University of Cambridge, UK
Department of Computer Science and Technology

Proceedings of the 2017 Scheme and Functional Programming Workshop

Oxford, UK—September 3rd, 2017

Edited by Nada Amin, University of Cambridge
and François-René Rideau, Metaphor

Preface

This report aggregates the papers presented at the eigtheenth annual Scheme and Functional Programming Workshop, hosted on September 3rd, 2017 in Oxford, UK and co-located with the twenty-second International Conference on Functional Programming.

The Scheme and Functional Programming Workshop is held every year to provide an opportunity for researchers and practitioners using Scheme and related functional programming languages like Racket, Clojure, and Lisp, to share research findings and discuss the future of the Scheme programming language.

Two full papers and three lightning talks were submitted to the workshop, and each submission was reviewed by three members of the program committee. After deliberation, all submissions were accepted to the workshop.

In addition to the two full papers and three lightning talks presented

- Sam Tobin-Hochstadt gave an invited keynote speech entitled *From Scheme to Typed Racket*,
- a panel including Michael Ballantyne, Arthur Gleckler, Kathy Gray, Alaric Snell-Pym, Andy Wingo as well as a lively audience debated on the future of Scheme,
- Alaric Snell-Pym presented an update on the R7RS standardization process, and
- Matt Might gave a closing invited talk on Precision Medecine.

Thanks to all presenters, panelists, participants, and members of the program committee.

Program Committee

Steering Committee

Contents

Scalar and Tensor Parameters for Importing Tensor Index Notation including Einstein Summation Notation

SATOSHI EGI, Rakuten Institute of Technology

In this paper, we propose a method for importing tensor index notation, including Einstein summation notation, into functional programming. This method involves introducing two types of parameters, i.e, scalar and tensor parameters, and simplified tensor index rules that do not handle expressions that are valid only for the Cartesian coordinate system, in which the index can move up and down freely. An example of such an expression is "$c = A_i B_i$". As an ordinary function, when a tensor parameter obtains a tensor as an argument, the function treats the tensor argument as a whole. In contrast, when a scalar parameter obtains a tensor as an argument, the function is applied to each component of the tensor. In this paper, we show that introducing these two types of parameters and our simplified index rules enables us to apply arbitrary user-defined functions to tensor arguments using index notation including Einstein summation notation without requiring an additional description to enable each function to handle tensors.

Additional Key Words and Phrases: tensor, index notation, Einstein summation notation, scalar parameters, tensor parameters, scalar functions, tensor functions

1 INTRODUCTION

Tensor analysis is one of the fields of mathematics in which we can easily find notations that have not been imported into popular programming languages [6, 13]. Index notation is one such notation widely used by mathematicians to describe expressions in tensor analysis concisely. This paper proposes a method for importing it into programming.

Tensor analysis is also a field with a wide range of application. For example, the general theory of relativity is formulated in terms of tensor analysis. In addition, tensor analysis plays an important role in other theories in physics, such as fluid dynamics. In fields more familiar to computer scientists, tensor analysis is necessary for computer vision [7]. Tensor analysis also appears in the theory of machine learning to handle multidimensional data. The importance of tensor calculation is increasing day by day even in computer science.

Concise notation for tensor calculation in programming will simplify technical programming in many areas. Therefore, it is important to develop a method for describing tensor calculation concisely in programs.

The novelty of this paper is that it introduces two types of parameters, *tensor parameters* and *scalar parameters*, to import tensor index notation and enables us to apply arbitrary user-defined functions to tensor arguments using index notation without requiring an additional description to enable each function to handle tensors. Tensor and scalar parameters are used to define two types of functions, *tensor functions* and *scalar functions*, respectively. Tensor functions are functions that involve contraction of the tensors provided as an argument, and scalar functions are not. For example, the inner product of vectors and the multiplication of matrices are tensor functions, because they involve contraction of the tensors. Most other functions such as "+", "-", "*", and "/" are scalar functions. We also simplified tensor index rules by removing a rule to handle the expressions such as "$c = A_i B_i$" that are valid only for the Cartesian coordinate system, in which the index can move up and down freely. It enables us to use the same index rules for all user-defined functions. We will discuss that in Sections 2 and 3.

The method proposed in this paper has already been implemented in the Egison programming language [5]. Egison is a functional language that has a lazy evaluation strategy and supports symbolic computation. In reading this paper, one can think of that language as an extended Scheme to support symbolic computation.

```
(define $min (lambda [$x $y] (if (less-than? x y) x y)))
```

Fig. 1. Definition of the min function

$$min(\begin{pmatrix}1\\2\\3\end{pmatrix}_i,\begin{pmatrix}10\\20\\30\end{pmatrix}_j) = \begin{pmatrix}min(1,10) & min(1,20) & min(1,30)\\ min(2,10) & min(2,20) & min(2,30)\\ min(3,10) & min(3,20) & min(3,30)\end{pmatrix}_{ij} = \begin{pmatrix}1 & 1 & 1\\ 2 & 2 & 2\\ 3 & 3 & 3\end{pmatrix}_{ij}$$

Fig. 2. Application of the min function to the vectors with different indices

$$min(\begin{pmatrix}1\\2\\3\end{pmatrix}_i,\begin{pmatrix}10\\20\\30\end{pmatrix}_i) = \begin{pmatrix}min(1,10) & min(1,20) & min(1,30)\\ min(2,10) & min(2,20) & min(2,30)\\ min(3,10) & min(3,20) & min(3,30)\end{pmatrix}_{ii} = \begin{pmatrix}min(1,10)\\ min(2,20)\\ min(3,30)\end{pmatrix}_i = \begin{pmatrix}1\\2\\3\end{pmatrix}_i$$

Fig. 3. Application of the min function to the vectors with identical indices

Figure 1 shows the definition of the `min` function as an example of a scalar function. The `min` function takes two numbers as arguments and returns the smaller one. "`$`" is prepended to the beginning of the parameters of the `min` function. It means the parameters of the `min` function are scalar parameters. When a scalar parameter obtains a tensor as an argument, the function is applied to each component of the tensor as Figures 2 and 3. As Figure 2, if the indices of the tensors of the arguments are different, it returns the tensor product using the scalar function as the operator. As Figure 3, if the indices of the tensors given as arguments are identical, the scalar function is applied to each corresponding component. Thus the `min` function can handle tensors even though it is defined without considering tensors. The function name "`$min`" is also prefixed by "`$`", but just as a convention of Egison. Thus it can be ignored.

Figure 4 shows the definition of the "`.`" function as an example of a tensor function. "`.`" is a function for multiplying tensors. "`%`" is prepended to the beginning of the parameters of the "`.`" function. It means the parameters of the "`.`" function are tensor parameters. As with ordinary functions, when a tensor is provided to a tensor parameter, the function treats the tensor argument as a whole. When a tensor with indices is provided, it is passed to the tensor function maintaining its indices.

In Figure 4, "`+`" and "`*`" are scalar functions for addition and multiplication, respectively. `contract` is a primitive function to contract a tensor that has pairs of a superscript and subscript with identical symbols. We will explain the semantics of `contract` expression in Section 3.2. Figure 5 shows the example for calculating the inner product of two vectors using the "`.`" function. We can use the "`.`" function for any kind of tensor multiplication such as tensor product and matrix multiplication as well as inner product.

Here we introduce a more mathematical example. The expression in Figure 6 from tensor analysis can be expressed in Egison as shown in Figure 7. When the same mathematical expression is expressed in a general way in the Wolfram language, it becomes a program such as the one shown in Figure 8. In the Wolfram language, it is assumed that all dimensions corresponding to each index of the tensor are a constant "`M`".

Note that a double loop consisting of the `Table` and `Sum` expressions appears in the program in the Wolfram language, whereas the program in Egison is flat, similarly to the mathematical expression. This is achieved by using tensor index notation in the program. In particular, the reason that the loop structure by the `Sum` expression in the Wolfram language does not appear in the Egison expression to express $\Gamma^m_{jk}\Gamma^i{}_{ml} - \Gamma^m_{jl}\Gamma^i{}_{mk}$ is that the "`.`" function in Egison can handle Einstein summation notation.

The part that we would like the reader to pay particular attention to in this example is the Egison program "`(∂/∂ Γ~i_j_k x~l)`" expressing $\frac{\partial \Gamma^i{}_{jk}}{\partial x^l}$ in the first term on the right-hand side. In the Wolfram language, the

```
(define $. (lambda [%t1 %t2] (contract + (* t1 t2))))
```

Fig. 4. Definition of the "." function

$$\begin{pmatrix}1\\2\\3\end{pmatrix}^{i} \cdot \begin{pmatrix}10\\20\\30\end{pmatrix}_{i} = contract(+, \begin{pmatrix}10 & 20 & 30\\20 & 40 & 60\\30 & 60 & 90\end{pmatrix}^{i}_{i}) = 10 + 40 + 90 = 140$$

$$\begin{pmatrix}1\\2\\3\end{pmatrix}_{i} \cdot \begin{pmatrix}10\\20\\30\end{pmatrix}_{i} = contract(+, \begin{pmatrix}10\\40\\90\end{pmatrix}_{i}) = \begin{pmatrix}10\\40\\90\end{pmatrix}_{i}$$

$$\begin{pmatrix}1\\2\\3\end{pmatrix}_{i} \cdot \begin{pmatrix}10\\20\\30\end{pmatrix}_{j} = contract(+, \begin{pmatrix}10 & 20 & 30\\20 & 40 & 60\\30 & 60 & 90\end{pmatrix}_{ij}) = \begin{pmatrix}10 & 20 & 30\\20 & 40 & 60\\30 & 60 & 90\end{pmatrix}_{ij}$$

Fig. 5. Application of the "." function

$$R^{i}{}_{jkl} = \frac{\partial \Gamma^{i}{}_{jl}}{\partial x^{k}} - \frac{\partial \Gamma^{i}{}_{jk}}{\partial x^{l}} + \Gamma^{m}_{jl}\Gamma^{i}{}_{mk} - \Gamma^{m}_{jk}\Gamma^{i}{}_{ml}$$

Fig. 6. Formula of Riemann curvature tensor

```
(define $R~i_j_k_l
  (with-symbols {m}
    (+ (- (∂/∂ Γ~i_j_l x~k) (∂/∂ Γ~i_j_k x~l))
       (- (. Γ~m_j_l Γ~i_m_k) (. Γ~m_j_k Γ~i_m_l)))))
```

Fig. 7. Egison program that represents the formula in Figure 6

```
R=Table[D[Γ[[i,j,l]],x[[k]]] - D[Γ[[i,j,k]],x[[l]]]
       +Sum[Γ[[m,j,l]] Γ[[i,m,k]]
          - Γ[[m,j,k]] Γ[[i,m,l]],
            {m,M}],
        {i,M},{j,M},{k,M},{l,M}]
```

Fig. 8. Wolfram program that represents the formula in Figure 6

differential function "D" is applied to each tensor component, but the differential function "∂/∂" is applied directly to the tensors in Egison.

The differential function "∂/∂" is defined in an Egison program as a scalar function. When a tensor is provided as an argument to a scalar function, the function is applied automatically to each component of the tensor. Therefore, when defining a scalar function, it is sufficient to consider only a scalar as its argument. That is, in the definition of the "∂/∂" function, the programmer need only write the program for the case in which the argument is a scalar value. Despite that, the program "(∂/∂ Γ~i_j_k x~l)" returns a fourth-order tensor.

The program "(∂/∂ Γ~i_j_k x~l)" returns a fourth-order tensor with superscript "i", subscript "j", subscript "k", and subscript "l" from left to right. Here the superscript "~l" of "x~l" is inverted and becomes the subscript "_l", because the differential operator is a special function in tensor analysis and the indices of the tensor applied as the denominator of $\frac{\partial}{\partial}$ are inverted upside down. We will discuss that in detail in Section 3.

Thus, we can naturally import tensor index notation including Einstein notation into programming if we clearly distinguish between tensor functions such as "`.`" and scalar functions such as "+" and "∂/∂". This paper explains this.

The remainder of this paper is structured as follows. In Section 2, we explain existing work to import index notation into programming and its problems. In Section 3, we describe our new method for importing index notation and explain how it solves the existing problems. In Section 4, we present a program for calculating the Riemannian curvature tensor in the Wolfram language and Egison, and we show how the expression is simplified by the method described in Section 3. In the final section, we summarize the contribution of this paper and future issues.

2 EXISTING WORK ON INDEX NOTATION

There are two existing methods for using index notation in programming, a method that introduces special operators supporting index notation and a method that introduces special syntax for index notation.

Using the first method enables index notation to be represented directly in a program. However, this has the disadvantage that index notation can be used only by functions that are specially prepared to use it.

In the second method, we describe the computation of the tensor using syntax such as the `Table` expression of the Wolfram language (Figure 8). This method has the advantage that we can use an arbitrary function defined for scalar values also for tensor operations, similarly to the differential function "`D`" in Figure 8. However, this method has the disadvantage that we cannot directly apply user-defined functions to tensor arguments using index notation. As the result, the description of a program becomes more complicated than the description directly using index notation, as we explained in the previous section using Figure 7 and Figure 8.

2.1 Introduction of Index Notation by Special Operators

For existing work using this method, there is Maxima [1, 15], a computer algebra system that introduces index notation through the extension library itensor, as well as Ahalander's work [3], which implements index notation on C++. These studies introduce index notation by implementing two special functions "+" and "·" that support tensor index notation.

"+" is a function that sums the components of two tensors given as arguments. "·" is a function that takes the tensor product of the two tensors given as arguments and takes the sum of the trace if there are pairs of the superscript and subscript with the same index variable.

"+" and "·" take different actions on tensors with the same index combination. For example, in the following expression, programs corresponding to (1), (4), and (5) are invalid. It is natural to implement index notation in this way because these expressions are meaningless in mathematics.

(1) $c_{ij} = a_i + b_j$
(2) $c_{ij} = a_i \cdot b_j$
(3) $c_{ij} = a_{ij} + b_{ij}$
(4) $c_{ij} = a_{ij} \cdot b_{ij}$
(5) $c = a^{ij} + b_{ij}$
(6) $c = a^{ij} \cdot b_{ij}$

Especially, Ahalander's work [3] can interpret the following expressions, which are meaningless when dealing with a general coordinate system other than the Cartesian coordinate system. In Ahalander's work [3], (7) is interpreted as equivalent to (3), and (8) is interpreted as equivalent to (6). In the range dealing with the Cartesian coordinate system, the index can move up and down freely, with the result that (7) and (3), and (8) and (6) are equivalent in mathematics, making such an interpretation useful.

(7) $c_{ij} = a^{ij} + b_{ij}$
(8) $c = a_{ij} \cdot b_{ij}$

	operator for tensor product	operator for contraction
$\cdot$	*	+
+	+	undefined
$*$	$*$	undefined
∂/∂	∂/∂	undefined

Fig. 9. All scalar functions are regarded as variations of the "·" function whose operator for tensor product is itself and one for contraction is undefined.

$$\begin{pmatrix}1\\2\\3\end{pmatrix}_i + \begin{pmatrix}10\\20\\30\end{pmatrix}_i = \begin{pmatrix}11 & 21 & 31\\12 & 22 & 32\\13 & 23 & 33\end{pmatrix}_{ii} = \begin{pmatrix}11\\22\\33\end{pmatrix}_i$$

Fig. 10. Interpretation of "+" as an operator for tensor product

Thus, in these work, the index rules of "+" and "·" for tensors are defined separately. As a result, "+" and "·" for tensors in the existing work are special operators prepared in the library for index notation, with the consequence that we need to edit the library directly to add new functions that support index notation. That is, it takes substantial effort to define the original operators for tensors.

There are important operations in tensor analysis other than simply adding and multiplying two tensors. For example, in tensor analysis, as we see in connection with "`(∂/∂ Γ~i_j_k x~l)`" in Figure 7, we often differentiate the components of a tensor with respect to the components of another tensor. It is a serious problem that programmers cannot add such operations easily.

2.2 Introduction of Tensor Index Notation by Special Syntax

The crucial difference of this method from the method in Section 2.1 is that we no longer consider the index rules for "+" and "·" separately. In this method, we regard "+" for tensors as a function that computes the tensor product using "+" for scalars, and the function for computing the sum of the trace for contraction is undefined, as shown in Figures 9. Then we can regard "+" as a variation of "·" that never contract the result of the tensor product. As shown Figure 10, we can interpret "+" correctly even if we regard "+" in the above manner. This idea simplifies the index rules. In this method, we control the way we combine tensors only through their indices.

Even if this idea is introduced, the expressions (2), (3), and (6) are still interpreted correctly as before. However, the expressions (1) and (4), which are invalid in mathematics, do not cause errors in this interpretation. Expression (5) still causes an error since the function for calculating the sum of the trace to contract the tensor is undefined for "+". Thus, the introduction of this idea renders the interpretation of some expressions meaningless in mathematics, but we had assigned a higher priority to making the index rules more concise.

This method is adopted primarily in the Wolfram Language, which has a tensor generation syntax, the Table expression [17]. In the Wolfram language, the `Table` expression plays a major role in describing operations that deal with tensors. Using this syntax, we can use arbitrary functions to deal with tensors using a notation with similar to that of index notation. Since this method has such advantages, a program using this method has been introduced by mathematicians in actual research. [9, 10]

For example, the expression $A_{ij} + B_{ij}$ is expressed as follows using the `Table` expression. In the following examples, it is assumed that all dimensions corresponding to each index of the tensors are a constant "`M`".

```
Table[A[[i, j]] + B[[i,j]],
     {i, M},{j, M}]
```

The expression $A_{ij}B_{kl}$ is expressed as follows.

```
Table[A[[i, j]] * B[[k,l]],
     {i, M},{j, M},{k, M},{l, M}]
```

In the Wolfram language, it is possible to express both addition of tensors and the tensor product in a unified manner using the `Table` expression. In addition, we can use arbitrary functions as an operator for tensor product, similarly to the manner in which the differential function "`D`" is used in Figure 8. Thus, parameterization of an operator for tensor product, as described in Section 2.1 is realized in the Wolfram language.

A program that contracts tensors is described by combining the `Table` and `Sum` expressions. For example, the expression $A^{ij}T_{jkl}$ is as follows. In the Wolfram language, we do not explicitly specify whether tensor indices are superscripts or subscripts. We need to determine from the context whether the index of the tensor in a program is a superscript or subscript.

```
Table[Sum[A[[i, j]] * T[[j, k, l]], {j, M}],
     {i, M},{k, M},{l, M}]
```

We can use a different aggregate function instead of the `Sum` expression. This means that the parameterization of an operator for contraction described in Section 2.1 is also realized in the Wolfram language.

In the Wolfram language, parameterizations of an operator for tensor product and contraction are both realized. However, it has a disadvantage that it always requires to use the `Table` expression for tensor operation using index notation. The Wolfram language does not allow to directly apply a function using index notation as Figures 2, 3, and 5 in Section 1. Furthermore, it is impossible to modularize an operation, such as "·" described in Section 2.1, whose behavior varies depending on the combination of indices of the tensors of the argument. That is, expressing such operations requires us to write the `Sum` expression nested in the `Table` expression every time, because the tensor of the Wolfram language does not contain index information. Consequently, the program becomes more complicated than the mathematical expression.

