Toward an unstaging translation for an environment classifier based multi-staged language

Mathias Sablé Meyer
Supervisor: Luke Ong

Last semester of the academic year 2014/2015

This is a report of the internship I had in the computer science department of Oxford under the supervision of Luke Ong. It was a 5 months research internship from March to July 2015 and was part of the first year of my Master Degree, the MPRI, that I'm attending as a computer science student from ENS Cachan.

After I took some time to discover and understand multi-staged programming and all the ways it was formalised depending on the various researcher's approach and goals, I focused on a specific logical approach and worked on verification for the subsequent language — more precisely, I worked on an unstaging translation for this language.

This report will be structured as such: after a long introduction on the notion of multi-stage programming that will try to cover the inherent benefits of this paradigm as well as the various relevant approaches, I will describe in a more comprehensive way a specific formalism based on environment classifiers, the $\lambda^{\blacktriangleright}$-calculus and its associated MiniML$^{\blacktriangleright}$, and describe the unstaging translation I worked on and the subsequent results.

Such an unstaging translation serves two goals: it is useful for verification purposes as the target language is already well known and posses various verification techniques, but more importantly from a theoretical point of view it fills the gap between the environment classifiers approach and the usual approaches as the profound links between these two techniques are yet undiscovered.

A few less important results I stumbled upon while working on the main subject are reported in the annex: while they are not part of the main work they are interesting to get intuitions on various subject I'm describing.

1 Introduction

This first part contains a short survey of the multi-staged programming field, as well as an introduction to its main concepts.

1.1 Notions of Multi-staged programming

Let's start with a short introduction to the concept of multi-staged programming, its advantages, to see how and why it can and should be used in some situations as an efficient programming tool.

1.1.1 General notions

For more than twenty years the study of meta-programming systems as formal systems has been very active. In a meta-programming system, a meta-program can manipulate other program, by combining them, observing them, or executing them. When these manipulation are done at run time, and when the programs manipulated are themselves run-time meta-programs, we talk about **multi-staged programming**.

The main idea when using meta-programming is to use partial evaluation to adapt a desired program: one uses partial informations about the environment to generate a case-specific program that solves this problem, rather than writing a general-purpose program.

After a short introduction on the *staging annotations*, an illustration will be given with some basic examples. A more complete introduction is to be found in [16, 17] while [14] gives a broad survey of the *state of the art* of meta-programming in general as it was in 2001, before the notion of environment classifiers was introduced.

1.1.2 Common notations

I will present the situation of staged-programming with annotations, in this context it is common to add these three annotations:

Brackets usually written $\langle expr \rangle$, are to surround an expression to delay its execution. The inner expression is then *staged* and can latter be combined or executed.

Escape usually written $\tilde{}expr$ is the combining operator, it allows one to build larger staged expressions using existing ones.

For example given $a = \langle e_1 \rangle$ and $b = \langle e_2 \rangle$, we can represent $\langle e_1 + e_2 \rangle$ with $\langle \tilde{}a + \tilde{}b \rangle$

Run sometimes written **run** *expr* and sometimes $!expr$, actually executes a staged expression.

For example $!\langle 2 \times 3 \rangle = 6$, and if we want to use all the constructors:

$$!\langle \tilde{}\langle 2 \rangle \times \tilde{}\langle 3 \rangle \rangle = 6$$

Sometime the **run** is ommited as it can be considered as a special case of **escape** at stage 0, and there is no confusion possible, however it is often useful to be able to distinguish easily between these two constructions when writing some code.

Remark : throughout the literature a lot of various notations and names were used for these annotations, such as quote/unquote, box/unbox, etc. I find this one easy to read and will use it for the introduction, but I may refer to bracket using either "boxing" or "staging" and to escape using "unboxing" or "unstaging".

The quote/unquote terminology comes from Lisp's quotation system that can be seen as a primitive multi-staged tool.

1.1.3 Examples

A simple example of a situation where adapted code is an interesting feature and easy to set up using staged-programming is the one of a program that would have to compute x^n often, for various x but for a fixed n that the program can only access at run-time.

In a purely functional style one would write the power function recursively, and call it whenever a new computed value is needed. But of every new x, the program needs to recursively compute x^n and therefore needs a recursive stack of size n to perform exactly the same sequence of operation.

Here is an example in an ML syntax:

```
let rec pow n x = match n with
  | 0 -> 1
  | 1 -> x
  | n -> x * (pow (n-1) x)
pow:  int -> int -> int

let rec pow_staged n x = match n with
  | 0 -> .< 1 >.
  | 1 -> .< x >.
  | n -> .< .~x * .~(pow_staged (n-1) x) >.
pow_staged:  int -> int code -> int code

let power_5 x = fun x ->
  !. .< .~(pow_staged 5 .< x >.) >.
power_5:  int -> int
```

Now the question is, how much better is the staged version? This is what power_5 looks like[1]:

```
powser_5 = fun x -> x * x * x * x * x
power_5:  int -> int
```

Which is often the most efficient way to write a function that computes x^5 without using the exponentiation by squaring. But it's as easy, though slightly less easy to read, to do the same thing for the exponentiation by squaring algorithm, and here's the output[2]:

```
power_5 = fun x -> x * (let y_1 = let y_0 = x in y_0 * y_0 in y_1 * y_1)
power_5:  int -> int
```

It's 3 multiplications as it would be in the recursive version, but there is no recursive call in the final version anymore, i.e. the result is a function that computes x^5 in the most efficient way possible in this language.

Of course in this example the difference is not significant as the only computational difference is the use or not of a recursive call stack of length 3, but it is easy to imagine situations where only

[1] This is the actual output of this code in MetaOCaml
[2] Idem

a certain amount of information is available when the program starts, we can precompute the adapted sub-program for this situation and then apply it : for example all the matrix operation used in machine learning would have to be written for arbitrary matrix, but the data set is of a fixed size hence we can compute all the adapted functions for this size and only use them afterward. See [4] for a relevant exemple on the Gaussian elimination problem.

One of the most interesting use of this is to be found in [16] where W. Taha gives an example of a interpreter for a small language with this paradigm. The staged version behaves like a transcription from the target language to the original language, which mean that the execution time for a given function in the target language is approximately the same as the same function written in the original language: there is *no cost* for the interpretation of the language in that interpreted code is as efficient as the equivalent code in the original language.

Another instructive example is that of a generalised Fibonacci function, requiring the two first terms as arguments of the function:

$$fibo(x, y, n) = \begin{cases} x & \text{if } n = 0 \\ y & \text{if } n = 1 \\ fibo(x, y, n-1) + fibo(x, y, n-2) & \text{otherwise} \end{cases}$$

I will be using hashtables in the following because it's a generic way to memoize function, although it is of course possible to implement efficiently this function without them.

```
let fibo x y n =
  let mem = Hashtbl.create 101 in
  let rec aux x y n = begin
    try Hashtbl.find mem n
    with Not_found ->
      let r = match n with
        | 0 -> x | 1 -> y
        | n -> (aux x y (n-1)) + (aux x y (n-2))
      in
      Hashtbl.add mem n r; r
  end in aux x y n
  fibo :  int -> int -> int -> int
```

This is the memoized recursive version of the function; it operates in n operations, and requires n recursive calls.

While it is possible to "naively" add annotations as we did for the power function, it appears quickly that this is not a satisfactory solution, as the staged code will not contain any **let** and the program will systematically go down to the base values to add them to the current situation, here is an example of such a version:

```
let fibo_staged_naive x y n =
  let mem = Hashtbl.create 101 in
  let rec aux (x:int) (y:int) n =
    begin try Hashtbl.find mem n
      with Not_found ->
      let r = match n with
        | 0 -> .<x>. | 1 -> .<y>.
        | n ->
          let r1 = aux x y (n-1) in
```

```
            let r2 = aux x y (n-2) in
            .<.~r1 + .~r2>.
        in Hashtbl.add mem n r; r
    end in aux x y n
fibo_staged_naive :  int -> int -> int -> int code
```

For example calling this function with x and y unbound and $n = 6$ will return the following piece of code:

$$((((y + x) + y) + (y + x)) + ((y + x) + y)) + (((y + x) + y) + (y + x))$$

While the result is correct, the expected number of operations is not met and it only adds the base value without remembering intermediate results: the staging is not efficiently used at all here.

This specific problem and how to address it systematically is discussed deeply in [15].

The key here is to rewrite the program in continuation passing style (CPS), to be able to add a **let** inside the staged part so that the computation does not backtrack each time down to the base values, we obtain the following program:

```
let fibo_staged x y n =
let mem = Hashtbl.create 101 in
let rec aux x y n k = begin
  try let r = Hashtbl.find mem n in .<.~(k r)>.
  with Not_found ->
    match n with
      | 0 -> k .<x>. | 1 -> k .<y>.
      | n ->
  aux x y (n-1) (fun r1 -> Hashtbl.add mem (n-1) r1;
  aux x y (n-2) (fun r2 -> Hashtbl.add mem (n-2) r2;
  .<let r = .~r1 + .~r2 in .~(k .<r>.) >.))
end
in aux x y n (fun x -> x)
fibo_staged :  int -> int -> int -> int code
```

Here is the output on unbound x and y for $n = 6$

```
fun x -> fun y ->
let r_1 = y + x in
let r_2 = r_1 + y in
let r_3 = r_2 + r_1 in
let r_4 = r_3 + r_2 in
let r_5 = r_4 + r_3 in r_5
```

This time the result is closer to what we expect and looks like the efficient way to compute the n^{th} term of the generalised Fibonacci function as it only requires n variables and n additions. Once again, compared to the recursive version, evaluating this function does not require any recursive call stack. A proof of correctness of this version is given in annex B.

And while this is more an illustrative example than an actual practical example, here again if you need to compute this function up to a given n multiple times, but you want to change the base values, this allows you to do so at run-time and saves you the cost of the recursive calls.

Hence multi-staged programming is an interesting feature to add to a language, but it can break other powerful feature already existing in the target language.

Remark : One important feature of a multi-stage programming language is how it handles variable capturing, and how it prevents open code from being executed. Let's have a small example, with our notation the identity function can be written:

$$\big(\lambda e.!\langle \lambda x.\tilde{}e\rangle\big)\langle x\rangle$$

Therefore we have to unbox an unknown expression and to run the resulting code. While there is no problem here as the x is indeed bound, a lot of issues arise around this specific kind of variable capturing terms. This is called "unhygienic variable capturing".

Some language would not accept this term as we are creating a piece of open code and that can be harmful. But other try to cope with them, which can be a desirable feature and was heavily used with Lisp's quotation system for example.

Another weaker question is "can a variable bound at a given stage be used in a later stage?", which is called Cross Stage Persistency (CSP), and is accepted by most languages though not all of them. The most simple exemple is $\lambda x.\langle x\rangle$, i.e. given a term can we make a piece of code which is exactly this term.

1.2 Literature survey

Let's have a look at the evolution of research about staged programming, trying to understand the current *state-of-the-art* and what led to this.

1.2.1 Foundations of Multi-Staged Programming

Logical Approach One way to discover and to give a theoretical foundation to MSP is to study specific extensions of the Curry-Howard isomorphism.

This approach first appears with [6] as R. Davies extends the Curry-Howard isomorphism to include the temporal operator "next" represented by $\bigcirc$, and thus creates the $\lambda^{\bigcirc}$-calculus, a conservative extension of the simple λ-calculus. This extension is proven in [6] to be equivalent to the most simple multi-staged language with the ability to manipulate open code — that is code with free variables.

One can also extend the isomorphism by adding the necessity operator $\Box$ from modal logic, as studied by R. Davies and F. Pfenning in [7] while creating MiniML$^{\Box}$. This extension also leads to a multi-staged language, but restricted to closed code unlike the previous one, with the advantage of resistance to the addition of some effects: it provides the insurance that an evaluated code object will not contain free variables as all code pieces are closed, but you can add references to the language easily, for example.

