CHALMERS

Very Lazy Evaluation

A new execution model for functional programming languages

Master of Science Thesis in Computer Science and Engineering

JAN ROCHEL

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Göteborg, Sweden, 2009-06-18

Very Lazy Evaluation (version 0.7.1)
A new execution model for functional programming languages

Jan Rochel

Examiner: Patrik Jansson

Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Göteborg, Sweden, 2009-06-18

Very Lazy Evaluation (version 0.7.1)

Jan Rochel *2009-06-12*

Acknowledgements

I would like to express my gratitude to both Carsten Sinz and Patrik Jansson not only for supervising my work, but also for giving me the excellent opportunity to carry out this project as a diploma thesis in the first place.

Abstract

In the recent years a multitude of functional language implementations has been developed, whereby those being intended for practical use share a very fundamental property: The execution models employed by implementations designed for efficient program execution are all based on some form of graph-reduction. This thesis introduces a new execution model for strongly-typed, higher-order, non-strict, purely-functional programming languages, that does not rely on graph-reduction but a new concept called *very lazy evaluation*. It uses an unconventional approach to evaluate expressions of the λ-calculus, that differs considerably from traditional term rewrite systems. Its method to perform function applications allows arguments to be handled in a way, that is more lazy than in existing graph-reduction based models. This leads to a new type of abstract machine, which promises very efficient program execution and might also be quite suitable to be implemented in hardware. A proof-of-concept implementation of the execution model is given, which consists of a compiler generating abstract machine code from a source program, and the abstract machine that interpretes the result. By using it to execute real programs it is shown, that the correct results are evaluated and that the approach therefore is a valid alternative to existing execution models.

Contents

1 Introduction

In this thesis a new execution model for functional programming languages is presented. A system is specified for executing programs written in a purely functional language. Such a system typically comprises a compiler for translating a program from the source language into a machine language and an abstract machine, which interpretes the machine language thus executing the program.

In contrast to already existing functional language implementations a new approach is chosen based on a new concept called *very lazy evaluation*. It differs considerably from previous execution models as it does not rely on term-rewriting but a more abstract mode of operation in which function arguments are treated more lazily.

1.1 Scope

The source languages covered by this model are *strongly-typed*, *higher-order*, *non-strict*, *purely-functional* languages (*2. The Source Language*) such as Haskell on which we focus.

The intention is to give a detailed explanation of the underlying concept as well as an accurate description of the system (*4. Very Lazy Evaluation*).

To show how it differs from existing concepts, at first a short overview over execution models (*3. Conventional Execution Models*) is given that are used in today's functional language implementations.

A formal specification of the abstract machine and the abstract machine language is presented (*5. Model Specification*), which is to constitute a solid foundation for a novel class of efficient implementations for functional programming languages.

A proof-of-concept implementation of the system's components is provided (*6. Implementation*) to show the applicability of the approach. The compiler processes Haskell as a source language and is written in Haskell 98, the interpreter in C. The implementation might help to estimate the performance that is to be expected from the model in comparison to existing implementations.

Finally, possibilities of further research are outlined that are opened by the introduced execution model as well as potentialities for optimisations and extensions of the presented system (*7. Perspectives*).

1.2 Motivation

In the last decades a lot of research work has been put into the compilation and execution of functional languages resulting in a number of different abstract machines such as the SECD [P. J. Landin (1963)], the CAM [G. Cousineau (1985)], the CMCM [Simon Thompson (1992)], the TIM [Jon Fairbairn (1987)], the ZAM [Xavier Leroy (1990)], the Krivine-machine [Cregut Pierre (1991)], the G-machine [Simon L. Peyton Jones (1987)], and the spineless tagless G-

machine (STG-machine) [Simon L. Peyton Jones (1992)] plus various adaptions [Ian Holyer (1998), Daan Leijen (2005)]. The attempt to make another contribution to this list is motivated by the fact that the presented model stands out from the previous ones. While the latter do differ amongst each other by various degrees, they share one fundamental property: Ultimately, they implement the λ-calculus by term-rewriting, either relying on the *push/enter* or the *eval/apply* model [Simon Marlow (2006)] to access function parameters. Here, this is solved by a *serve/request* mechanism in which function parameters are treated more lazily. This leads to a new type of execution model.

It is highly interesting how it performs (in terms of run-time efficiency) on actual machines compared to other methods. In particular the increased laziness of the model promises some potential in this respect. Currently, all notable Haskell implementations are more or less based on the STG-machine, thus one can fairly assume, that it is of the models above the one most suited for efficient execution on current computer systems. Therefore, to illustrate the characteristics of the presented model, in this thesis often comparisons with the well-established STG-machine are given. Although the match is far from fair, the most plausible competitor in terms performance is the highly optimized Glasgow Haskell Compiler GHC, `http://www.haskell.org/ghc/`).

Finally, the abstractness of the model holds its own beauty. While previous designs based on the concepts of term-rewriting and graph-reduction directly implement the λ-calculus by repeated application of reduction rules, which leads to iterated expression substitution, here a different approach is devised. It is due to the increased abstraction, that the abstract machine is on the one hand surprisingly simple and thus easy to implement (notably even in hardware), but at the same time for the same reason more difficult to understand.

Since it promises to allow execution on very simple architectures it might open up interesting perspectives in the field of embedded computing but could also motivate research in the areas of parallel computing, system security, and virtualisation (see *7. Perspectives*)

2 The Source Language

The execution model can be applied to any higher-order, strongly-typed, non-strict, purely-functional language. The most relevant member of this set of languages is Haskell. Other languages are among others its derivatives such as Agda-2 or Clean. But also (enriched) λ-calculus variants are supported.

2.1 Abstract Syntax

To cover this whole class of languages, we consider the abstract syntax below (Figure 1), which defines a simple untyped functional language. Bare of any dispensable language-specific constructs such as comments, parameter patterns for function definitions, or type-classes, it is still sufficiently expressive to act as a qualified representative for this class. Abstracting away from a concrete syntax helps to focus on the essence of the model rather than on tedious compilation details. Intermediate languages used by various compilers are often of similar appearance.[1]

To avoid defining a concrete syntax for the source language, we use Haskell 98 to denote the source code of example programs throughout this document.

$program$	$\rightarrow$	$datatype^*\ function^*$
$datatype$	$\rightarrow$	$constructor^*$
$constructor$	$\rightarrow$	*constructor-name parameter-name**
$function$	$\rightarrow$	*function-name parameter-name** $expression$
$expression$	$\rightarrow$	$variable \mid application \mid$ *case-discrimination* $\mid$ *let-expression* $\mid integer \mid float$
$variable$	$\rightarrow$	*constructor-name* $\mid$ *function-name* $\mid$ *parameter-name* $\mid primitive$
$application$	$\rightarrow$	$expression^+$
case-discrimination	$\rightarrow$	$expression\ alternative^*$
$alternative$	$\rightarrow$	$pattern\ expression$
let-expression	$\rightarrow$	$function^*\ expression$
$pattern$	$\rightarrow$	*constructor-name parameter-name*$^* \mid integer \mid$ `default`

Figure 1: Abstract syntax of the source language as a context-free grammar

Because correctness is guaranteed by the type-checker before translating the source program into this language, it causes no harm to leave out type information completely.[2] Ad-hoc polymorphism[3] can be resolved to ordinary case-discrimination.

[1] This is a very useful feature, because that makes it easy to use existing compilers as a front-end for the implementation of the execution model (see *6.1. The Compiler*).

[2] Therefore datatype definitions do not include a datatype name, since this is just type-information.

[3] polymorphism involving different behaviour depending on the concerning type

Many language elements commonly used in functional programming have been omitted in favour of others. In order to minimise the language, it is sensible to retain only one of two language elements that are capable of expressing the same concept.

Table 1: Some omitted language elements known in Haskell and the corresponding representation

λ-abstraction ($\lambda x_1 \ldots x_n . e$)	let-expression (let $f\ x_1 \ldots x_n = e$ in f)
function parameter pattern	case-discrimination
infix operator (`x + y`)	function application (`plus x y`)
character literals	either as integers or constructors
`if-then-else`-expression	case-discrimination

Higher-level constructs such as type-classes or popular language extensions are not discussed here, but are indirectly supported by the model, since it should be possible to eliminate them by a source-to-source transformation.

2.2 Semantics

The most common perception of a program given in this (or any other functional) language is to interprete it as a single λ-expression. For that purpose every function definition can be regarded as a named λ-abstraction. Beginning from a well-defined entry point (the `main`-function in Haskell) by recursively expanding every function application to the corresponding λ-abstraction, a (possibly infinitely large)[4] λ-expression is obtained.[5]

The source language merely enriches the expressiveness of the pure λ-calculus by a few additional language constructs. In the above language these are three concepts: Constants (integers, floats, and constructors), case-discrimination, and primitives. Although the λ-calculus is powerful enough to express case-discrimination and integer-arithmetics using the Church-encoding for numerals and booleans, it is not possible to implement this representation efficiently. Primitives give access to the underlying architecure's capacities (like I/O, interaction with the operating system, or arithmetic functions), and can also be used to force strictness[6].

We conclude, that the function definitions constituting a program, ultimately define a λ-expression, thus the program is executed by evaluating this expression. Therefore, the execution model of a functional language implementation is characterised by the very method it employs for processing a given λ-expression.

[4] if recursion or mutual recursion occurs, which should be the case for non-trivial programs

[5] see [Zena M. Ariola (1994)]

[6] In this document the term *strictness* is synonymic to *eagerness*

3 Conventional Execution Models

This section provides a basic insight into the fundamental approach used by current functional language implementations, in order to exhibit the uniqueness of the approach used by the presented execution model. Therefore, the explanations do not go deeper than it is necessary to serve that purpose.