Domain Specific Languages (DSLs) for computational chemistry [14] also support index notation including Einstein summarize notation in this method. For example, the expression $A_{ij}T_{jkl}$ is expressed as follows. In this work, superscripts and subscripts are not distinguished as the Wolfram language.

```
B["ikl"] = A["ij"] * T["jkl"]
```

The above program directly represents index notation. However, this method has two drawbacks.

First, it has a constraint that we always need to specify the type of the tensor returned by expressions using index notation. That is, we need to write a left-side expression as "`B["ikl"]`" in the above example. This constraint is caused by the ambiguity of the interpretation of the right-side expression of (4) in Section 2.1. This work allows to interpret the right-side expression of (4) in both manners, (4) and (8). Therefore, we need to specify the manner in which we interpret the expressions using index notation by specifying the type of the tensor returned by them.

Our proposal eliminates this ambiguity by distinguishing superscripts and subscripts, and ceasing the interpretation of the expressions such as (8) that are valid only for the Cartesian coordinate system in which index can move up and down freely. It enables us to directly apply a function to tensor arguments using index notation as Figures 2, 3, and 5 in Section 1.

The second disadvantage is that it requires additional description to temporarily change a function for calculating the tensor product and trace. In this work, the additive and multiplicative operators defined as the class methods of the class to which the elements of the tensor belong are used to caluculate the trace and tensor product, respectively. This is the reason why the above program do not specify the operator to use when calculating the trace corresponding to the `Sum` expression in the above Wolfram program. However, it requires additional description when we specify a function to use when calculating the tensor product as follows. In the following example, we specify the `min` function as the operator for calculating the tensor product.

```
((Transform<>)([] (double a, double b, double & c){ c = min(a,b); }))(A["i"],B["j"],C["ij"]);
```

The above program is equivalent to the following program.

```
for (int i=0; i<n; i++){
  for (int j=0; j<n; j++){
    C[i,j] = min(A[i],B[j]);
  }
}
```

We avoid this problem by introducing the concept of two types of functions, scalar and tensor functions. As the result, we can directly apply arbitrary user-defined functions using index notation, as we directly apply the "+", "-", "∂/∂", and "`.`" functions to tensors in Figure 7 in Section 1.

3 A NEW METHOD FOR IMPORTING INDEX NOTATION INTO PROGRAMMING

As mentioned in Section 2, the existing methods for importing tensor index notation have a disadvantage that we cannot directly apply arbitrary user-defined functions to tensor arguments using index notation. We can overcome this disadvantage if we satisfy all of the following three conditions at the same time.

No ambiguity in tensor index rules We do not need to specify the tensor type of the return value of an expression using index notation.

Parameterization of an operator for tensor product We can specify a function to use when calculating the tensor product with a parameter.

Parameterization of an operator for contraction We can specify a function to use when calculating the sum of the trace to contract tensors with a parameter.

In the work explained in section 2.1, the first condition is satisfied, but the second and third are not. In contrast, in the work explained in Section 2.2, the second and third conditions are satisfied, but the first is not. Therefore, both of the sections have relevant disadvantages.

This section discusses a means of satisfying all of these conditions simultaneously in programming languages.

3.1 Grammar

Figure 11 shows the syntax added to implement the proposed method in Egison. By implementing the same syntax, we can import index notation into programming languages other than Egison. In Section 3, we explain how to implement our proposal by explaining the semantics of this grammar.

In Figure 11, ⟨*scalar*⟩ represents scalar values such as numbers ("`1`", "`2`", "`(/ 3 2)`") and expressions ("`(+ x y)`", "`x^2`", "`(cos θ)`"). `tensor-map`, `contract`, `flip-indices`, and `generate-tensor` are primitive syntax to handle tensors.

We use Section 3.2 and 3.3 to explain our index reduction rules for tensors with symbolic indices. We show various examples to show their validity in the subsequent sections.

3.2 Reduction Rules for Tensors with Indices

In this section, we show the index reduction rules for a single tensor with indices.

First, we explain the notation for expressing tensors in Egison. Egison expresses a tensor by enclosing its components with "`[|`" and "`|]`". We express a higher-order tensor by nesting this description, as we do for an *n*-dimensional array.

To access the components of a tensor, we add indices to the tensor. Subscripts are represented by "`_`" followed by a natural number after the tensor. An arbitrary number of indices can be added to one tensor, though adding a number of indices larger than the rank of the target tensor results in an error.

```
[|[|11 12 13|] [|21 22 23|] [|31 32 33|]|]_2
;[|21 22 23|]
```

⟨*expr*⟩ ::= ⟨*tensor*⟩ | ‘(’ ⟨*expr*⟩ [⟨*expr*⟩ ...] ‘)’ | ‘(with-symbols {’ [⟨*symbol*⟩ ...] ‘}’ ⟨*expr*⟩ ‘)’ | ‘(tensor-map’ ⟨*function*⟩ ⟨*tensor*⟩ ‘)’ | ‘(contract’ ⟨*function*⟩ ⟨*tensor*⟩ ‘)’ | ‘(flip-indices’ ⟨*tensor*⟩ ‘)’

⟨*tensor*⟩ ::= ⟨*tensor-data*⟩ [⟨*index*⟩ ...]

⟨*tensor-data*⟩ ::= ⟨*variable-name*⟩ | ⟨*scalar*⟩ | ⟨*function*⟩ | ‘[|’ ⟨*tensor-data*⟩ ... ‘|]’ | ‘(generate-tensor ’ ⟨*function*⟩ ‘{’ [⟨*natural-number*⟩ ...] ‘})’

⟨*index*⟩ ::= ⟨*index-type*⟩ ⟨*natural-number*⟩ | ⟨*index-type*⟩ ⟨*symbol*⟩ | ⟨*index-type*⟩ ‘#’

⟨*index-type*⟩ ::= ‘~’ | ‘_’ | ‘~_’

⟨*function*⟩ ::= ‘(lambda’ ‘[’ [⟨*parameter*⟩ ...] ‘]’ ⟨*expr*⟩ ‘)’ | ⟨*builtin-scalar-function*⟩

⟨*parameter*⟩ ::= ‘%’ ⟨*variable-name*⟩ | ‘$’ ⟨*variable-name*⟩ | ‘*$’ ⟨*variable-name*⟩

⟨*builtin-scalar-function*⟩ ::= ‘+’ | ‘-’ | ‘*’ | ‘/’ | ...

Fig. 11. Grammar for index notation

```
[|[|11 12 13|] [|21 22 23|] [|31 32 33|]|]_2_1
;21
```

We can use symbols as well as natural numbers as indices. Egison is a computer algebra system and supports symbolic computation. Unbound variables are treated as symbols. We declare the indices of a tensor by using the symbols for the indices. If multiple indices of the same symbol appear, Egison converts it to the tensor composed of diagonal components for these indices. After this conversion, the leftmost index symbol remains. For example, the indices “`_i_j_i`” convert to “`_i_j`”.

```
[|[|11 12 13|] [|21 22 23|] [|31 32 33|]|]_i_j
;[|[|11 12 13|] [|21 22 23|] [|31 32 33|]|]_i_j

[|[|11 12 13|] [|21 22 23|] [|31 32 33|]|]_i_i
;[|11 22 33|]_i

[|[|[|1 2|] [|3 4|]|] [|[|5 6|] [|7 8|]|]|]_i_j_i
;[|[|1 3|] [|6 8|]|]_i_j
```

When three or more subscripts of the same symbol appear, Egison converts it to the tensor composed of diagonal components for all these indices.

```
[|[|[|1 2|] [|3 4|]|] [|[|5 6|] [|7 8|]|]|]_i_i_i
;[|1 8|]_i
```

Egison supports two types of indices, both superscripts and subscripts. A subscript is represented by “`_`”. A superscript is represented by “`~`”.

Superscripts and subscripts behave symmetrically. When only superscripts are used, they behave in exactly the same manner as when only subscripts are used.

```
[|[|11 12 13|] [|21 22 23|] [|31 32 33|]|]~1~1
;11

[|[|[|1 2|] [|3 4|]|] [|[|5 6|] [|7 8|]|]|]~i~j~i
;[|[|1 3|] [|6 8|]|]~i~j
```

The index reduction rules thus far are the same as those of the existing work [3].

```
E({A, xs}) =
  if e(xs) = [] then
    {A, xs})
  elsif e(xs) = [{k,j}, . . . ] & p(k,xs) = p(j,xs) then
    E({diag(k, j, A), remove(j, xs))
  elsif e(xs) = [{k,j}, . . . ] & p(k,xs) != p(j,xs) then
    E({diag(k, j, A), update(k, 0, remove(j, xs)))
```

Fig. 12. Pseudo code of index reduction

Let us consider a case in which the same symbols are used for a superscript and a subscript. In this case, the tensor is automatically contracted using "+" in the existing research. In contrast, Egison converts it to the tensor composed of diagonal components, as in the above examples. However, in that case, the summarized indices become a *supersubscript*, which is represented by "~_".

```
[|[|11 12 13|] [|21 22 23|] [|31 32 33|]|]~i_i
;[|11 22 33|]~_i
```

Even when three or more indices of the same symbol appear that contain both supersubscripts and subscripts, Egison converts it to the tensor composed of diagonal components for all these indices.

```
[|[|[|1 2|] [|3 4|]|] [|[|5 6|] [|7 8|]|]|]~i~i_i
;[|1 8|]~_i
```

The reason not to contract it immediately is to enable it to parameterize an operator for contraction. The components of supersubscripts can be contracted by using the `contract` expression. The `contract` expression receives a function to be used for contraction as the first argument, and a target tensor as the second argument.

```
(contract + [|11 22 33|]~_i)
;66
```

Figure 12 shows pseudo code of index reduction as explained in this section. `E(A,xs)` is a function for reducing a tensor with indices. `A` is an array that consists of tensor components. `xs` is a list of indices appended to `A`. For example, `E(A,xs)` works as follows with the tensor whose indices are "`~i_j_i`". We use "`1`", "`-1`", and "`0`" to represent a superscript, subscript, and supersubscript, respectively.

```
E({[|[|[|1 2|] [|3 4|]|] [|[|5 6|] [|7 8|]|]|],
  [{i,1}, {j,1}, {i,-1}]}) =
{[|[|1 3|] [|6 8|]|], [{i,0}, {j,1}]}
```

Next, we explain the helper functions used in Figure 12. `e(xs)` is a function for finding pairs of identical indices from `xs`. `diag(k, j, A)` is a function for creating the tensor that consists of diagonal components of `A` for the `k`-th and `j`-th order. `remove(k, xs)` is a function for removing the `k`-th element from `xs`. `p(k, xs)` is a function for obtaining the value of the `k`-th element of the assoc list `xs`. `update(xs, k, p)` is a function for updating the value of the `k`-th element of the assoc list `xs` to `p`. These functions work as follows.

```
e([{i, 1}, {j, -1}, {i, 1}])        = [{1,3}]
diag(1, 2, [|[|11 12|] [|21 22|]|]) = [|11 22|]
p(2, 0, [{i, 1}, {j, -1}])          = -1
remove(2, [{i, 1}, {j, -1}])        = [{i, 1}]
update(2, 0, [{i, 1}, {j, -1}])     = [{i, 1}, {j, 0}]
```

3.3 Scalar and Tensor Functions

In this section we introduce the concept of two types of functions: scalar and tensor functions. This enables us to introduce naturally the parameterization of an operator for tensor product and contraction when we combine it with the index reduction rules explained in Section 3.2.

Tensor functions are functions that involve contraction of the tensors provided as an argument, and scalar functions are not. For example, the inner product of vectors and matrix multiplication are tensor functions. In contrast, "+", " -", "*", and "/" are scalar functions.

We use the concept of two types of parameters, *scalar parameters* and *tensor parameters*, to specify whether the function we defined is a scalar or tensor function. Similar to Scheme, Egison generates a function using a `lambda` expression. In the `lambda` expression, we add "$" or "%" to the beginning of the parameters. A parameter to which "$" is prefixed is a scalar parameter. A parameter to which "%" is prefixed is a tensor parameter.

As with ordinary parameters, when a tensor parameter obtains a tensor as an argument, the function treats the tensor as it is. In contrast, when a scalar parameter obtains a tensor as an argument, the function is applied to each component of the tensor. A function with scalar parameters is converted to a function only with tensor parameters by using the `tensor-map` function as follows. In this way, we can implement scalar parameters.

```
(lambda [$x $y] ...)
;=>(lambda [%x %y]
     (tensor-map (lambda [%x]
                   (tensor-map (lambda [%y] ...)
                               y))
                 x))
```

As the name implies, the `tensor-map` function applies the function of the first argument to each component of the tensor provided as the second argument. If the result of applying the function of the first argument to each component of the tensor provided as the second argument is a tensor with indices, it moves those indices to the end of the tensor that is the result of evaluating the `tensor-map` function. We will see an example in a later part of this section.

Let's review the `min` function defined in Figure 1 in Section 1 as an example of a scalar function. This `min` function can handle tensors as arguments as follows.

```
(min [|1 2 3|]_i [|10 20 30|]_j)
;[|[|1 1 1|] [|2 2 2|] [|3 3 3|]|]_i_j

(min [|1 2 3|]_i [|10 20 30|]_i)
;[| 1 2 3 |]_i;
```

Note that the tensor indices of the evaluated result are "`_i_j`". If the `tensor-map` function simply applies the function to each component of the tensor, the result of this program will be similar to
"`[|[|1 1 1|]_j [|2 2 2|]_j [|3 3 3|]_j|]_i`". However, as explained above if the results of applying the function to each component of the tensor are tensors with indices, it moves those indices to the end of the tensor that is the result of evaluating the `tensor-map` function. This is the reason that the indices of the evaluated result are "`_i_j`". This mechanism enables us to directly apply scalar functions to tensor arguments using index notation as the above example.

The above evaluation result is equal to the result of specifying the `min` function as the operator of the tensor product. By defining a scalar function as described above, the parameterization of an operator for the tensor product is achieved without bringing it to programmers' attention. Proposing the concept of scalar functions is one of the major contributions of this paper.

Next, let's review the "`.`" function defined in Figure 4 in Section 1 as an example of a tensor function. All of the parameters of the "`.`" function are tensor parameters and "%" is prepended to the beginning of the parameters. This "`.`" function can handle tensors as arguments as follows.

```
(. [|1 2 3|]~i [|10 20 30|]_i)
;140

(. [|1 2 3|]_i [|10 20 30|]_j)
```

```
;[| [| 10 20 30 |] [| 20 40 60 |] [| 30 60 90 |] |]_i_j

(. [|1 2 3|]_i [|10 20 30|]_i)
;[| 10 40 90 |]_i
```

When a tensor with indices is given as an argument of a tensor function, it is passed to the tensor function maintaining its indices. Note that we can directly apply tensor functions to tensor arguments using index notation as in the above example.

By changing "*" and "+" appearing in Figure 4 to different functions, we can define a new tensor multiplication operator that uses the functions we specified to calculate the tensor product and the sum of the traces for contraction.

Since a tensor parameter is used only when defining a function that contracts tensors, in most cases only scalar parameters are used.

3.4 Application of Scalar Functions to Tensors

In Section 3.3, we introduced scalar and tensor parameters, and explained the behavior of scalar and tensor functions. The definition of scalar functions in Section 3.3 is highly compatible with the index reduction rules for a single tensor with indices explained in Section 3.2. In this section, we confirm this fact seeing various samples of a scalar function receiving indexed tensors as arguments.

In the following sample, the tensor that has the subscript "i" and the subscript "j" is applied to the scalar function "+". If the indices of the tensors of the arguments are different in this manner, it returns the tensor product using the scalar function as the operator, as we saw in Section 3.3.

```
(+ [|1 2 3|]_i [|10 20 30|]_j)
;[|[|11 21 31|] [|12 22 32|] [|13 23 33|]|]_i_j
```

In the following example, we add the same index "i" to both `[|1 2 3|]` and `[|10 20 30|]`. When the indices of the tensors given as arguments are identical, the scalar function is applied to each corresponding component. An error occurs if the dimensions are different even though the indices are identical. Note that this result is equal to the result of simplifying "`[|[|11 21 31|] [|12 22 32|] [|13 23 33|]|]_i_i`" by the reduction rules in Section 3.2. Review Figure 10 to clarify the idea underlying this transformation.

```
(+ [|1 2 3|]_i [|10 20 30|]_i)
;[|11 22 33|]_i
```

Both arguments are vectors in the above two examples. Next, let us see examples in which the arguments are higher-order tensors, as follows. We can see that the reduction rules work well even for high-order tensors.

```
(+ [|[|11 12|] [|21 22|] [|31 32|]|]_i_j [|100 200 300|]_i)
;[|[|111 112|] [|221 222|] [|331 332|]|]_i_j

(+ [|[|1 2 3|] [|10 20 30|]|]_i_j [|100 200 300|]_j)
;[|[|101 202 303|] [|110 220 330|]|]_i_j
```

As mentioned in Section 3.3, arbitrary scalar functions behave in the same manner as the above examples.

The "∂/∂" function appearing in Figure 7 is also a scalar function. However, "∂/∂" is not a normal scalar function. "∂/∂" is a scalar function that inverts indices of the tensor given as its second argument. For example, the program "`(∂/∂ Γ~i_j_k x~l)`" returns the fourth-order tensor with superscript "i", subscript "j", subscript "k", and subscript "l" from left to right.

To define scalar functions such as "∂/∂", we use *inverted scalar parameters*. Inverted scalar parameters are represented by "*$". A program that uses inverted scalar parameters is transformed as follows. Here, the `flip-indices` function is a primitive function for inverting the indices of a tensor provided as an argument upside down. Supersubscripts remain as supersubscripts even if they are inverted.

```
(define $∂/∂ (lambda [$f *$x] ...))
;=>(define $∂/∂ (lambda [%f %x]
     (tensor-map (lambda [%x] (tensor-map (lambda [%y] ...)
                                          (flip-indices x)))
                 f))
```

The definition of "∂/∂" can be seen in the GitHub repository[1].

In the following example, we apply "∂/∂" to tensors.

```
(∂/∂ [|(* r (sin θ)) (* r (cos θ))|]_i [|r θ|]_j)
;[|[|(sin θ) (* r (cos θ))|]
;  [|(cos θ) (* -1 r (sin θ))|]|]_i~j

(∂/∂ [|(* r (sin θ)) (* r (cos θ))|]_i [|r θ|]_i)
;[|(sin θ) (* -1 r (sin θ))|]~_i
```

In the writing of a program that deals with high-order tensors, the number of symbols used for indices increases. A dummy symbol is introduced to suppress that. "#" represents a dummy symbol. All instances of "#" are treated as different symbols. Using this mechanism makes it easier to distinguish indices that are important in the program, thereby also improving the readability of the program.

