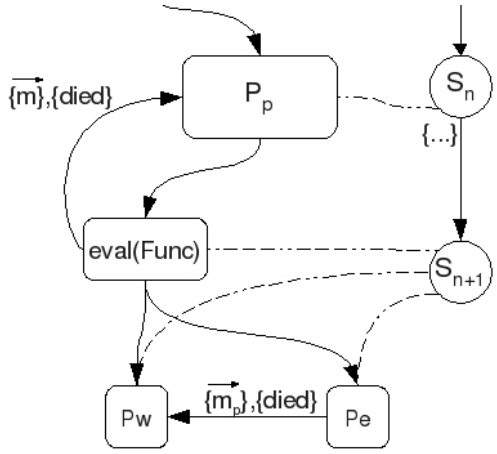

Figure 3.6: A general scheme of evaluation processes.

messages were compacted as vector of messages $\overrightarrow{m_i}$, only a matter of having this example more understandable.

3.2.4 Terms

Terms as `true`, `false`,`zero` etc are already in head normal form, then its evaluation return the own value.

$\mathsf{Teval}[\![E]\!] =$
    $\mathsf{send}(E,\ PPid)$
    $\mathsf{send}(\mathsf{died},PPid)$

Together with variables and nullary functions, terms are situated at the leaves of the evaluation tree, since no more processes are started. The Figure 3.8 confirms how simple is this evaluation, i.e. terms are already in normal form.

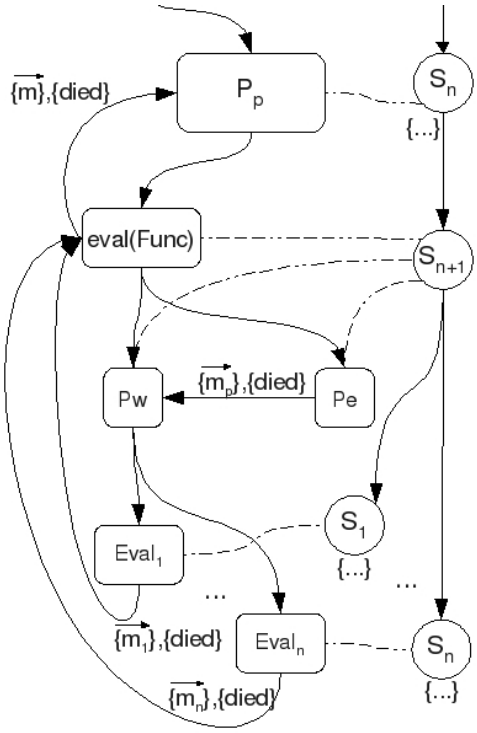

Figure 3.7: A general scheme of evaluation processes.

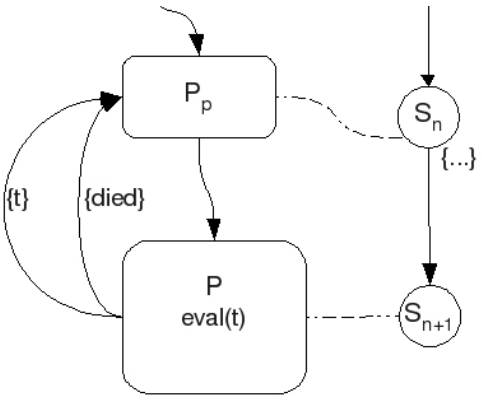

Figure 3.8: A general scheme of evaluation processes.

Chapter 4

The Translation

In this chapter, a transformation from a Curry program to an Erlang one will be defined. In short, will be shown the result of the transformation from Curry to FlatCurry and its conversion to Erlang functions, and also the formal translation rules. The result of the translation of Curry functions in Erlang code will be used by the evaluation system, detailed on the next chapter.

However, given the strict evaluation of Erlang, the results, specially functions, are returned as tuples and lists. For instances, a function that always returns the boolean value `true`, returns now `{true}`. Functions that returns another functions will not call the function as `module:function(args)`. It will be given as a list inside a tuple: `{[module, function, args]}`, avoiding the strict evaluation native in Erlang.

On the next sections, the basic translation will be shown, e.g. exporting functions, defining names and exporting the module name, then each rule will be introducing following the FlatCurry order: first types, than each type of expression, as the most significant difference among the function types. After this translation, the code will be connected with the evaluation code. Without this, that has no meaning regarding to the initial Curry program.

4.1 Basics

Given the FlatCurry data returned from PAKCS of a curry program, the translation starts from the `module`, `export` and `define` declarations, then the functions will be translated in the next sections. For better understanding of this process, a Curry program covering all types of functions was prepared, namely `exprs.curry`, then each part of it will be translated along this chapter. Below, we have the first part of the translation[1].

[1]The file exprs.curry can be found on Appendix A

$$\begin{aligned}
&\mathsf{Tprog} :: \mathsf{Prog} \rightarrow \mathsf{Erl} \\
&\mathsf{Tprog}\ [\![(Prog\ modname\ [imports]\ [TypeDecl]\ [FuncDecl]\ [OpDecl])]\!] = \\
&\qquad \text{-module}([\![modname]\!]). \\
&\qquad \text{-define(Modname, } [\![modname]\!]). \\
&\qquad \text{-import}([\![[imports]]\!]). \\
&\qquad \text{-export}(\mathsf{Texport}[\![[FuncDecl]]\!]) + [\![evalDecl]\!]. \\
&\qquad \mathsf{Tfunc}[\![[FuncDecl]]\!] \\
&\qquad evalFunc
\end{aligned}$$

Where *evalDecl* is the evaluation function declarations that will be added in the end of the Curry program translation(*evalFunc*). Next, Texport extract the function name of all functions declared at FuncDecl.

$$\begin{aligned}
&\mathsf{Texport} :: \mathsf{FuncDecl} \rightarrow \mathsf{String} \\
&\mathsf{Texport}[\![(Func\ (modname, fname)\ arity\ visibility\ type\ rule)]\!] = \\
&\qquad [\![fname]\!]/[\![arity]\!]
\end{aligned}$$

4.2 Function Declaration

Translate a function means transform it definitional tree represented in FlatCurry to Erlang functions, maintaining the semantic answers as the evaluation function needs. Thus, branches may be translated in a precise order, as functions on the same depth have the same function name: the initial function name added by the depth.

Therefore, a branch-and-bound algorithm is needed to translate first all the branches from the same level before start the a lower level. Thus, the function translation analyses and translates each level, and keep the descendant branches for the next translation round. This is done by the function Trule and Texpr_uber shown below:

Starting by the function declaration, for each function described in a FuncDecl, the following translation is applied:

$$\begin{aligned}
&\mathsf{Tfunc} :: \mathsf{FuncDecl} \rightarrow \mathsf{Erl} \\
&\mathsf{Tfunc}[\![(Func\ (modname, fname)\ arity\ visibility\ type\ rule)]\!] = \\
&\qquad \mathsf{Trule}[\![rule, fname]\!]
\end{aligned}$$

Trule will call Texpr_uber with the function name, the level equal to zero, since it is the first layer, and a structure representing the branch-and-bound algorithm, where there is no function already translated, and the *root* waiting for being translated.