3.1 Reduction

The natural way to evaluate λ-expressions is by applying the rules of the λ-calculus[7], which defines an equivalence relation on λ-expressions. The conversion of an expression into another is defined to be valid, if the expressions are equivalent. The λ-calculus defines three transformation rules: α-conversion, β-reduction, and η-conversion.

In a compiled setting, the abstract machine never needs to perform α-conversion. In the λ-calculus it is only required to prevent clashes of different variables with the same name. In an efficient implementation, the target language interpreted by the abstract machine does not reference variables by name, but rather relies on some unambiguous, numeric kind of identifier. Variable names are already resolved during the compilation of the program, which could be regarded as a form of static application of the α-conversion rule.

Also η-conversion does not play a too important role in this context. Some functional language implementations do not need to perform η-conversion at all, but even if it is done, it does not occur very frequently.

This leaves β-reduction as the central mechanism of the evaluation process. Indeed, the primary task of conventional abstract machines is to repeatedly perform β-reduction in order to transform the λ-expression step-by-step to its *normal form* (if existent), where the evaluation terminates.

3.2 Lazy Evaluation

These seem to be quite definite directions for the abstract machine, but an important aspect of the evaluation is still unsettled: the reduction order. In a λ-expression, each saturated λ-abstraction is a valid candidate for applying β-reduction. However this question has already been dismissed by limiting the scope only to non-strict (*lazy*) languages. Lazy evaluation implies, that arguments are not evaluated and is achieved by *normal order reduction*[8].

Normal order reduction states, that in a λ-expression $(f\ x\ y)$ always the *leftmost outermost* part (f) is to be considered for β-reduction. If the β-reduction rule is applicable that way, we speak of a *reducible expression* (or *redex*). An expression that can not be reduced by normal order reduction is in *weak head normal form* (*WHNF*).

[7] see [Peter Sestoft (2002)]

[8] see [Simon L. Peyton Jones (1987), 2.3. Reduction Order, p 23 ff]

3.3 Call-by-Need Strategy

Furthermore we postulate the *call-by-need* evaluation strategy, which amends lazy evaluation by the assertion, that a named expression that occurs multiple times in a certain context is evaluated at most once. This is an adequate choice for an implementation that is to be taken seriously. To give this guarantee, an evaluation model needs to include a *sharing*-mechanism, which preserves the evaluated form of a shared expression to deploy it in case of subsequent uses. (Sharing may not be considered a merely optional optimisation. In a functional language without a sharing mechanism, for any expression that is used multiple times it would be necessary to evaluate it for each occasion that its value is required.)

3.4 Interim Conclusion

Summarising, program execution corresponds to a *term rewrite system*, implemented by repeatedly performing normal order reduction to WHNF on a given redex. This imposes a strong assumption on the execution model's abstract machine design: Its configuration basically is a representation of the *current* λ-expression. In each step the abstract machine applies a transformation rule to it, the resulting expression being the *new* current expression of the abstract machine. This constricts the design space to only a few remaining aspects:

- term representation, i.e. how the abstract machine state (the current λ-expression) is encoded
- how β-reduction is performed efficiently on that representation
- sharing: how a computation result can be preserved and reused later

3.5 Graph Reduction

In all the practically relevant functional language implementations terms are represented as a graph. The nested structure of a λ-expression can be interpreted as a tree. Sharing implies, that a set of tree nodes, each representing the same shared expression, which occurs multiple times in the term, can be merged into a single node. Such a node then has more than one ancestor and the tree becomes a graph. A graph node is kept in a data structure called *closure* (or *thunk*). Program execution is achieved by performing β-reduction on this graph. This particular form of term rewriting is called *graph-reduction*.

In a real implementation on a system with finite resources the program graph can not be maintained in its expanded form.[9] Therefore the procedure of graph-reduction really is an alternation of expansions and reductions. The expansion of a reference to a function takes place by allocating a closure for the function definition, which holds its own bound variables. Depending on the implementation

[9] We keep in mind, that the expression might very well be infinitely large

this is either achieved by copying the function definition (which is held in form of a closure) or by building a closure from scratch by executing the function definition (which consists of the instructions to build the closure).

3.6 Argument Handling

The starting point of a β-reduction is a λ-abstraction applied to a number of arguments: $(\lambda x_{1 \ldots} x_n.\, e)\ a_{1 \ldots} a_m$. It is essential for the efficiency of current implementations not to treat such a λ-abstraction with multiple arguments as a cascade of applications of single-argument abstractions $\lambda x_1.\, (\lambda x_2.\, (\ldots (\lambda x_n.\, e) \ldots))\ a_{1 \ldots} a_m$. This approach was chosen by the G-machine, and rectified by the STG-machine.

In a higher-order language it can usually not be statically determined for a application $f\ a_{1 \ldots} a_m$ occuring somewhere in the program definition, how many arguments are expected by f.[10] For $f = \lambda x_{1 \ldots} x_n.\, a_{1 \ldots} a_m$ both $m > n$ (*oversaturation*) and $m < n$ (*undersaturation*) may occur. Therefore the model must include a mechanism to ensure, that always the right number of arguments is applied to a λ-abstraction.[11]

There are two strategies to pass arguments, both of which are used in existing implementations:[12]

- Push/Enter: The arguments are pushed on an argument stack before the function is entered. The function, which statically knows its own *arity* (parameter count), checks if enough arguments are present, and if so, takes the correct number of arguments from the stack and performs the evaluation.

- Eval/Apply: Before a function is applied, the caller must examine the function closure in order to determine its arity. Then the correct number of arguments are passed (possibly using registers and the stack) according to an specific argument passing convention (similarly as in imperative languages) and the function is called, knowing where its parameters are stored.

3.7 Free Variables

There are two methods to cope with free variables. One is to avoid free variables completely, which is achieved by a source-to-source transformation of the λ-calculus called λ-*lifting*[13]. It raises all λ-abstractions to top-level, binding free variables to additional parameters.

Relevant implementations however, do generally not perform λ-lifting, but rather bind them at runtime in closures. Every time a closure is allocated, for each

[10] In higher-order languages, f may be a parameter as in `flip f x y = f y x`. Therefore it is not clear at compile-time, which concrete function is applied on the right-hand side.

[11] In case of undersaturation the expression is already in WHNF.

[12] More detailed explanations and a comparison between the two strategies is given in [Simon Marlow (2006)]

[13] see [Simon L. Peyton Jones (1987), 13. Supercombinators and Lambda-Lifting, p 220 ff]

of its free variables, a value needs to be supplied. These values are stored at a predefined location in the closure and are accessible for the closure once it is being entered.[14]

3.8 Summary

This section's explanations can be rakishly subsumed in the formulars below. They define a rough schema of today's functional language implementations.

- program definition = λ-expression
- program execution = evaluation of the λ-expression
- evaluation of λ-expression = repeated β-reduction
- abstract machine state = current λ-expression
- term representation as graph → graph reduction
- lazy evaluation = normal order reduction
- β-reduction = push/enter or eval/apply

In the next section we will see how the presented execution model stands out from previous ones as it does not quite fit into the above schema. It relies on another more abstract approach, which emerges from lifting the concept of lazy evaluation to the abstract machine.

[14] see [Simon L. Peyton Jones (1992), 4. The STG Language, p 19 ff]

4 Very Lazy Evaluation

This section is dedicated to explain the idea behind the presented execution model and its implications for the design of the abstract machine. One very important characteristic is how function arguments are treated more lazily than in the push/enter or eval/apply models, which serves well as an entry into the matter.

4.1 Lazy Evaluation, revisited

Lazy evaluation describes the notion of delaying the evaluation of an expression to the moment its result is required. In conventional models this is achieved by normal order reduction of an expression to WHNF, with the result, that the arguments of the current expression are not evaluated. From a more abstract perspective, lazy evaluation connotes the idea to delay a task to the latest moment possible, speculating for the eventuality, that this moment never occurs, because the task has vanished somehow. In that case the effort of attending to the task is saved.

```
main = flip const a id b
flip f x y = f y x
const x y = x
id x = x
```

Figure 2: Trivial program

We will now see how we can squeeze out an additional portion of laziness by that idea, and thereby save some redundant effort. Therefore, we use an example (Figure 2) to try to identify a source of spare strictness in term-rewriting-based evaluation models.

```
main
-> flip const a id b
-> const id a b
-> id b
-> b
```

Figure 3: Normal order reduction

To avoid dealing with specifics of the Haskell standard library, we ignore the fact, that the main function does not have the `IO()`-type expected from a valid Haskell program. Normal order reduction (Figure 3) terminates after performing four β-reductions.[15]

It is due to lazy evaluation, that `a` never needs to be evaluated, since the `const`-function drops its second parameter. Despite the fact, that `a` is never used, it is

[15]The expansion from `main` to `flip const a id b` is not really a *reduction*, but can be treated as one, if it is regarded as a λ-abstraction without parameters.

evoked by `main`, passed on to `flip`, rearranged in the subsequent β-reduction, just to be discarded by `const` in the end. This is just the kind of redundant effort that one would expect to be avoided by laziness. Here it occurs here because arguments are handled in a strict way: They are passed just as soon as they are available, rather than at the moment they are required.

The latter approach is pursued our model. It avoids the redundant effort that occurs because arguments are evoked too early. It does not serve arguments on a silver plate for functions to consume them.[16] Instead, functions as soon as they require a parameter have to go fetch it. This way the effort of passing arguments is delayed to the moment the parameter is really accessed, and does not occur at all if the access does neither.

At first glance, the saved expense might seem rather low (after all maintaining arguments does not imply *evaluating* them). But a new argument often requires the allocation of a closure on the heap.

4.2 A new Model

We have shown, how (lazy) parameter retrieval promises the prospect of reduced evaluation costs compared to the conventional (strict) argument passing methods. Let us now see how this is realised in the presented execution model and how this appoints major characteristics of the abstract machine.