The idea of dummy symbols is not new. For example, there is syntax to generate local symbols in the Wolfram languages [16] and Maxima[2]. We can use such syntax to generate dummy symbols, though its primary purpose is to generate temporary symbols for substituting variables.

The novelty of this paper on dummy symbols is that we prepared one-character syntax "#" to generate dummy symbols to be used as indices of tensors. This is a very simple idea, but since this notation is not used even in mathematics, we think this idea is new.

```
(+ [|1 2 3|]_# [|10 20 30|]_#)
;[|[|11 21 31|] [|12 22 32|] [|13 23 33|]|]_#_#
```

Egison allows programmers to omit indices while recommending that programmers explicitly specify indices. If the indices are omitted, Egison handles the expression in the same manner as dummy symbols are omitted.

```
(+ [|1 2 3|] [|10 20 30|])
;[|[|11 21 31|] [|12 22 32|] [|13 23 33|]|]
```

Other computer algebra systems, including the Wolfram language, handle expressions in the same manner, as the same indices are added to all arguments where indices are omitted.

```
[10,20,30] + [1,2,3] = [11,22,33]
```

Egison selects the above specification to avoid creating difficulty for the interpreter in complementing the part of an expression in which programmers omit the description. Regardless of which specification is adopted, when a scalar function takes a scalar and a tensor as arguments, it performs as follows.

```
(+ [|1 2 3|] 10)
;[|11 12 13|]
```

3.5 The `with-symbols` Expression

The `with-symbols` expression is syntax for generating new local symbols, such as the `Module` expression in the Wolfram language [16].

One-character symbols that are often used as indices of tensors such as "`i`", "`j`", and "`k`" are often used in another part of a program. Generating local symbols using `with-symbols` expressions enables us to avoid variable conflicts for such symbols.

[1]https://github.com/egison/egison/blob/master/lib/math/analysis/derivative.egi

The `with-symbols` expression takes a list of symbols as its first argument. These symbols are valid only in the expression given in the second argument of the `with-symbols` expression.

```
(with-symbols {i} (contract + (* [|1 2 3|]~i [|10 20 30|]_i)))
;60
```

If the evaluation result of the body of the `with-symbols` expression contains the symbols generated by the `with-symbols` expressions, those symbols are converted into the dummy symbols described in Section 3.4. As a result of this mechanism, local symbols never appear in the result of the `with-symbols` expression. This mechanism enables us to use local symbols as indices of tensors that often remain in the evaluation result.

```
(with-symbols {i} (+ [|1 2 3|]_i [|10 20 30|]_i))
;[|11 22 33|]_#
```

3.6 Definitions of Tensor Functions

In this section, we define the function that calculates the inner product of vectors and matrix multiplication.

We can define the function for calculating the inner product as follows. This function is a tensor function, because it contracts the tensor.

```
(define $inner-product (lambda [%v1 %v2]
    (with-symbols {i} (contract + (* v1~i v2_i)))))
```

Unlike the "`.`" function defined in Section 3.3, this `inner-product` function assumes that no index is appended to the vectors given as arguments, such as "`(inner-product [|1 2 3|] [|10 20 30|])`". Therefore, the index is appended in the function definition as "`v1~i`" and "`v2_i`". If the vector provided as an argument has an index, it will be overwritten.

We can define the function for multiplying matrices as follows. This function is a tensor function as well.

```
(define $mat-mul (lambda [%m1 %m2]
    (with-symbols {j} (contract + (* m1~#~j m2_j_#)))))
```

The `mat-mul` function also assumes that no indices are appended to the matrices provided as arguments. Therefore, the indices are appended in the function definition as "`m1~#_j`" and "`m2_j_#`". Note that dummy symbols are again used effectively in this program. Without dummy symbols, the program "`(* m1~#~j m2_j_#)`" is represented "`(* m1~i~j m2_j_k)`". We avoid declaring the additional symbols "`i`" and "`k`" by using dummy symbols.

Thus, if we define a tensor function using `with-symbols` expressions, we can define functions that can be used without knowledge of index notation.

3.7 Tensor Generation Syntax and Pattern-Matching

Egison has a `generate-tensor` expression that is tensor generation syntax having essentially the same meaning as the `Table` expression in the Wolfram language. This syntax is even more powerful and provides for programs that initiate a complicated matrix more simply when combined with nonlinear pattern-matching implemented in Egison [4]. This section discusses this feature.

The `generate-tensor` expression takes a function as the first argument and the size of the tensor to be generated as the second argument. The number of arguments of the function of the first argument is equal to the rank of the tensor to be generated. Each component of the generated tensor is the result of applying the indices to the function of the first argument.

For example, a unit matrix can be initialized as follows.

```
(generate-tensor
  (match-lambda [integer integer]
    {[[$i ,i] 1] [[_ _] 0]})
  {4 4})
```

```
;[|[|1 0 0 0|] [|0 1 0 0|] [|0 0 1 0|] [|0 0 0 1|]|]
```

As a more complicated example, the matrix for generating the N-bonacci sequence can be initialized as follows. A program for calculating N-bonacci sequence using this matrix is available in the Egison website[2].

```
(generate-tensor
  (match-lambda [integer integer]
    {[[,1 _] 1]
     [[$x ,(- x 1)] 1]
     [[_ _] 0]})
  {4 4})
;[|[|1 1 1 1|] [|1 0 0 0|] [|0 1 0 0|] [|0 0 1 0|]|]
```

We consider the fact that pattern-matching is useful for initializing complicated tensors in a simple manner to be evidence for the necessity of pattern-matching with strong expressive power, as discussed in [4]. We will not discuss this in more detail here, because it is not the principal topic of this paper.

4 CALCULATION OF RIEMANN CURVATURE TENSOR

In this section, we present programs for calculating the Riemannian curvature tensor [6, 12, 13], the fourth-order tensor that expresses the curvature of a manifold, in the Wolfram language and Egison, to shows the advantages of our proposal.

Figures 13 and 14 show programs for calculating the Riemann curvature tensor of a torus in the Wolfram language and in Egison, respectively.

In Egison, when binding a tensor to a variable, we can specify the type of indices in the variable name. For example, we can bind different tensors to "`$g__`", "`$g~~`", "`$Γ___`", and "`Γ~__`". This feature is also implemented in the existing work described in Section 2.1. This feature simplifies variable names.

In Egison, some of the tensors are bound to a variable with symbolic indices such as "`$Γ_i_j_k`". It is automatically desugared as follows. This syntactic sugar renders a program closer to the mathematical expression. `transpose` is a function for transposing the tensor in the second argument as specified in the first argument.

```
(define $Γ_i_j_k ...)
;=>(define $Γ___
     (with-symbols {i j k} (transpose {i j k} ...)))
```

Except for that point, the programs for calculating the torus coordinates and the metric tensor differ only in the appearance of the syntax. On the other hand, our proposal introduces essential differences in the programs for calculating the local basis, Christoffel symbols of the first and second kind, and Riemann curvature tensor. This section explains these differences.

First of all, the Wolfram language uses the `Table` expression in the program for calculating the local basis, but Egison has a flat description using scalar functions. The `flip` function used in the Egison program is a function for swapping the arguments of a two-argument function. It is used to transpose the result matrix.

Next, let us examine the programs for calculating Christoffel symbols of the first and second kind, and the Riemann curvature tensor. They are defined in mathematics as follows.

$$\Gamma_{ijk} = \frac{1}{2}(\frac{\partial g_{ij}}{\partial x^k} + \frac{\partial g_{ik}}{\partial x^j} - \frac{\partial g_{kj}}{\partial x^i})$$

$$\Gamma^i_{kl} = g^{ij}\Gamma_{jkl}$$

[2]https://www.egison.org/math/tribonacci.html

```
(* Coordinates for Torus *)
M=2;
x={θ,φ};
X={(a*Cos[θ]+b)Cos[φ], (a*Cos[θ]+b)Sin[φ], a*Sin[θ]};

(* Local basis *)
e=Table[D[X[[j]],x[[i]]],{i,2},{j,3}] //ExpandAll//Simplify;

(* Metric tensor *)
g=Table[e[[i]].e[[j]],{i,M},{j,M}] //ExpandAll//Simplify;
Ig=Inverse[g] //ExpandAll//Simplify;

(* Christoffel symbols of the first kind *)
Γ1=Table[D[g[[i,j]],x[[k]]] + D[g[[i,k]],x[[j]]]
         - D[g[[j,k]],x[[i]]],
         {i,M},{j,M},{k,M}]/2 //ExpandAll//Simplify;

(* Christoffel symbols of the second kind *)
Γ2=Table[Sum[Ig[[i,j]]Γ1[[j,k,l]],{j,M}],
         {i,M},{k,M},{l,M}] //ExpandAll//Simplify;

(* Riemann curvature tensor *)
R=Table[D[Γ2[[i,j,l]],x[[k]]] - D[Γ2[[i,j,k]],x[[l]]]
       +Sum[Γ2[[m,j,l]]Γ2[[i,m,k]]
          - Γ2[[m,j,k]]Γ2[[i,m,l]],
            {m,M}],
        {i,M},{j,M},{k,M},{l,M}] //ExpandAll//Simplify;
```

Fig. 13. A program to calculate Riemann curvature tensor in Wolfram language

$$R^i{}_{jkl} = \frac{\partial \Gamma^i{}_{jl}}{\partial x^k} - \frac{\partial \Gamma^i{}_{jk}}{\partial x^l} + \Gamma^m_{jl}\Gamma^i{}_{mk} - \Gamma^m_{jk}\Gamma^i{}_{ml}$$

The essential difference between Figures 13 and 14 is that the Wolfram language uses `Table` expressions to represent the above three equations, whereas Egison uses index notation directly. In particular, in the program for calculating the Riemann curvature tensor, a double loop consisting of the `Table` and `Sum` expressions appears in the Wolfram language, whereas the Egison program is as flat as the mathematical expression.

From the above discussion, we can conclude that Egison expresses mathematical expressions more directly than the Wolfram language, though there is little difference in the number of lines in the programs.

5 CONCLUSION

5.1 Contributions

In this paper, we introduced index notation with simpler index rules than in the existing work, as explained in Sections 3.2 and 3.3. Additionally, we introduced the concept of two types of functions, scalar and tensor functions, as explained in Sections 3.3 and 3.4. We showed that the combination of these two ideas enables us to directly apply arbitrary user-defined functions to tensor arguments using index notation. We also proposed that these two types of functions can be defined using two types of parameters, scalar and tensor parameters. This proposal eliminates the need to consider the case in which the argument is a tensor when defining scalar functions, which appear in an overwhelming proportion of programs.

```
;; Coordinates for Torus
(define $x [|θ φ|])
(define $X [|(* '(+ (* a (cos θ)) b) (cos φ)) ; = x
             (* '(+ (* a (cos θ)) b) (sin φ)) ; = y
             (* a (sin θ))|])                ; = z

;; Local basis
(define $e ((flip ∂/∂) x~# X_#))

;; Metric tensor
(define $g__ (generate-tensor 2#(V.* e_%1 e_%2) {2 2}))
(define $g~~ (M.inverse g_#_#))

;; Christoffel symbols of the first kind
(define $Γ_i_j_k
  (* (/ 1 2)
     (+ (∂/∂ g_i_j x_k)
        (∂/∂ g_i_k x_j)
        (* -1 (∂/∂ g_j_k x_i)))))

;; Christoffel symbols of the second kind
(define $Γ~__ (with-symbols {i} (. g~#~i Γ_i_#_#)))

;; Riemann curvature tensor
(define $R~i_j_k_l
  (with-symbols {m}
    (+ (- (∂/∂ Γ~i_j_l x_k) (∂/∂ Γ~i_j_k x_l))
       (- (. Γ~m_j_l Γ~i_m_k) (. Γ~m_j_k Γ~i_m_l)))))
```

Fig. 14. A program to calculate Riemann curvature tensor in Egison

In addition, our proposal achieved lexical scoping of symbols used as tensor indices by the `with-symbols` expression and a dummy symbol "#", as explained in Sections 3.5 and 3.6. This is also our important contribution since there is no literature that discuss this topic.

This paper also showed the usefulness of a dummy symbol "#" and of using it as an index of tensors. This idea allows us to reduce the number of symbols used as indices by replacing symbols that appear only once with dummy symbols "#". Although this is a very simple idea, it improves the readability of programs by highlighting important indices.

5.2 Future Work

In this paper, we introduced several forms of syntax into a language to introduce the new concepts of scalar and tensor functions. However, we think it is possible to introduce the concepts of scalar and tensor functions even using a static type system or the overloading feature of object-oriented programming. For example, in a static type system, whether the parameter of a function is a scalar or tensor parameter can be specified when specifying the type of the function. Although we could not discuss this in this paper, it is an interesting research topic to think about how to incorporate the ideas proposed in this paper into existing programming languages naturally.

In particular, it is of substantial significance to incorporate this method into programming languages such as Formura [11] and Diderot [8] that have a compiler that generates code for executing tensor calculation in parallel. For example, incorporating this method into Formura would enable us to describe physical simulation using not only the Cartesian coordinate system but also more general coordinate systems such as the polar and spherical coordinate system in simple programs.

By the way, index notation as discussed in this paper is a notation invented over a century ago. Especially, it is well known that Einstein summation notation was invented by Einstein when he was working on general relativity theory. In addition to index notation, there might still be many notations in mathematics that are useful, but not yet introduced into programming. There might also be notations that describe the formulas of existing theories more concisely, but that mathematicians have not discovered yet.

We contend that it is very useful for those researching programming languages who are familiar with many programming paradigms and can flexibly create new programming languages to learn a wider range of mathematics for the future of programming languages.

ACKNOWLEDGMENTS

I thank Hiromi Hirano, Hidehiko Masuhara, and Michal J. Gajda for very helpful feedback on the earlier versions of the paper.

REFERENCES

[1] 2016. Maxima - a Computer Algebra System. (2016). http://maxima.sourceforge.net/.
[2] 2016. SymPy User's Guide - SymPy 1.0.1.dev documentation. (2016). http://docs.sympy.org/dev/guide.html.
[3] Krister Åhlander. 2002. Einstein summation for multidimensional arrays. *Computers & Mathematics with Applications* 44, 8-9 (2002), 1007–1017.
[4] Satoshi Egi. 2014. Non-Linear Pattern-Matching against Non-free Data Types with Lexical Scoping. *arXiv preprint arXiv:1407.0729* (2014).
[5] Satoshi Egi. 2016. The Egison Programming Language. (2016). https://www.egison.org.
[6] Daniel A Fleisch. 2011. *A student's guide to vectors and tensors.* Cambridge University Press.
[7] Richard Hartley and Andrew Zisserman. 2003. *Multiple view geometry in computer vision.* Cambridge university press.
[8] Gordon Kindlmann et al. 2016. Diderot: a domain-specific language for portable parallel scientific visualization and image analysis. *IEEE transactions on visualization and computer graphics* 22, 1 (2016), 867–876.
[9] Yoshiaki Maeda et al. 2010. Computation of the Wodzicki-Chern-Simons form in local coordinates. Computations for S^1 actions on $S^2 \times S^3$. (2010). http://math.bu.edu/people/sr/articles/ComputationsChernSimonsS2xS3_July_1_2010.pdf.
[10] Yoshiaki Maeda et al. 2016. The geometry of loop spaces II: Characteristic classes. *Advances in Mathematics* 287 (2016), 485–518.
[11] Takayuki Muranushi et al. 2016. Automatic generation of efficient codes from mathematical descriptions of stencil computation. In *Proceedings of the 5th International Workshop on Functional High-Performance Computing.* ACM, 17–22.
[12] Yann Ollivier. 2011. A visual introduction to Riemannian curvatures and some discrete generalizations. *Analysis and Geometry of Metric Measure Spaces: Lecture Notes of the 50th Séminaire de Mathématiques Supérieures (SMS), Montréal* (2011), 197–219.
[13] Bernard F Schutz. 1980. *Geometrical methods of mathematical physics.* Cambridge university press.
[14] Edgar Solomonik and Torsten Hoefler. 2015. Sparse Tensor Algebra as a Parallel Programming Model. *arXiv preprint arXiv:1512.00066* (2015).
[15] Viktor Toth. 2005. Tensor manipulation in GPL Maxima. *arXiv preprint cs/0503073* (2005).
[16] Wolfram. 2016. Module - Wolfram Language Documentation. (2016). http://reference.wolfram.com/language/ref/Module.html.
[17] Wolfram. 2016. Table - Wolfram Language Documentation. (2016). http://reference.wolfram.com/language/ref/Table.html.

Extending the LISP model: from cons cells to triples, from trees to hypergraphs

JOSEPH CORNELI AND RAYMOND PUZIO

Arxana is a higher-dimensional variant of LISP, based on nested semantic networks instead of cons cells. In contradistinction to LISP where the fundamental building block is a cell '`(a . b)`', Arxana's fundamental building block is a triple, '`(a c b)`'. Furthermore, in the language of the Semantic Web, every triple is 'reified'. Links and their constituent positions can contain further structure or be augmented with offset annotations: for example, we distinguish between '`((a d e) c b)`' and '`(a c b)` $\oplus_1$ `(a d e)`'. The first form models an assertion about the link '`(a d e)`', and the second models an assertion about the atom '`a`' within '`(a c b)`'. These facilities allow us to build, reason about, query, and program with hypergraphs rather than trees or networks. This representation strategy is useful for building runnable conceptual models of complex and recursive structures. Modelling informal mathematical discourse is a motivating application: this requires a different approach from the strictly deductive style of formal mathematics. Other programming languages which support a similar annotative style include Kurzweil's *Flare* and Nelson's *ZigZag*. Our Arxana prototypes are implemented in Emacs Lisp.

Toward Parallel CFA with Datalog, MPI, and CUDA

THOMAS GILRAY, University of Maryland
SIDHARTH KUMAR, University of Utah

We present our recent experience working to design parallel functional control-flow analysis (CFA) using an encoding in Datalog and underlying relational algebra implemented for SIMD coprocessors and supercomputers. Control-flow analysis statically models the possible propagations of data and control through a target program, finitely obtaining a bound on reachable expressions and environments and on possible return and argument values. We used Soufflé, a parallel CPU-based Datalog implementation from Oracle labs, and worked toward a new MPI-based distributed hash join implementation and an extension of the GPU-based relational algebra library RedFox.

In this paper, we provide introductions to functional flow analysis, Datalog, MPI, and CUDA, explaining the total process we are working on to bring these components together in an analysis pipeline toward the end of scaling functional program analyses by extracting their intrinsic parallelism in a principled manner.