Inside the `Rule` structure, the expressions will represent the functions having the function name given, the arguments represented by the list of variables (that also are expressions) provided by $[\![[varindex]]\!]$ and the content of *exprs*, that will determine the type of the respective function.

$$\begin{aligned}
&\mathsf{Trule} :: \mathsf{Rule} \rightarrow \mathsf{String} \rightarrow \mathsf{Erl} \\
&\mathsf{Trule}[\![(Rule\ [varindex]\ expr), fname]\!] = \\
&\quad \mathsf{Texpr_uber}[\![fname, level, ([], [([[\![varindex]]\!], expr)])]\!] \\
&\quad \mathsf{where} \\
&\qquad level{=}0
\end{aligned}$$

Texpr_uber returns the functions on the proper order to be written in Erlang: if a expression that just was translated return more branches to be executed, it is put on the end on the queue, together with the new function name, provided by increasing the level of it.

$$\begin{aligned}
&\mathsf{Texpr_uber} :: \mathsf{String} \rightarrow \mathsf{Int} \rightarrow (\mathsf{[String]},\ \mathsf{[([Expr],\ Expr)]}) \rightarrow \mathsf{Erl} \\
&\mathsf{Texpr_uber}[\![fname, level, ([func_str], [lexpr])]\!] = \\
&\quad \mathsf{if}\ lexpr\ \mathsf{do} \\
&\qquad [func_str]\ \mathsf{++}\ [l_func_str]\ \mathsf{++}\ \mathsf{Texpr_uber}[\![fname, (level+1), new_lexpr]\!] \\
&\quad \mathsf{where} \\
&\qquad (l_func_str, new_lexpr) = \mathsf{Texpr}[\![fname, [varindex], level, expr]\!]
\end{aligned}$$

The Erlang functions returned by Texpr depends on the expression type its represents. The examples on the subsections below will present Curry functions and a data definition, its respective FlatCurry definition and the final Erlang code. Then the formal translation is show, as above, from each `Expr` to Erlang code.

4.2.1 Data Definition

The Peano's natural numbers are a essential example of data defined by the user. In a Curry syntax, they are defined as:

```
data Nat = Zero | Succ Nat
```

In FlatCurry, the data definitions appear in a separated section, before the function definitions. Thus, the flat representation from the naturals defined above are:

```
[Type ("exprs","Nat") Public []
  [Cons ("exprs","Zero") 0 Public [],
   Cons ("exprs","Succ") 1 Public [TCons ("exprs","Nat") []]
  ]
]
```

where `exprs` is the module name, and `Public` declares that Nat is a public data type.

Thus, defined data types do have explicit a flat representation. However, Erlang is not typed, and the solution proposed is to manage them as tuples, therefore they will be adapted directly on the translation of data constructors, when they appear in some part of the FlatCurry code, e.g. `(Succ (Succ Zero))` must be `{succ, {succ, zero}}`.

It could be considered to translate these data types declarations into records in Erlang. Records are defines as:

```
-module(person).
-export([new/2]).

-record(person, {name, age}).

new(Name, Age) ->
    #person{name=Name, age=Age}.

1> person:new(ernie, 44).
{person,ernie,44}
```

In the end, records are tagged tuples with defined constructors. An initial idea for representing defined data types, e.g. the Peano naturals, in Erlang could be:

```
-record(nat, {succ, peano}).
```

Records may contain another different record on his composition, but recursive ones are not allowed. Thus, it is not possible to have a natural number on the successor record, i.e. substitute `peano` for a recursive call. Moreover, records are represented as tuples then data types will be defined directly as tuples. For instance, in the case of `Nat`: `{zero}` , `{succ,{zero}}`, `{succ,{succ,{zero}}}` ...

A practical example of data types will be detailed on the following sections, among functions examples.

4.2.2 Literals

Literal rules define functions leading to constant values: integers, floats or characters:

```
lit_int = 7
```

```
lit_float = 3.1416

lit_char = 'c'
```

And their respective FlatCurry definitions are:

```
Func ("exprs","lit_int") 0 Public
  (TCons ("Prelude","Int") [])
  (Rule [] (Lit (Intc 7))),

Func ("exprs","lit_float") 0 Public
  (TCons ("Prelude","Float") [])
  (Rule [] (Lit (Floatc 3.1416))),

Func ("exprs","lit_char") 0 Public
  (TCons ("Prelude","Char") [])
  (Rule [] (Lit (Charc 'c'))),
```

The functions type signature is already given by the FlatCurry representation. It facilitates the function type definition even when it is not explicitly given by the source code: it is inferred by the analysis leading on the FlatCurry.

On the example above, the signature of the function `lit_int` is expressed by `(TCons ("Prelude","Int") [])`, it means that `lit_int` returns predefined type: `Integer`.

Even having the function signature provided, the final Erlang code does not need its representation, since Erlang is dynamically typed and type safe. Therefore, this declaration is, by now, discarded. Finally, the Erlang code for these functions above are:

```
lit_int() ->
        {7}.

lit_float() ->
        {3.1416}.

lit_char() ->
        {'c'}.
```

The translation of literal functions are quite intuitive and do not demand much complexity:

Texpr :: String → [Expr] → Int → Expr → ([String], [([Expr], Expr)])

$$\mathsf{Texpr}[\![fname, [varindex], level, (Var\ i)]\!] =$$
$$(func\_str, [])$$
$$\text{where}$$
$$func\_str = [[\![fname, level]\!](\mathsf{ET}[\![[varindex]]\!]) \rightarrow \{\mathsf{ET}[\![(Var\ i)]\!]\}.]$$

$$\mathsf{ET} : \mathsf{Expr} \rightarrow \mathsf{Erl}$$
$$\mathsf{ET}[\![(Lit\ i)]\!] = \mathrm{i}$$

4.2.3 Variables

Simpler than previous expression, variable functions are the easiest translation since its characterised by functions that returns a variable, or its correspondent value, from the arguments.

```
func_var x = x

func_var' x y z = y
```

For the functions examples above, the related flat representation are:

```
Func ("exprs","func_var") 1 Public
  (FuncType (TVar 0) (TVar 0))
  (Rule [1] (Var 1)),

Func ("exprs","func_var'") 3 Public
  (FuncType
    (TVar 0)
    (FuncType (TVar 1) (FuncType (TVar 2) (TVar 1)))
  )
  (Rule [1,2,3] (Var 2)),
```

As can be observed above, the function signature of `func_var'` begins indexing variables by 0 and the expression starts describing the variables on the function by 1. The former shows the variables labelled by 0, 1 and 2 as parameters and the return of the function is the variable labelled by 1. It is, the signature of `func_var'` is `func_var' :: v0 -> v1 -> v2 -> v1`. On the latter, also represented inside the rule, the variables come as a vector of indexes `[1,2,3]` and then cited as `(Var i)` as could be inferred above.

And finally the simplest translation:

```
func_var(X1) ->
        {X1}.
```

```
func_varP(X1, X2, X3) ->
        X2.
```

$$\mathsf{Texpr}[\![fname,[varindex],level,(Lit\ l)]\!] =$$
$$(func\_str,[])$$
$$\text{where}$$
$$func\_str = [[\![fname,level]\!](\mathsf{ET}[\![[varindex]]\!]) \rightarrow \{\mathsf{ET}[\![(Lit\ l)]\!]\}.]$$

As variables in Erlang are uppercase, the name X was given to the variables in Erlang code, followed by the *v* number, given by FlatCurry.
$\mathsf{ET} : \mathsf{Expr} \rightarrow \mathsf{Erl}$
$\mathsf{ET}[\![(Var\ v)]\!] =$ Xv

4.2.4 Combinations

Combination rules represent functions that return callings to another functions or constructors. The two classes of combinations are distinguished by `CombType` on the `Comb` expression. Such classification comes to distinguish structures which may be a function with its arguments or a constructor, i.e. the Peano's natural numbers cited before.