Later on, new extensions appeared. While **environment classifiers** [19] were originally rather operationally motivated, T. Tsukada and A. Igarashi gave them a logical foundation in [21] by giving a new extension to the Curry-Howard isomorphism that deals with this notion.

This first leads to a CBV $\lambda^{\triangleright}$-calculus, which is proved to satisfy subject reduction, strong normalisation, confluence, and a notion of time-ordered normalization — stages are evaluated and rewritten in an incrising order which means that once a given stage n was reduced, all the subsequent reduction will take place at stage $> n+1$

From this was built MiniML$^{\triangleright}$, and extension with base types and (call-by-value) recursion of the $\lambda^{\triangleright}$-calculus.

A big-step semantics is given, as well as a proof that it implements staged execution of the code. Finally the induced proof system extracted from this language is proved to be sound and complete in a Kripke semantis.

This language does not have CSP for all types, and type inference for the whole language is not possible, but a subset of it satisfies this and is very close to MetaOCaml.

Three years later, Y. Hanada and A. Igarashi improved the $\lambda^{\triangleright}$ calculus with better CSP handling, creating the $\lambda^{\triangleright\%}$-calculus [8]. This call-by-value language satisfies subject reduction, strong-normalisation, confluence, and a specific property called Type-Safe Residualization — this property ensures that a well-typed object of code type has a body that is itself well-typed.

However this extension does not fill the absence of type inference already present in the original paper [21].

Operational approach On the other hand, staged computation was already known for a long time as it was present in various languages more or less hidden, for example with Lisp's *quotation* system or with macros in C. Some papers try to formalise this operational approach.

For example, while MiniML$^{\Box}$ appeared as a modal logic extension of the Curry-Howard isomorphism, similar results were obtained — using an operational motivation — by W. Taha et al. in [20], later corrected in [18] to preserve soundness of their language in the absence of exceptions and references.

This operational approach led to MetaML, a call-by-value, typed (monomorphic) calculus, and [18] proves that the core of this language, without any fixed-point operator, satisfies the Church-Rosser property, Subject-Reduction, and type safety. Yet unfortunately, it cannot execute open code nor distinguish between open and closed code with its type system.

To address this issue, E. Moggi, W. Taha et al. created a new extension called *An Idealized MetaML* [13], borrowing ideas from [6] for open code and from [7] for closed code, as well as the previous construction of MetaML in [18]. A big-step semantics was provided for either a call-by-name or for a call-by-value version, as well as type preservations results. They also prove that their language actually embeds MetaML, Mini-ML$^{\Box}$ and $\lambda^{\bigcirc}$.

What is more, major conversion functions from open-code to closed-code and backward remained correct in this formalism. The drawback of this is that the syntax had to be extremely verbose and unpleasant to use.

One year later, this system was extended using the notion of **closed types** [2] to include computational effects safely in a staged model. This led to a call-by-value language MiniML$^{BN}_{ref}$, with a fixed-point operator, ML-style references, and [2] proves type safety for this new language.

The main issue addressed by this extension is the existence, at run-time, of both references and free-variables, that used to lead to errors: the idea is to introduce a new **closedness** annotation specifically designed prevent this kind of error, by removing the ability to assign an open-code term to a variable that was designed for closed-code terms only.

E. Moggi and C. Calcagno extended it again [3] the same year so that the restriction to closed-code of this language should be a conservative extension of a non-staged ML language.

Another major idea to deal with type system appeared two years after with the idea of **environment classifiers** [19].

The underlying idea is that rather than only keeping information on the depth of various staged terms, this system also keeps tracks of which variables are free in a given term. Therefore manipulated open code is authorised while one can ensure that **run** is never applied on open code.

Instead of requiring explicit annotations for CSP terms, it offers an efficient and sound way to type a program, and allow the typing of open code. It embeds $\lambda^{\bigcirc}$ as well as MiniML$^{\Box}$. There is a proof of type safety in the monomorphic situation, and one of subject reduction if adding polymorphism.

Another result is to avoid the heavy syntax required by [13] and to be able to keep a Hindley-Milner type inference.

This idea was implemented in the curent public releases of **MetaOCaml** called MetaOCaml+BER (for Bracket, Escape, Run), an OCaml extension designed for staged programming.

Kim et al. had a more direct approach in [11], while wanting to add Lisp's constructs to ML. The resulting language, λ_{sim}^{open} (and its polymorphic version, λ_{open}^{poly}) is designed to manipulate open code, to behave well with imperative constructs, and accepts unhygienic variable capturing.

The type system is proved to satisfy type preservation in the given semantics. What's more, a sound and complete type inference algorithm is given.

1.2.2 Analysing a Multi-Staged program

Another important aspect is the question of **how to statically verify** an MSP program.

A first approach was to translate it into a common, well-known other language, to statically analyse the translated version, and then to be able to translate the analysis back into the original version.

This is the strategy adopted by W. Choi et al. in [5]. Improving the original idea found in [1], they define a staged calculus named λ_S, and then they introduce a simply-typed λ_ρ-calculus with record operations. Their idea is to store the information on what a program can manipulate and what it knows while inside a specific stage into a stack of records representing the environment, and to push/pop them when you get in/out of a stage. The advantage of such an approach is that it handles the variable capturing problem, allowing the analysis of unhygienic programs.

Their translations preserve type, and each step of the original languages can be simulated by the record, while a translated version is easy to translate back. On their target language you can apply well-known verification techniques, they use as an example **0CFA**. Finally, given a few weak assumptions, you can translate back the results to the original program.

Of course another orthogonal approach is to analyse the staged version directly, this is the method adopted by J. Inoue and W. Taha in [9] for example.

They answer three natural questions that are:

1. Is adding staging to a language conservative in general?
2. Is there such a thing as a notion of orthogonality between a program and its staging annotations?
3. What extensional properties do we have between programs — for example, are two function that agree on all arguments equivalent?

Sadly, but as expected, the first answer is "no" as reasoning under substitution is unsound in the case of staged programs.

But they are able to give a simple condition so that the **erasure theorem**, that answers the second question, holds.

They intruduce a very simple unstaging function on terms that just removes the anotations without touching anything else in the code, denoted $||\bullet||$, and the idea behind the erasure theorem is that if a terms e rewrites to an non-staged term $||t||$ then $e \hat{=} ||e||$ where $\hat{=}$ is their "provable equality" which is a reasonable subset of the observational equivalence (this is true for the call-by-name version, the call-by-value version is slightly more complicated).

Finally, they introduce then prove sound a notion of applicative bisimulation between programs, and their notion of bisimulation subsumes all extensional equivalence between programs. Roughly, the point is that when two programs are bisimilar, either they both diverge or they both terminate, and if they terminate their value must bisimulate again under **experiments** that test their behaviours (in an *experiment*, the environment can call functions, and run code values, but may never modify the constants).

The idea is that using the second and the third point, you can easily prove an equivalence between a staged program and its non-staged version, and in many cases the non-staged version is close to the program you would write naturally in a non-staged language, therefore easier to prove correct.

This approach is also used in [12] that focuses on providing a static analysis in the style of 0CFA for the "eval" construct of JavaScript, by modelling it by a multi-stage augmented version of λ_{JS}, a core calculus for JavaScript — the ultimate goal being to actually provide a translation from the language with **eval** constructor to this one. Then the idea is to extend 0CFA to handle the staging annotations, and to prove that the result is still sound.

1.2.3 Ease of use

While multi-staged programming has proved to be very efficient in some cases [16], it is not always straightforward to write and well-known algorithms might fail due to an unsafe addition of staging annotations: the previously shown Fibonacci example is relevant on that matter.

Therefore research was made on how to make the addition of staging annotations as safe as possible.

This issue is address in [15] using a monadic approach, designed to solve the specific memoization problem as noticed in the Fibonacci example, often by automatically translating it into a **continuation passing style** (CPS) version, while [4] extends and applies this solution to the problem of **gaussian elimination**.

Another way to achieve the same goal is to restrict the type system so that every program has the same restrictions on side effects as if it was written in **CPS** directly. This is the purpose of the $\lambda^{\varnothing}$-calculus as presented in [10]. The ability to run the generated code inside the program itself, as well as the cross-stage persistence, are traded against the ability to prove that the generated code is well-typed in the presence of various effects operators. This language satisfies type correctness and binding-time correctness.

This approach shares a lot with the idea of *weak-separability* as presented in [22] as a safe multi-staged extension of a subset of Java, called **Mint-Java**. The main difference is that weak-separability is a new way to prevent scope extrusion rather than focusing on effects themselves. This new restriction made it possible to prove various good properties on this extension of Java, such as type safety.

1.2.4 Summary

Let us give a quick comparison of these various approaches. One should remember that being hygienic is **not** always a desired feature, lisp's quotation system was not and this aspect was broadly used. It is however, with CSP, a very important property to understand how expressive a given formalism is.

This table is not exhaustive but gives a good idea of the dense and inhomogeneous fauna of multi-staged programming formalisms.

Only the most interesting properties are included and the absence of a ✓ does not necessarily mean that the property does not hold, but often it means that it was not mentioned in the original paper. On the other hand a ✗ means that the property does not hold and was disproved in the paper.

	Name	Motivation	Ch.-Rosser	Subj.-Red.	Str. norm.	Call-by-?	Type safe	Open code	Fixpoint	Hygienic	CSP	Embeds
1	$\lambda^{\bigcirc}$	Logical						✓	✗		✗	
2	MiniML$^{\Box}$	Logical		✓	✓[3]			✗	✓	✓		
3	MetaML	Operational	✓	✓			✓	✗	✗	✓	✓	
4	AIM[4]	Operational		✓		Both	✓	✓	✗		✓	1, 2, 3
5	MiniML$^{BN}_{ref}$	Operational				CBV	✓	✓	✓			
6	MetaOCaml	Operational		✓[5]		CBV	✓[6]	✓	✗		✓	1, 2
7	$\lambda^{\varnothing}$	Operational				CBV	✓	✓	✓			6
8	Mint-Java	Operational							✗	✓	✓	
9	$\lambda^{\blacktriangleright}$ - MiniML$^{\blacktriangleright}$	Logical	✓	✓	✓	CBV		✓	✓	✗	✓[7]	
10	$\lambda^{\triangleright\%}$	Logical		✓	✓	CBV	✓[8]	✓	✓	✓	✓	9
11	λ_S	Logical				CBV		✓	✓	✗		
12	λ^{poly}_{open}	Operational				CBV	✓	✓	✓	✗	✗	

Additional informations

6. Was implemented and available (see BER+MetaOCaml)

11. Equivalence to the λ_ρ-calculus, a calculus with record operations, used for verification, see [5]

12. Type can be inferred and an algorithm is given in [11]

2 $\lambda^{\blacktriangleright}$-calculus

During my internship I mainly focused on one specific core calculus, the $\lambda^{\blacktriangleright}$-calculus and its programming counterpart MiniML$^{\blacktriangleright}$. They were introduced by [21] and are purely logically founded as they are seen as an extension of the Curry-Howard isomorphism for a Multi-Modal logic. They give a foundation for the environment classifier approach initiated in [19].

Although they started working on the Kripke semantics and the completeness of the induced logic, I did not focus on that but rather on the programming aspect of the language.

2.1 The language

I will first give a formal description of the language including its syntax and a small-step semantics, as well as a type system.

2.1.1 Syntax

Given an alphabet Σ of *transition variables* designated by lower-case greek letters $\alpha, \beta, \cdots$, a transition is a finite sequence of transition variables, which we will write with upper-case Latin letters A, B,$\cdots$.

ε is the empty sequence of variables and AB is the naïve concatenation of two transitions (a transition will also be referred to as a stage).