We begin the evaluation by examining the function definition of `main`. But opposed to the conventional models, the arguments of the expression on its right-hand side are not considered. This will not be done before absolutely necessary, so for now, only the *leftmost* portion is used[17]. In the example above (Figure 2) instead of rewriting the term `main` as `flip const a id b` only `flip` as the leftmost portion of the expression is considered. That implies, that `main` may not be discarded, otherwise `flip`'s arguments (`const`, `a`, and `b`) would be lost.

4.3 Evaluation Stack

Therefore the original abstract machine state (initially only consisting of `main`) has to be transformed to a configuration containing both `flip` and `main` (for supplying arguments to `flip` when required). For that purpose the presented abstract machine maintains a stack, the *evaluation stack*, which is its primary data structure.[18] It contains a sequence of *stack tokens*, for now we assume solely *function tokens*. A function token represents an instance of a specific function and is a simple reference to its definition. We denote the evaluation stack in parentheses growing from right

[16] In the push/enter model, the function itself still needs to pick the correct amount of parameters off the silver plate (argument stack), while in the eval/apply model the correct amount is served in the most convenient manner.

[17] While this method is not based on *reduction*, it still pursues to proceed by *normal order*

[18] Whenever the term "the stack" is used, the evaluation stack is meant.

to left.[19] In the first state transition the initial state (`main`) is transformed to (`flip main`). This notation is by no means equal to the *expression* `flip main`. It rather expresses the situation, that the the function definition of `main` has been *partially* enforced, `flip` being the candidate for the next step. The transition is intuitively written as (`main`) → (`flip main`).

The evaluation procedure takes place in a sequence of steps, where in each step the topmost token of the evaluation stack is examined (without taking it off the stack) and (if it holds a function token) the leftmost portion of the associated definition is scrutinised, and action is taken depending on its value.

4.4 Parameters

The current configuration is interesting, because the leftmost portion of the topmost token `flip` is `f`, a function parameter, more specifically the first parameter in `flip`'s parameter list (`f`, `x`, `y`).

The ordinary argument passing rule for functions (in mathematics and most programming languages) states, that the nth parameter of a function correpsonds to the nth argument passed by its caller, in other words: Parameters are identified with their corresponding arguments by position.

The caller of `flip` is `main`, so the first parameter `f` of `flip` is identical with the first argument `const` in the definition of `main`. Just as in the preceding transition, we push a new function token on the stack, which corresponds to the leftmost part of the topmost function token: (`flip main`) → (`const flip main`). Only this time one level of indirection is required to determine the respective function.

In general, the method to access the nth parameter of a function token is to *request* the nth argument of its *predecessor* (the token immediately to the right on the evaluation stack). This works, because a function token is always pushed on the stack as the leftmost part of the previously topmost token. Thus it is always adjacent to its caller and the arguments of the caller correspond directly to its parameters.[20]

4.5 Evaluation Stack, revisited

For this method it is necessary to be able to read items *within* the evaluation stack, which contradicts the conventional conception of the stack data structure. Therefore the evaluation stack is a more potent data structure. Later we will see how it is in fact a hybrid of a heap and a stack, as it is destined to render a conventional heap obsolete.

[19] The terms "on top of" is equivalent to "left of", such as "topmost" can be used instead of "leftmost", etc.

[20] In case of unsaturated or oversaturated function calls, not all arguments are imperatively supplied directly by the caller (see *4.7. Currying*).

4.6 Registers

In the current abstract machine state (`const flip main`), the next evaluation step is similar to the previous one. Leftmost part of `const`'s definition is its first parameter, namely `x`. To obtain it, the first argument is requested from `const`'s predecessor `flip`. But this argument again is a parameter (`y`, third parameter of `flip`), so another request has to be performed to gather the third argument `id` of `main`, which is then put on the stack: (`const flip main`) $\rightarrow$ (`id const flip main`).

Here, for an update of the evaluation stack to occur, two requests had to be issued. In more complex cases, a complete cascade of requests might be necessary. Therefore it is sensible to model additional intermediate states between stack updates.

$(\texttt{A}_0, 1, \texttt{main}) \rightarrow$
$(\texttt{A}_0, 2, \texttt{flip main}) \rightarrow$
$(\texttt{A}_1, 1, \texttt{flip main}) \rightarrow$
$(\texttt{A}_0, 3, \texttt{const flip main}) \rightarrow$
$(\texttt{A}_1, 2, \texttt{const flip main}) \rightarrow$
$(\texttt{A}_3, 1, \texttt{const flip main}) \rightarrow$
$(\texttt{A}_0, 4, \texttt{id const flip main}) \rightarrow$
$(\texttt{A}_1, 3, \texttt{id const flip main})$

Figure 4: Trivial program evaluated: first part

For this purpose, two registers are employed, the *status register* (SR) and the *stack position register* (SP). The former expresses which operation is to be performed by the abstract machine during the next step, the latter addresses the stack token referenced by the operation. For now we know only of one operation, the argument request, which we denote by $\texttt{A}_i$, requesting the ith argument of the token addressed by SP. SP holds a natural number and addresses the SPth token on the evaluation stack (ES), the bottom token having the address 1. Whenever a function token is pushed, SP is adjusted to point to this new token, and SR is set to $\texttt{A}_0$. The leftmost portion of an expression is addressed as the 0th argument.

We extend the configuration of the abstract machine from (ES) to (SR, SP, ES). The evaluation of our example can then be depicted more fine-grained (Figure 4).

4.7 Currying

At this point however, the request for the first argument ($\texttt{A}_1$) can not be *served* by the token `const` at $SP = 3$, as its definition does not have an argument. This happens whenever unsaturated function applications occur, in this example `const` supplies not enough arguments to `id`. This can usually be remedied by *currying*, whenever

there exists a corresponding oversaturation. Here this is the case, as `main` supplies 4 arguments to `flip` while the latter has only an arity of 3.

$(\mathtt{A}_1, 3, \texttt{id const flip main}) \rightarrow$
$(\mathtt{A}_3, 2, \texttt{id const flip main}) \rightarrow$
$(\mathtt{A}_4, 1, \texttt{id const flip main}) \rightarrow$
$(\mathtt{A}_0, 5, \texttt{b id const flip main})$

Figure 5: Example evaluated: last part

The curry-mechanism of the presented abstract machine consists of a simple rule, which ensures that an unsatisfied request is redirected, such that it eventually ends up at the corresponding oversaturation: $(\mathtt{A}_i, SP, ES) \rightarrow (\mathtt{A}_{i-a+p}, SP-1, ES)$, where a is the argument count and p the arity of the function token at SP. After applying it twice, the evaluation can gently be completed (Figure 5).

4.8 Interim Conclusion

In this very simple example the basic procedure of the presented execution model could be depicted, but fundamental mechanisms for simple issues such as case-discrimination and let-expressions or more elaborate solutions for sophisticated problems like sharing are still to be discussed. Nevertheless some interesting features of the model could already be observerd:

- The procedure is not based on term-rewriting or ordinary λ-conversion rules. The evaluation can rather be seen as a series of partial function applications.
- Opposed to the conventional models, the state of the abstract machine is not equal to “the current expression”, or at least it relies on a much more abstract representation.
- As promised, the rendundant effort of dealing with arguments that are never required is avoided. The argument `a` of main remains untouched throughout the evaluation!
- The approach heavily relies on *requests* addressed at function tokens, which can be *served* if the corresponding function definition has enough arguments.

4.9 Subfunctions

One fundamental language construct did not appear in the example above at all: nested expressions. Whenever an *application* in the source language does not exclusively consist of atomic components (*variable*, *integer*, *float*), we speak of a *nested expression*.

During compilation from the source language to the target language, non-atomic sub-expressions are factored out into seperate function definitions, so called *subfunctions*. This leads to a flattened structure of the program definition, where the components of the function definitions are atomic. That way originally nested parts of an expression can be handled by the request-mechanism just like before by pushing the corresponding subfunction token on the evaluation stack. Consider the function definition `const' x y = id (flip const x (id y))`.

```
const' x y = id flip_const_x_id_y
flip_const_x_id_y = flip const x id_y
id_y = id y
```

Figure 6: Unnested form

If the outsourcing of nested expressions is performed without any precautions (Figure 6) a new problem arises: Previously bound variables might be torn out of their context. While in `const'` the variables `x` and `y` are bound, in `flip_const_x_id_y` and `id_y` they occur as free variables. Only `const'` is able to obtain these parameters by performing a request to its predecessor token, the two subfunctions do not have that possibility.[21] We therefore need a mechanism, by which `flip_const_x_id_y` and `id_y` can access the parameters of `const'`.[22]

We call `const'` the *parent* of `flip_const_x_id_y`, which in return is the *child* of `const'`. The above problem can be solved by redirecting parameter requests addressed at a subfunction token to its parent token, or in case of multi-level nestings to one of its *ancestor* tokens. To perform this redirection, the parent-child relationship needs to be articulated in the evaluation stack. This is achieved by establishing for each emerging subfunction token a reference (*parent-edge*) to the corresponding parent token. This is simple, because a subfunction can only be evoked due to a request served by its parent. That way even in the case of multi-level nestings, for every subfunction token, a connection to all its ancestors is guaranteed, through a cascade of parent-edges. Subfunctions can follow these edges to locate parameters that belong to an ancestor.

From now on we denote a function token at the stack position a that belongs to the function definition of f and has a parent-edge to the stack position p by ${}_a\mathtt{F}_p^f$.

4.10 Parameters, revisited

The subfunction approach implies, that to unambigously denote a specific parameter not only its index (position within the parameter list) needs to be defined, but also the function by which the parameter variable is bound. We demonstrate the

[21] The subfunctions may be requested from `const'` much later, and therefore be positioned at a very different location in the stack.