The formal translation is given before, since it is the same for the both cases, except for ET.

$$\mathsf{Texpr}[\![fname,[varindex],level,(Comb\ ctype\ (modname,cname)\ [expr])]\!] =$$
$$(func\_str,[])$$
$$\text{where}$$
$$func\_str = [[\![fname,level]\!](\mathsf{ET}[\![[varindex]]\!]) \rightarrow$$
$$\{\mathsf{ET}[\![(Comb\ ctype\ (modname,cname)\ [expr])]\!]\}.]$$

$$\mathsf{ET} : \mathsf{Expr} \rightarrow \mathsf{Erl}$$
$$\mathsf{ET}[\![(Comb\ ConsCall\ (modname,cname)\ [e_1 \ldots e_n])]\!] =$$
$$\{cname,\mathsf{ET}[\![e_1]\!],\ldots,\mathsf{ET}[\![e_n]\!]\}$$
$$\mathsf{ET}[\![(Comb\ ConsCall\ (modname,fname)\ [e_1 \ldots e_n])]\!] =$$
$$[modname,fname,[\mathsf{ET}[\![e_1]\!],\ldots,\mathsf{ET}[\![e_n]\!]]]$$

Function Combinations

Function combinations are the class of functions that leads to a call to another function or operator.

```
comb_func x = lit_int

comb_func' x y z = add x (add y z)

comb_func'' x = comb_func' x x x
```

And the related flat representation is given below. Notice that the `Comb` tag is followed by `FuncCall` as the `CombType`.

```
Func ("exprs","comb_func") 1 Public
  (FuncType (TVar 0) (TCons ("Prelude","Int") []))
  (Rule [1] (Comb FuncCall ("exprs","lit_int") [])),

Func ("exprs","comb_func'") 3 Public
  (FuncType
    (TCons ("exprs","Nat") [])
    (FuncType
      (TCons ("exprs","Nat") [])
      (FuncType
        (TCons ("exprs","Nat") []) (TCons ("exprs","Nat") [])
  ) ) )
  (Rule [1,2,3]
    (Comb FuncCall
      ("exprs","add")
      [Var 1,Comb FuncCall ("exprs","add") [Var 2,Var 3]]
  ) ),

Func ("exprs","comb_func''") 1 Public
  (FuncType (TCons ("exprs","Nat") [])
    (TCons ("exprs","Nat") [])
  )
  (Rule [1]
    (Comb FuncCall ("exprs","comb_func'") [Var 1,Var 1,Var 1])
  )
```

As can be seen above, the parameters of the combination functions are other expressions, that cam be another combinations, variables or literals. Thus the respective translation to Erlang is[2]:

[2] As Curry and Erlang have different rules for function names. The apostrophe ' found on names as comb_func' is being converted by a P, for prime.

```
comb_func(X1) ->
        {[exprs, lit_int, []]}.

comb_funcP(X1, X2, X3) ->
        {[exprs, add, [X1, [exprs, add, [X2, X3]]]]}.

comb_funcPP(X1) ->
        {[exprs, comb_funcP, [X1, X1, X1]]}.
```

Note how functions are translated. The list inside the tuple will indicate to the evaluator the result as a list Erlang-like, i.e. Erlang uses this syntax for representing functions for avoid strict evaluation. The given result may be spawn as a process or executed directly by, respectively, the functions `spawn` and `apply`:

```
spawn([modname, fname, [arguments]]).

apply([modname, fname, [arguments]]).
```

Constructor Combinations

Constructor combinations express functions that result in defined data types or prelude instances, e.g. `True`, `False`, `Zero`, `Succ (Succ Zero)` ... Examples below.

```
comb_cons = Zero

comb_cons' x = Succ x

comb_cons'' = True
```

On this flat representation, the `Comb` tag is followed by `ConsCall` as the `CombType`. It contains the constructor name or type and its arguments, if any.

```
Func ("exprs","comb_cons") 0 Public
  (TCons ("exprs","Nat") [])
  (Rule [] (Comb ConsCall ("exprs","Zero") [])),

Func ("exprs","comb_cons'") 1 Public
  (FuncType (TCons ("exprs","Nat") [])
    (TCons ("exprs","Nat") [])
  )
  (Rule [1] (Comb ConsCall ("exprs","Succ") [Var 1])),
```

```
Func ("exprs","comb_cons''") 0 Public
  (TCons ("Prelude","Bool") [])
  (Rule [] (Comb ConsCall ("Prelude","True") [])),

comb_cons() ->
        {{zero}}.

comb_consP(X1) ->
        {{succ, X1}}.

comb_consPP() ->
        {true}.
```

Here finally is given an example of data type definitions converted to tuples. Its the case of `{zero}` and `{succ, X1}`. The Boolean data type `True`, as all the others, must correspond to the Erlang syntax. In this case, `true` must be lowercase.

4.2.5 Let

Let express the introduction of a bound variable in the function declaration. This fresh variable has the scope only within the function it is contained, there bound variables are set to ground values: combinations (function call or constructors) or constants (literals: an integer, a float or a character).

```
func_let =
  let x = (Succ Zero) in comb_cons' x

func_let' = comb_cons' x
  where x = (Succ Zero)
```

On the Curry syntax it may be a `let` or `where` declaration, the former is declared before its application and the latter, after the function declaration. In both cases variables are being bound to ground values. However it may be the case that variables will be bound to previously declared variables in the scope, but this brings a more complicated case and will be cited further.

```
  Func ("exprs","func_let") 0 Public
  (TCons ("exprs","Nat") [])
  (Rule []
```

```
    (Let
      [(1, Comb ConsCall ("exprs","Succ")
        [Comb ConsCall ("exprs","Zero") []]
      )]
    (Comb FuncCall ("exprs","comb_cons'") [Var 1])
    )
  ),

Func ("exprs","func_let'") 0 Public
  (TCons ("exprs","Nat") [])
  (Rule []
    (Let
      [(1,
      Comb ConsCall ("exprs","Succ")
      [Comb ConsCall ("exprs","Zero") []]
      )]
    (Comb FuncCall ("exprs","comb_cons'") [Var 1])
    )
  ),
```

On the final code the new variable introduced is named by the index number gave by the FlatCurry representation. As can be seeing in the final translation above, both `let` and `where` declarations are indistinguishable, given the same flat representation.

```
func_let() ->
        X1 = {succ, {zero}},
        {[exprs, comb_consP, [X1]]}.


func_letP() ->
        X1 = {succ, {zero}},
        {[exprs, comb_consP, [X1]]}.
```

The translation for `Let` and `where` is a special case of a new statement inclusion inside the other types of functions. It is passed as argument to the nested expression and put at first between their statements (as seem above).

$$\mathsf{Texpr}[\![fname, [varindex], level, (Let\ [(varidx, exprl)]\ expr)]\!] =$$
$$\quad \mathsf{Texpr}[\![exprs]\!] \leftarrow let\_str$$
$$\quad \text{where}$$

$$
\begin{aligned}
&let\_str = \\
&\quad \mathsf{ET}[\![(Var\ varidx_1)]\!] = \mathsf{ET}[\![exprsl_1]\!], \\
&\quad \ldots \\
&\quad \mathsf{ET}[\![(Var\ varidx_n)]\!] = \mathsf{ET}[\![exprsl_n]\!],
\end{aligned}
$$

4.2.6 Free

Also using the same `let` declaration, new free variables are introduced as `let x free in`, where `x` is a fresh variable name within its scope. Or also as `where x free`, as below.

```
func_free =
  let x free in comb_cons' x

func_free' =
  comb_func x
  where x free
```

Thus, in the flat representation, `Free` shows what variables are free regarding to its index number. Notice the scope of the free variable is under the expression (`Expr`) under the `Free` rule.

```
Func ("exprs","func_free") 0 Public
  (TCons ("exprs","Nat") [])
  (Rule []
    (Free [1]
      (Comb FuncCall ("exprs","comb_cons'") [Var 1])
    )
  ),

Func ("exprs","func_free'") 0 Public
  (TCons ("Prelude","Int") [])
  (Rule []
    (Free [1]
      (Comb FuncCall ("exprs","comb_func") [Var 1])
    )
  ),
```

Finally, the translation to Erlang is given by introducing a syntax for free variables in Erlang. As known, Erlang does not accept unbound variables as argument

to any function. The solution is given by a structure like `{var, var_x`$_i$`}` where i stands for the index number of the new variable, not existing in the server name.

This structure representing new variables will be needed when arguments are checked as grounded or not on functions of `Case Flex` type, as will be shown further. By now, one must record how free variables are.

```
func_free() ->
        X1 = {var, var_x1},
        {[exprs, comb_consP, [X1]]}.


func_freeP() ->
        X1 = {var, var_x1},
        {[exprs, comb_func, [X1]]}.
```

As on `Let` statements, `Free` expressions have the same translation from Curry to Erlang due to the same flat representation. The free variable will be passed as the argument to the function nested on its scope.