The set of Terms is defined as follow :

[3] no proof given
[4] An Idealized MetaML
[5] polymorphic
[6] monomorphic
[7] partly
[8] partly

$$\begin{array}{rl} \text{Terms} & M ::= x \mid \lambda x : \tau.e \mid e\ e \\ & \qquad \Lambda\alpha.M \mid \blacktriangleleft_\alpha\ M \mid \blacktriangleright_\alpha\ M \mid MA \end{array}$$

While $\blacktriangleright_\alpha$ relates to the boxing mecanism, we will see later that $\blacktriangleleft_\alpha$ is not only the unboxing but also the **run**. These terms are anotated with transition variables because of the two other new terms: one can abstract (resp. apply) transition variable (resp. transition), and the $\blacktriangleright_\alpha$ / $\blacktriangleleft_\alpha$ might have to be duplicated subsequently.

Finally $\Lambda\alpha.M$ is the abstraction of a transition variable in M while MA is the application of a transition to a transition variable.

Although my work was not relying on types for this language, it is worth noting that a strong type system was given in [21] based on the following grammar.

$$\tau ::= b \mid \tau \to \tau \mid \triangleright_\alpha \tau \mid \forall\alpha.\tau$$

where b is in a given set of base types.

The two new types are

- The *code type* written $\triangleright_\alpha \tau$ that will be given to boxed expression
- The α-*closed type* which states that α is closed inside the terms it refers to.

2.1.2 Semantics

Because not only usual variables, but also transition variables can be abstracted in the language we need to define the notion of substitution.

The substitution for usual variables being the common one, it will be omited here, and the substitution on type is the straightforward non-capturing one, thus I will only focus on $M[\alpha/A]$:

$$\begin{array}{rcl} (\lambda x : \tau.M)[\alpha/A] & = & \lambda x : \tau[\alpha/A].M[\alpha/A] \\ (MN)[\alpha/A] & = & (M[\alpha/A])(N[\alpha/A]) \\ (MB)[\alpha/A] & = & (M[\alpha/A])(B[\alpha/A]) \\ (\blacktriangleright_\beta\ M)[\alpha/A] & = & \blacktriangleright_{\beta[\alpha/A]}\ M[\alpha/A] \\ (\blacktriangleleft_\beta\ M)[\alpha/A] & = & \blacktriangleleft_{\beta[\alpha/A]}\ M[\alpha/A] \\ (\Lambda\alpha.M)[\alpha/A] & = & M \\ (\Lambda\beta.M)[\alpha/A] & = & \Lambda\beta.M[\alpha/A] \\ (\Lambda\alpha.M)[\alpha/A] & = & M \end{array}$$

It's worth noting that $\blacktriangleright_\varepsilon\ M = M$, the same goes for $\blacktriangleleft_\varepsilon$ and this is reflected in the type, of course conversely they will be duplicated for a transition A of length > 1.

The semantics is then defined as the smallest relation closed under the three following rules (and α-equivalence):

$$\begin{array}{rcl} (\lambda x : \tau.M)N & \longrightarrow & M[x/N] \\ (\Lambda\alpha.M)A & \longrightarrow & M[\alpha/A] \\ \blacktriangleleft_\alpha\ (\blacktriangleright_\alpha\ M) & \longrightarrow & M \end{array}$$

The two first rules should not be a surprise, the third one is the reason why this formalism does not need a **run** rule as it is here a special case of unboxing at stage 0 or, in this context, ε.

In the following a rewriting from a term M of stage A to a term N of stage A will sometime be written $M \xrightarrow{\mathrm{A}} N$ which means "M rewrites to N at stage A".

2.1.3 Type system

The type system is where the environment classifiers make a difference in the construction of the language.

Our contexts Γ are finite sets $\{x_1 : T_1@A_1, \cdots, x_n : T_n@A_n\}$ where the x_is are distinct variables. This means that the variable x_i is given type τ_i at stage A_i.

The set of free transition variables for a context, denoted $FTV(\Gamma)$ is defined by the union of, for all $x_i : T_i@A_i$, the transition variables composing A_i (which is denoted $FTV(A_i)$) and the free transition variables of τ_i[9].

Our judgments are then of the form $\Gamma \vdash^{\mathrm{A}} M : \tau$ meaning "Under the context Γ and at stage A, M is given the type τ" and our type system is:

$$\frac{}{\Gamma, x : \tau@A \vdash^A x : t}(\textsc{var})$$

$$\frac{\Gamma, x : \tau@A \vdash^A M : \sigma}{\Gamma \vdash^A \lambda x : \tau.M : \tau \to \sigma}(\textsc{abs}) \qquad \frac{\Gamma \vdash^A M : \tau \to \sigma \qquad \Gamma \vdash^A N : \tau}{\Gamma \vdash^A MN : \sigma}(\textsc{app})$$

$$\frac{\Gamma \vdash^{A\alpha} M : \tau}{\Gamma \vdash^A \blacktriangleright_\alpha M : \triangleright_\alpha \tau}(\blacktriangleright) \qquad \frac{\Gamma \vdash^A M : \triangleright_\alpha \tau}{\Gamma \vdash^{A\alpha} \blacktriangleleft_\alpha M : \tau}(\blacktriangleleft)$$

$$\frac{\Gamma \vdash^A M : \tau \qquad \alpha \notin FTV(\Gamma) \cup FTV(A)}{\Gamma \vdash^A \Lambda\alpha.M : \forall\alpha.\tau}(\textsc{abs}_{env}) \qquad \frac{\Gamma \vdash^A M : \forall\alpha.\tau}{\Gamma \vdash^A M\, B : \tau[\alpha/B]}(\textsc{app}_{env})$$

The first top three rules are standard, with added safeguards about the stages: (VAR) ensures that the variable is indeed declared at this stage, (APP) requires typability at the same stages under the same context and conversely (ABS) requires that the body and the argument are of the same stage.

The boxing and unboxing rules are not surprising either as they directly reflect the current stage on the type.

The abstraction of an environment variable ensures that the abstracted variable does not depend on the context, i.e. no variable in M has an α-annotation either for its type or stage.

And finally the rule ($\textsc{app}_{env}$) perform the substition inside the type of M without changing anything else.

Unlike systems without the notion of *environment classifiers*, this one has a quite fine granularity over which variable are bound at which stages. Previous major systems had to carry the full context through the typing to make sure that no open code would be evaluated; λ_S from [5] is a good example of such a way to solve this issue.

[9] this is defined by all the unbound transition variables appearing in the type

2.2 Main Properties

In this core language we have a few interesting proprieties that hold, which is why it's an interesting base for my work and a good candidate for a reference language to implement, it is indeed quite close to the one chosen as a base for MetaOCaml.

I won't prove them here (the extensive proofs are in [21]) but this section shows the most interesting properties of this language.

2.2.1 Rewriting properties

First of all, this semantics is confluent:

Theorem. *(Confluence)*
If $M \to^* N_1$ and $M \to^* N_2$ then there is an N such as $N_1 \to^* N$ and $N_2 \to^* N$.

Now if we take the types into account this language also satifies Subject Reduction and Strong Normalisation:

Theorem. *(Subject Reduction)*
If M is typable at stage A of type τ and $M \to^* N$ then

- *N is also a term of stage A*
- *At stage A, N is typable of type τ*

Theorem. *(Strong Normalisation)*
If M is typable, then there is no infinite sequence of reductions starting from M.

There's another interesting result, which is something only designed for multi-staged programming: you might want to know exactly where, in term of stages, your rewrites takes place.

A relevant property is called Time Ordered Normalisation, and it states that the when a rewrite occurs at stage at least T, the resulting term will not have any rewrite at stage lower than T, or formally:

Definition. *M is said to be T-normal if there is no $A \leq T$ and no N such that $M \xrightarrow{A} N$ where $A \leq T$ means that there is an environment $B \neq \varepsilon$ such that $AB = T$.*

Theorem. *(Time Ordered Normalisation)*
If M is typable and T-normal and $M \to^* N$ then N is T-normal.

Combining this and the Strong Normalisation, we know that for every typable term M, there is a rewriting such that:

- the stages are getting higher and higher (or more and more nested) during the rewriting
- there is a final term N that will by definition be in normal form
- the type is preserved during the rewriting

These properties make $\lambda^{\blacktriangleright}$ a good choice of language to represent the idea of multi-staged programming, and a useful language to work on.

2.2.2 No context lemma

Let's have a look at the general behaviours of this language in the untyped situation, and prove that the *context lemma* does not hold.

Lemma 1. *(context lemma)*

- *For all contexts C, for all terms t_1 and t_2 and for all $e_1, e_2, \ldots, e_n$, if $C[t_1]$ and $C[t_2]$ behave the same way (both converge or both diverge) then $t_1\, e_1\, e_2 \cdots e_n$ and $t_2\, e_1\, e_2 \ldots e_n$ behave the same way.*
- *For all contexts C, for all terms t_1 and t_2 and for all $e_1, e_2, \ldots, e_n$, if $C[t_1]$ and $C[t_2]$ behave differently then so do $t_1\, e_1\, e_2 \cdots e_n$ and $t_2\, e_1\, e_2 \ldots e_n$*

The idea here is that if we are able to differentiate two terms using contexts, then we can do it only by observing how they react when given arguments, that is with *applicative contexts*.

This is a good intuitive vision of how a purely functionnal language behaves, and we will see that this result does not hold here, leading to the idea that there is something inherently non-functionnal about multi-staged programming.

Let's take

$$t_1 = \blacktriangleright_\alpha\ (\lambda x.x)$$
$$t_2 = \blacktriangleright_\alpha\ (\lambda x.xx)$$

and

$$C[\bullet] = (\lambda x.(\blacktriangleleft_\alpha\ x)(\blacktriangleleft_\alpha\ x))\bullet$$

Then

$$\begin{array}{rcl} C[t_1] & \to & (\blacktriangleleft_\alpha\blacktriangleright_\alpha\ \lambda x.x)(\blacktriangleleft_\alpha\blacktriangleright_\alpha\ \lambda x.x) \\ & \to & (\lambda x.x)(\lambda x.x) \\ & \to & \lambda x.x \end{array}$$

$$\begin{array}{rcl} C[t_2] & \to & (\blacktriangleleft_\alpha\blacktriangleright_\alpha\ \lambda x.xx)(\blacktriangleleft_\alpha\blacktriangleright_\alpha\ \lambda x.xx) \\ & \to & (\lambda x.xx)(\blacktriangleleft_\alpha\blacktriangleright_\alpha\ \lambda x.xx) \\ & \to & (\blacktriangleleft_\alpha\blacktriangleright_\alpha\ \lambda x.xx)(\blacktriangleleft_\alpha\blacktriangleright_\alpha\ \lambda x.xx) \\ & \to & \cdots \end{array}$$

Hence $C[t_2]\uparrow$ but $\neg(C[t_1]\uparrow)$.

Yet both t_1 and t_2 are values and can't take any argument, thus behave the same in any *applicative context* i.e. context that could just feed them with arguments. Thus we are able to distinguish two terms using contexts but not using applicative contexts.

Even though the type system prevents a term like t_2 from occuring, this result proves that the context lemma does not hold fot untyped contexts.

3 MiniML$^{\blacktriangleright}$ – Toward an unstaging translation

Being familiar with the core language I can now describe the programming language, MiniML$^{\blacktriangleright}$ as well as the results I was working on.