Additional Key Words and Phrases: Control-flow analysis; CFA; Datalog; CUDA; MPI; Parallelism; Static analysis

1 INTRODUCTION

A *control-flow analysis* (CFA) of a functional programming language models the propagation of data flows and control flows through a target program, across all possible executions. In static analyses of functional languages generally, a model of where functional values (i.e., lambdas, closures) can flow is required to model where control can flow from a call site, and vice versa. At a call site in Scheme, `(f x)`, the possible values for `f` determine which lambda bodies can be reached. Likewise, the possible callers for the lambda that binds `f`, `(lambda (... f ...) ...)`, influence the lambdas that can flow into `f`. This mutual dependence of data flow and control flow can be handled using an abstract interpretation, simultaneously modeling all interdependent language features. There are systematic approaches to designing analyses such as these, however the traditional worklist algorithms used to implement them in practice are inefficient and have difficulty scaling. Even with optimizations such as global-store widening and flat environments, the analysis is in $O(n^3)$ in the flow-insensitive case or in $O(n^4)$ in the flow-sensitive case. Using more advanced forms of polyvariance or context-sensitivity, the analysis becomes significantly more expensive.

In this paper, we describe our ongoing work to encode these analyses as declarative datalog programs and implement them as highly parallel relational algebra (RA), on the GPU and across many CPUs. Relational algebra operations, derived from Datalog analysis specifications, are computationally intensive and memory-bound in nature. GPUs provide massive fine-grained parallelism and great memory bandwidth, potentially making them the ideal target for solving these operations. We use Redfox [Wu et al. 2014], an open source tool which executes queries expressed in a specialized query language on GPUs. We also modify Redfox, adding capabilities to perform fixed-point iterations essential for solving RA operations derived from Datalog. We are also pursuing a Message Passing Interface (MPI)-based backend for solving RA operations across many compute nodes on a network. This approach is also particularly promising, given that HPC is increasingly mainstream and supercomputers are getting faster cores and lower latency interconnects.

2 CONTROL-FLOW ANALYSIS

This section introduces control-flow analysis by instantiating it for the continuation-passing-style (CPS) λ-calculus. We follow the abstracting abstract machines (AAM) methodology, a systematic process for developing a static analysis (an approximating semantics) from a precise operational semantics of an abstract machine.

Static analysis by abstract interpretation proves properties of a program by running code through an interpreter powered by an *abstract semantics* that approximates the behavior of an exact *concrete semantics*. This process is a general method for analyzing programs and serves applications such as program verification, malware/vulnerability detection, and compiler optimization, among others [Cousot and Cousot 1976, 1977, 1979; Midtgaard 2012]. Van Horn and Might's approach of *abstracting abstract machines* (AAM) uses abstract interpretation of abstract machines for *control-flow analysis* (CFA) of functional (higher-order) programming languages [Johnson et al. 2013; Might 2010; Van Horn and Might 2010]. The AAM methodology is flexible and allows a high degree of control over how program states are represented. AAM provides a general method for automatically abstracting an arbitrary small-step abstract-machine semantics to obtain an approximation in a variety of styles. Importantly, one such style aims to focus all unboundedness in a semantics on the machine's address-space. This makes the strategy used for the allocation of addresses crucial to the tradeoff struck between precision and complexity [Gilray et al. 2016a], and results in a highly flexible and tunable analysis infrastructure. More broadly, the approach has been used to instantiate both traditional finite flow analyses and heavy-weight program verification [Nguyen et al. 2014].

2.1 A Concrete Operational Semantics

This section reviews the process of producing a formal operational semantics for a simple language [Plotkin 1981], specifically, the untyped λ-calculus in *continuation-passing style* (CPS). CPS constrains call sites to tail position so that functions may never return; instead, callers explicitly pass a continuation forward to be invoked on the return value [Plotkin 1975]. This makes our semantics tail recursive (small-step) and easier to abstract while entirely eliding the challenges of manually managing a stack and its abstraction, a process previously discussed in the context of AAM [Johnson and Van Horn 2014; Van Horn and Might 2010]. Using an AAM that explicitly models the stack in a precise manner, while allowing for adjustable allocation, has also been recently addressed [Gilray et al. 2016b].

The grammar structurally distinguishes between call-sites *call* and atomic-expressions *ae*:

$$
\begin{aligned}
call \in \mathsf{Call} &::= (ae\ ae\ \ldots) \mid (\texttt{halt}) \\
lam \in \mathsf{Lam} &::= (\lambda\ (x\ \ldots)\ call) \\
ae \in \mathsf{AE} &::= lam \mid x \\
x \in \mathsf{Var} &\ \textit{is a set of program variables}
\end{aligned}
$$

Instead of specifically affixing each expression with a unique label, we assume two identical expressions occurring separately in a program are not equal. While a direct-style language with a variety of continuations (e.g., argument continuations, `let`-continuations, etc.), or extensions such as recursive-binding forms, conditionals, mutation, or primitive operations, would add complexity to any semantics, they do not affect the concepts we are exploring and so are left out.

We define the evaluation of programs in this language using a relation ($\rightarrow_\Sigma$), over states of an abstract-machine, which determines how the machine transitions from one state to another. States (ς) range over control expression (a call site), binding environment, and value store components:

$$\begin{aligned} \varsigma \in \Sigma &\triangleq \mathsf{Call} \times \mathit{Env} \times \mathit{Store} \\ \rho \in \mathit{Env} &\triangleq \mathsf{Var} \rightharpoonup \mathit{Addr} \\ \sigma \in \mathit{Store} &\triangleq \mathit{Addr} \rightharpoonup \mathit{Value} \\ a \in \mathit{Addr} &\triangleq \mathsf{Var} \times \mathbb{N} \\ v \in \mathit{Value} &\triangleq \mathit{Clo} \\ \mathit{clo} \in \mathit{Clo} &\triangleq \mathsf{Lam} \times \mathit{Env} \end{aligned}$$

Environments (ρ) map variables in scope to an address for the visible binding. Value stores (σ) map these addresses to values (in this case, closures); these may be thought of as a model of the heap. Both these functions are partial and accumulate points as execution progresses.

Evaluation of atomic expressions is handled by an auxiliary function ($\mathcal{A}$) which produces a value (*clo*) for an atomic expression in the context of a state (ς). This is done by a lookup in the environment and store for variable references (x), and by closure creation for λ-abstractions (*lam*). In a language containing syntactic literals, these would be translated into equivalent semantic values here.

$$\begin{aligned} \mathcal{A} &: \mathsf{AE} \times \Sigma \rightharpoonup \mathit{Value} \\ \mathcal{A}(x,\ (\mathit{call},\ \rho,\ \sigma)) &\triangleq \sigma(\rho(x)) \\ \mathcal{A}(\mathit{lam},\ (\mathit{call},\ \rho,\ \sigma)) &\triangleq (\mathit{lam},\ \rho) \end{aligned}$$

The transition relation $(\rightarrow_\Sigma) : \Sigma \rightharpoonup \Sigma$ yields at most one successor for a given predecessor in the state-space Σ. This is defined:

$$\overbrace{((ae_f\ ae_1\ \ldots\ ae_j), \rho, \sigma)}^{\varsigma} \rightarrow_\Sigma (\mathit{call}', \rho', \sigma')$$

$$\begin{aligned} \text{where} \quad ((\lambda\ (x_0 \ldots x_j)\ \ \mathit{call}'), \rho_\lambda) &= \mathcal{A}(ae_f,\ \varsigma) \\ v_i &= \mathcal{A}(ae_i,\ \varsigma) \\ \rho' &= \rho_\lambda[x_i \mapsto a_i] \\ \sigma' &= \sigma[a_i \mapsto v_i] \\ a_i &= (x_i, |\mathit{dom}(\sigma)|) \end{aligned}$$

Execution steps to the call-site body of the lambda invoked (as given by the atomic-evaluation of ae_f). This closure's environment (ρ_λ) is extended with a binding for each variable x_i to a fresh address a_i. A particular strategy for allocating a fresh address is to pair the variable being allocated for with the current number of points in the store (a value that increases after each set of new allocations). The store is extended with the atomic evaluation of ae_i for each of these addresses a_i. A state becomes stuck if `(halt)` is reached or if the program is malformed (*e.g.*, a free variable is encountered).

To fully evaluate a program call_0 using these transition rules, we *inject* it into our state space using a helper $\mathcal{I} : \mathsf{Call} \rightarrow \Sigma$:

$$\mathcal{I}(\mathit{call}) \triangleq (\mathit{call}, \varnothing, \varnothing)$$

We may now perform the standard lifting of $(\rightarrow_\Sigma)$ to a collecting semantics defined over sets of states:

$$s \in S \triangleq \mathcal{P}(\Sigma)$$

Our collecting relation $(\rightarrow_s)$ is a monotonic, total function that gives a set including the trivially reachable state $\mathcal{I}(call_0)$ plus the set of all states immediately succeeding those in its input.

$$s \rightarrow_s \{\varsigma' \mid \varsigma \in s \land \varsigma \rightarrow_\Sigma \varsigma'\} \cup \{\mathcal{I}(call_0)\}$$

If the program $call_0$ terminates, iteration of $(\rightarrow_s)$ from $\bot$ (i.e., the empty set $\varnothing$) does as well. That is, $(\rightarrow_s)^n(\bot)$ is a fixed point containing $call_0$'s full program trace for some $n \in \mathbb{N}$ whenever $call_0$ is a terminating program. No such n is guaranteed to exist in the general case (when $call_0$ is a non-terminating program) as our language (the untyped CPS λ-calculus) is Turing-equivalent, our semantics is fully precise, and the state-space we defined is infinite.

2.2 An Abstract Operational Semantics

Now that we have formalized program evaluation using our concrete semantics as iteration to a (possibly infinite) fixed point, we are ready to design a computable approximation of this fixed point (the exact program trace) using abstract interpretation. Previous work has explored a wide variety of approaches to systematically abstracting a semantics like these [Johnson et al. 2013; Might 2010; Van Horn and Might 2010]. Broadly construed, the nature of these changes is to simultaneously finitize the domains of our machine while introducing non-determinism both into the transition relation (multiple successor states may immediately follow a predecessor state) and the store (multiple values may become conflated at a single address). We use a finite address space to cut the otherwise mutually recursive structure of values (closures) and environments. (Without addresses and a value store, environments map variables directly to closures and closures contain environments). A finite address space yields a finite state space overall and ensures the computability of our analysis. Typographically, components unique to this *abstract* abstract machine wear hats so we can tell them apart without confusing essential underlying roles:

$$\begin{aligned}
\hat{\varsigma} \in \hat{\Sigma} &\triangleq \mathsf{Call} \times \widehat{Env} \times \widehat{Store} \\
\hat{\rho} \in \widehat{Env} &\triangleq Var \rightharpoonup \widehat{Addr} \\
\hat{\sigma} \in \widehat{Store} &\triangleq \widehat{Addr} \rightarrow \widehat{Value} \\
\hat{a} \in \widehat{Addr} &\triangleq \mathsf{Var} \\
\hat{v} \in \widehat{Value} &\triangleq \mathcal{P}(\widehat{Clo}) \\
\widehat{clo} \in \widehat{Clo} &\triangleq \mathsf{Lam} \times \widehat{Env}
\end{aligned}$$

Value stores are now total functions mapping abstract addresses to a *flow set* $(\hat{v})$ of zero or more abstract closures. This allows a range of values to merge and inhabit a single abstract address, introducing imprecision into our abstract semantics, but also allowing for a finite state space and a guarantee of computability. To begin, we use a monovariant address set $\widehat{Addr}$ with a single address for each syntactic variable. This choice (and its alternatives) is at the heart of our present topic and will be returned to shortly.

Evaluation of atomic expressions is handled by an auxiliary function $(\hat{\mathcal{A}})$ which produces a flow set $(\hat{v})$ for an atomic expression in the context of an abstract state $(\hat{\varsigma})$. In the case of closure creation, a singleton flow set is produced.

$$\begin{aligned}
\hat{\mathcal{A}} &: \mathsf{AE} \times \hat{\Sigma} \rightharpoonup \widehat{Value} \\
\hat{\mathcal{A}}(x,\ (call,\ \hat{\rho},\ \hat{\sigma})) &\triangleq \hat{\sigma}(\hat{\rho}(x)) \\
\hat{\mathcal{A}}(lam,\ (call,\ \hat{\rho},\ \hat{\sigma})) &\triangleq \{(lam,\ \hat{\rho})\}
\end{aligned}$$

The abstract transition relation $(\rightsquigarrow_{\hat{\Sigma}}) \subseteq \hat{\Sigma} \times \hat{\Sigma}$ yields any number of successors for a given predecessor in the state-space $\hat{\Sigma}$. As mentioned when introducing AAM, there are two fundamental changes required using this approach. Because abstract addresses can become bound to multiple closures in the store and atomic evaluation produces a flow set containing zero or more closures, one successor state results for each closure bound to the address for ae_f. Also, due to the relationality of abstract stores, we can no longer use strong update when extending the store $\hat{\sigma}'$.

$$\overbrace{((ae_f\ ae_1\ \ldots\ ae_j), \hat{\rho}, \hat{\sigma})}^{\hat{\varsigma}} \rightsquigarrow_{\hat{\Sigma}} (call', \hat{\rho}', \hat{\sigma}')$$

$$\begin{aligned} \text{where}\quad & ((\lambda\ (x_0 \ldots x_j)\ \ call'), \hat{\rho}_\lambda) \in \hat{\mathcal{A}}(ae_f,\ \hat{\varsigma}) \\ & \hat{v}_i = \hat{\mathcal{A}}(ae_i,\ \hat{\varsigma}) \\ & \hat{\rho}' = \hat{\rho}_\lambda[x_i \mapsto \hat{a}_i] \\ & \hat{\sigma}' = \hat{\sigma} \sqcup [\hat{a}_i \mapsto \hat{v}_i] \\ & \hat{a}_i = x_i \end{aligned}$$

A weak update is performed on the store instead which results in the least upper bound of the existing store and each new binding. Join on abstract stores distributes point-wise:

$$\hat{\sigma} \sqcup \hat{\sigma}' \triangleq \lambda\hat{a}.\ \hat{\sigma}(\hat{a}) \cup \hat{\sigma}'(\hat{a})$$

Unless it is desirable, and provably safe to do so [Might and Shivers 2006], we never remove closures already seen. Instead, we strictly accumulate every closure bound to each $\hat{a}$ (i.e., abstract closures which simulate closures bound to addresses which $\hat{a}$ simulates) over the lifetime of the program. A flow set for an address $\hat{a}$ indicates a range of values which over-approximates all possible concrete values that can flow to any concrete address approximated by $\hat{a}$. For example, if a concrete machine binds $(y, 345) \mapsto clo_1$ and $(y, 903) \mapsto clo_2$, its monovariant approximation might bind $y \mapsto \{\widehat{clo}_1, \widehat{clo}_2\}$. Precision is lost for $(y, 345)$ both because its value has been merged with $\widehat{clo}_2$, and because the environments for $\widehat{clo}_1$ and $\widehat{clo}_2$ in-turn generalize over many possible addresses for their free variables (the environment in $\widehat{clo}_1$ is less precise than that in clo_1).

To approximately evaluate a program according to these abstract semantics, we first define an abstract injection function, $\hat{\mathcal{I}}$, where the store begins as a function, $\bot$, that maps every abstract address to the empty set.

$$\hat{\mathcal{I}} : \mathsf{Call} \to \hat{\Sigma}$$

$$\hat{\mathcal{I}}(call) \triangleq (call, \varnothing, \bot)$$

We again lift $(\rightsquigarrow_{\hat{\Sigma}})$ to obtain a collecting semantics $(\rightsquigarrow_{\hat{S}})$ defined over sets of states:

$$\hat{s} \in \hat{S} \triangleq \mathcal{P}(\hat{\Sigma})$$

Our collecting relation $(\rightsquigarrow_{\hat{S}})$ is a monotonic, total function that gives a set including the trivially reachable finite-state $\hat{\mathcal{I}}(call_0)$ plus the set of all states immediately succeeding those in its input.

$$\hat{s} \rightsquigarrow_{\hat{S}} \hat{s}', \text{where}$$

$$\hat{s}' = \{\hat{\varsigma}' \mid \hat{\varsigma} \in \hat{s} \wedge \hat{\varsigma} \rightsquigarrow_{\hat{\Sigma}} \hat{\varsigma}'\} \cup \{\hat{\mathcal{I}}(call_0)\}$$

Because $\widehat{Addr}$ (and thus $\hat{\Sigma}$) is now finite, we know the approximate evaluation of even a non-terminating $call_0$ will terminate. That is, for some $n \in \mathbb{N}$, the value $(\rightsquigarrow_{\hat{S}})^n(\bot)$ is guaranteed to be a fixed point containing an approximation of $call_0$'s full concrete program trace [Tarski 1955].

2.2.1 Widening and Extension to Larger Languages. Various forms of widening and further approximations may be layered on top of the naïve analysis ($\rightsquigarrow_{\hat{s}}$). One such approximation is store widening, which is necessary for our analysis to be polynomial-time in the size of the program. This structurally approximates the analysis above, where each state contains a whole store, by pairing a set of states without stores, with a single, global store that overapproximates all possible bindings. This global store is maintained as the least-upper-bound of all bindings that are encountered in the course of analysis.

Setting up a semantics for real language features such as conditionals, primitive operations, direct-style recursion, or exceptions, is no more difficult, if more verbose. Supporting direct-style recursion, for example, requires an explicit stack as continuations are no longer baked into the source text by CPS conversion. Handling other forms is often as straightforward as including an additional transition rule for each.

3 DATALOG-BASED FLOW ANALYSIS

In this section, we will give an overview of Datalog as a declarative logic-programming language, show how it may be executed using a bottom-up fixed-point algorithm, and discuss our encoding of abstract semantics from the previous section.

3.1 Datalog

A datalog program consists of a set of relations, along with the *rules* pertaining to them. A relation encodes a set of tuples known as *facts*. For example, if we have a relation *Parent*, the fact $Parent(p, c)$ may assert that p is a parent of c. A rule then takes the form $a_0\ :-\ a_1, \ldots, a_j$ where a comma denotes conjunction and each atom a_i is of the form $r(x, \ldots)$, where r is a relation and each x is a variable. In Datalog the turnstile means "is implied by"; this is because the rules are formally an implication form of a Horn clause. Horn clauses are disjunctions where all but one of the atoms are negated: $a_0 \vee \neg a_1 \vee \ldots \vee \neg a_j$. This is the same as $a_0 \vee \neg(a_1 \wedge \ldots \wedge a_j)$ by De Morgan's laws, which is the same as an implication: $a_0 \impliedby a_1 \wedge \ldots \wedge a_j$. For example, a rule for computing grandparents can be specified:

$$GrandParent(gp, c)\ :-\ Parent(gp, p), Parent(p, c).$$

3.2 Datalog solvers

A number of strategies exist for executing a datalog program—that is, finding a sound least-fixed-point that contains all the input facts and obeys every rule. Unlike solvers for Prolog and other logic programming systems, bottom-up approaches have tended to prevail [Ullman 1989] although this can depend on extensions to the language and its use. Bottom-up solving begins with the set of facts provided as input, and iterates a monotonic function encoding the set of rules until a least-fixed-point is reached. The data structure for relations and the operation of these increasing iterations varies by approach.

The datalog solver *bddbddb* uses binary decision diagrams (BDDs) to encode relations and supports efficient relational algebra (RA) over these structures [Whaley et al. 2005]. BDDs are decision trees that encode arbitrary relations by viewing each bit in the input as a binary decision. They have the potential to be a compact and efficient representation of a relation, but have a worst-case exponential space complexity and are highly sensitive to variable ordering (in what order are bits of input branched on).