Similarly to `Let` statements, free variables must be introduces before the last statement. The formal translation create this new variable and pass it to the nested expression, to be placed in the right position.

$$
\begin{aligned}
&\mathsf{Texpr}[\![fname, [varindex], level, (Free\ [varidx]\ expr)]\!] = \\
&\quad \mathsf{Texpr}[\![exprs]\!] \leftarrow let\_str \\
&\quad \text{where} \\
&\qquad free\_str = \\
&\qquad\quad \mathsf{ET}[\![(Var\ varidx_1)]\!] = \{var,\ new\_var_j\}, \\
&\qquad\quad \ldots \\
&\qquad\quad \mathsf{ET}[\![(Var\ varidx_n)]\!] = \{var,\ new\_var_{j+n}\},
\end{aligned}
$$

4.2.7 Case Flex

Expressions `Case Flex` are the most important ones in this work. Despite of the simplicity of the final result of this case translation, the real possibility of multi-thread computation is justified by this sort of expression.

Cases represent the creation of new branches, the place where an unbound variable may be assigned by some values, reduce an argument, and split the computation to s many branches as the assignations.

An example of this behaviour can be checked in functions where, among the parameters, there are bound values or structures for pattern matching.

```
fcase_flex 1 2 = 3
fcase_flex 1 3 = 4
fcase_flex 2 3 = 5
```

The FlatCurry representation indicates the pivot variable among the arguments, i.e. the *i-th* argument will be confronted with the branch possible values. If it is bound and its value match with the pattern of some branch, then the computation follows. otherwise, there is no solution for that value on that position.

Case the pivot is a free variable, it may be assigned to each of the patterns from the branches:

FlatCurry:

```
Func ("exprs","fcase_flex") 2 Public
  (FuncType
    (TCons ("Prelude","Int") [])
    (FuncType
      (TCons ("Prelude","Int") [])
      (TCons ("Prelude","Int") [])
    )
  )
  (Rule [1,2]
    (Case Flex (Var 1)
      [Branch (LPattern (Intc 1))
        (Case Flex (Var 2)
          [Branch (LPattern (Intc 2)) (Lit (Intc 3)),
          Branch (LPattern (Intc 3)) (Lit (Intc 4))]
        ),
      Branch (LPattern (Intc 2))
        (Case Flex (Var 2)
          [Branch (LPattern (Intc 3)) (Lit (Intc 5))]
        )
      ]
    )
  ),
```

The formal translation keeps the branch-and-bound navigation of the `Case Flex` tree:

$$\mathsf{Texpr}[\![fname,[varindex],level,(Case\ Flex\ (var\ i)\ [branches])]\!] =$$
$$(func\_str, new\_branches)$$
$$\text{where}$$
$$new\_branches = \mathsf{Tbranch}[\![[branches],[varindex]]\!]$$

$$
\begin{aligned}
&func\_str = \\
&\quad [i,\ \mathsf{Tpattern}[\![[branches]]\!],\ [\![fname, level+1]\!]]
\end{aligned}
$$

Where Tpattern returns the list of the pivots values from *branches*. With this syntax, all the information about the tree is given in the answer, since the next step of the function is defined in the function f_{n+1}. The translation of those functions will be done in the next step of the translation: the branches are passed on the queue for the next translations.

$$
\begin{aligned}
&\mathsf{Tbranch} :: \mathsf{BranchExpr} \rightarrow [\mathsf{Expr}] \rightarrow ([\mathsf{Expr}], \mathsf{Expr}) \\
&\mathsf{Tbranch}[\![branch, [v_1, \ldots, v_n]]\!] = \\
&\quad (new\_varindex, expr) \\
&\quad \mathsf{where} \\
&\quad\quad (Branch\ pattern\ expr) = branch \\
&\quad\quad new\_varindex = [v_1, \ldots, [\![pattern]\!]_{pivot}, \ldots, v_n]
\end{aligned}
$$

```
fcase_flex(X1, X2) ->
        {1, [1, 2], fcase_flex1}.

fcase_flex1(1, X2) ->
        {2, [2, 3], fcase_flex2};

fcase_flex1(2, X2) ->
        {2, [3], fcase_flex2}.

fcase_flex2(1, 2) ->
        {3};

fcase_flex2(1, 3) ->
        {4};

fcase_flex2(2, 3) ->
        {5}.
```

Chapter 5

Execution Model

At this point, all the base for the evaluation is done: the functions were translated to Erlang in a proper way to do not be strictly evaluated, and its data is only accessible by its explicit execution. On this chapter, the expressions evaluation will be defined in order to access that information, set the necessaries substitutions and make the evaluation.

Given the goal of evaluating expressions, each evaluation step will be done spreading the work through processes. First we have the name server, a tree of process that will keep the proper set of substitutions to the correspondent evaluation processes. Then the evaluation functions are introduced, and its recurvise execution is related with the operational semantics of Curry.

5.1 Name server

The design of a name server demands a simple tree structure which keeps all the variable bindings for the respective process it is linked. An initial name server is created together with the first evaluation process, and passed on as parent name server to the new evaluation branches. When a variable needs to be assigned to a value in a new environment, a new name server is created and linked with the parent name server, keeping all the bindings made previously and attaching the new one only for the respective evaluation environment.

To be most precise, a new name server process is only created when a function is evaluated with variables, and those are free in the parent name server. For carrying o with the computation, a new name server process is created for each new evaluation process, as many as values assigned to the free variable in question.

Each leaf of the name server tree corresponds to a substitution set that may reach a solution. When a solution is found, it is sent together with its correspondent name server, in order to keep track of the bindings. In case the solutions

reach the initial process that asked for the evaluation, the solution is printed and all the bindings from that branch also, until it reaches the first name server.

5.2 Concurrent Evaluation

On this evaluation step, it will be show how Curry evaluates its expressions (constructors, functions, variables...) using lazy narrowing. Then, the semantically correspondence to this proposed evaluation will be shown, in order to support the multi-thread evaluation.

5.2.1 Operational semantics

Curry works with a lazy narrowing strategy, i.e. narrow a inner position only if it is really necessary for the outermost evaluation, by pattern matching the pivot on the left hand side of some rule. The evaluation to normal form of each expression is shown on the Figure 5.1.

$$\frac{[x \mapsto t] \in \sigma}{x \Downarrow t \mid \sigma} \tag{5.1}$$

$$\frac{}{c \Downarrow c \mid \varepsilon} \tag{5.2}$$

$$\frac{f \rightsquigarrow r \qquad r \Downarrow t \mid \sigma}{f \Downarrow t \mid \sigma} \tag{5.3}$$

$$\frac{}{f \Downarrow f \mid \sigma} \tag{5.4}$$

$$\frac{e_1 \rightsquigarrow c\ r_1 \ldots r_m \mid \sigma}{(e_1\ e_2) \Downarrow c\ r_1 \ldots r_m\ e_2 \mid \sigma' \circ \sigma} \tag{5.5}$$

$$\frac{e_1 \Downarrow f \mid \sigma \qquad e_2\sigma \Downarrow r' \mid \sigma' \qquad (f\ r')\sigma' \rightsquigarrow_{\sigma''} e \qquad e \Downarrow t \mid \sigma'''}{(e_1\ e_2) \Downarrow t \mid \sigma''' \circ \sigma'' \circ \sigma' \circ \sigma} \tag{5.6}$$

Figure 5.1: The big step operational semantics based on lazy narrowing to head normal form

Respectively:

5.1 A variable x evaluates to t if x was narrow to t.

5.2 Constructors are already head normal form.

5.3 If f is a nullary function symbol, it computes r and r evaluates to t.

5.4 Otherwise f is in normal form.

5.5 A narrowing construction.

5.6 Narrowing function application.

The next steps will show that the proposed Curry translation and evaluation keeps the operational semantics of Curry. Evaluate a expression mean execute it in a evaluation function called `eval`. For all types of expressions, this function will return all the solutions attached with each correspondent substitutions.

In Erlang semantics, a process will be launched for evaluate a expression and will return n messages containing answers and at last one informing that the computation is over.

In general, a evaluation process start by a parent process [PPid] with a given name server [Ns] and an expression to evaluate. If the process is able to evaluate the expression itself, it will send the answers to the PPid and then the sign of its dead. The PPid will wait for the answers and resend them to it own parent process while messages are arriving. They will wait for answers until the process they started send the message indication its death. In this case, the PPid itself, if it is not waiting for messages from other processes it started, can ”die”.