3.1 MiniML$^{\blacktriangleright}$

3.1.1 Syntax

Let's describe what is added to the base calculus. Compared to the core language we add a fixpoint operator, as well as branching, integers and booleans:

$$
\begin{array}{rl}
\text{Terms} & M ::= b \mid n \mid x \mid \lambda x : \tau.e \mid e\ e \mid \textbf{fix}\ f : \tau \to \sigma.M \mid \\
& \quad \textbf{if}\ M\ \textbf{then}\ M\ \textbf{else}\ M \mid M \diamond M \mid M = M \mid \\
& \quad \Lambda\alpha.M \mid \blacktriangleleft_\alpha\ M \mid \blacktriangleright_\alpha\ M \\
\text{Values} & v^\varepsilon ::= n \mid \textbf{true} \mid \textbf{false} \mid \lambda x : \tau.M \mid \blacktriangleright_\alpha\ v^\alpha \mid \Lambda\alpha.v^\varepsilon \\
(A \neq \varepsilon) & v^A ::= n \mid \textbf{true} \mid \textbf{false} \mid x \mid \lambda x : \tau.v^A \mid \textbf{fix}\ f : \tau \to \sigma.v^A \mid \\
& \quad v^A v^A \mid \blacktriangleright_\alpha\ v^{A\alpha} \mid \Lambda\alpha.v^A \mid v^A B \mid \\
& \quad \blacktriangleleft_\alpha\ v^{A'} (\text{with } A'\alpha = A \text{ and } A' \neq \varepsilon)
\end{array}
$$

Where $\diamond$ ranges in some usual operations for integers: $\{+, -, \times\}$.

Remark : this is reflected in the type system to take into account booleans and integers but I will not describe that any further as I will not use the types in my work.

3.1.2 Semantics

The language has a call-by-value semantics and the reductions depend on the current stage, $M \xrightarrow{\mathrm{A}} M'$ is to be read "M rewrites to M' at stage A"

The substitution is defined as a straightforward extension of the one that was defined for the core language.

The original language and its semantics includes an "error" value, yet although it will not be described any further here it's worth noting that it also gives a proof of type soundness for this semantics such that the adage "Well typed program do not go wrong" holds, which is why I decided not to include the error cases in my version.

The proof of the equivalence between the small-step semantics presented here and the big-step semantics from the original paper is given in annex A.

Given the explanations on the semantics of the core version, none of the rule here should be really surprising; the computing rules can only happen at stage ε, and the only relevant rules that can happen at higher stages are linked with the boxing/unboxing.

$$\frac{}{\textbf{if true then } M \textbf{ else } M' \xrightarrow{\varepsilon} M}(if_\top) \qquad \frac{}{\textbf{if false then } M \textbf{ else } M' \xrightarrow{\varepsilon} M'}(if_\bot)$$

$$\frac{m = n}{m = n \xrightarrow{\varepsilon} \textbf{true}}(eq_\top) \qquad \frac{m \neq n}{m = n \xrightarrow{\varepsilon} \textbf{false}}(eq_\bot)$$

$$\frac{m \diamond m' = n \qquad \diamond \in \{+, -, \times\}}{m \diamond m' \xrightarrow{\varepsilon} n}(\diamond_{comp})$$

$$\frac{M \xrightarrow{\mathrm{A}} M' \qquad \diamond \in \{+, -, \times, =\}}{M \diamond N \xrightarrow{\mathrm{A}} M' \diamond N}(\diamond_{left}) \qquad \frac{M \xrightarrow{\mathrm{A}} M' \qquad \diamond \in \{+, -, \times, =\}}{N \diamond M \xrightarrow{\mathrm{A}} N \diamond M'}(\diamond_{right})$$

$$\frac{M \xrightarrow{\mathrm{A}} M'}{\textbf{if } M \textbf{ then } N \textbf{ else } N' \xrightarrow{\mathrm{A}} \textbf{if } M' \textbf{ then } N \textbf{ else } N'}(if_{if})$$

$$\frac{M \xrightarrow{\mathrm{A} \neq \varepsilon} M'}{\textbf{if } N \textbf{ then } M \textbf{ else } N' \xrightarrow{\mathrm{A}} \textbf{if } N \textbf{ then } M' \textbf{ else } N'}(if_{then})$$

$$\frac{M \xrightarrow{\mathrm{A} \neq \varepsilon} M'}{\textbf{if } N \textbf{ then } N' \textbf{ else } M \xrightarrow{\mathrm{A}} \textbf{if } N \textbf{ then } N' \textbf{ else } M'}(if_{else})$$

$$\frac{M \xrightarrow{\mathrm{A}} M'}{MN \xrightarrow{\mathrm{A}} M'N}(APP_{fun}) \qquad \frac{N \xrightarrow{\mathrm{A}} N'}{VN \xrightarrow{\mathrm{A}} VN'}(APP_{arg})$$

$$\frac{M \xrightarrow{\mathrm{A}} M'}{\textbf{fix } f : (\tau \to \sigma).M \xrightarrow{\mathrm{A}} \textbf{fix } f : (\tau \to \sigma).M'}(APP_{fix}) \qquad \frac{}{\textbf{fix } f : (\tau \to \sigma).M \xrightarrow{\varepsilon} M[f/\textbf{fix } f : (\tau \to \sigma).M]}(EVAL_{fix})$$

$$\frac{M \xrightarrow{\mathrm{A}} M'}{\lambda x : \tau.M \xrightarrow{\mathrm{A}} \lambda x : \tau.M'}(ABS) \qquad \frac{}{(\lambda x : \tau.M)V \xrightarrow{\varepsilon} M[x/V]}(EVAL)$$

$$\frac{M \xrightarrow{\mathrm{A}} M'}{\Lambda\alpha.M \xrightarrow{\mathrm{A}} \Lambda\alpha.M'}(ABS_{env}) \qquad \frac{}{(\Lambda\alpha.M)A \xrightarrow{\varepsilon} M[\alpha = A]}(EVAL_{env})$$

$$\frac{M \xrightarrow{\mathrm{A}} M'}{MB \xrightarrow{\mathrm{A}} M'B}(APP_{env})$$

$$\frac{M \xrightarrow{\mathrm{A}} M'}{\blacktriangleleft_\alpha M \xrightarrow{\mathrm{A}\alpha} \blacktriangleleft_\alpha M'}(BOX) \qquad \frac{M \xrightarrow{\mathrm{A}\alpha} M'}{\blacktriangleright_\alpha M \xrightarrow{\mathrm{A}} \blacktriangleright_\alpha M'}(UNBOX)$$

$$\frac{}{\blacktriangleleft_\alpha \blacktriangleright_\alpha M \xrightarrow{\alpha} M}(RUN)$$

3.1.3 An example of programming in MiniML$^{\blacktriangleright}$

We will write the already described power function in our new language MiniML$^{\blacktriangleright}$ to see how transition variables work together.

$$\begin{aligned} power_{abs} &= \Lambda\alpha.\mathbf{fix}\ f.\lambda n.\lambda x. \\ &\quad \mathbf{if}\ n = 0\ \mathbf{then}\ (\blacktriangleright_\alpha\ 1) \\ &\quad\ \mathbf{else}\ \ \blacktriangleright_\alpha\ ((\blacktriangleleft_\alpha\ x) * (\blacktriangleleft\ f(n-1)x)) \end{aligned}$$

$$power = \lambda n.\Lambda\beta.\ \blacktriangleright_\beta\ (\lambda x.\ \blacktriangleleft_\beta\ (power_{abs}\ \beta\, n\, (\blacktriangleright_\beta\ x)))$$

The first function is here to *manipulate* some code: it takes an int n and a piece of code that returns an int, and returns a piece of code that returns an int. The underlying idea is that it will concatenate the pice of code given n times with the symbol $\times$.

The second function is designed to *create* the code itself : it takes an int and output a piece of code of type int $\to$ int. Indeed if we call it with with 3 and α as argument then it rewrites to a piece of code that takes x and compute x^3: $\blacktriangleright_\alpha\ (\lambda x : int.(x \times x \times x \times 1))$

That means that as soon as we know n we can precompute the adapted function $power\, n$, and once this is computed we can use it at any stage, including at "top-level", by giving it as 2^{nd} argument the corresponding transition. Hence for example at "top-level" we can give it ε as we are not currently in any stage and get $power\, n\, \varepsilon = \lambda x : int.(x \times x \times \cdots \times x \times 1)$ which is exactly the expected output.

3.2 The unstaging translation

3.2.1 The base language

Our base language is a subset of MiniML$^{\blacktriangleright}$ that lacks operations on integers and the if statement, while keeping the fixpoint operator.

$$\begin{aligned} \text{Terms}\quad & M ::= i \mid x \mid \lambda x : \tau.e \mid e\ e \mid \mathbf{fix}\ f : \tau \to \sigma.M \mid \\ & \qquad \Lambda\alpha.M \mid \blacktriangleleft_\alpha\ M \mid \blacktriangleright_\alpha\ M \\ \text{Values}\quad & v^\varepsilon ::= i \mid \lambda x : \tau.M \mid \blacktriangleright_\alpha\ v^\alpha \mid \Lambda\alpha.v^\varepsilon \\ (A \neq \varepsilon)\quad & v^A ::= i \mid x \mid \lambda x : \tau.v^A \mid \mathbf{fix}\ f : \tau \to \sigma.v^A \mid \\ & \qquad v^A v^A \mid \blacktriangleright_\alpha\ v^{A\alpha} \mid \Lambda\alpha.v^A \mid v^A B \mid \\ & \qquad \blacktriangleleft_\alpha\ v^{A'} (\text{with } A'\alpha = A \text{ and } A' \neq \varepsilon) \end{aligned}$$

The reason for that is that I'm using an already existing language as target language and while it would be easy to add to this language some features my goal is to use some properties of this language to have an even broader result.

For the same reason the integers were also replaced by a set of constants denoted i to help with the translation.

Its semantics is, as expected, the straightforward restriction of the full semantics to the appropriate terms.

3.2.2 The destination language

Our destination language is Choi et al.'s λ_ρ as described in [5], that will be described again here.

This language was designed as a target language for an unstaged translation for a language that did not have environment classifiers.

$$\begin{array}{rl} \text{Terms} & M ::= i \mid x \mid \lambda x : \tau.M \mid M\ M \mid \mathbf{fix}\ f : \tau \to \sigma.M \mid \\ & \quad r \mid r \cdot \mathbf{x} \mid \mathbf{let}\ x = M\ \mathbf{in}\ M \\ & r ::= \{\} \mid \rho \mid r + \{\mathbf{x} = x\} \mid r + \{\mathbf{x} = v\} \\ \text{Values} & v ::= i \mid \lambda x.M \mid \mathbf{fix}\ f.M \mid v_r \\ & v_r ::= \{\} \mid v_r + \{\mathbf{x} = v\} \end{array}$$

Its small-step semantics is given bellow. It is close to the standard λ-calculus with a fixpoint operator, with the addition of records as base terms.

Note that while $r \cdot \mathbf{x}$ *is* a term of the language, the result of the rule $(LOOK_{\mathcal{R}})$ never has this form, but instead is actually replaced by its value.

Records can either be a variable ρ, or defined recursively by the empty record $\{\}$ and $r + \{\mathbf{x} = e\}$.

$$\frac{M \xrightarrow{\mathcal{R}} M'}{MN \xrightarrow{\mathcal{R}} M'N}(APP_{\mathcal{R}}) \qquad \frac{N \xrightarrow{\mathcal{R}} N'}{VN \xrightarrow{\mathcal{R}} VN'}(APP_{\mathcal{R}})$$

$$\frac{}{(\lambda x.M)V \xrightarrow{\mathcal{R}} M[x/V]}(EVAL_{\mathcal{R}})$$

$$\frac{}{(\mathbf{fix}\ f.M) \xrightarrow{\mathcal{R}} M[f/\mathbf{fix}\ f.M]}(EVAL_{\mathcal{R}})$$

$$\frac{M \xrightarrow{\mathcal{R}} M'}{\mathbf{let}\ x = M\ \mathbf{in}\ N \xrightarrow{\mathcal{R}} \mathbf{let}\ x = M'\ \mathbf{in}\ N}(LET_{\mathcal{R}})$$

$$\frac{}{\mathbf{let}\ x = V\ \mathbf{in}\ N \xrightarrow{\mathcal{R}} N[x/V]}(LEVAL_{\mathcal{R}})$$

$$\frac{}{v_r \cdot \mathbf{x} \xrightarrow{\mathcal{R}} v_r(\mathbf{x})}(LOOK_{\mathcal{R}})$$

Let $v_r = v'_r + \{\mathbf{y} = e\}$, if $\mathbf{x} = \mathbf{y}$ and $e = v$ then $v_r(\mathbf{x}) = v$ else $v_r(\mathbf{x}) = v'_r(\mathbf{x})$

Figure 1: Small-step semantics for λ_ρ

While the **let** construction will not be used by our translation — it was only useful for the backward translation in the original paper — I will keep it as I want to keep the original language.