[22] A valid solution would be to perform λ-lifting, which however leads to very inefficient results.

parameter notation in the target language by taking the compilation of the example (Figure 6) one step further (Figure 7). We also include the definitions of `flip`, `const`, and `id`.

```
const' 2 = id flip_const_x_id_y
flip_const_x_id_y 0 = flip const P_1^const' id_y
id_y 0 = id P_2^const'
flip 3 = P_1^flip P_3^flip P_2^flip
const 2 = P_1^const
id 1 = P_1^id
```

Figure 7: Unnested form with correcly resolved parameters

The integer that follows each function name in each definition denotes the arity of the function. It is always 0 for subfunctions. The symbol $\mathtt{P}_i^f$ simply denotes the f's ith parameter. Whenever such a parameter atom is served, the abstract machine goes into a corresponding parameter state $SR = \mathtt{P}_i^f$.

If SP addresses the correct function f in which the parameter was bound, the corresponding argument is requested ($\mathtt{A}_i$) from the predecessor. Otherwise $\mathtt{P}_i^f$ represents a free variable and is bound somewhere in the surrounding environment reachable through the parent edge of the function token at SP. The process of following parent-edges until f is reached is called *backtracking*. In the evaluation (Figure 8) of the example above (Figure 7 the new mechanisms can be seen at work.

We see, that in this execution model it is quite hard to follow the evaluation even for a very simple example. To understand this sequence it might be useful to consult the first few of the inference rules given below (Figure 17). Just a few central aspects are to be pointed out here:

- Whenever a token is pushed on the stack, the parent edges are easily established by using SP as the pointer value.
- Backtracking takes place in the last four lines, where the SP-register is decreased from 6 to 3 and then to 1. This is where `id_y` requests a parameter of its second-generation ancestor (`const'`), therefore the parent-edge has to be followed twice, until it is addressed to the right token.
- The sequence of the pushed functions (can be obtained by reading the tokens in the last evaluation stack from right to left) corresponds exactly to the sequence of function applied by normal order reduction: `const' x y` $\rightarrow$ `id flip_const_x_id_y` $\rightarrow$ `flip_const_id_x_id_y` $\rightarrow$ `flip const x id_y` $\rightarrow$ ...
- The evaluation terminates, because in the last line SP points outside the evaluation stack. This indicates, that the WHNF of the evaluated expression

$$
\begin{array}{l}
(\mathtt{A}_0, 1, {}_1\mathtt{F}_0^{const'}) \longrightarrow \\
(\mathtt{A}_0, 2, {}_2\mathtt{F}_1^{id}{}_1\mathtt{F}_0^{const'}) \longrightarrow \\
(\mathtt{A}_0, 2, {}_2\mathtt{F}_1^{id}{}_1\mathtt{F}_0^{const'}) \longrightarrow \\
(\mathtt{P}_1^{id}, 2, {}_2\mathtt{F}_1^{id}{}_1\mathtt{F}_0^{const'}) \longrightarrow \\
(\mathtt{A}_1, 1, {}_2\mathtt{F}_1^{id}{}_1\mathtt{F}_0^{const'}) \longrightarrow \\
(\mathtt{A}_0, 3, {}_3\mathtt{F}_1^{flip\_const\_x\_id\_y}{}_2\mathtt{F}_1^{id}{}_1\mathtt{F}_0^{const'}) \longrightarrow \\
(\mathtt{A}_0, 4, {}_4\mathtt{F}_3^{flip}{}_3\mathtt{F}_1^{flip\_const\_x\_id\_y}{}_2\mathtt{F}_1^{id}{}_1\mathtt{F}_0^{const'}) \longrightarrow \\
(\mathtt{P}_1^{flip}, 4, {}_4\mathtt{F}_3^{flip}{}_3\mathtt{F}_1^{flip\_const\_x\_id\_y}{}_2\mathtt{F}_1^{id}{}_1\mathtt{F}_0^{const'}) \longrightarrow \\
(\mathtt{A}_1, 3, {}_4\mathtt{F}_3^{flip}{}_3\mathtt{F}_1^{flip\_const\_x\_id\_y}{}_2\mathtt{F}_1^{id}{}_1\mathtt{F}_0^{const'}) \longrightarrow \\
(\mathtt{A}_0, 5, {}_5\mathtt{F}_3^{const}{}_4\mathtt{F}_3^{flip}{}_3\mathtt{F}_1^{flip\_const\_x\_id\_y}{}_2\mathtt{F}_1^{id}{}_1\mathtt{F}_0^{const'}) \longrightarrow \\
(\mathtt{P}_1^{const}, 5, {}_5\mathtt{F}_3^{const}{}_4\mathtt{F}_3^{flip}{}_3\mathtt{F}_1^{flip\_const\_x\_id\_y}{}_2\mathtt{F}_1^{id}{}_1\mathtt{F}_0^{const'}) \longrightarrow \\
(\mathtt{A}_1, 4, {}_5\mathtt{F}_3^{const}{}_4\mathtt{F}_3^{flip}{}_3\mathtt{F}_1^{flip\_const\_x\_id\_y}{}_2\mathtt{F}_1^{id}{}_1\mathtt{F}_0^{const'}) \longrightarrow \\
(\mathtt{P}_3^{flip}, 4, {}_5\mathtt{F}_3^{const}{}_4\mathtt{F}_3^{flip}{}_3\mathtt{F}_1^{flip\_const\_x\_id\_y}{}_2\mathtt{F}_1^{id}{}_1\mathtt{F}_0^{const'}) \longrightarrow \\
(\mathtt{A}_3, 3, {}_5\mathtt{F}_3^{const}{}_4\mathtt{F}_3^{flip}{}_3\mathtt{F}_1^{flip\_const\_x\_id\_y}{}_2\mathtt{F}_1^{id}{}_1\mathtt{F}_0^{const'}) \longrightarrow \\
(\mathtt{A}_0, 6, {}_6\mathtt{F}_3^{id\_y}{}_5\mathtt{F}_3^{const}{}_4\mathtt{F}_3^{flip}{}_3\mathtt{F}_1^{flip\_const\_x\_id\_y}{}_2\mathtt{F}_1^{id}{}_1\mathtt{F}_0^{const'}) \longrightarrow \\
(\mathtt{A}_0, 7, {}_7\mathtt{F}_6^{id}{}_6\mathtt{F}_3^{id\_y}{}_5\mathtt{F}_3^{const}{}_4\mathtt{F}_3^{flip}{}_3\mathtt{F}_1^{flip\_const\_x\_id\_y}{}_2\mathtt{F}_1^{id}{}_1\mathtt{F}_0^{const'}) \longrightarrow \\
(\mathtt{P}_1^{id}, 7, {}_7\mathtt{F}_6^{id}{}_6\mathtt{F}_3^{id\_y}{}_5\mathtt{F}_3^{const}{}_4\mathtt{F}_3^{flip}{}_3\mathtt{F}_1^{flip\_const\_x\_id\_y}{}_2\mathtt{F}_1^{id}{}_1\mathtt{F}_0^{const'}) \longrightarrow \\
(\mathtt{A}_1, 6, {}_7\mathtt{F}_6^{id}{}_6\mathtt{F}_3^{id\_y}{}_5\mathtt{F}_3^{const}{}_4\mathtt{F}_3^{flip}{}_3\mathtt{F}_1^{flip\_const\_x\_id\_y}{}_2\mathtt{F}_1^{id}{}_1\mathtt{F}_0^{const'}) \longrightarrow \\
(\mathtt{P}_2^{const'}, 6, {}_7\mathtt{F}_6^{id}{}_6\mathtt{F}_3^{id\_y}{}_5\mathtt{F}_3^{const}{}_4\mathtt{F}_3^{flip}{}_3\mathtt{F}_1^{flip\_const\_x\_id\_y}{}_2\mathtt{F}_1^{id}{}_1\mathtt{F}_0^{const'}) \longrightarrow \\
(\mathtt{P}_2^{const'}, 3, {}_7\mathtt{F}_6^{id}{}_6\mathtt{F}_3^{id\_y}{}_5\mathtt{F}_3^{const}{}_4\mathtt{F}_3^{flip}{}_3\mathtt{F}_1^{flip\_const\_x\_id\_y}{}_2\mathtt{F}_1^{id}{}_1\mathtt{F}_0^{const'}) \longrightarrow \\
(\mathtt{P}_2^{const'}, 1, {}_7\mathtt{F}_6^{id}{}_6\mathtt{F}_3^{id\_y}{}_5\mathtt{F}_3^{const}{}_4\mathtt{F}_3^{flip}{}_3\mathtt{F}_1^{flip\_const\_x\_id\_y}{}_2\mathtt{F}_1^{id}{}_1\mathtt{F}_0^{const'}) \longrightarrow \\
(\mathtt{A}_2, 0, {}_7\mathtt{F}_6^{id}{}_6\mathtt{F}_3^{id\_y}{}_5\mathtt{F}_3^{const}{}_4\mathtt{F}_3^{flip}{}_3\mathtt{F}_1^{flip\_const\_x\_id\_y}{}_2\mathtt{F}_1^{id}{}_1\mathtt{F}_0^{const'})
\end{array}
$$

Figure 8: Evaluation with Backtracking

exists and is unsaturated. More about the relationship between WHNF and very lazy evaluation below (`VWHNF`)).

4.11 Continuation Stack

To permit useful computations two more issues need to be taken care of: case-discrimination and primitives, both of which share the necissity for a certain kind of strictness. The selection of the right alternative in a case-discrimination depends on the value of its scrutinee, just as primitive operators can only be applied to fully evaluated operands.

Therefore a mechanism is required, that enforces the evaluation of the required value and ensures, that after its computation the evaluation returns with the result to the point of origin that initiated this computation.[23] This is achieved by introducing another stack, the *continuation stack*, which contrary to the evaluation stack is an ordinary stack and can only be accessed at its top. It can contain two types of continuation tokens, *case-continuation tokens* and *operator tokens*. We extend the abstract machine configuration from (SR, SP, ES) to (SR, SP, ES, CS).