The datalog solver Soufflé uses the semi-naïve bottom up algorithm with partial-evaluation-based optimization [Scholz et al. 2016]. A relational algebra machine (RAM) executes the program in bottom-up fashion by using efficient underlying data structures for relations (such as prefix trees) and directly selects and propagates tuples in a series of nested loops. This RAM then has a partial-evaluation applied to it that optimizes the interpretation for the specific set of datalog rules. The tool accepts a datalog program as input, performs this partial evaluation and writes a C++ program that parallelizes the outside loops of each RA operation using pthreads.

3.3 Encoding CFA

Various program analysis have been implemented using Datalog. Both bddbddb and Soufflé presented program analyses in their experiments. Another prominent example is the DOOP framework for Java [Smaragdakis et al. 2011], used to demonstrate a generalization of object-sensitive flow analysis.

To encode a control-flow analysis as a Datalog problem, we first represent the abstract syntax tree (AST) as a set of relations where each expression and variable in the program has been enumerated. In the soufflé syntax, the encoding for Lambda expressions, conditional expressions, and Variable references is as follows:

```
  .decl SyntaxLambdaExp(e:Exp, x:Var, ebody:Exp) input
  .decl SyntaxIfExp(e:Exp, x:Var, e0:Exp, e1:Exp) input
  .decl SyntaxVarExp(e:Exp, x:Var) input
```

A tuple (e_0, x, e_1) in the `SyntaxLambdaExp` relation indicates that the expression e_0 is a lambda with the formal parameter x and the expression body e_1. A tuple (e_0, x, e_1, e_2) in the `SyntaxIfExp` relation indicates that the expression e_0 is a conditional branching on the variable x with the true branch e_1 and false branch e_2. A tuple (e, x) in the `SyntaxVarExp` relation indicates that the expression e is a variable reference returning the value of variable x.

We then encode the constraints of our analysis directly as rules in datalog, such as:

```
StoreEdge(x,y) :-
  SyntaxVarExp(e,x),
  ReachableExp(e,kaddr),
  KStore(kaddr,y,ebody,kaddr0).
```

This rule adds a flow from x to y when a reachable expression e is returning the value at x and its continuation binds the variable y.

4 MANY-NODE INNER-JOINS AND FUTURE WORK

Modern supercomputers' use of very low latency interconnect networks and highly optimized compute cores opens up the possibility of implementing highly parallel relational algebra using Message-Passing Interface (MPI). MPI is a portable standard library interface for writing parallel programs in a HPC setting and has been highly optimized on a variety of computing infrastructures, from small clusters to high-end supercomputers.

4.1 Parallel Join

Radix-hash join and merge-sort join are two of the most popularly used parallel implementations of the inner join operation. Both these algorithms involve partitioning the input data so that they can be efficiently distributed to the participating processes. For example, in the radix-hash approach a tuple is assigned to a process based on the hash output of the column-value on which the join operation is keyed. With this approach, tuples on both relations that share the same hash value are always assigned to the same process. For every tuple in the left-hand side of the join relation is matched against all the tuples of the right-hand side of the join relation. Fast lookup data-structures like hash tables, or radix-trees (TRIE) can be used to organize the tuples within every process. The initial distribution of data using hashing reduces the overall computation overhead by a factor of the number of processes (n).

More recently [Barthels et al. 2015, 2017], there has been a concerted effort to implement JOIN operations on clusters using an MPI backend. The commonly used radix-hash join and merge-sort join have been re-designed for this purpose. Both these algorithms involve a hash-based partitioning of data so that they are be efficiently distributed to the participating processes and are designed such that inter-process communication is minimized. In both of these implementations one-sided communication is used for transferring data between process. With one-sided communication the initiator of a data transfer request can directly access parts of the remote memory and

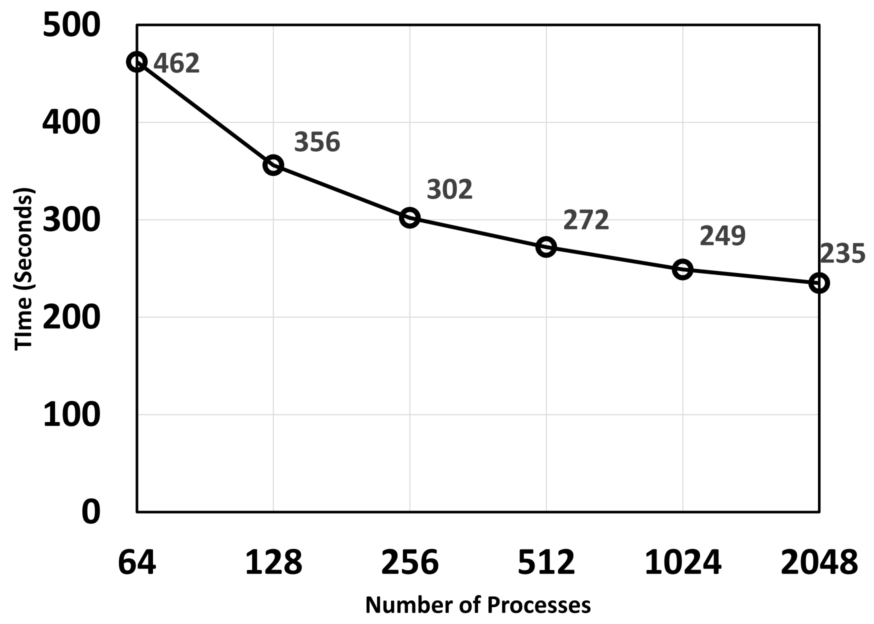
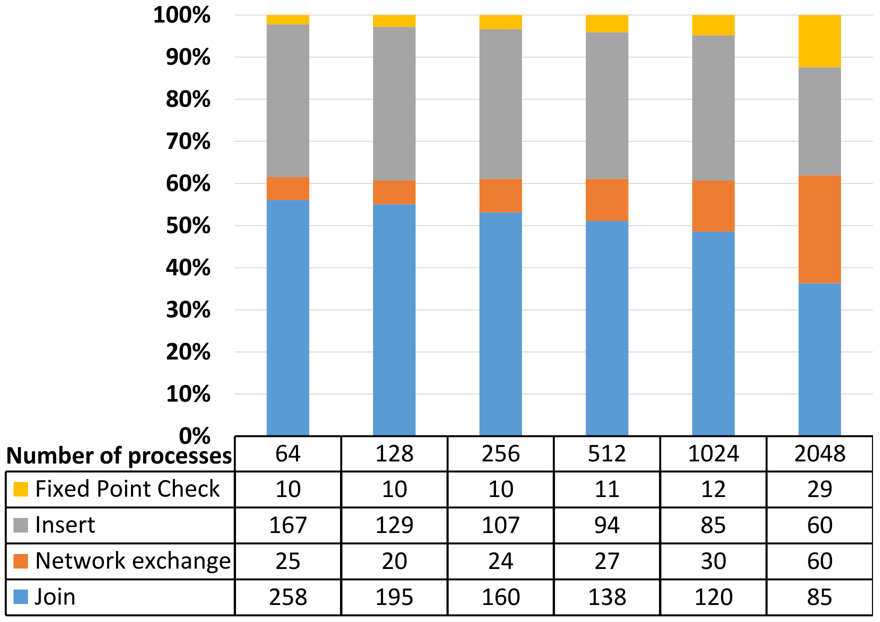


Fig. 1. Strong-scaling results.



Number of processes	64	128	256	512	1024	2048
🟨 Fixed Point Check	10	10	10	11	12	29
⬜ Insert	167	129	107	94	85	60
🟧 Network exchange	25	20	24	27	30	60
🟦 Join	258	195	160	138	120	85

Fig. 2. Strong-scaling timing breakdown.

has full control where the data will be placed. Read and write operations are executed without any involvement of the target machine. This approach of data transfer involves minimal synchronization between particiapting processes and have been shown to scale better that traditional two-sided communication. The implementation of parallel join has shown promising performance numbers; for example, the parallel join algorithm of [Barthels et al. 2017] ran successfully at 4,096 processor cores with up to 4.8 terabytes of input data.

4.2 Benchmarking: transitive closure

Computing the transitive closure of a graph involves repeated join operations until a fixed point is reached. We use the previously discussed radix-hash join algorithm to distribute the tuples across all processes. The algorithm can then be roughly divided into four phases: 1) Join 2) network communication 3) insertion 4) checking for a fixed point. In our join phase every process concurrently computes the join output of the local tuples. In the next phase every process sends the join output results to the relevant processes. This is a all-to-all communication phase, which we implemet using MPI's all_to_all routines. The next step involves inserting the join output result received from the network to the output graph's local partition. In the final step we check if the size of the output graph changed on any process, if it does then we have not yet reached a fixed point and we continue to another iteration of these 4 steps.

We performed a set of strong-scaling experiments to compute the transitive closure of graph with 412148 edges—the largest graph in the U. Florida Sparse Matrix set [Davis and Hu 2011]. We used the Quartz supercomputer at the Lawrence Livermore National Laboratory (LLNL). For our runs, we varied the number of processes from 64 to 2048. A fixed point was attained after 2933 iterations, with the resulting graph containing 1676697415 edges. As can be seen in Figure 1, our approach takes 462 seconds at 64 cores and 235 seconds at 2048 cores, corresponds to an overall efficiency of 6.25%. We investigated these timings further by plotting the timing breakdown of by the four major components (join, network communication, join, fixed-point check) of the algorithm. We observe (see Figure 2) that for all our runs the total time is dominated by computation rather than communication; insert and join together tended to take up close to 90% of the total time. This is quite an encouraging result as it shows that we are not bound primarily by the network bandwidth (at these scales and likely moderately higher ones) and it gives us the opportunity to optimize the computation phase.

5 PARALLELIZING DATALOG ON THE GPU

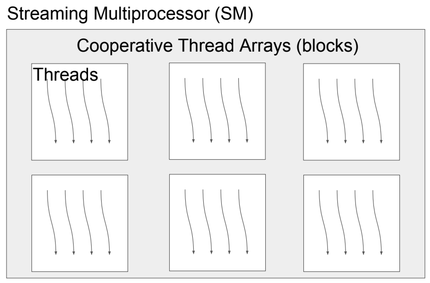
Programmable GPUs provide massive fine-grained parallelism, higher raw computational throughput, and higher memory bandwidth compared with multi-core CPUs. As a result they are a favorable alternative over traditional CPUs when it comes to high throughput implementations of applications. GPU implementations can potentially provide several orders of magnitude in performance improvement over traditional CPUs. As a result, GPU technology is increasingly widespread and has been successfully adopted by significant number of data-intensive scientific applications such as molecular dynamics [Anderson et al. 2008], physical simulations [Mosegaard and SÃÿrensen 2005], and ray tracing in graphics [Parker et al. 2010].

5.1 GPU architecture

Threads provide the finest level of parallelism in a GPU. A GPU application is composed of a series of multi-threaded data-parallel kernels. Data-parallel kernels are composed of a grid of parallel work-units called Cooperative Thread Arrays (CTAs) which in turn consist of an array of threads. In such processors, threads within a CTA are grouped into logical units known as warps that are mapped to SIMD units called Stream Multiprocessors (SMs) (see Figure 3). The programmer divides work into threads, threads map to thread blocks (CTAs), and thread blocks map to a grid. The compute work distributor allocates thread blocks to SMs. Once a thread block is distributed to an SM the resources for the thread block are allocated (warps and shared memory) and threads are divided into groups of (typically) 32 threads called warps.

5.2 Redfox

We use Redfox [Wu et al. 2014] a GPU-based open-source tool to run the relational algebra (RA) kernels translated from Datalog. Redfox is used for compiling and executing queries expressed in a specialized query language on GPUs. Typically, the parallelism involved in solving relational-algebra operations on GPUs is challenging due to unstructured and irregular data access as opposed to other domain-specific operations, such as those common to dense linear algebra. Redfox tackles these issues and provides an ecosystem to accelerate relational computation including algorithm design, system implementation, and compiler optimizations. It bridges the semantic gap between relational queries and GPU execution models, allowing its clients to operate exclusively in terms of RA semantics, and maintains significant performance speedup relative to the baseline CPU implementation.


Streaming Multiprocessor (SM)
Cooperative Thread Arrays (blocks)
Threads
Fig. 3. High-level overview of GPU.

Redfox takes advantage of the fine-grained massive parallelism offered by GPUs. It is comprised of (a) a specialized language front-end that allows the specification of sequences of RA operations that combine to form a query, (b) an RA to GPU compiler, (c) an optimized GPU implementation of select RA operators, and (d) a supporting runtime. The relational data is stored as a key-value store to support a range of workloads corresponding to queries over data sets. We use our own system to transform datalog queries into RA kernels. Redfox provides a GPU implementation of the following set operations: union, intersection, difference, cross product, inner-join, project and select. Among all the RA primitive operators, inner-join is the most complex and is more compute intensive than the rest of the RA primitives. Another problem with joins is that their output size can vary, i.e. between zero to the product of the sizes of the two inputs. One of the salient contributions of redfox is an optimal implementation of the join operation [Haicheng Wu and Yalamanchili 2014].

5.3 Fixed-point iterations with Redfox

One of the major challenges in adapting redfox to solve RA kernels derived from datalog queries was to perform fixed-point iterations. For fixed-point iterations redfox needed to process loops and, until now, Redfox was only used in a sequential mode, where a block unconditionally transitioned to the next block. In the original Redfox paper's experiments, the authors did use a fixed-point computation, but had manually unrolled the benchmark to the needed number of iterations. In our application, we need the ability to run basic blocks (each a straight-line sequence of RA kernels), in a loop, until the relation in contention does not change and the system reaches a fixed point—regardless of how many loops this requires. In order to facilitate execution of loops in redfox, we have added conditional branches, that allows execution to choose a target basic block based on the equality of two input relations. We used the COND kernel of GPU and use the outcome of the kernel to schedule the target block. Typically, in fixed-point iterations we check if the values stored in relation after execution of a

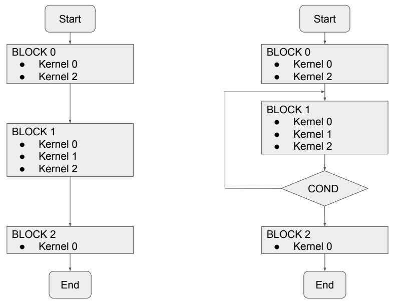


Fig. 4. Redfox execution with (right) and without (left) conditional branches.

certain kernel changed or not, if it remains unchanged then we have attained a fixed point and the execution can move forward, otherwise the set of kernel is executed again (see Figure 4).

5.4 Preliminary results

We evaluated the performance of Redfox in computing the transitive closure of large-sized graphs. For benchmarking we used the open source graphs available at [Davis and Hu 2011]. Out of all relation operations used in computing the transitive closure, join is computationally the most complex. We found that the join operation manage to scale decently well with larger graphs. Time consumed performing join operation across 188 iterations for input graph of 25,674 edges (output size is 6,489,757 edges) took 3.6 seconds. Total time for other kernel operation (project, union, copy) along with I/O time was 3.3 seconds. This total time (3.6 + 3.3 seconds) is almost comparable to the time taken by the highly optimized code Souffle (5.6 seconds) to compute the transitive closure of the same graph. We surmise, that Souffle is able to extract parallelism sufficient enough to solve the problem for this graph. Our hypothesis is that the GPU performance may become significantly faster than Souffle for very large scale graphs.

6 FUTURE WORK AND CONCLUSION

We have outlined a possible pipeline for extracting parallelism from a control-flow analysis in a principled way and have implemented GPU-based and MPI-based transitive closure algorithms to experiment with parallizing this kind of problem. We are also interested in writing PGAS based backends for our RA kernels. Partitioned global address space (PGAS) is a commonly used parallel programming model, that follows the ideals of shared memory access but operates in a distributed setting—it assumes a global memory address space that is logically partitioned, portions of which are local to each process. The two main implementations of this programming model are chapel [Chamberlain et al. 2007] and UPC++.

REFERENCES

Joshua A. Anderson, Chris D. Lorenz, and A. Travesset. 2008. General purpose molecular dynamics simulations fully implemented on graphics processing units. *J. Comput. Phys.* 227, 10 (2008), 5342 – 5359. DOI:http://dx.doi.org/10.1016/j.jcp.2008.01.047

Claude Barthels, Simon Loesing, Gustavo Alonso, and Donald Kossmann. 2015. Rack-Scale In-Memory Join Processing Using RDMA. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. ACM, New York, NY, USA, 1463–1475. DOI:http://dx.doi.org/10.1145/2723372.2750547

Claude Barthels, Ingo Müller, Timo Schneider, Gustavo Alonso, and Torsten Hoefler. 2017. Distributed Join Algorithms on Thousands of Cores. *Proc. VLDB Endow.* 10, 5 (Jan. 2017), 517–528. DOI:http://dx.doi.org/10.14778/3055540.3055545

B.L. Chamberlain, D. Callahan, and H.P. Zima. 2007. Parallel Programmability and the Chapel Language. *The International Journal of High Performance Computing Applications* 21, 3 (2007), 291–312. DOI:http://dx.doi.org/10.1177/1094342007078442 arXiv:http://dx.doi.org/10.1177/1094342007078442

Patrick Cousot and Radhia Cousot. 1976. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*. Paris, France, 106–130.

Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Symposium on Principles of Programming Languages*. ACM Press, New York, Los Angeles, CA, 238–252.

Patrick Cousot and Radhia Cousot. 1979. Systematic design of program analysis frameworks. In *Proceedings of the Symposium on Principles of Programming Languages*. ACM Press, New York, San Antonio, TX, 269–282.

Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (Dec. 2011), 25 pages. DOI:http://dx.doi.org/10.1145/2049662.2049663

Thomas Gilray, Michael D. Adams, and Matthew Might. 2016a. Allocation Characterizes Polyvariance: A Unified Methodology for Polyvariant Control-flow Analysis. *Proceedings of the International Conference on Functional Programming (ICFP)* (September 2016).

Thomas Gilray, Steven Lyde, Michael D. Adams, Matthew Might, and David Van Horn. 2016b. Pushdown Control-Flow Analysis For Free. *Proceedings of the Symposium on the Principles of Programming Languages (POPL)* (January 2016).

Molham Aref Haicheng Wu, Daniel Zinn and Sudhakar Yalamanchili. 2014. Multipredicate Join Algorithms for Accelerating Relational Graph Processing on GPUs. In *The 5th International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage*

Architectures (ADMS).

J. Ian Johnson, Nicholas Labich, Matthew Might, and David Van Horn. 2013. Optimizing Abstract Abstract Machines. In *Proceedings of the International Conference on Functional Programming.*

J. Ian Johnson and David Van Horn. 2014. Abstracting Abstract Control. In *Proceedings of the ACM Symposium on Dynamic Languages.*

Jan Midtgaard. 2012. Control-flow analysis of functional programs. *Comput. Surveys* 44, 3 (Jun 2012), 10:1–10:33.