3.2.3 The translation

The goal of this section is to show how staged code can be represented in the record-calculus.

One should note that a boxed expression is frozen, as is an abstraction, therefore the boxing will somehow be represented by functions.

Now an unboxed expression may suddenly capture variables that were free, therefore our translation should make sure that these variables are closed after unboxing.

With that in mind one should see that a boxing will be relevantly represented by the abstraction of a record, and that this record will be given at unboxing time to provide a meaning to the variables. Therefore bindings should update these envorinments to keep track of variables.

A simple example will be given here, without environment classifiers first:

$$\blacktriangleright ((\lambda x. \blacktriangleleft (\blacktriangleright x))i)$$

This code could be represented by the follwing λ_ρ term:

$$\lambda\rho.((\lambda\rho'.\rho' \cdot x)\{\mathbf{x} = i\})$$

But $\blacktriangleleft M$ inside a stage will be evaluated to a block of code, therefore we want to move the unstaged expression outside the staging, and this is done by using contexts: an $\blacktriangleleft M$ will be given the form $(\lambda h.[\bullet])M'$, and as the semantics is call-by-value the argument is evaluated before the substitution and the order is respected.

Therefore we want the translation to look like:

$$(\lambda h.\lambda\rho.h\,(\rho + \{\mathbf{x} = i\}))(\lambda\rho'.\rho' \cdot x)$$

A translation of the form $R \vdash M \mapsto (M', K)$ has the meaning "the expression M of $\lambda^{\blacktriangleright}$ under the environment stack R translates to an expression M' of λ_ρ with the tree of labelled contexts K".

The environment stack is designed to keep track of the bound variables at a given stage. On the other hand the context tree is a way to keep track of what was described before for the unstaging.

The difference between this translation and the one presented in the Choi et al. paper lies in the *environment classifier* and the way they manage free variables. An intuitive idea behind the modification is that rather than having a linear staging/unstaging we can have branchings, hence we need a tree of context rather than a stack.

A consequent major difference is that the size of a $\blacktriangleright_\alpha M$ can suddenly change if α is bound to a stack, possibly empty, of transition variables.

This also means that our translation needs to be α-equivalent for the transition variable too as $(\Lambda\alpha.M)\beta$ is a valid term, therefore even though the resulting term does not keep track of the name of the environment we need to take care of that, and we will do it at translation-time.

$$\frac{}{R \vdash i \mapsto (i, \bot)}\ (CST_{\mapsto})$$

$$\frac{}{R, r \vdash x \mapsto (r(\mathrm{x}), \bot)}\ (VAR_{\mapsto})$$

$$\frac{R, r + \{\mathrm{x} = x\} \vdash M \mapsto (M', K)}{R, r \vdash \lambda x : \tau . M \mapsto \left(\lambda x.M', \begin{array}{c}\bot \\ | \\ K\end{array}\right)}\ (ABS_{\mapsto})$$

$$\frac{R, r + \{\mathrm{f} = f\} \vdash M \mapsto (M', K)}{R, r \vdash \mathbf{fix}\ f : \tau \to \sigma . M \mapsto \left(\mathbf{fix}\ f.M', \begin{array}{c}\bot \\ | \\ K\end{array}\right)}\ (FIX_{\mapsto})$$

$$\frac{R \vdash M \mapsto (M', K) \qquad R \vdash N \mapsto (N', K')}{R \vdash MN \mapsto \left(M'N', \begin{array}{c}\bot \\ \diagup\ \diagdown \\ K \quad K'\end{array}\right)}\ (APP_{\mapsto})$$

$$\frac{R \vdash M \mapsto (M', K)}{R \vdash \Lambda\alpha.M \mapsto \left(M', \begin{array}{c}\alpha \\ | \\ K\end{array}\right)}\ (ENV_{\mapsto})$$

$$\frac{R, r, \rho \vdash M \mapsto (M', K)}{R, r \vdash \blacktriangleright_{\alpha} M \mapsto \left(\lambda\rho.M', \begin{array}{c}\blacktriangleright_{\alpha}, r \\ | \\ update_1(r, K)\end{array}\right)}\ (BOX_{\mapsto})$$

$$\frac{R \vdash M \mapsto \left(M', \begin{array}{c}\blacktriangleright_{\alpha}, r \\ | \\ K\end{array}\right)}{R, r \vdash \blacktriangleleft_{\alpha} M \mapsto (M'\, r, K)}\ (UBOX_{\mapsto})$$

$$\frac{R \vdash M \mapsto (M', K)}{R, r \vdash \blacktriangleleft_{\alpha} M \mapsto \left(hr, \begin{array}{c}\blacktriangleleft_{\alpha} = (\lambda h.[\bullet])M' \\ | \\ update_1(r, K)\end{array}\right)}\ (UBOX'_{\mapsto})$$

$$\frac{R \vdash M \mapsto (M', K)}{R \vdash M\beta \mapsto (M', update_2(\beta, K))}\ (APP''_{\mapsto})$$

$$\frac{R \vdash M \mapsto (M', K)}{R \vdash M\varepsilon \mapsto (M', update_3(K))}\ (APP''_{\mapsto})$$

Figure 2: Translation from $\lambda^{\blacktriangleright}$ to λ_{ρ}

$$update_1(\rho, K) = \begin{cases} - & \begin{array}{c} \blacktriangleright, (\rho \cdot r) \\ | \\ update_1(\rho, K') \end{array} \text{ if } K = \begin{array}{c} \blacktriangleright, r \\ | \\ K' \end{array} \\ - & \text{Simple recursive call on the sub-trees otherwise.} \end{cases}$$

$$update_2(\beta, K) = \begin{cases} - & \begin{array}{c} \bot \\ | \\ update_2(\beta, K') \end{array} \text{ if } K = \begin{array}{c} \bot \\ | \\ K' \end{array} \text{ and } \begin{array}{ccc} & \bot & \\ \diagup & & \diagdown \\ update_2(\beta, K') & & K'' \end{array} \text{ if } K = \begin{array}{ccc} & \bot & \\ \diagup & & \diagdown \\ K' & & K'' \end{array} \\ - & collapse\,(update'_2(\alpha, \beta, K')) \text{ if } K = \begin{array}{c} \alpha \\ | \\ K' \end{array} \end{cases}$$

$$update'_2(\alpha, \beta, K) = \begin{cases} - & \begin{array}{c} \blacktriangleright_\beta, r \\ | \\ translation'_2(\alpha, \beta, K') \end{array} \text{ if } K = \begin{array}{c} \blacktriangleright_\alpha, r \\ | \\ K' \end{array} \\ - & \begin{array}{c} \blacktriangleleft_\beta = \kappa \\ | \\ translation'_2(\alpha, \beta, K') \end{array} \text{ if } K = \begin{array}{c} \blacktriangleleft_\alpha = \kappa \\ | \\ K' \end{array} \\ - & K \text{ if } K = \begin{array}{c} \alpha \\ | \\ K' \end{array} \\ - & \text{The application of } update'_2 \text{ to the subtree in any other case.} \end{cases}$$

collapse searches through the tree for structures like $\begin{pmatrix} K \\ | \\ \blacktriangleleft_\alpha = \kappa \\ | \\ \blacktriangleright_\alpha, r \\ | \\ K' \end{pmatrix}$

in which case the tree is collapsed to $\begin{pmatrix} K \\ | \\ K' \end{pmatrix}$

and the current term is inserted in κ's context hole.

$$update_3(K) = \begin{cases} - & \begin{array}{c} \bot \\ | \\ update_3(K') \end{array} \text{ if } K = \begin{array}{c} \bot \\ | \\ K' \end{array} \text{ and } \begin{array}{ccc} & \bot & \\ \diagup & & \diagdown \\ update_3(K') & & K'' \end{array} \text{ if } K = \begin{array}{ccc} & \bot & \\ \diagup & & \diagdown \\ K' & & K'' \end{array} \\ - & update'_3(\alpha, K') = \text{if } K = \begin{array}{c} \alpha \\ | \\ K' \end{array} \end{cases}$$

$$update'_3(\alpha, K) = \begin{cases} - & update'_3(\alpha, K') \text{ if } K = \begin{array}{c} \blacktriangleleft_\alpha = \kappa \\ | \\ K' \end{array} \text{ and the } \textit{current} \text{ term } M \text{ is changed to } \kappa[(\lambda h.M)(\lambda r.h)] \\ - & update'_3(\alpha, K') \text{ if } K = \begin{array}{c} (\blacktriangleright_\alpha, r) \\ | \\ K' \end{array} \text{ and the } \textit{corresponding} \text{ term } M \text{ is changed to } M\,r \\ - & K \text{ if } K = \begin{array}{c} \alpha \\ | \\ K' \end{array} \\ - & \text{The application of } update'_3 \text{ on the subtree in any other cases.} \end{cases}$$

Figure 3: Definition of required function for the translation.

Definition 1. *(Context Closure)*
If T is a tree of contexts and M is a term, then $T(M)$ is defined as as follow:

$$T(M) = \begin{cases} T'(\kappa[M]) \text{ if } T = \begin{array}{c} \blacktriangleleft = \kappa \\ | \\ T' \end{array} \\ T'(M) \text{ if } T = \begin{array}{c} T'' \\ | \\ T' \end{array} \text{ or } T = \begin{array}{c} T'' \\ \diagup\diagdown \\ T' \quad T''' \end{array} \\ M \text{ if } T = \bot \end{cases}$$

Theorem 1. *(Translation is semantic-preserving)*
If

- *M is a term of $\lambda^{\blacktriangleright}$*
- *$M \xrightarrow{A}{}^{*} N$*
- *$R \vdash M \mapsto (M', K)$*
- *$R' \vdash N \mapsto (N', K')$*

Then $K(M') \xrightarrow{\mathcal{R}}{}^{*} K'(N')$ up to α-renaming.

Remark : the converse is not true as some naming information is lost after the translation is done, when we fill the contexts – see $(\lambda x.\ \blacktriangleleft_\beta\ x)(\blacktriangleright_\alpha\ i)$.

3.3 Proof

Because this translation is not the one I originally wrote I don't have yet the proof of the major theorem presented here. And although most cases are just straightforward induction, I've had surprises with my previous translation system when it came to increasing or decreasing the length of an abstracted transition

To prevent these problem from happening this translation is doing some computation at translation-time, and the proof becomes substantially more complex.

But I had it working on all the example I could find, and a more thorough example of how the full translation works can be found in annex C.

4 Conclusion

This translation is the first step toward a reunification of the two current dominant ways of formalising multi-staged programming.

Indeed, our destination language has a back-and-forth translation to a multi-staged language that does not use *environment classifiers*, and this multi-staged language is itself embedded naively in MiniML$^{\blacktriangleright}$.

This means that we have a back and forth translation that preserves semantic as described in the main theorem between two multi-staged languages, using different approaches. And while the environment classifiers one has strong theoretical results such as an underlying Kripke semantics and logical completeness, the usual one already have tools for verifications now.

The next step is, using this translation more more importantly the underlying equivalence result, to export the most interesting result from one field to the other and vice-versa, and as

it would seem that environment classifiers were adopted as the core of MetaOCaml a practical result could be to write a 0CFA tool for MetaOCaml code.

References

[1] T B Aktemur. *Improving efficiency and safety of Program Generation.* PhD thesis, 2009.