4.12 Case-Discrimination

Whenever a function token is pushed on the evaluation stack, whose function definition incorporates a case-discrimination, a case-continuation token is pushed on the continuation stack. It holds a reference to this function token. The evaluation proceeds just as usual, interpreting the scrutinee as the function definition's right-hand side. As soon as the subsequent computation yields a *constant* value[24] v, indicated by $SR = \mathtt{C}_v$, the case-continuation token is popped off the continuation stack and the case-discrimination can be concluded by selecting the appropriate case-alternative in the function definition of the referenced function token, according to the computed value. The alternative's function token is then pushed on the evaluation stack and the evaluation can be resumed as usual.

The method comes with the extra benefit, that in the resulting machine state the function token that accounts for the evokation of the constant directly precedes the alternative's function token. That way in the target language, constructors do not need to be included, and be represented by a simple integer constant. The constructor's parameter list can be discarded, because they can be accessed just as regular function parameters by the alternative's function.

Note, how in the example (Figure 9) the right-hand side of the case-alternative `Nothing -> id` is outsourced in the compilation result (Figure 10) into the function `id'`. The reason why this is necessary, even though with only a sole argument (`id`) the right-hand side is atomic, is explained later (*5. Model Specification*).

[23] Note: What is required is controlled strictness.

[24] only integer values are possible, since characters are represented as integers in the source language and constructors are resolved to integers in the compilation. Case-discrimination over non-integral values is not supported.

```
main = f 4 3
maybe = Just
f x = case maybe x of
  Nothing -> id
  Just x -> const x
```

Figure 9: Case-discrimination and constructors

```
main 0 = f 4 3
maybe 0 = 1
f 1 = maybe P_1^f
      0 → id'
      1 → const_x
const_x 0 = const P_1^{const_x}
const 2 = P_1^{const}
id' 0 = id
id 1 = P_1^{id}
```

Figure 10: Example compiled with `const` and `id` included

$(\mathtt{A}_0, 1, {}_1\mathtt{F}_0^{main}, \epsilon) \rightarrow$
$(\mathtt{A}_0, 2, {}_2\mathtt{F}_0^f {}_1\mathtt{F}_0^{main}, \mathtt{C}^2) \rightarrow$

Figure 11: Evaluation - part 1

According to the explanations above, in part 1 a continuation token is pushed on the continuation stack, since `f`'s definition contains a case-discrimination.

$$
\begin{array}{l}
(\mathtt{A}_0, 3, {}_3\mathtt{F}_2^{maybe}\,{}_2\mathtt{F}_0^{f}\,{}_1\mathtt{F}_0^{main}, \mathtt{C}^2) \rightarrow \\
(\mathtt{A}_0, 4, {}_4\mathtt{F}_3^{const}\,{}_3\mathtt{F}_2^{maybe}\,{}_2\mathtt{F}_0^{f}\,{}_1\mathtt{F}_0^{main}, \mathtt{C}^2) \rightarrow \\
(\mathtt{P}_1^{const}, 4, {}_4\mathtt{F}_3^{const}\,{}_3\mathtt{F}_2^{maybe}\,{}_2\mathtt{F}_0^{f}\,{}_1\mathtt{F}_0^{main}, \mathtt{C}^2) \rightarrow \\
(\mathtt{A}_1, 3, {}_4\mathtt{F}_3^{const}\,{}_3\mathtt{F}_2^{maybe}\,{}_2\mathtt{F}_0^{f}\,{}_1\mathtt{F}_0^{main}, \mathtt{C}^2) \rightarrow \\
(\mathtt{C}_1, 3, {}_4\mathtt{F}_3^{const}\,{}_3\mathtt{F}_2^{maybe}\,{}_2\mathtt{F}_0^{f}\,{}_1\mathtt{F}_0^{main}, \mathtt{C}^2) \rightarrow
\end{array}
$$

Figure 12: Evaluation - part 2

At the end of part 2 the continuation token is used to return to f and to select the alternative based on the propagated constant, which is 1. Therefore the appropriate alternative function is pushed (`const_x`).

$$
\begin{array}{l}
(\mathtt{A}_0, 5, {}_5\mathtt{F}_3^{const\_x}\,{}_4\mathtt{F}_3^{const}\,{}_3\mathtt{F}_2^{maybe}\,{}_2\mathtt{F}_0^{f}\,{}_1\mathtt{F}_0^{main}, \epsilon) \rightarrow \\
(\mathtt{A}_0, 6, {}_6\mathtt{F}_5^{const}\,{}_5\mathtt{F}_3^{const\_x}\,{}_4\mathtt{F}_3^{const}\,{}_3\mathtt{F}_2^{maybe}\,{}_2\mathtt{F}_0^{f}\,{}_1\mathtt{F}_0^{main}, \epsilon) \rightarrow \\
(\mathtt{P}_1^{const}, 6, {}_6\mathtt{F}_5^{const}\,{}_5\mathtt{F}_3^{const\_x}\,{}_4\mathtt{F}_3^{const}\,{}_3\mathtt{F}_2^{maybe}\,{}_2\mathtt{F}_0^{f}\,{}_1\mathtt{F}_0^{main}, \epsilon) \rightarrow \\
(\mathtt{A}_1, 5, {}_6\mathtt{F}_5^{const}\,{}_5\mathtt{F}_3^{const\_x}\,{}_4\mathtt{F}_3^{const}\,{}_3\mathtt{F}_2^{maybe}\,{}_2\mathtt{F}_0^{f}\,{}_1\mathtt{F}_0^{main}, \epsilon) \rightarrow \\
(\mathtt{P}_1^{const\_x}, 5, {}_6\mathtt{F}_5^{const}\,{}_5\mathtt{F}_3^{const\_x}\,{}_4\mathtt{F}_3^{const}\,{}_3\mathtt{F}_2^{maybe}\,{}_2\mathtt{F}_0^{f}\,{}_1\mathtt{F}_0^{main}, \epsilon) \rightarrow
\end{array}
$$

Figure 13: Evaluation - part 3

`const_x` obtains the constructor-parameter of `Just` by treating it as if it was its own function parameter.

$$
\begin{array}{l}
(\mathtt{A}_1, 4, {}_6\mathtt{F}_5^{const}\,{}_5\mathtt{F}_3^{const\_x}\,{}_4\mathtt{F}_3^{const}\,{}_3\mathtt{F}_2^{maybe}\,{}_2\mathtt{F}_0^{f}\,{}_1\mathtt{F}_0^{main}, \epsilon) \rightarrow \\
(\mathtt{P}_1^{const}, 4, {}_6\mathtt{F}_5^{const}\,{}_5\mathtt{F}_3^{const\_x}\,{}_4\mathtt{F}_3^{const}\,{}_3\mathtt{F}_2^{maybe}\,{}_2\mathtt{F}_0^{f}\,{}_1\mathtt{F}_0^{main}, \epsilon) \rightarrow \\
(\mathtt{A}_1, 3, {}_6\mathtt{F}_5^{const}\,{}_5\mathtt{F}_3^{const\_x}\,{}_4\mathtt{F}_3^{const}\,{}_3\mathtt{F}_2^{maybe}\,{}_2\mathtt{F}_0^{f}\,{}_1\mathtt{F}_0^{main}, \epsilon) \rightarrow \\
(\mathtt{A}_1, 2, {}_6\mathtt{F}_5^{const}\,{}_5\mathtt{F}_3^{const\_x}\,{}_4\mathtt{F}_3^{const}\,{}_3\mathtt{F}_2^{maybe}\,{}_2\mathtt{F}_0^{f}\,{}_1\mathtt{F}_0^{main}, \epsilon) \rightarrow \\
(\mathtt{P}_1^{f}, 2, {}_6\mathtt{F}_5^{const}\,{}_5\mathtt{F}_3^{const\_x}\,{}_4\mathtt{F}_3^{const}\,{}_3\mathtt{F}_2^{maybe}\,{}_2\mathtt{F}_0^{f}\,{}_1\mathtt{F}_0^{main}, \epsilon) \rightarrow \\
(\mathtt{A}_1, 1, {}_6\mathtt{F}_5^{const}\,{}_5\mathtt{F}_3^{const\_x}\,{}_4\mathtt{F}_3^{const}\,{}_3\mathtt{F}_2^{maybe}\,{}_2\mathtt{F}_0^{f}\,{}_1\mathtt{F}_0^{main}, \epsilon) \rightarrow \\
(\mathtt{C}_4, 1, {}_6\mathtt{F}_5^{const}\,{}_5\mathtt{F}_3^{const\_x}\,{}_4\mathtt{F}_3^{const}\,{}_3\mathtt{F}_2^{maybe}\,{}_2\mathtt{F}_0^{f}\,{}_1\mathtt{F}_0^{main}, \epsilon)
\end{array}
$$

Figure 14: Evaluation - part 4

Indeed at the end of part 4 the request does arrive at `main`, which serves the request with the correct argument. The evaluation terminates, since there are no continuation tokens on the stack with which the constant in the status register could be processed with. This corresponds roughly to WHNF.

4.13 Primitives

Primitive operators can be handled almost the same way. When an operator is served, an *operator token* is pushed on the continuation stack. Operator-continuations have to be treated sligtly different than case-continuations. First, the continuation token must specify the employed operator. Second, the operator might require more than one constant as an operand. Therefore its operands must be acquired one by one.

4.14 Sharing

In this version of the document, a sharing-mechanism is not included.

5 Model Specification

The presented execution model consists of two components, the first being the abstract machine language, given below along with directions for the compilation from the source language (Figure 1). The second component is the abstract machine, with a description of how to interprete the target language.

5.1 Target Language

The language interpreted by the abstract machine is a very simple, untyped, functional language with a flat structure, i.e. its abstract syntax does not involve (mutual) recursion. This property harmonises well with the request/serve mechanism, as it can be implemented most efficiently, if every component of a function definition's is atomic. That way, the ith argument of a function definition can be easily accessed by reading the ith atom relative to the function definition's address, an operation, which is very well supported on current computer architectures.