Matthew Might. 2010. Abstract Interpreters for free. In *Static Analysis Symposium.* 407–421.

Matthew Might and Olin Shivers. 2006. Improving flow analyses via ΓCFA: abstract garbage collection and counting. In *ACM SIGPLAN Notices*, Vol. 41. ACM, 13–25.

J. Mosegaard and T. S. SÃÿrensen. 2005. Real-time Deformation of Detailed Geometry Based on Mappings to a Less Detailed Physical Simulation on the GPU. In *Eurographics Symposium on Virtual Environments*, Erik Kjems and Roland Blach (Eds.). The Eurographics Association. DOI: http://dx.doi.org/10.2312/EGVE/IPT_EGVE2005/105-111

Phúc C. Nguyen, Sam Tobin-Hochstadt, and David Van Horn. 2014. Soft Contract Verification. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14).* ACM, New York, NY, USA, 139–152. DOI: http://dx.doi.org/10.1145/2628136.2628156

Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. 2010. OptiX: A General Purpose Ray Tracing Engine. In *ACM SIGGRAPH 2010 Papers (SIGGRAPH '10).* ACM, New York, NY, USA, Article 66, 13 pages. DOI: http://dx.doi.org/10.1145/1833349.1778803

G. D. Plotkin. 1975. Call-by-name, call-by-value and the lambda-calculus. In *Theoretical Computer Science 1.* 125–159.

Gordon D Plotkin. 1981. A structural approach to operational semantics. (1981).

Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. 2016. On fast large-scale program analysis in datalog. In *Proceedings of the 25th International Conference on Compiler Construction.* ACM, 196–206.

Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhotak. 2011. Pick Your Contexts Well: Understanding Object-Sensitivity. In *Symposium on Principles of Programming Languages.* 17–30.

Alfred Tarski. 1955. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.* 5, 2 (1955), 285–309.

Jeffrey D Ullman. 1989. Bottom-up beats top-down for datalog. In *Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems.* ACM, 140–149.

David Van Horn and Matthew Might. 2010. Abstracting Abstract Machines. In *International Conference on Functional Programming.* 51.

John Whaley, Dzintars Avots, Michael Carbin, and Monica S Lam. 2005. Using datalog with binary decision diagrams for program analysis. In *Asian Symposium on Programming Languages and Systems.* Springer, 97–118.

Haicheng Wu, Gregory Diamos, Tim Sheard, Molham Aref, Sean Baxter, Michael Garland, and Sudhakar Yalamanchili. 2014. Red Fox: An Execution Environment for Relational Query Processing on GPUs. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '14).* ACM, New York, NY, USA, Article 44, 11 pages. DOI: http://dx.doi.org/10.1145/2544137.2544166

Gerbil on Gambit, as they say Racket on Chez

DIMITRIS VYZOVITIS

Gerbil is a new opinionated dialect of Scheme designed for Systems Programming with a state of the art macro and module system on top of the Gambit runtime. Gerbil wants to fill the the needs of seasoned Schemers and Common LISP refugees who want to do their systems programming with modern macro facilities and without loss of performance.

The system implements modularity and language extensibility facilities equivalent to Racket's, including the `#lang` reader. Gerbil implements this with a macro expander, compiler, and standard library sitting on top of Gambit. As such, it brings these facilities for the first time to a highly performant Ahead of Time natively compiled environment.

The relationship between Gerbil and Gambit is symbiotic: Gerbil provides the top-half and Gambit the bottom-half ot the system, similar to how Racket will run on Chez in the not so distant future.

Source code: https://github.com/vyzo/gerbil

+

{lambda talk}

Alain Marty Engineer Architect
Villeneuve de la Raho, France
marty.alain@free.fr

ABSTRACT

The {lambda way} project is a web application built on two engines, {lambda talk} and {lambda tank}. {lambda talk} is a purely functional language unifying authoring, styling and scripting in a single and coherent Lisp-like syntax, working in {lambda tank}, a tiny wiki built as a thin overlay on top of any web browser. In this paper we forget {lambda tank}, mainly a PHP engine managing text files on the server side, and progressively introduce {lambda talk}, a Javascript engine evaluating code in realtime on the client side. The making of {lambda talk} is done in three stages:

- 1) we define the minimal set of rules making {lambda talk} a programming language, *complete even if unusable*,
- 2) we progressively add numbers, operators, data and control structures making {lambda talk} *more usable*,
- 3) we finally build on the browsers' full functionalities a set of *libraries* making {lambda talk} *usable and much more efficient*.

As a guilding line, at each stage, we compute with a total precision the factorial of **5** and **50**:

```
5!  = 120
50! = 30414093201713378043612608166064
      76884437764156896051200000000000
```

In a last section, APPENDICE, some explanations are given on the {lambda talk}'s Javascript implementation.

KEYWORDS

- Information systems~Wikis
- Theory of computation~Regular languages
- Theory of computation~Lambda calculus
- Software and its engineering~Functional languages
- Software and its engineering~Extensible Markup Language

INTRODUCTION

{lambda talk} expressions are written in an editor frame, evaluated in real time, displayed in the wiki's viewer frame, then saved and published on the WEB. « *A wiki is a web application which allows collaborative modification, extension, or deletion of its content and structure.*[1] » The father of this concept, Ward Cunningham[2], gives a simple and clear introduction to {lambda talk}:

- **1)** Away from curly braces {} *words are just words*:

```
Hello World!
-> Hello World!
```

- **2)** Expressions are written in a *prefix notation*:

```
2+3 is equal to {b {+ 2 3}}
-> 2+3 is equal to 5
```

- **3)** Functions are created with *lambda* and named with *def*:

```
{def SMART_ADD
 {lambda {:a :b}
  :a+:b is equal to {b {+ :a :b}}}}
-> SMART_ADD

{SMART_ADD 2 3}
-> 2+3 is equal to 5
```

These examples use the "+" Math operator, the "b" HTML/CSS markup operator and Javascript numbers. In the following we want to introduce {lambda talk} built upon the deepest foundation possible, *simple words*, and we will ignore everything but words until the third section.

1. WORDS

We present the structure and evaluation of a {lambda talk} expression.

1.1. expressions

{lambda talk} is mainly built on three rules freely inspired by the *λ-calculus*[3]. An expression is defined recursively as follows:

```
expression is [word|abstraction|application]*
```

where

```
1) word         is [^\s{}]*
2) abstraction  is {lambda {word*} expression}
3) application  is {abstraction expression}
```

An expression is made of `words`, `abstractions` and `applications` where 1) a `word` is any character except spaces "`\s`" and curly braces "`{}`", 2) an `abstraction` is the "process" (called a *function*) selecting a sequence of `words` (called *arguments*) in an `expression` (called *body*), 3) an `application` is the process calling an `abstraction` to replace selected `words` by some other `words` (called *values*).

The evaluation follows these rules:

1. words are not evaluated,
2. abstractions are evaluated before applications,
3. an abstraction is evaluated to a single word, a reference to an anonymous a function stored in a global dictionary, initially empty,

4. an application is progressively evaluated from inside out, to a sequence of words,
5. the evaluation stops when all expressions have been reduced to a sequence of words.

What can we do with that?

1.1.1. words

```
Hello World
-> Hello World
```

Words are not evaluated and are displayed as they are.

1.1.2. abstraction

```
{lambda {o a} oh happy day!}
-> _LAMB_6
```

The abstraction selects `o` and `a` as characters whose occurences in the expression `oh happy day!` are to be replaced by some future values, and returns the reference to an anonymous function.

1.1.3. application (1)

```
{{lambda {o a} oh happy day!} oOOOo aaAAaa}
-> oOOOoh haaAAaappy daaAAaay!
```

The abstraction is defined and immediately called. The abstraction is first evaluated to a word, say `_LAMB_6`, the application `{_LAMB_6 oOOOo aaAAaa}` gets the given values, calls the abstraction which makes the substitution and returns the result, `oOOOoh haaAAaappy daaAAaay!`.

1.1.4. application (2)

```
{{lambda {z}
 {z {lambda {x y} x}}}
  {{lambda {x y z}
   {z x y}} Hello World}}
-> Hello

{{lambda {z}
 {z {lambda {x y} y}}}
  {{lambda {x y z}
   {z x y}} Hello World}}
-> World
```

These expressions return respectively the first and the second words of `Hello World`, recalling a pair and its accessors. Let's trace the evaluation leading to Hello:

- 1) Nested lambdas are first evaluated:

```
1: {{lambda {z} {z {lambda {x y} x}}}
    {{lambda {x y z} {z x y}} Hello World}}
2: {{lambda {z} {z _LAMB_1}}
    {_LAMB_2 Hello World}}
3: {_LAMB_3 {_LAMB_2 Hello World}}
where
  _LAMB_1 replaces {lambda {x y} x}
  _LAMB_2 replaces {lambda {x y z} {z x y}}
  _LAMB_3 replaces {lambda {z} {z f1}}
```

- 2) Then simple forms are evaluated:

```
1: {_LAMB_3 {_LAMB_2 Hello World}}
2: {_LAMB_3 {{lambda {x y z} {z x y}} Hello
World}}
```

`{{lambda {x y z} {z x y}} Hello World}` is a *partial application* replacing "x" and "y" by "Hello" and "World", creating a new lambda waiting for the third value, `{lambda {z} {z Hello World}}`, and returning a reference, `_LAMB_4`.

```
3: {_LAMB_3 _LAMB_4}
4: {{lambda {z} {z _LAMB_1}} _LAMB_4}
5: {_LAMB_4 _LAMB_1}
6: {{lambda {z} {z Hello World}} _LAMB_1}
7: {_LAMB_1 Hello World}
8: {{lambda {x y} x} Hello World}
9: Hello
```

To sum up on lambdas:

- `1) lambdas` are *first class* functions,
- `2) lambdas` accept *partial function application*: a lambda called with a number of values lesser than its arity memorizes the given values and returns a new lambda waiting for the rest.
- `3) lambdas` *don't create closures*, inner lambdas have no access to outer lambdas' arguments, there is no lexical scoping, no nested environments, no free variables.

Lambdas are pure black boxes, independant of any context, as are mathematical functions.

1.1.5. application (3)

```
{{lambda {n} {{lambda {g n} {g g n}} {lambda {g
n} {{lambda {p t f g n} {{{p n} {{lambda {x y z}
{z x y}} t f}} g n}} {lambda {n} {{lambda {n} {n
{lambda {x} {lambda {z} {z {lambda {x y} y}}}}
{lambda {z} {z {lambda {x y} x}}}}} n}} {lambda
{g n} {{lambda {n f x} {f {{n f} x}}} {lambda {f
x} x}}} {lambda {g n} {{lambda {n m f} {m {n
f}}} n {g g {{lambda {n} {{lambda {z} {z {lambda
{x y} x}}} {{n {lambda {p} {{lambda {x y z} {z x
y}} {{lambda {z} {z {lambda {x y} y}}} p}
{{lambda {n f x} {f {{n f} x}}} {{lambda {z} {z
{lambda {x y} y}}} p}}}}} {{lambda {x y z} {z x
y}} {lambda {f x} x} {lambda {f x} x}}}}} n}}}}
g n}} n}} {lambda {f x} {f {f {f {f {f x}}}}}}}
-> _LAMB_167
```

At this point, *you should believe* that **this unreadble expression** evaluated to an anonymous function *is* the factorial of **5**, **5! = 120**, computed using its recursive definition:

```
fac(n) = 1 if n == 0 else fac(n) = n*fac(n-1)
```

1.2. names

In order to make code more readable we introduce a second special form `{def word expression}`, with which we will populate the *global dictionary* with *constants* and give *names* to anonymous functions. Let's rewrite with names the previous examples.

1.2.1. words

Any sequence of words can be given a name:

```
{def HI Hello World}
-> HI
```

```
HI {HI}
-> HI Hello World
```

Note that the word `HI` out of curly braces {} is not evaluated. Remember that, in a spreadsheet, one must write `=PI()` to get the value associated to **PI**.

1.2.2. abstractions

An anonymous functions can be given a name:

```
{def GOOD_DAY
 {lambda {:o :a} :oh h:appy day!}}
-> GOOD_DAY
```

1.2.3. application $^{(1)}$

That makes several applications easier:

```
{GOOD_DAY oOOOo aaAaa}
-> oOOOoh haaAaappy day!

{GOOD_DAY ♠ ♥}
-> ♠h h♥ppy day!
```

Note: arguments and their occurences in the function's body have been prefixed with a colon "**:**". Doing so prevents unintentional substitutions in the function's body, for instance the word **day** hasn't been replaced by **daaAaay** or **d ♥ y**. Escaping/marking arguments, for instance prefixing them with a colon "`:`", is highly recommended if not always mandatory. We will do it systematically and we add this constraint to the previous rules: « *In lambda expressions arguments* `arg` *must at least be tagged by some escaping character, for instance* `:arg`*, or for a better security, bracketted between two, for instance* `:arg:`.

1.2.4. application $^{(2)}$

```
{def CONS {lambda {:x :y :z} {:z :x :y}}}
-> CONS
{def CAR  {lambda {:z} {:z {lambda {:x :y}
:x}}}}
-> CAR
{def CDR  {lambda {:z} {:z {lambda {:x :y}
:y}}}}
-> CDR

{CAR {CONS Hello World}}
-> Hello
{CDR {CONS Hello World}}
-> World
```

In fact we just have built and used a *pair* and its *accessors*. Where LISP[4] and SCHEME[5] use a closure to define `cons`:

```
(def cons (lambda (x y) (lambda (z) (z x y))))
(def car  (lambda (z)   (z (lambda (x y) x))))
(def cdr  (lambda (z)   (z (lambda (x y) y))))
```

{lambda talk} uses partial application. There is no outer environment storing accessible values, values are stored inside lambdas.

1.2.5. application $^{(3)}$

We rewrite the example 1.1.5. using two names:

```
{def FOO {lambda {:n} {{lambda {:g :n} {:g :g
:n}} {lambda {:g :n} {{lambda {:b :t :f :g :n}
{{{:b :n} {{lambda {:x :y :z} {:z :x :y}} :t
:f}} :g :n}} {lambda {:n} {{lambda {:n} {:n
{lambda {:x} {lambda {:z} {:z {lambda {:x :y}
:y}}}} {lambda {:z} {:z {lambda {:x :y} :x}}}}}
:n}} {lambda {:g :n} {{lambda {:n :f :x} {:f
{{:n :f} :x}}} {lambda {:f :x} :x}}} {lambda {:g
:n} {{lambda {:n :m :f} {:m {:n :f}}} :n {:g :g
{{lambda {:n} {{lambda {:z} {:z {lambda {:x :y}
:x}}} {{:n {lambda {:p} {{lambda {:x :y :z} {:z
:x :y}} {{lambda {:z} {:z {lambda {:x :y} :y}}}
:p} {{lambda {:n :f :x} {:f {{:n :f} :x}}}
{{lambda {:z} {:z {lambda {:x :y} :y}}} :p}}}}}
{{lambda {:x :y :z} {:z :x :y}} {lambda {:f :x}
:x} {lambda {:f :x} :x}}}}} :n}}}} :g :n}} :n}}
}
-> FOO

{def BAR
 {lambda {:f :x} {:f {:f {:f {:f {:f :x}}}}}}}
-> BAR

{FOO BAR }
-> _LAMB_8
```

We notice that `FOO` is applied to `BAR`, an anonymous function applying *5 times* `:f` to `:x`. We can now better understand that the result is a reference to an anonymous function which might be associated to the factorial of **5**. And we guess that applying **50 times** `:f` to `:x` would lead to an anonymous function associated to the factorial of **50** ... provided we had a huge memory and thousands years before us!

Concluding this first section we note that, until now, {lambda talk} knows nothing but text substitution and that the dictionary contains no built-in primitive. In the following section, *still without using any Javascript Math object*, we progressively build **numbers**, **operators**, **data** and **control structures** to compute **5!** and **50!**.

2. NUMBERS

Following *"Collected Lambda Calculus Functions"*[6] we progressively add numbers, operators, data and control structures.

2.1. numbers

We define the so-called *Church numbers*:

```
{def ZERO  {lambda {:f :x} :x}}
-> ZERO
{def ONE   {lambda {:f :x} {:f :x}}}
-> ONE
{def TWO   {lambda {:f :x} {:f {:f :x}}}}
-> TWO
{def THREE {lambda {:f :x} {:f {:f {:f :x}}}}}
-> THREE
{def FOUR  {lambda {:f :x} {:f {:f {:f {:f
:x}}}}}}
-> FOUR
{def FIVE  {lambda {:f :x} {:f {:f {:f {:f {:f
:x}}}}}}}
-> FIVE
```

Applied to a couple of any words, we get strange things:

```
{ZERO . .} -> .
{ONE  . .} -> (. .)
```

```
{TWO  . .} -> (. (. .))
{FIVE . .} -> (. (. (. (. (. .)))))
```

We define the function CHURCH which translates Church numbers in a more familiar shape:

```
{def CHURCH
 {lambda {:n}
  {{:n {lambda {:x} {+ :x 1}}} 0}}}
-> CHURCH
{CHURCH ZERO} -> 0
{CHURCH ONE}  -> 1
{CHURCH FIVE} -> 5
```

Note: the CHURCH function is built on numbers, **[0,1]**, and a function, **'+'**, coming with Javascript, which are not supposed to exist at this point. Consider that it's only for readability.

2.2. operators

Based on Church numbers, which are **iterators by themselves**, we can easily define and test a first set of operators:

```
{def SUCC {lambda {:n :f :x} {:f {{:n :f} :x}}}}
-> SUCC
{def ADD {lambda {:n :m :f :x} {{:n :f} {{:m :f}
:x}}}}
-> ADD
{def MUL {lambda {:n :m :f} {:m {:n :f}}}}
-> MUL
{def POWER {lambda {:n :m} {:m :n}}}
-> POWER

{CHURCH {SUCC ZERO}}        -> 1
{CHURCH {SUCC ONE}}         -> 2
{CHURCH {SUCC THREE}}       -> 3
{CHURCH {ADD TWO THREE}}    -> 5  // 2+3
{CHURCH {MUL TWO THREE}}    -> 6  // 2*3
{CHURCH {POWER THREE TWO}} -> 9  // 3^2
```

Building "opposite" functions like PRED, SUBTRACT, DIVIDE is not so easy - and Church himself avoided them in the primitive version of λ-calculus. The answer was given by Stephen Cole Kleene[7], the father of Regular Expressions: *Church numbers can be used to* ***iterate*** *and pairs to* ***aggregate***. This is how:

- we define a function PRED.PAIR getting a pair [a,a] and returning a pair [a,a+1],
- the function PRED computes n iterations of PRED.PAIR starting on the pair [0,0] and leading to the pair [n-1,n] and returns the first, n-1:

```
{def PRED.PAIR {lambda {:p}
 {CONS {CDR :p} {SUCC {CDR :p}}}}}
-> PRED.PAIR
{def PRED {lambda {:n}
 {CAR {{:n PRED.PAIR} {CONS ZERO ZERO}}}}}
-> PRED

{CHURCH {PRED FIVE}}
-> 4
```

2.3. « To Iterate is Human, ...