[2] C. Calcagno, E. Moggi, and W. Taha. Closed Types as a Simple Approach to Safe Imperative Multi-Stage Programming. *Automata, Languages and Programming*, pages 25–36, 2000.

[3] C. Calcagno and E. Moggi1. Multi-Stage Imperative Languages: A Conservative Extension Result. *Semantics, Applications, and Implementation of Program Generation*, pages 92–107, 2000.

[4] Jacques Carette and Oleg Kiselyov. Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code. *Science of Computer Programming*, 76(5):349–375, 2011.

[5] Wontae Choi, Baris Aktemur, Kwangkeun Yi, and Makoto Tatsuta. Static analysis of multi-staged programs via unstaging translation. *ACM SIGPLAN Notices*, 46:81, 2011.

[6] Rowan Davies. A temporal-logic approach to binding-time analysis. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, USA, July 27-30, 1996*, pages 184–195, 1996.

[7] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. *Journal of the ACM*, 48(3):555–604, 2001.

[8] Yuichiro Hanada and Atsushi Igarashi. On cross-stage persistence in multi-stage programming. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 8475 LNCS:103–118, 2014.

[9] Jun Inoue and Walid Taha. Reasoning about multi-stage programs. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7211 LNCS:357–376, 2012.

[10] Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-Chieh Shan. Shifting the stage. *Journal of Functional Programming*, 21(06):617–662, 2011.

[11] Ik-Soon Kim, Kwangkeun Yi, and Cristiano Calcagno. A polymorphic modal type system for lisp-like multi-staged languages. *ACM SIGPLAN Notices*, 41(1):257–268, 2006.

[12] Martin Lester, Luke Ong, and Max Schaefer. Information flow analysis for a dynamically typed language with staged metaprogramming. *Proceedings of the Computer Security Foundations Workshop*, pages 209–223, 2013.

[13] Eugenio Moggi, Walid Taha, Zine-El-Abidine Benaissa, and Tim Sheard. An Idealized MetaML: Simpler, and More Expressive. In *Proceedings of theEuropean Symposium on Programming*, pages 193–207, 1999.

[14] Tim Sheard. Accomplishments and Research Challenges in Meta-programming. In *Proceedings of Semantics, Applications, and Implementation of Program Generation, Second International Workshop, SAIG 2001*, pages 2–44, 2001.

[15] Kedar Swadi, Walid Taha, Oleg Kiselyov, and Emir Pasalic. A monadic approach for avoiding code duplication when staging memoized functions. *Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation - PEPM '06*, page 160, 2006.

[16] Walid Taha. A gentle introduction to multi-stage programming. In *Domain-specific Program Generation, LNCS*, pages 30–50. Springer-Verlag, 2004.

[17] Walid Taha. A gentle introduction to multi-stage programming, part II. *Lecture Notes in Computer Science*, 5235 LNCS(Part II):260–290, 2008.

[18] Walid Taha, Zine-el-abidine Benaissa, and Tim Sheard. Multi-Stage Programming: Axiomatization and Type Safety. pages 1–12, 1998.

[19] Walid Taha and Michael Florentin Nielsen. Environment classifiers. *ACM SIGPLAN Notices*, 38:26–37, 2003.

[20] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. *ACM SIGPLAN Notices*, 32:203–217, 1997.

[21] Takeshi Tsukada and Atsushi Igarashi. A logical foundation for environment classifiers. *Logical Methods in Computer Science*, 6(4):1–43, 2010.

[22] Edwin Westbrook, Mathias Ricken, Jun Inoue, Yilong Yao, Tamer Abdelatif, and Walid Taha. Mint : Java Multi-stage Programming Using Weak Separability. *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, (Section 6):400–411, 2010.

Annex

A Equivalence between the small-step and the big-step semantics.

The reference big-step semantics for this language is given in [21], here will be given a proof of the equivalence between their semantics and the small-step one given in this paper.

Theorem If $\vdash^{\mathrm{A}} M \Downarrow M'$ then $M \xrightarrow{\mathrm{A}}{}^{*} M'$.

Proof By induction on the big-step rules. Let's show that every rule can be successfully evaluated using the small-step semantics presented here.

Only a few interesting cases will be presented here.

Handling operations

$$\frac{\vdash^{\varepsilon} M \Downarrow m \qquad \vdash^{\varepsilon} N \Downarrow n \qquad m \diamond n = k}{\vdash^{\varepsilon} M \diamond N \Downarrow k}$$

By induction

$$M \xrightarrow{\varepsilon}{}^{*} m \quad \text{and} \quad N \xrightarrow{\varepsilon}{}^{*} n$$

hence using several times the rules $(\diamond_{left})$ and $(\diamond_{right})$ with $A = \varepsilon$ yields

$$M \diamond N \xrightarrow{\varepsilon}{}^{*} m \diamond n$$

Now using the hypothesis that $m \diamond n = k$, the rule $(\diamond_{comp})$ gives $m \diamond n \xrightarrow{\varepsilon} k$, and finally

$$M \diamond N \xrightarrow{\varepsilon}{}^{*} k$$

Call-by-value

$$\frac{\vdash^{\varepsilon} M \Downarrow (\lambda x : \tau.M') \qquad \vdash^{\varepsilon} N \Downarrow V \qquad \vdash^{\varepsilon} M'[x/V] \Downarrow R}{\vdash^{\varepsilon} MN \Downarrow R}$$

By induction,

$$M \xrightarrow{\varepsilon}{}^{*} \lambda x : \tau.M'$$

$$N \xrightarrow{\varepsilon}{}^{*} V$$

$$M[x/V] \xrightarrow{\varepsilon}{}^{*} R$$

As stated by *theorem 4.3*, $(\lambda x : \tau.M')$ is a value. Therefore, applying several time (APP_{fun}) and then several time (APP_{arg}) gives us $MN \xrightarrow{\varepsilon}{}^{*} (\lambda x : \tau.M')V$.

Thus $(EVAL)$ can be applied, and $MN \xrightarrow{\varepsilon}{}^{*} M'[x/V]$, and by induction $MN \xrightarrow{\varepsilon}{}^{*} R$

Staging

$$\frac{\vdash^{\mathrm{A}\alpha} M \Downarrow M'}{\vdash^{\mathrm{A}} \blacktriangleright_\alpha \Downarrow \blacktriangleright_\alpha M'}$$

By induction,

$$M \xrightarrow{\mathrm{A}\alpha}{}^{*} M'$$

hence applying several times the rule (BOX) immediately gives the expected result:

$$\blacktriangleright_\alpha M \xrightarrow{\mathrm{A}}{}^{*} \blacktriangleright_\alpha M'$$

Theorem If $M \xrightarrow{\mathrm{A}}{}^{*} R$ and R is a value, i.e. can not be rewritten any further, then $\vdash^{\mathrm{A}} M \Downarrow R$.

Proof Let's prove this by induction on M.

- if M is a value of stage A, then $M = R$ and $\vdash^{\mathrm{A}} M \Downarrow R$
- Case study over the structure of M:
  - M is of the form **if** M' **then** M'' **else** M'''

    If $A = \varepsilon$ we can successfully evaluate the **if** and take either one branch or the other in the small-step semantics.

    The small step gives $M' \xrightarrow{\mathrm{A}}{}^{*} B$, $M'' \xrightarrow{\mathrm{A}}{}^{*} R''$ and $M''' \xrightarrow{\mathrm{A}}{}^{*} R'''$, and the induction rule yields:

$$\begin{array}{ccc} \vdots\,\Delta'_\varepsilon & \vdots\,\Delta''_\varepsilon & \vdots\,\Delta'''_\varepsilon \\ \vdash^{\varepsilon} M' \Downarrow B & \vdash^{\varepsilon} M'' \Downarrow R'' & \vdash^{\varepsilon} M''' \Downarrow R''' \end{array}$$

    If $B = \mathbf{true}$ then $R = R''$ and

$$\frac{\begin{array}{cc} \vdots\,\Delta'_\varepsilon & \vdots\,\Delta''_\varepsilon \\ \vdash^{\varepsilon} M' \Downarrow \mathbf{true} & \vdash^{\varepsilon} M'' \Downarrow R \end{array}}{\vdash^{\varepsilon} \mathbf{if}\ M'\ \mathbf{then}\ M''\ \mathbf{else}\ M''' \Downarrow R}$$

    The same holds for $B = \mathbf{false}$ with $R = R'''$

    If $A \neq \varepsilon$ there is no rule in either semantics to evaluate the if, hence R itself is of the form **if** $\bullet$ **then** $\bullet$ **else** $\bullet$ The induction rule yields:

$$\begin{array}{ccc} \vdots\,\Delta'_{\mathrm{A}} & \vdots\,\Delta''_{\mathrm{A}} & \vdots\,\Delta'''_{\mathrm{A}} \\ \vdash^{\mathrm{A}} M' \Downarrow R' & \vdash^{\mathrm{A}} M'' \Downarrow R'' & \vdash^{\mathrm{A}} M''' \Downarrow R''' \end{array}$$

    Then the following holds in the big step semantics:

$$\frac{\begin{array}{ccc}\vdots\,\Delta'_{\mathrm{A}} & \vdots\,\Delta''_{\mathrm{A}} & \vdots\,\Delta'''_{\mathrm{A}}\\ \vdash^{\mathrm{A}} M'\Downarrow R' & \vdash^{\mathrm{A}} M''\Downarrow R'' & \vdash^{\mathrm{A}} M'''\Downarrow R'''\end{array}}{\vdash^{\mathrm{A}} \textbf{if } M' \textbf{ then } M'' \textbf{ else } M''' \Downarrow \textbf{if } R' \textbf{ then } R'' \textbf{ else } R'''}$$

– M is of the form $M' = M''$
If $A = \varepsilon$, the small step semantics can evaluate M' and M'' and then the equality $M' \xrightarrow{\varepsilon}{}^{*} m'$ and $M'' \xrightarrow{\varepsilon}{}^{*} m''$ and finally $m' = m'' \xrightarrow{\varepsilon} b$ where $b \in \textbf{true}, \textbf{false}$
The induction gives :

$$\begin{array}{cc}\vdots\,\Delta'_{\varepsilon} & \vdots\,\Delta''_{\varepsilon}\\ \vdash^{\varepsilon} M'\Downarrow m' & \vdash^{\varepsilon} M''\Downarrow m''\end{array}$$

And we have the last rule in the big-step semantic to conclude:

$$\frac{\begin{array}{ccc}\vdots\,\Delta'_{\varepsilon} & \vdots\,\Delta''_{\varepsilon} & \\ \vdash^{\varepsilon} M'\Downarrow m' & \vdash^{\varepsilon} M''\Downarrow m'' & m' = (resp. \neq) m''\end{array}}{\vdash^{\varepsilon} M' = M'' \Downarrow \textbf{true}\ (resp.\ \textbf{false})}$$

If $A \neq \varepsilon$ the same induction applies with $M' \xrightarrow{\mathrm{A}}{}^{*} N'$ and $M'' \xrightarrow{\mathrm{A}}{}^{*} N''$ but the equality cannot be evaluated, hence the following derivation valid in the big-step semantics:

$$\frac{\begin{array}{cc}\vdots\,\Delta'_{\mathrm{A}} & \vdots\,\Delta''_{\mathrm{A}}\\ \vdash^{\mathrm{A}} M'\Downarrow N' & \vdash^{\mathrm{A}} M''\Downarrow N''\end{array}}{\vdash^{\varepsilon} M' = M'' \Downarrow N' = N''}$$