$program$	$\rightarrow$	*function-binding**
function-binding	$\rightarrow$	$type\ arity\ atom^*\ alternative^*$
$type$	$\rightarrow$	TLF | ASF | RSF
$arity$	$\rightarrow$	$integer$
$atom$	$\rightarrow$	$parameter \mid function \mid integer \mid float$
$parameter$	$\rightarrow$	$function\ index$
$function$	$\rightarrow$	*function-address*
$alternative$	$\rightarrow$	$integer\ function \mid$ `default`$function$
$index$	$\rightarrow$	$integer$

Figure 15: Abstract syntax of the target language as a context-free grammar

As in the source language (*2. The Source Language*), a *program* definition comprises a set of function definitions. The most important members of such a *function-binding* are its *arity* and the list of *atom*s that correspond to its right hand-side expression.

The *type* is TLF for functions defined at top-level in the source language. Stack tokens of top-level functions do not need to maintain a parent-edge in the evaluation stack. Subfunctions (function definitions resulting from the outsourcing of nested expressions) have the type ASF (alternative subfunction) if the outsourced expression was the right-hand side of a case alternative in the source language, otherwise it is a RSF (regular subfunction).

If the function defines a case-discrimination, it also has a non-empty set of *alternative*s, each of which assigns a function to a `default`- or integer-pattern.

Functions references in *alternative*s, *function*- and *parameter-atoms* are represented by their *function-address*, which constitutes a unique identifier for each function definition.

The target language representation of a program is gained by a straightforward compilation process from the source language. Any kind of nesting is factored out into subfunctions, thus the result has a flat structure. Constructors are resolved to mere integers.

Table 2: Transformation of some source language elements

nested expression	subfunction
let-expression	subfunctions
datatype	association of every member constructor with a different integer
variable	depending on the binding with the same name, either a *function*-reference or *integer* (encoding a constructor)
application	list of *atom*s
case-discrimination	list of *argument*s

5.2 Abstract Machine Configuration

The state space of the abstract is specified in a context-free grammar (Figure 16).

(configuration)	ST	$\rightarrow$	(SR, SP, ES, CS)
(status register)	SR	$\rightarrow$	$\mathtt{F}^f \mid \mathtt{C}_c \mid \mathtt{P}^f_{int} \mid \mathtt{A}_{int} \mid \mathtt{O}_{op}$
(current stack position)	SP	$\rightarrow$	sp
(stack position)	sp	$\rightarrow$	$usp \mid \mathtt{1} \mid \mathtt{2} \mid \ldots \mid \mathtt{n}$
(undefined stack position)	usp	$\rightarrow$	$\mathtt{0}$
(evaluation stack)	ES	$\rightarrow$	${}_{\mathtt{n}}st\,{}_{\mathtt{n-1}}st \ldots {}_{\mathtt{1}}st$
(stack token)	st	$\rightarrow$	$\mathtt{F}^f_{sp}$
(continuation stack)	CS	$\rightarrow$	ct^*
(continuation token)	ct	$\rightarrow$	$\mathtt{C}^{sp} \mid \mathtt{O}^{sp}_{op}(c^*)$
(operator code)	op	$\rightarrow$	int
(function address)	f	$\rightarrow$	int
(constant)	c	$\rightarrow$	$int \mid float$

Figure 16: Configuration grammar

- The configuration (ST) comprises a status register (SR), a stack position register (SP), the evaluation stack (ES), and the continuation stack (CS).

- Except for the request ($\mathtt{A}_i$), the possible values for the status register (SR) correspond to the *atom*s of the target language above: $\mathtt{F}^f$ (*function*), $\mathtt{C}_c$ (constant: *integer* or *float*), $\mathtt{P}_i^f$ (*parameter*), $\mathtt{O}_{op}$ (*primitive*). The status register is set to one of these values, when the corresponding atom is served.

- The evaluation stack is an indexed array of n stack tokens growing from right to left, indexed from 1 to n. A stack position (sp) is a number, addressing a token in the evaluation stack by index. Whenever there is the need to leave a stack position undefined, the value $\mathtt{0}$ (*usp*) can be used.

- A stack token represents an instance of a function definition *f* that has a parent token referenced by *p*.

- The continuation stack is a stack of continuation tokens (*ct*). Both case-continuation tokens $\mathtt{C}^{sp}$ and operator-continuation tokes $\mathtt{O}_{op}^{sp}(c^*)$ define the function token (addressed by *sp*), where the evaluation is to be continued. For the operator-continuation also the corresponding operator (*op*) needs to be defined along with the operands (c^*) that have already been computed.

5.3 Abstract Machine Semantics

In the operational semantics (Figure 17) the transitions are given in a special inference rule format. It comprises a set of *situations*, each of which specifies a configuration pattern and a set of transitions. A transition may include a condition. In each evaluation step, the machine configuration matches at least one of the situations' patterns, and of the belonging transitions there should be exactly one, of which the associated condition is satisfied. Once, the transition rule has been selected, the machine configuration is transformed according to the inference from the situation's pattern to the transition's pattern.

- Initial State: At the beginning of the evaluation, both stacks are empty, and SP is undefined. The status register indicates, that the $main$ function token is to be pushed on the evaluation stack.

- Function: Indicates the intent to push a function token on the stack, typically the result of the function token at stack position a serving a function atom, but besides the initial state, also case-continuations lead here. A function token for the function f is pushed on the evaluation stack with a parent-reference to the originating function's token at a. SR and SP are set to request the leftmost part of f. If it performs a case-discrimination also a case-continuation has to be pushed onto the continuation stack, pointing to the f's function token.

- Request: A request for the ith argument of the function token at a can only be served, if the definition of the associated function f has enough atoms on

Initial State: $(\mathtt{F}^{main}, \mathtt{0}, \epsilon, \epsilon)$

Function: $(\mathtt{F}^f, a, \ldots, \ldots)$		
$\rightarrow (\mathtt{A}_0, n, {}_n\mathtt{F}^f_a \ldots, \ldots)$	$\lvert alts(f)\rvert = 0$	*(Push)*
$\rightarrow (\mathtt{A}_0, n, {}_n\mathtt{F}^f_a \ldots, \mathtt{C}^n_{\mathtt{0}} \ldots)$	$\lvert alts(f)\rvert > 0$	*(Scrutinise)*
Request: $(\mathtt{A}_i, a, \ldots {}_a\mathtt{F}^f_p \ldots {}_p\mathtt{F}^g_{\_} \ldots, \ldots)$		
$\rightarrow (\mathtt{A}_{i-\lvert args(f)\rvert + arity(f)}, a-1, \ldots {}_a\mathtt{F}^f_p \ldots {}_p\mathtt{F}^g_{\_} \ldots, \ldots)$	$\lvert args(f)\rvert < i$ $type(f) \neq \mathtt{ASF}$	*(Curry)*
$\rightarrow (\mathtt{A}_{i-\lvert args(f)\rvert + arity(g)}, p-1, \ldots {}_a\mathtt{F}^f_p \ldots {}_p\mathtt{F}^g_{\_} \ldots, \ldots)$	$\lvert args(f)\rvert < i$ $type(f) = \mathtt{ASF}$	*(Redirect)*
$\rightarrow (args_i(f), a, \ldots {}_a\mathtt{F}^f_p \ldots {}_p\mathtt{F}^g_{\_} \ldots, \ldots)$	$\lvert args(f)\rvert \geq i$	*(Serve)*
Parameter: $(\mathtt{P}^f_i, a, \ldots {}_a\mathtt{F}^g_p \ldots, \ldots)$		
$\rightarrow (\mathtt{P}^f_i, p, \ldots {}_a\mathtt{F}^g_p \ldots, \ldots)$	$f \neq g$	*(Backtrack)*
$\rightarrow (\mathtt{A}_i, a-1, \ldots {}_a\mathtt{F}^g_p \ldots, \ldots)$	$f = g$	*(Request)*
Operator: $(\mathtt{O}_{op}, \_, {}_nt \ldots, \ldots)$		
$\rightarrow (\mathtt{A}_1, n, {}_nt \ldots, \mathtt{O}^n_{op}())$		*(1st Operand)*
Operand: $(\mathtt{C}_v, \_, \ldots, \mathtt{O}^a_{op}(v_1, \ldots, v_c) \ldots)$		
$\rightarrow (\mathtt{C}_{apply_{op}(v_1,\ldots,v_c,v)}, \_, \ldots, \ldots)$	$arity(op) = c$	*(Apply Operator)*
$\rightarrow (\mathtt{A}_n, a, \ldots, \mathtt{O}^a_{op}(v_1, \ldots, v_n, v) \ldots)$	$arity(op) > c$	*(Next Operand)*
Scrutinee: $(\mathtt{C}_c, \_, \ldots {}_a\mathtt{F}^f_p \ldots, \mathtt{C}^a_{\mathtt{0}} \ldots)$		
$\rightarrow (alts_c(f), a, \ldots {}_a\mathtt{F}^f_p \ldots, \ldots)$	$c \in int$	*(Alternative)*

Figure 17: Operational semantics of the abstract machine

its right-hand side. A function definition's atoms in the program definition can be accessed by $args(f)$, which returns a sequence of arguments. To extract a single atom, the sequence can be indexed (beginning with 0 for the leftmost atom) by $args_i(f)$. In the argument count $|args(f)|$ the obligatory leftmost atom is not counted. If the function application is unsatisfied, the curry rule needs to be applied, in case of an alternative subfunction (ASF the curried request needs to be diverted to the discriminating function (see *4.12. Case-Discrimination*)

- Parameter: If a parameter request is not addressed to the function token of the intended function f, it must be of an descendant of f. Therefore the parameter request needs to backtrack to the parent token. A parameter request to the correct function can be translated to the corresponding argument request to the predecessor (see *4.4. Parameters*)

- Operator: When an operator has been served, an operator continuation token is pushed on the continuation stack, which has a reference to the current topmost evaluation stack token. This token is an instance of the function that produced this operator as its leftmost atom. Future requests for operands have to be addressed to this token. After setting up the operator continuation, the first operand is requested.