All a process must do is try to evaluate a expression and send the answers to the process that started it. There are cases that a processes must start more than one process and wait for the answers of all of them before die. These differences appear on each type of expression, shown on the next sections, relating their evaluation with the Curry operational semantics shown on Figure 5.1.

5.2.2 Variables

In Curry, variables are mapped to values if they were already evaluated, i.e. if they were assigned to a value on the respective name server. However, if they do not occur in the name server, the evaluation function returns the same expression, the variable.

Variables are represented on name servers as a tuple of its name and a value. Syntactically, variables are represented as tuples $\{var,\ X\}$ where X is the atom that represent the variable. Thus, the *eval* function is:

$\mathsf{Teval}[\![PPid, \{var,\ X\}, Ns]\!] =$
$\quad V \leftarrow \mathsf{retrieve}\ X\ \mathsf{from}\ Ns$
$\quad \mathsf{case}\ V\ \mathsf{is\ bound}$

```
        send(V, PPid)
        send(died, PPid)
    else
        send({var, X}, PPid)
        send(died, PPid)
```

As in Curry, variables evaluates to values if they are in the name server, otherwise returns themselves as $\{var, v\}$, where v is the correspondent identifier. After the response from the name server, the process sends a message with the data to its parent process and dies.

5.2.3 Peano Natural Numbers

The evaluation of Peano natural numbers must check all the nested constructors of the expression, i.e. in case E, in $succ,\ E$, is a ground value, a variable or even a function. Thus, what its evaluation does is to extract the inner expression and to ask to another process the evaluation.

Case E is $\{zero\}$ or a variable, it is in head normal form.

```
Teval⟦PPid, {succ, E}, Ns⟧ =
    spawn(Teval(Self, X, Ns))
    while messages
        receive(M)
        send(M, PPid)
    send(died, PPid)
```

where $Self$ is the own process identifier.

5.2.4 Functions

The evaluation of functions will depend more on how they are defined than by the arity of a function. Functions that are flexible ones, i.e. may assign values to free variables, differ from the functions that do not need to evaluate any argument for returning a expression. The former fit on the definition of nullary functions (see 5.1), since they return a term to be evaluated, and do not need any type of substitution. The latter needs to evaluate the argument on the pivot position, the one that needs to match the possible values or be assigned to them.

For the first case, the evaluation is:

$\mathsf{Teval}[\![PPid, [Mod, Func, Args], Ns]\!] =$
$D \leftarrow \mathsf{apply}(Mod,\ Func,\ Args))$
$\mathsf{Teval}(PPid,\ D,\ Ns)$

The Erlang function apply executes the function, given that it returns a ground value, a variable or another nullary function application (non flexible). D will store this result and will be evaluated, then answers will be sent directly to the PPid.

For flexible functions, the evaluation is slightly more complicated. When evaluation the pivot argument, it may return more than one value. And all these values must took the place of the pivot to be executed on the next level of the function.

As seen before, a flexible function return a tuple composed by the pivot position, the possible values and the next function (that must be evaluated when the pivot is exchanged). Keeping this scheme in mind, the evaluation process becomes more clear.

$\mathsf{Teval}[\![PPid, [Mod,\ Func,\ Args], Ns]\!] =$
$\{Pivot,\ Values,\ Next\_func\} \leftarrow \mathsf{apply}(Mod,\ Func,\ Args))$
$\mathsf{Dest} = \mathsf{spawn}(\mathsf{while\_pivot\_msg}(self,\ \{Pivot,\ Values,\ Next\_func\}))$
$\mathsf{spawn}(\mathsf{Teval}((Dest,\ Args!!Pivot,\ Ns))$
while messages
    $\mathsf{receive}(M)$
    $\mathsf{send}(M,\ PPid)$
$\mathsf{send}(\mathsf{died},\ PPid)$

5.2.5 Terms

Terms as `true`, `false`,`zero` etc are already in head normal form, then its evaluation return the own value.

$\mathsf{Teval}[\![E]\!] =$
$\mathsf{send}(E,\ PPid)$
$\mathsf{send}(\mathsf{died}, PPid)$

Together with variables and nullary functions, terms are situated at the leaves of the evaluation tree, since no more processes are started.

Recursively, the evaluation scheme skip all expressions that do not need to be evaluated, specially when evaluation functions. The flexible function guarantee

that only the pivot position is evaluated that time, leaving the others arguments waiting for the following function call and, maybe, being converted into normal form.

Having now all the process of translation and evaluation, the next section will compare various solutions from this work and from Curry itself.

Chapter 6

Experimental Results

As experimental results, we will show how some small, however important, functions behave in our evaluator and at Packs. It will be tested with some types of each evaluation function, specially variables, constructors and functions.

The most important characteristic to be noted is whether some solution is lost, i.e. when our system do not return all the appropriated values and substitutions.

For a sort of simplicity, shell interactions will be omitted, and the goals will be defined on each section. Also the translation steps in both Pakcs and Erlang. For a complete documentation, the archives used in the Chapter 4 are available on the Appendix A, B and C.

6.1 Example 1

First we compare the solutions over constructors and nullary functions. Even not being our approach to deal with integers, chars and other types, they are evaluated as other terms, and then included here as an extra. The program for terms is the following:

```
fint = 7

ffloat = 3.1416

fchar = 'c'

czero = Zero

ctrue = True
```

```
fvar x = x

cint x y = fint
```

The test performs the goals above for $x = True$ and $y = False$, and for both programs the answers were the same: 7; 3.1416; $'c'$; *Zero*; *True*; *True*; 7. However, they look syntactically different, given the representation of booleans and Peano numbers in Erlang: 7; 3.1416; *c*; $\{zero\}$; *true*; *true*; 7

Trying to reach the following goal: `fvar z where z free`, we have, for Packs:

```
za> fvar z where z free
Free variables in goal: z
Result: z
Bindings:
z=z ?
```

In Erlang, the interaction must be done by a function called `shell` for the moment. The interface was not a priority, but will certainly be a task for a future work:

```
17> za:shell([za, fvar, [{var, z}]]).
{var,z}
no_more_solutions
```

In our system, to express that a variable is free, it comes as above. There is no need to write a long statement.

6.2 Example 2

Now we will test function evaluations with all the arguments bound, and then with free variables.

```
add  Zero x = x
add (Succ x) y = Succ (add x y)

mand False _ = False
mand True x = x

eqb True x = x
eqb False False = True
```

```
eqb False True = False

fflex 1 2 = 3
fflex 1 3 = 4
fflex 2 3 = 5

addall x y z = add x (add y z)

goal1 = add Zero (Succ Zero)

goal2 = add (Succ (Succ Zero)) (Succ (Succ Zero))

goal3 = mand False True

goal4 = eqb False False

goal5 = eqb (mand False True) False

goal6 = fflex 1 3

goal7 = fflex 2 2
```

Again, all the results were equivalent. For Curry: *Succ Zero*; *Succ* (*Succ* (*Succ* (*Succ Zero*))); *False*; *True*; *True*; 4; *No more solutions.*. And for Erlang: $\{succ, \{zero\}\}$; $\{succ, \{succ, \{succ, \{succ, \{zero\}\}\}\}\}$; $false$; $true$; $true$; 4; $no\_more\_solutions$.