We already have all what is needed to evaluate complex expressions like **1*2*3*...*n**. Inspired by the PRED operator:

- we define a function ITER.PAIR getting a pair [a,b] and returning a pair [a+1,a*b],
- the function ITER computes n iterations of ITER.PAIR, starting on the pair [1,1] and leading to the pair [n,n!] and returns the second, n!

```
{def ITER
 {def ITER.PAIR
  {lambda {:p}
   {CONS {SUCC {CAR :p}}
         {MUL {CAR :p} {CDR :p}}}}}
 {lambda {:n}
  {CDR {{:n ITER.PAIR} {CONS ONE ONE}}}}}
-> ITER

{CHURCH {ITER TWO}}   -> 2
{CHURCH {ITER THREE}} -> 6
{CHURCH {ITER FOUR}}  -> 24
{CHURCH {ITER FIVE}}  -> 120
```

2.4. ... to Recurse, Divine »[8]

If we want to define the factorial using its recursive mathematical definition:

```
  fac(n) = 1 if n == 0 else fac(n) = n*fac(n-1)
```

we need to build a few boolean operators:

```
{def TRUE {lambda {:z} {:z {lambda {:x :y}
:x}}}}
-> TRUE
{def FALSE {lambda {:z} {:z {lambda {:x :y}
:y}}}}
-> FALSE
{def IF {lambda {:x :y :z} {:z :x :y}}}
-> IF
{def ISZERO {lambda {:n}
  {:n {lambda {:x} FALSE} TRUE}}}
-> ISZERO
```

Note that TRUE, FALSE, IF are aliases to CAR, CDR, CONS.

Here is the tricky part! We remember that all expressions except abstractions are evaluated *eagerly*: functions' arguments are *called by value* and not *called by name*. Inside the IF function every arguments are evaluated before the call and this would lead to an *infinite loop* in a recursive process. A workaround is to use abstraction to introduce *manually* some kind of lazyness. This is an answer:

```
{def FAC
 {lambda {:n}
  {{lambda {:b :t :f :n}
   {{{:b :n} {IF :t :f}} :n}}
    {lambda {:n} {ISZERO :n}}    -> :b
    {lambda {:n} ONE}            -> :t
    {lambda {:n}
      {MUL :n {FAC {PRED :n}}}} -> :f
    :n                           -> :n
}}}
-> FAC

{CHURCH {FAC FIVE}}
-> 120
```

We can see that expressions {ISZERO :n} and {MUL :n {FAC {PRED :n}}} are hidden via lambdas behind names :b and `:f`. As long as `{:b :n}` is evaluated to `false`, `{IF :t :f}` returns the word `:f` which is **then** evaluated to `{MUL :n {FAC {PRED :n}}}` and the process recurses until zero, leading to `{* 5 {* 4 {* 3 {* 2 1}}}} = 120`.

Let's introduce some **Y-combinator** making recursive an almost recursive function:

```
{def Y {lambda {:g :n} {:g :g :n}}} -> Y

{def IFTHENELSE {lambda {:b :t :f :g :n}
  {{{:b :n} {IF :t :f}} :g :n} }}
-> IFTHENELSE

{def ALMOST_FAC {lambda {:g :n}
  {IFTHENELSE
   {lambda {:n} {ISZERO :n}}
   {lambda {:g :n} ONE}
   {lambda {:g :n} {MUL :n {:g :g {PRED :n}}}}
   :g :n }}}
-> ALMOST_FAC

{CHURCH {Y ALMOST_FAC FIVE}}
-> 120
```

Let's mix the both:

```
{def YFAC {lambda {:n}
 {{lambda {:g :n} {:g :g :n}}
 {lambda {:g :n}
  {IFTHENELSE
   {lambda {:n} {ISZERO :n}}
   {lambda {:g :n} ONE}
   {lambda {:g :n} {MUL :n {:g :g {PRED :n}}}}
   :g :n}}
   :n}}}
-> YFAC

{CHURCH {YFAC FIVE}}
-> 120
```

Throwing away the name, let's define and immediately call the lambda on the value **5**:

```
{CHURCH
 {{lambda {:n} {{lambda {:g :n} {:g :g :n}}
  {lambda {:g :n}
   {IFTHENELSE
    {lambda {:n} {ISZERO :n}}
    {lambda {:g :n} ONE}
    {lambda {:g :n} {MUL :n {:g :g {PRED :n}}}}
    :g :n}} :n}} FIVE}}
-> 120
```

Finally, let's replace all constants by their lambda based values to get a pure λ-calculus expression made of `words`, `abstractions` and `applications`:

```
{CHURCH {{lambda {:n} {{lambda {:g :n} {:g :g
:n}} {lambda {:g :n} {{lambda {:b :t :f :g :n}
{{{:b :n} {{lambda {:x :y :z} {:z :x :y}} :t
:f}} :g :n}} {lambda {:n} {{lambda {:n} {:n
{lambda {:x} {lambda {:z} {:z {lambda {:x :y}
:y}}}} {lambda {:z} {:z {lambda {:x :y} :x}}}}}}
:n}} {lambda {:g :n} {{lambda {:n :f :x} {:f
{{:n :f} :x}}} {lambda {:f :x} :x}}} {lambda {:g
:n} {{lambda {:n :m :f} {:m {:n :f}}} :n {:g :g
{{lambda {:n} {{lambda {:z} {:z {lambda {:x :y}
:x}}} {{:n {lambda {:p} {{lambda {:x :y :z} {:z
:x :y}} {{lambda {:z} {:z {lambda {:x :y} :y}}}
:p} {{lambda {:n :f :x} {:f {{:n :f} :x}}}
{{lambda {:z} {:z {lambda {:x :y} :y}}} :p}}}}}
{{lambda {:x :y :z} {:z :x :y}} {lambda {:f :x}
:x} {lambda {:f :x} :x}}}}}} :n}}}} :g :n}} :n}}
{lambda {:f :x} {:f {:f {:f {:f {:f :x}}}}}}}}
-> 120
```

Concluding this section, using nothing but words and text replacement processes, forgetting limitations of Javascript numbers and so theoretically regardless of its size and with a total precision, we can compute the factorial of any natural number. But if computing **5!** is *relatively* fast, computing **50!** would still be too long! It's time to remember that we can use the power of modern browsers to make things easier and much more faster!

3. {LAMBDA TALK}

In this section Church numbers and their related operators built as user defined functions are *forgotten* and replaced by primitive functions built on the browser's foundations. We use Javascript's numbers, Math operators and functions, HTML tags, CSS rules, SVG and more. We add *aggregate datas* like `pairs, lists, arrays` and some others specific to the wiki context. We add new special forms, `[if, let, quote, macro]`. Note that there is no `set!` special form, {lambda talk} is purely functional. This is the current dictionary:

DICTionary: *(250) [debug, browser_cache, lib, eval, apply, <, >, <=, >=, =, not, or, and, +, -, *, /, %, abs, acos, asin, atan, ceil, cos, exp, floor, pow, log, random, round, sin, sqrt, tan, min, max, PI, E, date, serie, map, reduce, equal?, empty?, chars, charAt, substring, length, first, rest, last, nth, replace, cons, cons?, car, cdr, cons.disp, list.new, list, list.disp, list.null?, list.length, list.reverse, list.first, list.butfirst, list.last, list.butlast, array.new, array, array.disp, array.array?, array.null?, array.length, array.item, array.first, array.last, array.rest, array.slice, array.concat, array.set!, array.push!, array.pop!, array.unshift!, array.shift!, array.reverse!, array.sort!, @, div, span, a, ul, ol, li, dl, dt, dd, table, tr, td, h1, h2, h3, h4, h5, h6, p, b, i, u, center, hr, blockquote, sup, sub, del, code, img, pre, textarea, canvas, audio, video, source, select, option, object, svg, line, rect, circle, ellipse, polygon, polyline, path, text, g, mpath, use, textPath, pattern, image, clipPath, defs, animate, set, animateMotion, animateTransform, br, input, iframe, mailto, back, hide, long_mult, turtle, drag, note, note_start, note_end, show, lightbox, minibox, editable, forum, lisp, BN.DEC, BN.new, BN.+, BN.-, BN.*, BN./, BN.%, BN.pow, BN.compare, BN.negate, BN.abs, BN.intPart, BN.valueOf, BN.round, BN.fac, sheet, sheet.new, sheet.input, sheet.output, SMART_ADD, HI, GOOD_DAY, CONS, CAR, CDR, BAR, ZERO, ONE, TWO, THREE, FOUR, FIVE, CHURCH, SUCC, ADD, MUL, POWER, PRED.PAIR, PRED, ITER.PAIR, ITER, TRUE, FALSE, IF, ISZERO, FAC, Y, IFTHENELSE, ALMOST_FAC, YFAC, FACT, TFAC.rec, TFAC, BI.bigint2pol.rec, BI.bigint2pol, BI.pol2bigint.rec, BI.pol2bigint, BI.simplify.rec, BI.simplify, BI.pk, BI.p+, BI.p*, BI.tfac.r, BI.tfac, castel.interpol, castel.sub, castel.point, castel.build, svg.dot, p0, p1, p2, p3, red_curve, green_curve, QUOTIENT, SIGMA, PAREN, mul, TITLE, WRAP, ref, back_ref, space, COLUMNS]*

where can be seen the user defined functions starting at `SMART_ADD`, the first constant created for the current document.

In order to illustrate some of these new capabilities we will write effective recursive factorials, compute big numbers, play with tabular data in a spreadsheet, explore intensive computing with javascripts, build regular expressions based macros.

3.1. recursion

In the previous section we have seen how tricky it was to write a recursive algorithm. We had to build *manually* a lazy behaviour. Using the third `{if bool then one else two}` special form and its built-in *lazy evaluation* the way is opened to efficient recursive algorithms. It's now possible to write the factorial function following its mathematical definition:

```
{def FACT
 {lambda {:n}
  {if {< :n 0}
   then {b n must be positive!}
   else {if {= :n 0} then 1
   else {* :n {FACT {- :n 1}}}}}}}
-> FACT

{FACT -1} -> n must be positive!
{FACT 0}  -> 1
{FACT 5}  -> 120
{FACT 50} -> 3.0414093201713376e+64
```

Let's write the tail-recursive version:

```
{def TFAC
 {def TFAC.rec
  {lambda {:a :n}
   {if {< :n 0}
    then {b n must be positive!}
    else {if {= :n 0} then :a
    else {TFAC.rec {* :a :n} {- :n 1}}}}}}
 {lambda {:n} {TFAC.rec 1 :n}}}
-> TFAC

{TFAC 5}  -> 120
{TFAC 50} -> 3.0414093201713376e+64
```

The recursive part is called by a *helper function* introducing the accumulator "`:a`". {lambda talk} doesn't know lexical scoping - the `TFAC.rec` inner function is global - and this leads to some pollution of the dictionary. The *Y-combinator* mentionned above, *making recursive an almost-recursive function*, will help us to discard this helper function. The *Y-combinator* and the almost-recursive function can be defined and used like this:

```
{def Y {lambda {:f :a :n} {:f :f :a :n}}}
-> Y

{def ALMOST_FAC
 {lambda {:f :a :n}
  {if {< :n 0}
   then {b n must be positive!}
   else {if {= :n 0} then :a
   else {:f :f {* :a :n} {- :n 1}}}}}}
-> ALMOST_FAC

{Y ALMOST_FAC 1 5}
-> 120
```

We can do better. Instead of applying the Y combinator to the almost recursive function we can define a function composing both:

```
{def YFAC {lambda {:n}
 {{lambda {:f :a :n}
  {:f :f :a :n}}
   {lambda {:f :a :n}
    {if {< :n 0}
     then {b n must be positive!}
     else {if {= :n 0} then :a
     else {:f :f {* :a :n} {- :n 1}}}}} 1 :n}}}
-> YFAC

{YFAC 5}
-> 120
{YFAC 50}
-> 3.0414093201713376e+64
```

We can `map` this first-class function to a sequence of numbers:

```
{map YFAC {serie 0 20}}
-> 1 1 2 6 24 120 720 5040 40320 362880 3628800
39916800 479001600 6227020800 87178291200
1307674368000 20922789888000 355687428096000
6402373705728000 121645100408832000
2432902008176640000
```

It's much fast but there is a last point to fix: `{FAC 50}`, `{TFAC 50}` and `{YFAC 50}` return a rounded value **3.0414093201713376e+64** which is obvioulsy not the exact value. We must go a little further and build some tools capable of processing big numbers.

3.2. big numbers

The way the Javascript Math object is implemented puts the limit of natural numbers to $\mathbf{2^{54}}$. Beyond this limit last digits are rounded to zeros, for instance, as we will demonstrate later, the four last digits of $\mathbf{2^{64}}$ **= {pow 2 64} = 18446744073709552000** should be **1616** and are rounded to **2000**. And beyond $\mathbf{2^{69}}$ natural numbers are replaced by float numbers with a maximum of 15 valid digits. In order to overcome this limitation we come back to the definition of a natural number: *A natural number* $a_0a_1...a_n$ *is the value of a polynomial* $\Sigma_{i=0}^{n}a_ix^i$ *for some value of x, called the base*. For instance $12345 = 1*10^4+2*10^3+3*10^2+4*10^1+5*10^0$. We build a set of user defined functions defining `addition, multiplication` of polynomials and some helper functions:

```
{def BI.bigint2pol
 {def BI.bigint2pol.rec
  {lambda {:n :p :i}
   {if {< :i 0}
    then :p
    else {BI.bigint2pol.rec
          :n
          {cons {charAt :i :n} :p} {- :i 1}}}}}
 {lambda {:n}
  {BI.bigint2pol.rec :n nil {- {chars :n} 1}}}}
-> BI.bigint2pol

{def BI.pol2bigint
 {def BI.pol2bigint.rec
  {lambda {:p :n}
   {if {equal? :p nil}
    then :n
    else {BI.pol2bigint.rec
           {cdr :p} {car :p}:n}}}}
 {lambda {:p}
  {let { {:q {list.reverse :p}} }
   {BI.pol2bigint.rec {cdr :q} {car :q}}}}}
-> BI.pol2bigint

{def BI.simplify
 {def BI.simplify.rec
  {lambda {:p :q :r}
   {if {and {equal? :p nil} {= :r 0}}
    then :q
    else {if {equal? :p nil} then {cons :r :q}
    else {BI.simplify.rec
          {cdr :p}
```

```
           {cons {+ {% {car :p} 10} :r} :q}
           {floor {/ {car :p} 10}} }} }}}
  {lambda {:p}
   {BI.simplify.rec {list.reverse :p} nil 0}}}
-> BI.simplify

{def BI.pk
 {lambda {:k :p}
  {if {equal? :p nil}
   then nil
   else {cons {* :k {car :p}}
              {BI.pk :k {cdr :p}}} }}}
-> BI.pk

{def BI.p+
 {lambda {:p1 :p2}
  {if {and {equal? :p1 nil} {equal? :p2 nil}}
   then nil
   else {if {equal? :p1 nil} then :p2
   else {if {equal? :p2 nil} then :p1
   else {cons {+ {car :p1} {car :p2}}
              {BI.p+ {cdr :p1} {cdr :p2} }}}}}}
-> BI.p+

{def BI.p*
 {lambda {:p1 :p2}
  {if {or {equal? :p1 nil} {equal? :p2 nil}}
   then nil
   else {if {not {cons? :p1}}
   then {BI.pk :p1 :p2}
   else {BI.p+ {BI.pk {car :p1} :p2}
               {cons 0 {BI.p* {cdr :p1}
:p2}}}}}}}
-> BI.p*
```

Using this set of functions we can now compute the factorial of natural numbers of any size with an exact precision:

```
{def BI.tfac
 {def BI.tfac.r {lambda {:n :p}
  {if {< :n 1}
   then :p
   else {BI.tfac.r {- :n 1}
        {BI.simplify
         {BI.p* {BI.bigint2pol :n} :p}}}}}}
 {lambda {:n}
  {BI.pol2bigint
   {BI.simplify
    {BI.tfac.r :n {BI.bigint2pol 1}}}}}}
-> BI.tfac

{BI.tfac 50}
-> 30414093201713378043612608166064
   768844377641568960512000000000000
```

We finally reached the goal: with the subset of special forms `[lambda, def, if]`, the `cons` and `lists` structures, and *without any external library* we have *effectively* computed the exact value of **50!** But we need to go a little further.

3.3. when {lambda talk} calls Javascript

Until now user defined functions were exclusively created in the {lambda talk} syntax, with a large speed penalty when comes intensive computation. We can add any Javascript code using the {script ... Javascript code ...} special form. And when the set of user defined functions written in {lambda talk} or Javascript syntaxes increazes in size, it's time to externalize code in some other wiki page used as a **library** and called via a (require library_name). This helps {lambda talk} to stay minimal, coherent, orthogonal, and any user to create, add and maintain his own specific **library**. In the following we illustrate some of these capabilities.

3.3.1. the lib_BN library

Jonas Raoni Soares Silva[9] has written a small (150 lines) and smart Javascript library, `BigNumber`, ready to be called via {lambda talk} wrapping functions, everything being stored in another wiki page, `lib_BN`. We just write the factorial function using the multiplicate operator `BN.*` redefined for big numbers:

```
{def BN.fac
 {lambda {:n}
  {if {= :n 0}
   then 1
   else {BN.* :n {BN.fac {- :n 1}}}}}}
-> BN.fac
```

and call it on the number **50**:

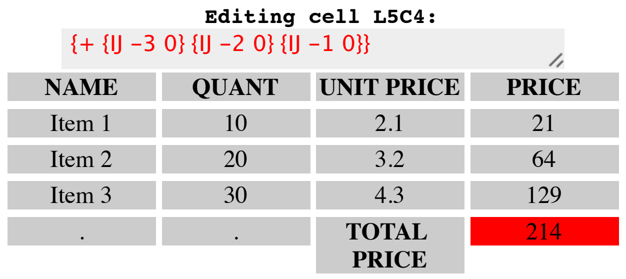
```
{BN.fac 50}
-> 30414093201713378043612608166064
   768844377641568960512000000000000
```

Obviously it's the fastest and best choice!

Note: Using this library, we can control that the value of *2^{64}* given by the Javascript Math object, *{pow 2 64} = 18446744073709552000* is not its exact value *{BN.pow 2 64} = 18446744073709551616*!

3.3.2. the lib_sheet library

A spreadsheet is a good illustration of functional languages, (Simon Peyton-Jones [10]). A spreadsheet is an interactive computer application for organization, analysis and storage of data in tabular form. The basic idea is that each cell contains the input - *words and expressions* - and displays the output. Calling a set of Javascript and {lambda talk} functions stored in a wiki page, `lib_sheet`, and writing {sheet 4 5} displays the following table of 5 rows and 4 columns of editable cells:



Editing cell L5C4:

{+ {IJ -3 0} {IJ -2 0} {IJ -1 0}}

NAME	QUANT	UNIT PRICE	PRICE
Item 1	10	2.1	21
Item 2	20	3.2	64
Item 3	30	4.3	129
.	.	TOTAL PRICE	214

[local storage]

Two specific functions are added for linking cells:

- `{LC i j}` returns the value of the cell L_iC_j as an *absolute reference*,
- `{IJ i j}` returns the value of the cell $L_{[i]}C_{[j]}$ as a *relative reference*. For instance writing `{IJ -1 -1}` in `L2C2` will return the value of `L1C1`.