- Operand: Whenever a constant is served, the continuation stack is examined. If the topmost token is a operator continuation, the constant is an operand to the operator. In case the operator has acquired its required amount of operands, it is applied to the operands in a primitive operation, and the result is propagated as another constant. Otherwise the operator requires yet another operand, and the current constant is stored along with the operator continuation and the next operand is requested from the token that evoked the operand.

- Scrutinee: When a constant is served, and the topmost continuation token is a case-continuation token, the constant is the result of evaluating the scrutinee of a case-discrimination. The continuation token refers to the token of the function that has the corresponding case-discrimination on its right-hand side, thus the constant is used to select the right alternative from that function. Similar as the $args$-function, $alts(f)$ returns a sequence of function-references each representing one of f's alternatives.

6 Implementation

To prove the applicability of the execution model, it has been implemented as a part of this project. For the source code, see the appendix. The implementation supports Haskell as a source language and is compatible with a variety of architectures. The system comprises two components:

- The compiler, which produces for a given source program the corresponding program in the target language. It is written in Haskell (about 2000 lines of code).
- The interpreter is written in C (about 400 lines of code) and implements the abstract machine for executing a program given in the target language.

6.1 The Compiler

To avoid the effort of implementing a complete Haskell-compiler, while even so supporting Haskell as a source language, currently the York Haskell Compiler (YHC, `http://www.haskell.org/yhc/`) is used as a front-end. YHC provides an interface to its intermediate language YHC-Core, a functional language, which is basically simplified Haskell 98. In the compilation from Haskell to YHC-Core typechecking is already performed. Therefore it is an easy task for the presented compiler to convert YHC-Core to its own internal untyped representation of the source language (Figure 18).[25] In fact, the languages are very similar, the same goes for most intermediate languages used by other compilers like GHC or JHC.

The fact, that the compiler does not operate on a concrete programing language but the rather abstract source language (Figure 1) displays, how not only one specific functional language is eligible for the execution model, but rather a whole class of languages.[26] This is respected by the compiler architecture (Figure 20), in which besides the parts of the system (bold) that have been implemented also other possible solutions are shown.

The real compilation takes place in the translation step from the intermediate source to the intermediate target language (Figure 19).[27] The main task of the compiler during the transformation is to recursively outsource nested expressions[28] and therefore flatten the module structure. Thereby variables that address parameters are resolved to the corresponding (function-name, index)-pair.

[25] While in this document the possible module structure of source languages is ignored, it is regarded by the implemented compiler, and therefore in the intermediate source language.

[26] The language class being higher-order, strongly-typed, non-strict, purely-functional languages

[27] *intermediate* languages as opposed to a source language such as Haskell or a concrete (binary or textual) form of the target language

[28] The recursive structure of the source language can be observed in the source language module (Figure 18), where e.g. `Expression` appears on the right-hand side of its own data-type definition.

```
data Module = Module
  {modName  :: String,
   imports  :: [String],
   bindings :: [Binding]}

data Binding = Binding
  {exported   :: Bool,
   definition :: Either Function DataType}

type DataType = [Constructor]
data Constructor = Constructor {cName :: String, cParams :: [String]}

data Function = Function
  {fName :: String, fParams :: [String], fExpr :: Expression}

data Expression
  = Variable {var :: String]}
  | Application {components :: [Expression]}
  | Case {scrutinee :: Expression, cases :: [Alternative]}
  | Let {functions :: [Function], inExpr :: Expression}
  | Int {intValue :: Int}
  | Float {floatValue :: Float}
  | Primitive {name :: String}

data Alternative = Alternative
  {condition :: Condition, alternative :: Expression}

data Condition = CaseConst {const :: String, params :: [String]}
               | CaseInt Int
               | Default
```

Figure 18: `CHaOS.Language.IntermediateSource`

```
data Module = Module
  {modName  :: String,
   imports  :: [String],
   bindings :: [Binding]}

data Binding = Binding
  {name       :: String,
   exported   :: Bool,
   definition :: Definition}

data Definition
  = Constructor {code :: Int}
  | Function {fType :: FunctionType, expr :: [Atom], cases :: [Case]}

data FunctionType = TopLevelFunction | RegularSubFunction | CaseAlternative

type Variable = String -- function or constructor name

data Atom
  = Parameter Variable Int
  | Variable Variable
  | Int Int
  | Float Float
  | Primitive String

data Case
  = Equals (Either Int Variable) Variable
  | Default Variable
```

Figure 19: `CHaOS.Language.IntermediateTarget`

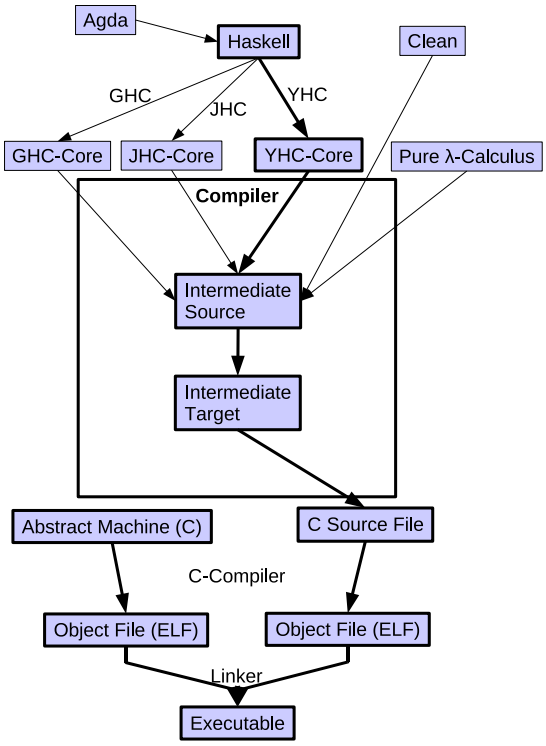

Figure 20: Compiler pipeline

Note, that in the target language, functions and constructors are still referenced by name, rather than by address (for functions) or integer code (for constructors). The reason for this is again the idea, to leave tasks that are not inherent to the execution model to other well-established software, in this case an external linker program.

This strategy harmonises nicely with the approach to use the C programming language as another intermediate step to represent the target language.[29] In order to obtain a concrete binary form from the target language, the compiler produces a C source file, in which every function definition appears as an array of structs, where every member of the array represents one atom or case-discrimination. The name of the array is identical to the function's name.

This makes it also very easy to bind together the interpreter, which is also written in C, simply by using a C compiler to produce an object file for both the interpreter and the program definition and linking them into a single executable using a conventional linker. Thereby the symbolic references within the program definition are resolved to their respective addresses.

6.2 The Interpreter

The particular characteristics of the presented execution model suggest certain aspects concerning the implementation of the abstract machine. Here we explore what kind of architecture might be most adequate for this purpose. The abstract machine comprises two kinds of components that need to be included in the target system:

- *Dynamic* components: evaluation stack, continuation stack, status register, stack position register. Their contents are constantly modified by the abstract machine during the execution.

- *Static* components: program definition, operational semantics implementation. These remain unaltered throughout the program run.

An important feature of the execution model is the simplicity of the operational semantics. It operates on the other components and requires no further state information. This might permit an efficient implementation of the operational semantics in hardware (making the abstract machine concrete). The task of implementing non-strict, functional languages in hardware based on other execution models has been attempted multiple times, but has not yet lead to lasting success.

An operational implementation is however much more easily gained by using current computer systems to run the abstract machine as a program given in the architecture's machine code. Typically the program code is kept in a special non-resizable section for executable code in the program's address space, along with the program definition (of the interpreted program given in the abstract machine

[29] In fact, C is generated from yet another representation based on C macros.

language) in an also non-resizable data-section. To avoid an early exhaustion of the address space, it is most convenient to locate the two stacks at the opposite bounds of the remaining address space growing towards each other.

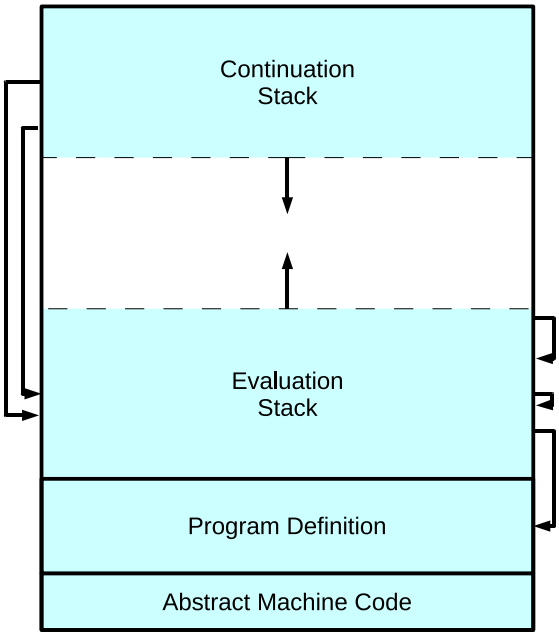

Figure 21: Typical address space of a software implementation

A different approach employed by the STG-machine is to implement the operational semantics by compiling the program definition from the abstract machine code to machine code, and thereby mangling the semantics into the program definition.[30] For several reasons, this seems not to be a very good choice for the presented execution model in terms of efficency:

- In this model, a function definition is not evaluated at once as a whole, but rather partially, atom by atom. Therefore only very short sequences of instructions would be executed, before performing a jump to another atom.
- These jump instructions and the frequent case-distinctions on what inference rule to apply next, would bloat the code for each single atom, and therefore the complete program definition. Yet, for efficient program execution the size of the cache-footprint is a very important factor on current architectures and is in this model vastly determined by the size of the program definition.
- In this model a fast request/serve-mechanism is crucial. If compiled to machine-code, the resulting code for each atom would certainly not have the

[30] See [Simon L. Peyton Jones (1992), Part III: Mapping the abstract machine to stock hardware, p 41 ff]

same size. If this is ensured however, a specific atom $\texttt{A}_i$ of a function f can be efficiently accessed by addressing it relative to f's function definition[31], an operation well supported by most architecures. The specified abstract machine language (Figure 15) is designed to allow a compact, equally sized representation for atoms in a concrete binary format.