6.3 Example 3

Now we introduce `let` and `where` statements in order to provide free variables to the goals. The results, now, must present all the possible assignments to the free variable on the position it occurs. The goals are (continuing the code above):

```
goal8 = add (Succ Zero) x where x free

goal9 = let x free in mand True x

goal10 = mand x y where x,y free

goal11 = eqb x y where x,y free
```

```
goal12 = fflex x (fflex x y) where x, y free

goal13 = eqb (mand x y) (eqb y z) where x, y, z free
```

When asking for the goal in the terminal, Curry returns a non-friendly answer, and still does not show the bindings, e.g.

```
za> goal13
Result: True ? ;
Result: False ? ;
Result: False ? ;
Result: True ? ;
Result: _G4790 ? ;
Result: False ? ;
Result: True ? ;
No more solutions.
```

But, when called explicitly from the shell:

```
za> add (Succ Zero) x where x free
Free variables in goal: x
Result: Succ x
Bindings:
x=x ? ;
No more solutions.

za>   let x free in (mand True x)
Free variables in goal: x
Result: x
Bindings:
x=x ? ;
No more solutions.

za> mand x y where x,y free
Free variables in goal: x, y
Result: False
Bindings:
x=False
y=y ? ;
Result: y
Bindings:
```

```
x=True
y=y ? ;
No more solutions.

za> eqb x y where x,y free
Free variables in goal: x, y
Result: y
Bindings:
x=True
y=y ? ;
Result: True
Bindings:
x=False
y=False ? ;
Result: False
Bindings:
x=False
y=True ? ;
No more solutions.

za> fflex x (fflex x y) where x, y free
Free variables in goal: x, y
Result: 4
Bindings:
x=1
y=2 ? ;
No more solutions.

za> eqb (mand x y) (eqb y z) where x, y, z free
Free variables in goal: x, y, z
Result: True
Bindings:
x=False
y=True
z=False ? ;
Result: False
Bindings:
x=False
y=True
z=True ? ;
Result: False
```

```
Bindings:
x=False
y=False
z=False ? ;
Result: True
Bindings:
x=False
y=False
z=True ? ;
Result: z
Bindings:
x=True
y=True
z=z ? ;
Result: False
Bindings:
x=True
y=False
z=False ? ;
Result: True
Bindings:
x=True
y=False
z=True ? ;
No more solutions.
```

In Erlang, the same set of answers and substitutions was returned, but sometimes in a different order. Not only because of different algorithms and structures implemented, but also for the messages that may arrive from different processes.

```
30> za:shell([za, goal8, []]).
{succ,{var,var_x1}}
no_more_solutions

31> za:shell([za, goal9, []]).
{var,var_x1}
no_more_solutions

32> za:shell([za, goal10, []]).
{var_x1 = false} | false
```

```
{var_x1 = true} | {var,var_x2}
no_more_solutions

33> za:shell([za, goal11, []]).
{var_x1 = true} | {var,var_x2}
{var_x2 = false}, {var_x1 = false} | true
{var_x2 = true}, {var_x1 = false} | false
no_more_solutions

34> za:shell([za, goal12, []]).
{var_x2 = 2}, {var_x1 = 1} | 4
no_more_solutions

35> za:shell([za, goal13, []]).
{var_x2 = true}, {var_x1 = true} | {var,var_x3}
{var_x3 = false}, {var_x2 = true}, {var_x1 = false} | true
{var_x3 = true}, {var_x2 = true}, {var_x1 = false} | false
{var_x3 = false}, {var_x2 = false}, {var_x1 = false} | false
{var_x3 = true}, {var_x2 = false}, {var_x1 = false} | true
{var_x3 = false}, {var_x2 = false}, {var_x1 = true} | false
{var_x3 = true}, {var_x2 = false}, {var_x1 = true} | true
no_more_solutions
```

6.4 Infinite Solutions

May occur that one function loops indefinitely, when a variable is reduced to a constructor with another free variable. This variable on that position will never be bound to a ground value, because, recursively, the new variable will be bound to another new one.

It is the case of the function `add`, when called with a variable in the first argument:

```
add  Zero x = x
add (Succ x) y = Succ (add x y)
```

If we try to evaluate `add x Zero where x free`, `x` will be reduced to `Zero` and `(Succ x1)`. When `add Zero Zero` is called, there is no problem, and the solution comes from the matching with the first clause: `Zero`. But `add (Succ x1) Zero` will be reduced to `Succ (add x1 Zero)` again and again.

In these cases, lets observe the solutions gave by Packs: (only answers)

```
Result: Zero
x=Zero ? ;

Result: Succ Zero
x=Succ Zero ? ;

Result: Succ (Succ Zero)
x=Succ (Succ Zero) ? ;

Result: Succ (Succ (Succ Zero))
x=Succ (Succ (Succ Zero)) ? ;

Result: Succ (Succ (Succ (Succ Zero)))
x=Succ (Succ (Succ (Succ Zero))) ? ;

...
```

And in Erlang:

```
{x = {zero}} | {zero}
{nVar3 = {zero}}, {x = {succ,{var,nVar3}}} | {succ,{zero}}

=ERROR REPORT==== 6-Oct-2008::15:26:00 ===
Too many processes

no_more_solutions
```

As processes are created without control, an infinite branch have been exploited and too many processes created. This case needs more investigation, since the answers are not incorrect, but some search mechanism. It is perfectly possible to control this kind of process explosion and it may be included as future works.

6.5 Conclusion

Unfortunately was not possible to test these implementation in clusters or machines with many cores. But the great news are that the proposal was reached since the system computes perfectly right answers and have the great advantage of being concurrent. We hope we can soon expand the types and functions to have a more complex test measuring the complexity and time performance.

Chapter 7

Conclusions and Future work

In this work, we proposed and implemented a multi-thread evaluator of Curry programs and queries. it was possible by transforming a Curry code in a flat and common language, FlatCurry, by means of Pakcs, a Curry implementation with many back-ends available for translating Curry. Then, after the translation, the code was incremented by a Erlang system to control variables and names substitutions and to evaluate expressions in a concurrent way.

It is the first time that Curry have been translated and evaluated in Erlang. Previous trials were done converting Curry to Prolog, C or Java. But the process communication features of Erlang were not subestimated on this work. They showed to be more simple than expected, and more reliable than it looked. Thus, Erlang was a right choice to make this proposal work and opened the door for many others possibilities and new research.

We consider the results achieved as excellent given the powerful of the recursive multi-thread processing and message passing. Surly this work may derives many others, as adapting the evaluation system for other types, control the spawning of processes and integrate constraints to the Functional Logic Programming.

For covering even more Curry programs, this implementation may include rigid functions and OR trees, dealing better with non-determinism and completing *narrowing* with *residuation*.

Appendix A

exprs.curry

```
data Nat = Zero | Succ Nat

add :: Nat -> Nat -> Nat
add  Zero x = x
add (Succ x) y = Succ (add x y)

mand False _ = False
mand True x = x

eqb True x = x
eqb False False = True
eqb False True = False

func_var x = x

func_var' :: x -> y-> z-> y
func_var' x y z = y

lit_int = 7

lit_float = 3.1416

lit_char = 'c'

comb_func x = lit_int

comb_func' x y z = add x (add y z)
```

```
comb_func'' x = comb_func' x x x

comb_cons = Zero

comb_cons' x = Succ x

comb_cons'' = True

func_let =
  let x = (Succ Zero) in comb_cons' x

func_let' = comb_cons' x
  where x = (Succ Zero)

func_free =
  let x free in comb_cons' x

func_free' =
  comb_func x
  where x free

fcase_flex 1 2 = 3
fcase_flex 1 3 = 4
fcase_flex 2 3 = 5
```