– M is of the form $M' \diamond M''$ where $\diamond \in \{+, \times, -\}$ behaves just as the previous case but evaluates to a natural number instead of a boolean.
– M is of the form $(\lambda x : \tau . M')M''$, because of the rules (APP_{fun}), (APP_{arg}) and (ABS), at any stage M' will be rewritten as long as possible to reaches V' a value, and then M'' will be rewritten to V''. Finally if $A = \varepsilon$, then the application is evaluated.
The induction derivations are therefore:

$$\begin{array}{cc}\vdots\,\Delta'_{\mathrm{A}} & \vdots\,\Delta''_{\mathrm{A}}\\ \vdash^{\mathrm{A}} (\lambda x : \tau . M')\Downarrow (\lambda x : \tau . V') & \vdash^{\mathrm{A}} M''\Downarrow V''\end{array}$$

If $A = \varepsilon$ it rewrites as $M \xrightarrow{\varepsilon}{}^{*} V'[x/V''] \xrightarrow{\varepsilon}{}^{*} R$, and as the second part is smaller again the induction gives us:

$$\begin{array}{c}\vdots\,\pi_{\varepsilon}\\ \vdash^{\varepsilon} V'[x/V''] \Downarrow R\end{array}$$

Putting things together gives the following complete derivation:

$$\dfrac{\begin{array}{ccc}\vdots\,\Delta'_{\varepsilon} & \vdots\,\Delta''_{\varepsilon} & \vdots\,\pi_{\varepsilon}\\ \vdash^{\varepsilon}(\lambda x:\tau.M')\Downarrow(\lambda x:\tau.V') & \vdash^{\varepsilon} M''\Downarrow V'' & \vdash^{\varepsilon} V'[x/V'']\Downarrow R\end{array}}{\vdash^{\varepsilon}(\lambda x:\tau.M')M''\Downarrow R}$$

If $A \neq \varepsilon$ then the computation is stuck in both semantics and the derivation is:

$$\dfrac{\begin{array}{cc}\vdots\,\Delta'_{\mathrm{A}} & \vdots\,\Delta''_{\mathrm{A}}\\ \vdash^{\mathrm{A}}(\lambda x:\tau.M')\Downarrow(\lambda x:\tau.V') & \vdash^{\mathrm{A}} M''\Downarrow V''\end{array}}{\vdash^{\mathrm{A}}(\lambda x:\tau.M')M''\Downarrow(\lambda x:\tau.V')V''}$$

- Both other cases of evaluation, $(EVAL_{fix})$ and $(EVAL_{env})$ are similar to the previous one.
- M is of the form $\blacktriangleleft_\alpha \blacktriangleright_\alpha N$
  If $N \xrightarrow{\alpha}{}^{*} R$ then $M \xrightarrow{\alpha}{}^{*} R$ because of rule (RUN). In this case $A = \alpha$ and the induction yields:

$$\begin{array}{c}\vdots\,\Delta_{\alpha}\\ \vdash^{\alpha} N\Downarrow N'\end{array}$$

  Hence the following derivation:

$$\dfrac{\dfrac{\begin{array}{c}\vdots\,\Delta_{\alpha}\\ \vdash^{\alpha} N\Downarrow R\end{array}}{\vdash^{\varepsilon}\blacktriangleright_\alpha N\Downarrow\blacktriangleright_\alpha R}}{\vdash^{\alpha}\blacktriangleleft_\alpha\blacktriangleright_\alpha N\Downarrow R}$$

  If $A \neq \varepsilon$ then $M \xrightarrow{A\alpha}{}^{*}\blacktriangleleft_\alpha\blacktriangleright_\alpha R$ and with the same derivation rule gives:

$$\dfrac{\dfrac{\begin{array}{c}\vdots\,\Delta_{\mathrm{A}}\\ \vdash^{\mathrm{A}\alpha} N\Downarrow R\end{array}}{\vdash^{\mathrm{A}}\blacktriangleright_\alpha N\Downarrow\blacktriangleright_\alpha R}}{\vdash^{\mathrm{A}\alpha}\blacktriangleleft_\alpha\blacktriangleright_\alpha N\Downarrow\blacktriangleleft_\alpha\blacktriangleright_\alpha R}$$

- The other cases are just "context-passing" cases that are straightforwards as the rules are fairly identical.

Thus the general result by induction: $M \xrightarrow{\mathrm{A}}{}^{*} R$ and R is a value then $\vdash^{\mathrm{A}} M\Downarrow R$.

B Proof of correctness for the generalised Fibonacci

As I give a staged generalised version of the Fibonacci function and a few hindsight on why it works and what happens with the CPS version, here is a proof of its correctness I wrote initially for myself.

It uses a different formalism than the rest of the paper and is based on the techniques exposed in [9] that do not rely on environment classifier, but it was a way for me to get used to their reasoning tools. It was also my first practical approach of programming in MetaOCaml and was an excellent exercise for me.

The main point here is that the proof is concise and does only ensure that the output is valid. It says nothing about the complexity, and as stated earlier without the CPS the program would still work but highly inefficiently.

```
let rec fibo_memo n =
  if Hashtbl.mem mem n then Hashtbl.find mem n
  else begin
    let ret = match n with
      | 0 | 1 -> 1
      | n -> (frec (n-1)) + (frec (n-2))
    in Hashtbl.add mem n ret;
    ret
  end
  int -> int

let fibo_cps_memo_staged n =
  let rec aux n k =
    if Hashtbl.mem mem n then
      let r = Hashtbl.find mem n in
      .<.~(k r)>.
    else
      match n with
  | 0 | 1 -> k .<1>.
        | n ->
            aux (n-1) (fun r1 ->
              Hashtbl.add mem (n-1) r1;
              aux (n-2) (fun r2 ->
                Hashtbl.add mem (n-2) r2;
                .<let r = .~r1 + .~r2 in .~(k .<r>.) >.))
  end in aux n (fun x -> x)
  int -> int code
```

Both algorithm compile in MetaOCaml (the first one does also in OCaml obviously), and both assume that the environment provides a hash table mem. They both run in linear time with n.

The output of the staged version is interesting :

```
  let x0 = 1 in
  let x1 = 1 in
```

```
let x2 = x0 + x1 in
let x3 = x1 + x2 in
...
```

Theorem. $\forall k \geq 0$, *!(fibo_cps_memo_staged k) = fibo_memo k*

Proof. The proof will require to demonstrate two equalities :

1. $\forall k \geq 0, ||$ fibo_cps_memo_staged $|| \; k =$ fibo_memo k
2. $\forall k \geq 0, ||$ fibo_cps_memo_staged $|| \; k =$!(fibo_cps_memo_staged k)

The first one comes immediatly when applying η-reductions to the left term, as we then reach a cps version equivalent to the right term.

The second one will use *Lemma 30* borrowed from the technical report from Inoue and Taha [9], that states that if a term e reduces to a non-staged term $||t||$, then $e = ||e||$.

By induction on k :

- for $k = 0$ and $k = 1$, aux returns $k\langle 1 \rangle$ hence we obtain $!\langle 1 \rangle$ that is 1, a term without any staging annotation.
- Supposing that for $k' < k$, (fibo_cps_memo_staged k) is of the form $\langle ||e|| \rangle$, then r_1 (resp. r_2) is of the form $\langle ||e_1|| \rangle$ (resp. $\langle ||e_2|| \rangle$). The program answers :

$$\begin{aligned} \langle \text{let } r = \tilde{} r_1 + \tilde{} r_2 \text{ in } \tilde{}(k\langle r \rangle) \rangle &= \langle \tilde{}(k\langle \tilde{} r_1 + \tilde{} r_2 \rangle) \rangle \\ &= \langle \tilde{}(\langle ||e_1|| + ||e_2|| \rangle) \rangle \\ &= \langle ||e_1|| + ||e_2|| \rangle \end{aligned}$$

  This is of the form $\langle ||e|| \rangle$, hence:

$$!(\text{fibo\_cps\_memo\_staged } k) = ||e||$$

  which concludes second point of the proof

We then immediatly have

$$\forall k \geq 0, !(\text{fibo\_cps\_memo\_staged k}) = \text{fibo\_memo k}$$

which concludes the proof and the theorem holds. □

C A concrete example of translation

One important thing to do was to test the practical use of the translation on a common example. One will quickly see the drawbacks, in terms of verbosity, of this translation, but it's original purpose is rather theoretical.

In this section the environment classifier multi-staged power function will be translated from our usual formalism to the λ_ρ record calculus, and we will check that the result actually behaves as we expect it to.

Let's translate a common example of a multi-staged function *power*, as presented by TI

$$
\begin{aligned}
power_{abs} &= \Lambda\alpha.\mathbf{fix}\ f.\lambda n.\lambda x. \\
&\quad\ \mathbf{if}\ n = 0\ \mathbf{then}\ (\blacktriangleright_\alpha\ 1) \\
&\quad\ \ \mathbf{else}\ \ \blacktriangleright_\alpha\ ((\blacktriangleleft_\alpha\ x) * (\blacktriangleleft\ f(n-1)x)) \\
\\
power &= \lambda n.\Lambda\beta.\ \blacktriangleright_\beta\ (\lambda x.\ \blacktriangleleft_\beta\ (power_{abs}\ \beta\ n\ (\blacktriangleright_\beta\ x)))
\end{aligned}
$$

Let's translate $power_{abs}$ first:

$$
\dfrac{\dfrac{}{R, r \vdash x \mapsto r(\mathbf{x})}}{R, r, r' \vdash \blacktriangleleft_\alpha\ x \mapsto \begin{pmatrix} hr', & \blacktriangleleft_\alpha = (\lambda h.[\bullet])(r(\mathbf{x})) \\ & \vdots \\ & \bot \end{pmatrix}} \quad \dfrac{\begin{array}{c}\vdots \\ R, r \vdash f\ (n-1)\ x \mapsto f\ (n-1)\ (r(\mathbf{x})), \bot\end{array}}{R, r, r' \vdash \blacktriangleleft_\alpha\ (f\ (n-1)\ x) \mapsto \begin{pmatrix} hr', & \blacktriangleleft_\alpha = (\lambda h.[\bullet])(f\ (n-1)\ r(\mathbf{x})) \\ & \vdots \\ & \bot \end{pmatrix}}
$$

$$
R, r, r' \vdash (\blacktriangleleft_\alpha\ x) \times (\blacktriangleleft_\alpha\ (f\ (n-1)\ x)) \mapsto \left(hr' \times hr', T = \begin{array}{ccc} & \bot & \\ \blacktriangleleft_\alpha = (\lambda h.[\bullet])(r(\mathbf{x})) & & \blacktriangleleft_\alpha = (\lambda h.[\bullet])(f\ (n-1)\ r(\mathbf{x})) \\ \vdots & & \vdots \\ \bot & & \bot \end{array} \right)
$$

$$
R, r \vdash \text{IfFalse} = \blacktriangleright_\alpha\ ((\blacktriangleleft_\alpha\ x) \times (\blacktriangleleft_\alpha\ (f\ (n-1)\ x))) \mapsto \begin{pmatrix} \lambda r'.(hr' \times hr'), & \blacktriangleright_\alpha, r \\ & \vdots \\ & T \end{pmatrix}
$$

Figure 4: If = false

We can use this first translation to translate the whole term:

$$\frac{R, r \vdash n = 0 \mapsto (r(\mathbf{n}) = 0, \bot) \qquad \dfrac{R, r, \rho \vdash 1 \mapsto 1, \bot}{R, r \vdash \blacktriangleright_\alpha\ 1 \mapsto \left(\lambda\rho.1,\ \begin{array}{c}\blacktriangleright_\alpha, r\\ |\\ \bot\end{array}\right)} \qquad \text{(Figure 4)}}{R, r \vdash \mathbf{if}\ n = 0\ \mathbf{then}\ \blacktriangleright_\alpha\ 1\ \mathbf{else}\ \text{IfFalse} \mapsto \left(\mathbf{if}\ r(\mathbf{n}) = 0\ \mathbf{then}\ \lambda\rho.1\ \mathbf{else}\ \lambda r'.(hr' \times hr'),\ \begin{array}{ccc}&\bot&\\ \bot & \blacktriangleright_\alpha, r & T\\ & |& \\ & \bot &\end{array}\right)}$$

$$\vdots$$

$$R, r_{fnx} \vdash power_{abs} \mapsto \left(\mathbf{fix}\ .f\lambda n.\lambda x.\mathbf{if}\ r(\mathbf{n}) = 0\ \mathbf{then}\ \lambda\rho.1\ \mathbf{else}\ \lambda r'.(hr' \times hr'),\ \begin{array}{ccc}&\alpha&\\ &|&\\ &\bot&\\ &|&\\ &\bot&\\ &|&\\ &\bot&\\ \bot & \blacktriangleright_\alpha, r & T\\ & |& \\ & \bot &\end{array}\right)$$