Datas are stored in the browser's *localStorage*. The **[local storage]** button opens a window where the spreadsheet can be alternatively edited in a JSON format:

```
["{b NAME}","{b QUANT}","{b UNIT PRICE}","{b
PRICE}","Item 1","10","2.1","{* {IJ 0 -2} {IJ 0
-1}}","Item 2","20","3.2","{* {IJ 0 -2} {IJ 0
-1}}","Item 3","30","4.3","{* {IJ 0 -2} {IJ 0
-1}}","","","{b TOTAL PRICE}","{+ {IJ -3 0} {IJ
-2 0} {IJ -1 0}}","4"]
```

3.3.3. graphics

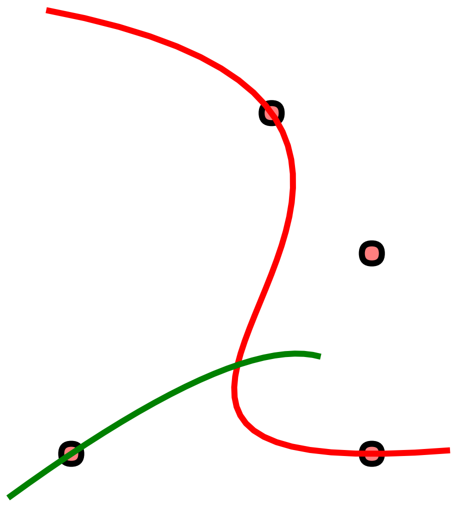
The **de Casteljau** recursive algorithm[11] allows drawing Bezier curves of any degree, i.e controlled by any number of points. Defining points as pairs and control polylines as lists, we build a small set of {lambda talk} user defined functions feeding the points attributes of SVG polylines:

```
{def castel.interpol {lambda {:p0 :p1 :t}
 {cons {+ {* {car :p0} {- 1 :t}}
          {* {car :p1} :t}}
       {+ {* {cdr :p0} {- 1 :t}}
          {* {cdr :p1} :t}}
}}}
-> castel.interpol

{def castel.sub {lambda {:l :t}
 {if {equal? {cdr :l} nil}
  then nil
  else {cons
   {castel.interpol {car :l} {car {cdr :l}} :t}
   {castel.sub {cdr :l} :t}}}}}
-> castel.sub

{def castel.point {lambda {:l :t}
 {if {equal? {cdr :l} nil}
  then {car {car :l}} {cdr {car :l}}
  else {castel.point {castel.sub :l :t} :t}}}}
-> castel.point

{def castel.build {lambda {:l :a :b :d}
 {map {castel.point :l} {serie :a :b :d}}}}
-> castel.build

{def svg.dot {lambda {:p}
 {circle {@ cx="{car :p}" cy="{cdr :p}" r="5"
           stroke="black" stroke-width="3"
           fill="rgba(255,0,0,0.5)"}}}}
-> svg.dot
```

For instance the following code:

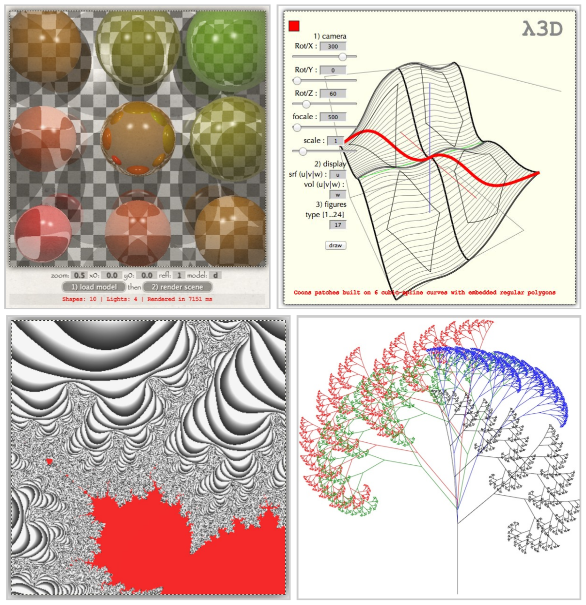
```
{def p0 {cons 150 80}}  -> p0
{def p1 {cons 200 150}} -> p1
{def p2 {cons 50 250}}  -> p2
{def p3 {cons 200 250}} -> p3
{def red_curve {castel.build
  {list {p0} {p1} {p2} {p3}}
   -0.3 1.1 {pow 2 -5}}}
-> red_curve
{def green_curve {castel.build
  {list {p2} {p1} {p3}}
   -0.1 0.6 {pow 2 -5}}}
-> green_curve

{svg.dot {p0}}
{svg.dot {p1}}
{svg.dot {p2}}
{svg.dot {p3}}

{polyline {@ points="{red_curve}"
  stroke="red" fill="transparent"
  stroke-width="3"}}
{polyline {@ points="{green_curve}"
  stroke="green" fill="transparent"
  stroke-width="3"}}
```

draws a cute **λ** in an SVG frame:



3.3.4. intensive computing

For intensive computing, it's obvioulsly more efficient to call the underlying language, Javascript. These are screenshots of {lambda tank}'s pages dedicated to **ray-tracing,[12] curved shapes modeling[13], fractals[14], turtle graphics drawing[15]**.



3.3.5. mathML

$$i\hbar\frac{\partial\psi}{\partial t}(x,t) = \left(mc^2\alpha_O - i\hbar c\sum_{j=1}^{3}\alpha_j\frac{\partial}{\partial x_j}\right)\psi(x,t)$$

The Dirac equation in the form originally proposed by Dirac {lambda talk} forgets the **MathML** markup set which is not implemented in Google Chrome[16]. A set of functions, *exclusively built on standard HTML and CSS rules*, can be defined to render Math Symbols. For instance the above **Dirac equation** is not a picture but the result of the code below:

```
i{del h}{QUOTIENT 30 ∂ψ ∂t}(x,t) = {PAREN 3 (}
mc{sup 2}α{sub 0} - i{del h}c {SIGMA 30 j=1 3}
α{sub j}
{QUOTIENT 30 ∂ ∂x{sub j}} {PAREN 3 )} ψ(x,t)
```

calling three user defined {lambda talk} functions:

```
{def QUOTIENT
 {lambda {:s :num :denom}
  {table
   {@ style="width::spx;
             display:inline-block;
             vertical-align:middle;
             text-align:center;"}
 {tr {td {@ style="border:0 solid;
                   border-bottom:1px
solid;"}:num}}
 {tr {td {@ style="border:0 solid;"}:denom}} }}}
-> QUOTIENT

{def SIGMA
 {lambda {:s :one :two}
  {table
   {@ style="width::spx;
             display:inline-block;
             vertical-align:middle;
             text-align:center;"}
 {tr {td {@ style="border:0 solid;"}:two}}
 {tr {td {@ style="border:0 solid;
                   font-size:2em;
                   line-height:0.7em;"}Σ}}
 {tr {td {@ style="border:0 solid;"}:one}} }}}
-> SIGMA

{def PAREN
 {lambda {:s :p}
  {span {@ style="font:normal :sem arial;
                  vertical-align:-0.15em;"}:p}}}
-> PAREN
```

3.4. what about macros?

A language without macros is not a *true* language, isnt'it? {lambda talk} macros bring (*a little bit of*) the power of **regular expressions** directly in the language.

3.4.1. make it variadic

{lambda talk} comes with some variadic primitives, for instance `[+, -, *, /, list, ...]`. But at first sight, user functions can't be defined variadic, for instance:

```
{def mul {lambda {:x :y} {* :x :y}}} -> mul
{* 1 2 3 4 5}   -> 120 // * is variadic
{mul 1 2 3 4 5} -> 2   // 3, 4, 5 are ignored
```

In order to make `mul` variadic we glue values in a list and define an helper function, `variadic`:

```
{def variadic
 {lambda {:f :args}
  {if   {equal? {cdr :args} nil}
   then {car :args}
   else {:f {car :args}
            {variadic :f {cdr :args}}}}}}
-> variadic

{variadic mul {list 1 2 3 4 5}}
-> 120
```

But it's ugly and doesn't follow a standard call. We can do better using a macro:

```
1) defining:
{macro {mul* (.*?)}
    to {variadic mul {list €1}}}

2) using:
{mul* 1 2 3 4 5}
-> 120
```

Now `mul*` is a variadic function which can be used as any other primitive or user function, except that it is not, as in most Lisps, a first class function.

3.4.2. titles, paragraphs & links

As a last example, {lambda talk} comes with a predefined small set of macros allowing writing without curly braces titles, paragraphs, list items, links:

```
_h1 TITLE ¬
  stands for: {h1 TITLE}

_p Some paragraph ... ¬
  stands for: {p Some paragraph ...}

[[PIXAR|http://www.pixar.com/]]
  stands for: {a {@
href="http://www.pixar.com/"}PIXAR}

[[sandbox]]
  stands for: {a {@ href="?
view=sandbox"}sandbox}
```

These simplified alternatives, avoiding curly braces as much as possible, are fully used in the current document.

CONCLUSION

{lambda talk} takes benefit from the extraordinary power of modern web browsers, simply adding a coherent and unique syntax, *without re-inventing the wheel*, just using existing tools, HTML/CSS, the DOM and Javascript. *Standing on the shoulders of such giants*, {lambda talk} can be built as a minimal regexp based implementation of the λ-calculus, where the repeated substitutions inside the code string overcomes limitations of regular language, where the lack of closure is balanced by the built-in partial application functionality, where a dictionary initially empty can be extended "inline" via user defined libraries. More can be seen in the following APPENDICE.

The {lambda way} project is a thin overlay - about 100kb - built upon any modern browser, proposing a small interactive development environment, {lambda tank}, and a coherent language, {lambda talk}, without any external dependencies and thereby easy to download and install on a web account provider running PHP. From any web browser on any system, complex web pages can be created, enriched, structured and (algorithms) tested in real time on the web. The current document has been created in this wiki page, `http://lambdaway.free.fr/workshop/?view=oxford` then directly printed from the browser as a PDF document.

.

Alain Marty, 2017/07/28

APPENDICE

In this section we present the minimal set of JavaScript functions necessary and sufficient to implement `abstractions`, `applications`, `definitions` and the `ifthenelse` control structure.

1. evaluation

Working on the client side the {lambda talk} evaluator is a Javascript **IIFE** (Immediately Invoked Function Expression), `LAMBDATALK`, returning the public function `eval()`. This function is called at every keyboard entry and replaces the string code by its evaluation, without building any Abstract Syntaxic Tree.

```
 var LAMBDATALK = (function() {
 var eval = function(str) {
   str = pre_processing(str);
   str = abstract_lambdas(str); // abstraction
   str = abstract_defs(str);    // abstraction
   // some other special forms
   str = eval_forms(str);       // application
   str = post_processing(str);
   return str;
 };
 return {eval:eval}
 })();
```

2. application

In a single loop, using a single regular expression[17], simple forms `{first rest}` are recursively evaluated from the leaves to the root and replaced by words. The evaluator stops when simple forms are reduced to a sequence of words, actually a valid HTML code sent to the browser's engine for the final evaluation and display. Using a *regular expression based window*, the evaluator literally loops over the code string, skips the words and progressively replaces *in situ* forms by words. *The repeated substitutions inside the code string overcomes limitations of regular language*. A kind of Turing machine[18] ...

```
 var eval_forms = function( str ) {
   while (str !=
 (str=str.replace(leaf,eval_leaf)))
     ; // does nothing!
   return str
 };
 var leaf = /\{([^\s{}]*)(?:[\s]*)([^{}]*)\}/g;
 var eval_leaf = function(_,f,r) {
   return (DICT.hasOwnProperty(f)) ?
           DICT[f].apply(null,[r]) : '('+f+'
 '+r+')'
 };
 var DICT = {}; // initially empty
```

3. abstraction

Special forms `{lambda {arg*} body}` are matched and evaluated **before** simple forms and replaced by a reference to an anonymous function added to the dictionary. The following code demonstrates that:

- **1) lambdas** are **first class** functions,
- **2) lambdas** accept **partial function application**, when called with a number of values lesser than their arity, they memorize the given values and return a lambda waiting for the rest,
- **3) lambdas don't create closures**, inner lambdas have no access to outer lambdas' arguments, there is no lexical scoping, no environment, no free variables. Like mathematical functions lambdas are pure black boxes.

```
var abstract_lambdas = function(str) {
  while ( str !== ( str =
    form_replace(str,'{lambda',
abstract_lambda)));
  return str
};

var abstract_lambda = function(s){
 s = abstract_lambdas( s );  // nested lambdas
 var index = s.indexOf('}'),
     args = supertrim(s.substring(1,
index)).split(' '),
     body = s.substring(index+2).trim(),
     name = '_LAMB_' + LAMB_num++,
     reg_args = [];
 for (var i=0; i < args.length; i++)
   reg_args[i] = RegExp( args[i], 'g');
 body = abstract_ifs( body ); // {ifthenelse}
 DICT[name] = function() {
  var vals =
      supertrim(arguments[0]).split(' ');
  return function(bod) {
   bod = ifthenelse( bod, reg_args, vals );
   if (vals.length < args.length) {
    for (var i=0; i < vals.length; i++)
     bod = bod.replace(reg_args[i],vals[i]);
    var _args=args.slice(vals.length).join(' ');
    bod = '{' + _args + '} ' + bod;
    bod = abstract_lambda(bod);// return a
lambda
   } else {                    // return a form
    for (var i=0; i <  args.length; i++)
     bod = bod.replace(reg_args[i],vals[i]);
   }
   return bod;
  }(body);
 };
 return name;
};

var form_replace = function(str,sym,func,flag){
  sym += ' ';
  var s = catch_form( sym, str );
  return (s==='none')?
     str:str.replace(sym+s+'}',func(s,flag))
};

var catch_form = function( symbol, str ) {
  var start = str.indexOf( symbol );
  if (start == -1) return 'none';
  var d0, d1, d2;
  if (symbol === "'{")     { d0=1; d1=1; d2=1;}
  else if (symbol === "{") { d0=0; d1=0; d2=1;}
  else          { d0=0; d1=symbol.length; d2=0;}
  var nb = 1, index = start+d0;
  while(nb  > 0) { index++;
       if ( str.charAt(index) == '{' ) nb++;
    else if ( str.charAt(index) == '}' ) nb--;
  }
  return str.substring( start+d1, index+d2 )
};
```

4. definition

Special forms `{def name expression}` are matched and evaluated **before** simple forms and replaced by `name` as a reference to `expression` added to the dictionary.

```
var abstract_defs = function(str, flag) {
  while ( str !== ( str =
    form_replace( str, '{def', abstract_def,
flag ))) ;
  return str
};
var abstract_def = function (s, flag) {
  flag = (flag === undefined)? true : false;
  s = abstract_defs( s, false );
  var index = s.search(/\s/), // match spaces
      name  = s.substring(0, index).trim(),
      body  = s.substring(index).trim();
  if (body.substring(0,6) === '_LAMB_') {
    DICT[name] = DICT[body];
    delete DICT[body];
  } else {
     body = eval_forms(body);
     DICT[name] = function() { return body };
  }
  return (flag)? name : '';
};
```

5. ifthenelse

Special forms `{if bool then one else two}` are matched in lambda's bodies and replaced by the reference to an array `[bool, one, two]`. When the lambda is called with some values, `one` or `two` is returned according to the **bool** value.

```
var abstract_ifs = function(str) {
  while ( str !== ( str =
    form_replace( str, '{if', abstract_if ))) ;
  return str
};
var abstract_if = function(s){
  s = eval_ifs( s );
  var name = '_COND_' + COND_num++;
  var index1 = s.indexOf( 'then' ),
      index2 = s.indexOf( 'else' ),
      bool = s.substring(0,index1).trim(),
      one = s.substring(index1+5,index2).trim(),
      two = s.substring(index2+5).trim();
  COND[name] = [bool,one,two];
  return name;
};
var eval_ifs = function(bod, reg_args, vals) {
   var m = bod.match( /_COND_\d+/ );
   if (m === null) {
     return bod
   } else {
     var name = m[0];
     var cond = COND[name];
     if (cond === undefined) return bod;
     var bool=cond[0], one=cond[1], two=cond[2];
     if (reg_args !== undefined) {
      for (var i=0; i < vals.length; i++) {
       bool = bool.replace(reg_args[i],
vals[i]);
       one = one.replace(reg_args[i], vals[i]);
       two = two.replace(reg_args[i], vals[i]);
      }
     }
     var boolval = (eval_forms(bool)==='true')?
                   one : two;
     bod = bod.replace( name, boolval );
     return eval_ifs( bod, reg_args, vals )
   }
};
```

REFERENCES

[1] The Wiki way: http://dl.acm.org/citation.cfm?id=375211
[2] Ward_Cunningham: http://ward.asia.wiki.org/view/testing-microtalk
[3] A Tutorial Introduction to the Lambda Calculus (Raul Rojas): http://www.inf.fu-berlin.de/lehre/WS03/alpi/lambda.pdf
[4] Lisp: http://www.cs.utexas.edu/~cannata/cs345/Class%20Notes/06%20Lisp.pdf
[5] Scheme: https://mitpress.mit.edu/sicp/full-text/book/book.html
[6] Collected Lambda Calculus Functions: http://jwodder.freeshell.org/lambda.html
[7] Stephen_Cole_Kleene: https://en.wikipedia.org/wiki/Stephen_Cole_Kleene
[8] L. Peter Deutsch https://fr.wikipedia.org/wiki/L._Peter_Deutsch https://sites.google.com/a/gertrudandcope.com/info/Publications/Patterns/C--Report/SpaceIII
[9] Jonas Raoni Soares Silva http://jsfromhell.com/classes/bignumber
[10] Simon_Peyton_Jones: https://www.microsoft.com/en-us/research/people/simonpj/#
[11] De_Casteljau's_algorithm: http://www.malinc.se/m/DeCasteljauAndBezier.php
[12] raytracing: http://lambdaway.free.fr/workshop/?view=raytracing
[13] pForms: http://lambdaway.free.fr/workshop/?view=pforms
[14] fractal: http://lambdaway.free.fr/workshop/?view=mandel
[15] turtle: http://lambdaway.free.fr/workshop/?view=turtle_tree
[16] google-subtracts-mathml-from-chrome: https://www.cnet.com/news/google-subtracts-mathml-from-chrome-and-anger-multiplies/
[17] Regular Expressions: http://blog.stevenlevithan.com/archives/reverse-recursive-pattern
[18] Turing machines implemented in JavaScript http://www.turing.org.uk/book/update/tmjavar.html

{λ way^2} v.20170717