Appendix B

exprs.fcy

```
Prog "exprs"

 ["Prelude"]

 [Type ("exprs","Nat") Public []
  [Cons ("exprs","Zero") 0 Public [],
  Cons ("exprs","Succ") 1 Public [TCons ("exprs","Nat") []]]]

 [Func ("exprs","add") 2 Public
  (FuncType (TCons ("exprs","Nat") [])
   (FuncType (TCons ("exprs","Nat") []) (TCons ("exprs","Nat") []))
  )
  (Rule [1,2]
   (Case Flex (Var 1)
    [Branch (Pattern ("exprs","Zero") []) (Var 2),
    Branch (Pattern ("exprs","Succ") [3])
     (Comb ConsCall ("exprs","Succ")
      [Comb FuncCall ("exprs","add") [Var 3,Var 2]])])),

  Func ("exprs","mand") 2 Public (FuncType (TCons ("Prelude","Bool") [])
   (FuncType (TCons ("Prelude","Bool") []) (TCons ("Prelude","Bool") []))
  )
  (Rule [1,2] (Case Flex (Var 1)
   [Branch (Pattern ("Prelude","False") [])
    (Comb ConsCall ("Prelude","False") []),
   Branch (Pattern ("Prelude","True") []) (Var 2)])),

  Func ("exprs","eqb") 2 Public
```

```
 (FuncType (TCons ("Prelude","Bool") [])
  (FuncType (TCons ("Prelude","Bool") [])
  (TCons ("Prelude","Bool") [])
  )
 )
 (Rule [1,2] (Case Flex (Var 1)
  [Branch (Pattern ("Prelude","True") []) (Var 2),
  Branch (Pattern ("Prelude","False") [])
   (Case Flex (Var 2)
    [Branch (Pattern ("Prelude","False") [])
     (Comb ConsCall ("Prelude","True") []),
    Branch (Pattern ("Prelude","True") [])
     (Comb ConsCall ("Prelude","False") [])
     ]
    )
   ]
  )
 ),

Func ("exprs","func_var") 1 Public (FuncType (TVar 0) (TVar 0))
(Rule [1] (Var 1)),

Func ("exprs","func_var'") 3 Public
(FuncType (TVar 0) (FuncType (TVar 1)
 (FuncType (TVar 2) (TVar 1))))
(Rule [1,2,3] (Var 2)),

Func ("exprs","lit_int") 0 Public (TCons ("Prelude","Int") [])
(Rule [] (Lit (Intc 7))),

Func ("exprs","lit_float") 0 Public (TCons ("Prelude","Float") [])
(Rule [] (Lit (Floatc 3.1416))),

Func ("exprs","lit_char") 0 Public (TCons ("Prelude","Char") [])
(Rule [] (Lit (Charc 'c'))),

Func ("exprs","comb_func") 1 Public (FuncType (TVar 0)
 (TCons ("Prelude","Int") []))
 (Rule [1] (Comb FuncCall ("exprs","lit_int") [])),

Func ("exprs","comb_func'") 3 Public
```

```
 (FuncType (TCons ("exprs","Nat") [])
 (FuncType (TCons ("exprs","Nat") [])
 (FuncType (TCons ("exprs","Nat") [])
 (TCons ("exprs","Nat") [])))))
(Rule [1,2,3] (Comb FuncCall ("exprs","add")
 [Var 1,Comb FuncCall ("exprs","add") [Var 2,Var 3]])),

Func ("exprs","comb_func''") 1 Public
 (FuncType (TCons ("exprs","Nat") []) (TCons ("exprs","Nat") []))
 (Rule [1] (Comb FuncCall ("exprs","comb_func'") [Var 1,Var 1,Var 1])),

Func ("exprs","comb_cons") 0 Public (TCons ("exprs","Nat") [])
 (Rule [] (Comb ConsCall ("exprs","Zero") [])),

Func ("exprs","comb_cons'") 1 Public
 (FuncType (TCons ("exprs","Nat") []) (TCons ("exprs","Nat") []))
 (Rule [1] (Comb ConsCall ("exprs","Succ") [Var 1])),

Func ("exprs","comb_cons''") 0 Public (TCons ("Prelude","Bool") [])
  (Rule [] (Comb ConsCall ("Prelude","True") [])),

Func ("exprs","func_let") 0 Public (TCons ("exprs","Nat") [])
  (Rule []
   (Let [(1,Comb ConsCall ("exprs","Succ")
    [Comb ConsCall ("exprs","Zero") []])]
   (Comb FuncCall ("exprs","comb_cons'") [Var 1]))),

Func ("exprs","func_let'") 0 Public (TCons ("exprs","Nat") [])
 (Rule [] (Let [(1,Comb ConsCall ("exprs","Succ")
   [Comb ConsCall ("exprs","Zero") []])]
   (Comb FuncCall ("exprs","comb_cons'") [Var 1]))),

Func ("exprs","func_free") 0 Public (TCons ("exprs","Nat") [])
(Rule [] (Free [1] (Comb FuncCall ("exprs","comb_cons'") [Var 1]))),

Func ("exprs","func_free'") 0 Public (TCons ("Prelude","Int") [])
(Rule [] (Free [1] (Comb FuncCall ("exprs","comb_func") [Var 1]))),

Func ("exprs","fcase_flex") 2 Public
 (FuncType (TCons ("Prelude","Int") [])
  (FuncType (TCons ("Prelude","Int") []) (TCons ("Prelude","Int") [])))
```

```
   (Rule [1,2] (Case Flex (Var 1)
     [Branch (LPattern (Intc 1)) (Case Flex (Var 2)
       [Branch (LPattern (Intc 2)) (Lit (Intc 3)),
       Branch (LPattern (Intc 3)) (Lit (Intc 4))]),
     Branch (LPattern (Intc 2)) (Case Flex (Var 2)
       [Branch (LPattern (Intc 3)) (Lit (Intc 5))])
     ]
     )
   )
 ]

 []
```