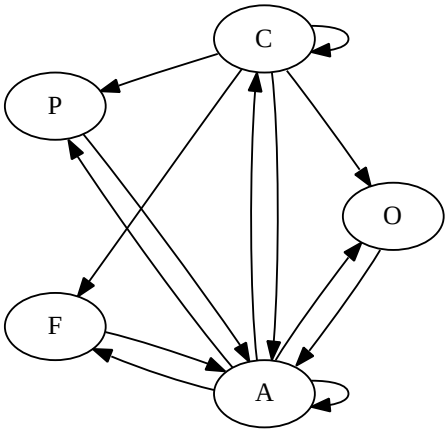

Figure 22: Flow graph of the operational semantics

The operational semantics however can only be kept in a non-growing section in the address space, if in the implementation the depth of (mutually) recursive function-calls can be bounded by a constant. Otherwise it would require a dynamically growing function-call stack (also called *return* stack). The control flow of the operational semantics however exibits a structure (Figure 22) that does not harmonise very well with an implementation that encapsulates routines of the operational semantics in functions. In fact, the operational semantics can be implemented without relying on function-calls at all.

That means, that control flow is achieved solely by (conditional) jumps, a programming style often referred to as spaghetti code.[32] That way no function-call stack needs to be maintained, which leaves the function-call mechanisms possibly supplied by the underlying architecure unused. On certain platforms these might be abused to manage the evaluation and/or continuation stack for an additional portion of performance.

The given implementation of the abstract machine in C uses *inlined* functions to encapsulate operations used at different locations in the code. One has to keep in mind, that this leads to replication of the the encapsulated code.

[31] $args_i(f)$ is at: $address(f) + i * sizeof(atom)$

[32] Usually spaghetti code is not recommended regardless of the project size and rightly so. This is however a special setting, where the intended behaviour deviates from conventional programs.

The stack position register can be represented by a regular integer variable, but is due to frequent usage probably best kept in a machine register (if enough are available), rather than in memory.[33] In a software implementation, the status register is automatically partly encoded by the instruction pointer (also called *program counter*), since every inference rule of the operational semantics has a corresponding fragment in the code section, therefore modifying the status is achieved performing a jump to its code location. Thereby only the status register's parameters need to be put into (at most 2) variables, for example $\mathtt{P}_i^f$ requires one for each f and i.

6.3 Implementation Status

While the implementation of the compiler is fully functional and even generates all information required for sharing and garbage collection, the current version of the interpreter is merely fit to demonstrate the general applicability of the execution model. It is not optimised at all and therefore not qualified to yield useful runtime measurements that could be compared with GHC. In fact for many source programs, the evaluation currently fails, because support for most primitives of YHC's Haskell libraries needs still to be implemented.

Sharing is not yet included, neither a garbage collection mechanism, the hints given by the compiler are simply ignored. Therefore the stack grows very fast and is never compacted.[34]

However, for programs that make only use of the implemented primitives, the evaluation delivers the same results as when run using other Haskell implementations. This is a strong evidence for the correctness of the execution model and its operational semantics. The reassurence that in the still early stadium of the project, it can be used to run real programs is the most important outcome of the implementation work.

[33] Due to optimising C compilers, this does not need to be explicitely specified in the abstract machine implementation.

[34] Without sharing, problems of linear complexity can easily require exponential runtime.

7 Perspectives

There are a few matters that would have been appropriate to include in the thesis and are to be covered in future versions of this document:

- Currently, the affirmation of the correctness of the execution model relies on pure reasoning and repeated validation of the evaluation result. While this delivers strong evidence, it still would be desirable to verify the correctness of the operational semantics by a formal deduction from the denotational semantics of a source language like Haskell or the λ-calculus.
- Several important issues have been mentioned that are not covered by this thesis, such as sharing and garbage collection. They require an updated, more complex revision of the abstract machine's operational semantics.
- If a method could be found to systematically infer this execution model from a graph-reduction based approach (preferably the STG-machine), one could make a formal comparison of the two execution models' run-time behaviour (execution time, memory usage, timing parameters). Again, the assumption, that the presented execution model is at least as efficient as conventional models, relies on pure reasoning.
- Support for parallel execution on a multi-processor (or multi-core processor) system.

Also a lot of work needs to be put into the yet very crude implementation, particularly the interpreter:

- The lack of a sharing-mechanism in the interpreter constitutes the main drawback, which impedes to benchmark the system's efficiency and to compare it to other functional language implementations (preferably GHC). But also garbage collection is mandatory for more complex programs.
- Unfortunately the YHC project is not maintained anymore. Therefore sooner or later the compiler should include support for other front-ends. Attractive candidates are GHC and Jhc (`http://repetae.net/computer/jhc/`).
- To support all given programs in the source language, the complete set of primitives used by the front-end's standard libraries needs to be implemented.
- The applicability of different kinds of optimisations to the execution model is to be explored. Some of the traditional optimisations based on program analysis by the compiler might not be adaptable to the execution model. The novel type of abstract machine on the other hand, might lead to new kinds of optimisations for the interpreter.[35]

[35] Presently, not even the most obvious optimisations or those described in this document are included in the current implementation.

Finally, once the execution model is proven to constitute a serious competitor for current functional language implementations, there are visions for future research that are to be explored, like how the execution model could be supported on a operating-system level, or even building an operating system based on the model. One might even think about implementing the operational semantics on a specialised processor. These approaches might prove useful especially for simple, embedded systems where the simplicity of the operational semantics could be very valuable.

In this thesis it could be shown, how the observation of the strictness of argument handling in the normal order reduction led to a new idea for the approach of evaluating a program in a functional programming language and how the idea evolved into a fully operational execution model. For the model a proof-of-concept implementation was developed by which the applicability of the concept could be demonstrated.

References

[Rémi Douence] *A Systematic Study of Functional Language Implementations*, Rémi Douence, Pascal Fradet. INRIA / IRISA.

[Zena M. Ariola (1994)] *Cyclic Lambda Graph Rewriting*, Zena M. Ariola, Jan Willem Klop. 1994. pp 416-425. (Symposium on Logic in Computer Science, 1994. Proceedings)

[Peter Sestoft (2002)] *Demonstrating Lambda Calculus Reduction*, Peter Sestoft. 2002. Department of Mathematics and Physics Royal Veterinary and Agricultural University, Denmark and IT University of Copenhagen, Denmark . pp 420-435. *The Essence of Computation: Complexity, Analysis, Transformation*, T. Mogensen, D. Schmidt, I. H. Sudburough (eds.) Lecture Notes in Computer Science 2566.

[Simon L. Peyton Jones (1992)] *Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine - Version 2.5*, Simon L. Peyton Jones. 1992. Department of Computing Science, University of Glasgow G12 8QQ .

[Cregut Pierre (1991)] *Machines à environnement pour la réduction symbolique et l'évaluation partielle*, Cregut Pierre, Cousineau Guy. *Travaux Universitaires - Thèse nouveau doctorat*, 1991. Université de Paris 07. p 168 ff.

[Simon Marlow (2006)] Simon Marlow, Simon Peyton Jones. *Making a fast curry: push/enter vs. eval/apply for higher-order languages*, 2006. Microsoft Research, Cambridge, UK. pp 415-449. *Journal of Functional Programming*, Cambridge University Press (ed.) Volume 16 2006.

[Ian Holyer (1998)] *The Brisk Machine: A Simplified STG Machine*, Ian Holyer, Eleni Spiliopoulou. 1998. University of Bristol, Department of Computer Science .

[G. Cousineau (1985)] G. Cousineau, P. L. Curien, M. Mauny. *The categorical abstract machine*, *Proc. of a conference on Functional programming languages and computer architecture*, 1985. Springer-Verlag New York, Inc.. pp 50-64.

[Simon Thompson (1992)] Simon Thompson, Rafael Lins. *The Categorical Multi-Combinator Machine: CMCM*, 1992. pp 170-176. *The Computer Journal*, The British Computer Society (ed.) Volume 35, Number 2.

[Simon L. Peyton Jones (1987)] *The implementation of functional programming languages*, Simon L. Peyton Jones, Philip Wadler, Peter Hancock, David Turner. 1987. Prentice Hall International (UK) Ltd.

[Daan Leijen (2005)] *The Lazy Virtual Machine specification*, Daan Leijen. 2005. Institute of Information and Computing Sciences, Utrecht University .

[P. J. Landin (1963)] P. J. Landin. *The mechanical evaluation of expressions*, 1963. pp 308-320. *The Computer Journal*, The British Computer Society (ed.) Volume 6, Number 4.

[Xavier Leroy (1990)] Xavier Leroy. *The ZINC Experiment: An economical Implementation of the ML language*, 1990. Institut National de Rercherche en Informatique et en Automatique.

[Jon Fairbairn (1987)] *Tim: A simple, lazy abstract machine to execute supercombinators*, Jon Fairbairn, Stuart Wray. *Functional Programming Languages and Computer Architecture*, pp 34-45. 1987. Springer Berlin / Heidelberg. *Lecture Notes in Computer Science*, Volume 274/1987.

Appendix

This document is available along with the implementation of the execution model at `http://rochel.info/`. The material will be updated with future versions as the project evolves.