Appendix C

exprs.erl

```
-module(exprs).
-define(Modname, exprs).

-export([add/2, add1/2, mand/2, mand1/2, eqb/2, eqb1/2, eqb2/2,
func_var/1, func_varP/3, lit_int/0, lit_float/0, lit_char/0,
comb_func/1, comb_funcP/3, comb_funcPP/1, comb_cons/0, comb_consP/1,
 comb_consPP/0, func_let/0, func_letP/0, func_free/0, func_freeP/0,
 fcase_flex/2, fcase_flex1/2, fcase_flex2/2, shell/1, eval/3,
 nameserver/1, whilepivot/5]).

add(X1, X2) ->
{1, [{zero}, {succ, {var, nVar3}}], add1}.

add1({zero}, X2) ->
{X2};

add1({succ, X3}, X2) ->
{{succ, [exprs, add, [X3, X2]]}}.


mand(X1, X2) ->
{1, [false, true], mand1}.

mand1(false, X2) ->
{false};

mand1(true, X2) ->
{X2}.
```

```
eqb(X1, X2) ->
{1, [true, false], eqb1}.

eqb1(true, X2) ->
{X2};

eqb1(false, X2) ->
{2, [false, true], eqb2}.

eqb2(false, false) ->
{true};

eqb2(false, true) ->
{false}.


func_var(X1) ->
{X1}.


func_varP(X1, X2, X3) ->
{X2}.


lit_int() ->
{7}.


lit_float() ->
{3.1416}.


lit_char() ->
{'c'}.


comb_func(X1) ->
{[exprs, lit_int, []]}.
```

```
comb_funcP(X1, X2, X3) ->
{[exprs, add, [X1, [exprs, add, [X2, X3]]]]}.


comb_funcPP(X1) ->
{[exprs, comb_funcP, [X1, X1, X1]]}.


comb_cons() ->
{{zero}}.


comb_consP(X1) ->
{{succ, X1}}.


comb_consPP() ->
{true}.


func_let() ->
X1 = {succ, {zero}},
{[exprs, comb_consP, [X1]]}.


func_letP() ->
X1 = {succ, {zero}},
{[exprs, comb_consP, [X1]]}.


func_free() ->
X1 = {var, var_x1},
{[exprs, comb_consP, [X1]]}.


func_freeP() ->
X1 = {var, var_x1},
{[exprs, comb_func, [X1]]}.
```

```
fcase_flex(X1, X2) ->
{1, [1, 2], fcase_flex1}.

fcase_flex1(1, X2) ->
{2, [2, 3], fcase_flex2};

fcase_flex1(2, X2) ->
{2, [3], fcase_flex2}.

fcase_flex2(1, 2) ->
{3};

fcase_flex2(1, 3) ->
{4};

fcase_flex2(2, 3) ->
{5}.



%%%%%%%%%% Evaluation functions %%%%%%%%%%

shell(E) ->
  case is_atom(whereis(root)) of
    false -> unregister(root);
    _ -> true
  end,
  register(root, self()),
  Pns = spawn(?Modname, nameserver, [root]),
  From = spawn(?Modname, eval, [self(), E, Pns]),
  receive_shell(From, Pns).

receive_shell(From, PNs) ->
  receive
    {From, X, Ns} ->
      ppdump(dump(Ns)),
      io:format("~p~n", [X]),
      receive_shell(From, PNs);
    {died, From} ->
      no_more_solutions
  end.
```

```
eval(PPid, [Mod, Func, Args], Pns) ->
  try
    Data =  apply(Mod, Func, Args),
    case Data of
    {R} ->
      apply(?Modname, eval, [PPid, R, Pns]);
    {Pivot, Values, Nfunc} ->
      Dest = spawn(?Modname, whilepivot, [self(), [Mod, Args],
 {Pivot, Values, Nfunc}, Pns, 0]),
      spawn(?Modname, eval, [Dest, lists:nth(Pivot, Args), Pns]),
      whileanswer(PPid, Dest, 0, -1)
    end
  catch
      throw:Term -> Term;
      exit:_ ->
        PPid ! {died, self()};
      error:_ ->
        PPid ! {died, self()}
  end;

eval(PPid, {var, X}, Pns) ->
  R = retrieve(Pns, X),
  case R of
    false ->
      PPid ! {self(), {var, X}, Pns},
      PPid ! {died, self()};
    {bound, Value} ->
      PPid ! {self(), Value, Pns},
      PPid ! {died, self()}
  end;

eval(PPid, {succ, X}, Pns) ->
  spawn(?Modname, eval, [self(), X, Pns]),
  whilesucc(self(), PPid);

eval(PPid, Atom, Pns) ->
  PPid ! {self(), Atom, Pns},
  PPid ! {died, self()}.

whilesucc(From, PPid) ->
```

```
  receive
    {_, X, Pns} ->
      PPid ! {From, {succ, X}, Pns},
      whilesucc(From, PPid);
    {died, _} ->
      PPid ! {died, From}
  end.

whileanswer(PPid, Dest, Rcv, Total) ->
  if
    Rcv == Total ->
      PPid ! {died, self()};
    true ->
      receive
        {died, Dest, Nchild} ->
          whileanswer(PPid, Dest, Rcv, Nchild);
        {_, X, New_ns} ->
          PPid ! {self(), X, New_ns},
          whileanswer(PPid, Dest, Rcv, Total);
        {died, _} ->
          whileanswer(PPid, Dest, Rcv+1, Total)
      end
  end.

whilepivot(Dad, [Mod, Args], {Pivot, Values, Nfunc}, Pns, Nchild) ->
  receive
    {_, {var, X}, New_ns} ->
      NNchild = propag(Values, Dad, [Mod, Nfunc, Args],
  Pivot, {var, X}, New_ns, Nchild),
      whilepivot(Dad, [Mod, Args], {Pivot, Values, Nfunc}, Pns, NNchild);
    {_, A, New_ns} ->
      New_args = change(A, Pivot, Args),
      spawn(?Modname, eval, [Dad, [Mod, Nfunc, New_args], New_ns]),
      whilepivot(Dad, [Mod, Args], {Pivot, Values, Nfunc}, Pns, Nchild + 1);
    {died, _} -> % //there are not more pivots
      Dad ! {died, self(), Nchild}
  end.

propag([V|Vs], Dad, [Mod, Func, Args], Pivot, {var, Var}, Pns, Nchild) ->
  New_ns = spawn(?Modname, nameserver, [Pns]),
  R = assign_or_retrieve(New_ns, Var, V),
```

```
  case R of
    true ->
      New_args = change(V, Pivot, Args),
      spawn(?Modname, eval, [Dad, [Mod, Func, New_args], New_ns]),
      propag(Vs, Dad, [Mod, Func, Args], Pivot, {var, Var}, Pns, Nchild + 1);
    false ->
      propag(Vs, Dad, [Mod, Func, Args], Pivot, {var, Var}, Pns, Nchild)
  end;

propag([], _, _, _, _, _, Nchild) -> Nchild.

change(Val, Pivot, Args) ->
  {Fst, Snd} = lists:split(Pivot-1, Args),
  {_, Tail} = lists:split(1, Snd),
  Fst ++ [Val] ++ Tail.

nameserver(PPid) ->
  regchild(PPid, self()),
  while_ns(PPid, [], []).

while_ns(PPid, CPid, Assigns) ->
  receive
    {getsubs, From} ->
      From ! {list, Assigns, PPid},
      while_ns(PPid, CPid, Assigns);
    {assign, Var, Value} ->
      New_assigns = Assigns ++ [{Var, Value}],
      while_ns(PPid, CPid, New_assigns);
    {retrieve, From, Var} ->
      R = lists:keysearch(Var, 1, Assigns),
      case R of
        {value, {Var, Value}} ->
          From ! {bound, Var, Value, PPid};
        false ->
          From ! {unbound, PPid}
      end,
      while_ns(PPid, CPid, Assigns);
    {child, C} ->
        while_ns(PPid, CPid++[C], Assigns);
    {exit, PPid} ->
      sendexit(CPid)
```

```
  end.

stopns(Ns, PPid) ->
  Ns ! {exit, PPid}.

sendexit([]) -> end_exiting_child;
sendexit([P|Ps]) ->
  stopns(P, self()),
  sendexit(Ps).

regchild(PPid, Ns) ->
  PPid ! {child, Ns}.

dump(Pid) ->
  Pid ! {getsubs, self()},
  receive
    {list, PidList, PPid} ->
      if PPid == root ->
        PidList;
      true ->
        PidList ++ dump(PPid)
      end
  end.

ppdump([]) -> do_nothing;

ppdump([{Var, {Value}}]) -> io:format("{~p = ~p} | ", [Var, Value]);

ppdump([{Var, {Value}}|Vs]) ->
  io:format("{~p = ~p}, ", [Var, Value]),
  ppdump(Vs).

retrieve(Pid, Var) ->
  Pid ! {retrieve, self(), Var},
  receive
    {bound, Var, Value, _} ->
      case Value of
        {var, _} -> false;
        {V} -> {bound, V}
      end;
    {unbound, PPid} ->
```

```
      if PPid == root ->
        false;
      true ->
        retrieve(PPid, Var)
      end
  end.
assign(Pid, Var, Value) ->
  Pid ! {assign, Var, {Value}}.

assign_or_retrieve(Pid, Var, Value) ->
  case retrieve(Pid, Var) of
    false -> assign(Pid, Var, Value), true;
    {bound, Value} -> true;
    {bound, _} -> false
  end.
```

Bibliography

[AH94] Sergio Antoy and Michael Hanus. A needed narrowing strategy. In *Journal of the ACM*, pages 268–279. ACM Press, 1994.

[Car01] Julio Mariño Carballo. *Semantics and Analysis of Functional Logic Programs*. PhD thesis, 2001.

[GM05] Emilio Jesús Gallego Arias and Julio Mariño. An overview of the Sloth2005 Curry system: system description. In *WCFLP '05: Proceedings of the 2005 ACM SIGPLAN workshop on Curry and functional logic programming*, pages 66–69, New York, NY, USA, 2005. ACM Press.

[Han] Michael Hanus. Curry: An integrated functional logic language. Available at http://www.informatik.uni-kiel.de/ mh/curry/.

[Han02] Michael Hanus. Distributed programming in a multi-paradigm declarative language. In *In Proc. of the International Conference on Principles and Practice of Declarative Programming (PPDP'99*, pages 376–395. Springer LNCS, 1702.

[Han94] Michael Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19:583–628, 1994.

[Han97] Michael Hanus. A unified computation model for functional and logic programming. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 80–93, New York, NY, USA, 1997. ACM.

[Han07] Michael Hanus. Multi-paradigm declarative languages. In *In Proceedings of the International Conference on Logic Programming (ICLP 2007)*, pages 45–75. Springer LNCS, 2007.

[HIA97] Michael Hanus, Informatik Ii, and Rwth Aachen. A unified computation model for functional and logic programming. In *In Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris*, pages 80–93. ACM, 1997.

[HKMN95] M. Hanus, H. Kuchen, and J. Moreno-Navarro. Curry: A truly functional logic language. In *In Proc. ILPS'95 Workshop on Visions for the Future of Logic Programming*, 1995.

[HP03] Michael Hanus and Christian Prehofer. Higher-order narrowing with denitional trees. In *In Proc. Seventh International Conference on Rewriting Techniques and Applications (RTA'96*, pages 13815–2. Springer LNCS, 1103.

[TA03] Andrew Tolmach and Sergio Antoy. A monadic semantics for core curry. In *In Proc. of WFLP 2003*. Elsevier, 2003.