Where $(r_{fnx} + \{\mathbf{n} = n\} + \{\mathbf{x} = n\} + \{\mathbf{f} = f\}) = r$

We now need to translate *power*. First, note that applying a transition variable has a special effect in the translation, but applying a term is just a simple translation, hence we will leave $power_{abs}$ hidden as long as possible but we need to study first $power'_{abs} = power_{abs}\,\beta$, therefore we will write the whole tree.

$$R, r, r_{fnx} \vdash power_{abs}\,\beta \quad \mapsto \quad M = \mathbf{fix}\ .f\lambda n.\lambda x.\mathbf{if}\ r(\mathbf{n}) = 0\ \mathbf{then}\ \lambda\rho.1\ \mathbf{else}\ \lambda\rho.(hr' \times hr'),$$

$$T' = \left(\begin{array}{ccccc}&&\bot&&\\ &&|&&\\ &&\bot&&\\ &&|&&\\ &&\bot&&\\ &\swarrow&|&\searrow&\\ \bot & & \blacktriangleright_\beta, r & & \bot\\ & & | & & \\ & & \bot & & \\ & \blacktriangleleft_\beta = (\lambda h.[\bullet])(r(\mathbf{x})) & & \blacktriangleleft_\beta = (\lambda h.[\bullet])(f\,(n-1)\,(r(\mathbf{x}))) & \\ & | & & | & \\ & \bot & & \bot &\end{array}\right)$$

Hence the following translation tree:

$$\vdots$$

$$\dfrac{R, r_{fnx} \vdash power'_{abs} \mapsto M, T' \qquad R, r_{fnx} \vdash n \mapsto r_{fnx}(\mathbf{n}), \bot}{R, r_{fnx} \vdash power'_{abs} \mapsto \left(M\, n,\ \begin{array}{c} \bot \\ \diagup\ \diagdown \\ \bot \quad T' \end{array} \right)} \qquad \dfrac{R, r_{fnx}, s \vdash x \mapsto s(\mathbf{x}), \bot}{R, r_{fnx} \vdash \blacktriangleright_\beta\ x \mapsto \left(\lambda s.(s(\mathbf{x})),\ \begin{array}{c} \blacktriangleright_\beta, r_{fnx} \\ | \\ \bot \end{array} \right)}$$

$$\dfrac{}{R, r_{fnx} \vdash power'_{abs}\, n\, (\blacktriangleright_\beta\ x) \mapsto \left(M\, n\, (\lambda s.s(\mathbf{x})),\ \begin{array}{c} \bot \\ \diagup\ \diagdown \\ \bot \qquad \blacktriangleright_\beta, r_{fnx} \\ \diagup\ \diagdown \quad | \\ \bot \quad T' \quad \bot \end{array} \right)}$$

$$\dfrac{}{R, r_{fnx}, \rho \vdash \blacktriangleleft_\beta\ (power'_{abs}\, n\, (\blacktriangleright_\beta\ x)) \mapsto \left(\begin{array}{ll} & \blacktriangleleft_\beta = (\lambda h.[\bullet])(M\, n\, (\lambda s.s(\mathbf{x}))) \\ & | \\ & \bot \\ h\, \rho, & \diagup\ \diagdown \\ & \bot \qquad \blacktriangleright_\beta, r_{fnx} \cdot \rho \\ & \diagup\ \diagdown \quad | \\ & \bot \quad T' \quad \bot \end{array} \right)}$$

$$\dfrac{}{R, r_{fnx}, \rho_x \vdash \lambda x.\ \blacktriangleleft_\beta\ (power'_{abs}\, n\, (\blacktriangleright_\beta\ x)) \mapsto \left(\begin{array}{ll} & \bot \\ & | \\ & \blacktriangleleft_\beta = (\lambda h.[\bullet])(M\, n\, (\lambda s.s(\mathbf{x}))) \\ & | \\ \lambda x.h\, \rho, & \bot \\ & \diagup\ \diagdown \\ & \bot \qquad \blacktriangleright_\beta, r_{fnx} \cdot \rho \\ & \diagup\ \diagdown \quad | \\ & \bot \quad T' \quad \bot \end{array} \right)}$$

$$\dfrac{}{R, r_{fnx} \vdash \blacktriangleright_\beta\ \big(\lambda x.\ \blacktriangleleft_\beta\ (power'_{abs}\, n\, (\blacktriangleright_\beta\ x))\big) \mapsto \left(\begin{array}{ll} & \blacktriangleright_\beta, r_{fnx} \\ & | \\ & \bot \\ & | \\ & \blacktriangleleft_\beta = (\lambda h.[\bullet])(M\, n\, (\lambda s.s(\mathbf{x}))) \\ \lambda \rho_x.\lambda x.h\, \rho, & | \\ & \bot \\ & \diagup\ \diagdown \\ & \bot \qquad \blacktriangleright_\beta, r_{fnx} \cdot \rho \\ & \diagup\ \diagdown \quad | \\ & \bot \quad T' \quad \bot \end{array} \right)}$$

$$\vdots$$

$$\dfrac{}{R, r_{fnx} \vdash power \mapsto \left(\begin{array}{ll} & \bot \\ & | \\ & \beta \\ & | \\ & \blacktriangleright_\beta, r_{fnx} \\ & | \\ & \bot \\ & | \\ \lambda n.\big(\lambda \rho_x.\lambda x.h\, \rho\big), & \blacktriangleleft_\beta = (\lambda h.[\bullet])(M\, n\, (\lambda s.s(\mathbf{x}))) \\ & | \\ & \bot \\ & \diagup\ \diagdown \\ & \bot \qquad \blacktriangleright_\beta, r_{fnx} \cdot \rho \\ & \diagup\ \diagdown \quad | \\ & \bot \quad T' \quad \bot \end{array} \right)}$$

And is it working? What does $power\,1\,\varepsilon$ yield?
Let's keep going and translate $power\,1\,\varepsilon$.

$$R, r_{fnx} \vdash power \mapsto \left(\lambda n.\big(\lambda\rho_x.\lambda x.h\,\rho\big),\ \begin{array}{c} \bot \\ | \\ \beta \\ | \\ \blacktriangleright_\beta, r_{fnx} \\ | \\ \bot \\ | \\ \blacktriangleleft_\beta = (\lambda h.[\bullet])(M\,n\,(\lambda s.s(\mathbf{x}))) \\ | \\ \bot \\ \diagup\ \diagdown \\ \bot \quad \blacktriangleright_\beta, r_{fnx} \cdot \rho \\ \diagup\ \diagdown \quad | \\ \bot \quad T' \quad \bot \end{array} \right) \qquad R, r_{fnx} \vdash 1 \mapsto (1, \bot)$$

$$R, r_{fnx} \vdash power\,1 \mapsto \left(\lambda n.\big(\lambda\rho_x.\lambda x.h\,\rho\big)\,1,\ \begin{array}{c} \bot \\ \diagup\ \diagdown \\ \bot \quad\quad \bot \\ | \\ \beta \\ | \\ \blacktriangleright_\beta, r_{fnx} \\ | \\ \bot \\ | \\ \blacktriangleleft_\beta = (\lambda h.[\bullet])(M\,n\,(\lambda s.s(\mathbf{x}))) \\ | \\ \bot \\ \diagup\ \diagdown \\ \bot \quad \blacktriangleright_\beta, r_{fnx} \cdot \rho \\ \diagup\ \diagdown \quad | \\ \bot \quad T' \quad \bot \end{array} \right)$$

Now applying ε will deeply change the tree and the term, as every $\blacktriangleright$ and $\blacktriangleleft$ will be nullified as per the translation down to all the leaves. The resulting tree will only contain $\bot$ and we will omit it, let's have a look at the other transformations.

Let's call M' the result of transforming M, we will focus on that later. We first have to deal with the $\blacktriangleright_\beta$ in the tree corresponding to the $(\lambda\rho_x.\lambda x.h\,\rho)$ to which we apply r_{fnx} to get $\lambda x.h\,\rho$

Then we have to deal with the $\blacktriangleleft_\beta$, using the ε transformation (essentially, getting rid oh the r in $h\,r$):

$$\Big(\lambda h.\big((\lambda h.((\lambda n.\lambda x.h\,\rho)\,1))(\lambda r.h)\big)\Big)(M' n(\lambda s.s(\mathbf{x})))$$

This rewrites as expected to

$$(\lambda n.\lambda x.(M'\,n\,(\lambda s.s(\mathbf{x}))))\,1$$

And finally we need to apply $r_{fns} \cdot \rho$ to $\lambda s.s(\mathbf{x})$ as we follow down the term, to get x and therefore for the whole term:

$$(\lambda n.\lambda x.M'\, n\, x)\, 1$$

Now, let's see how M is changed: we apply $r \cdot \rho$ to $\lambda\rho.1$ and insert the hr' correspondingly in the right context with the special transformation to get:

$$\begin{array}{rll} M' = \mathbf{fix}\ .f\lambda n.\lambda x. & \mathbf{if}\ r(\mathbf{n}) = 0 & \\ & \mathbf{then} & (\lambda\rho.1)(r_{fnx} \cdot \rho) \\ & \mathbf{else}\ \lambda\rho. & \Big(\big((\lambda h.\big((\lambda h.h\, r')(\lambda r.\lambda x.x)h\big)\big)(r(\mathbf{x}))\big) \times \\ & & \big((\lambda h.\big((\lambda h.h\, r')(\lambda r.\lambda x.x)h\big)\big)(f\,(r(\mathbf{n}) - 1)\, r(\mathbf{x}))\big)\Big) \end{array}$$

Knowing that $r(\mathbf{x}) = x$ and $r(\mathbf{n}) = n$ this rewrites to:

$$M' = \mathbf{fix}\ .f\lambda n.\lambda x.\mathbf{if}\ n = 0\ \mathbf{then}\ 1\ \mathbf{else}\ x \times (f\,(n-1)\,x)$$

And the whole term becomes (after rewriting the 1 and because $\rho(\mathbf{x}) = x$)

$$\begin{array}{rcl} \lambda x.(M'\,1\,x) & \xrightarrow{\mathcal{R}} & \lambda x.((\mathbf{fix}\ f.\lambda n.\lambda x.\mathbf{if}\ n = 0\ \mathbf{then}\ 1\ \mathbf{else}\ x \times (f\,(n-1)\,x))\,1\,x) \\ & \xrightarrow{\mathcal{R}} & \lambda x.(\mathbf{fix}\ f.\mathbf{if}\ 1 = 0\ \mathbf{then}\ 1\ \mathbf{else}\ x \times (f\,0\,x)) \\ & \xrightarrow{\mathcal{R}} & \lambda x.(x \times ((\mathbf{fix}\ f.\lambda n.\lambda x.\mathbf{if}\ n = 0\ \mathbf{then}\ 1\ \mathbf{else}\ x \times (f\,(n-1)\,x))\,0\,x)) \\ & \xrightarrow{\mathcal{R}} & \lambda x.(x \times (\mathbf{fix}\ f.\mathbf{if}\ 0 = 0\ \mathbf{then}\ 1\ \mathbf{else}\ x \times (f\,(0-1)\,x))) \\ & \xrightarrow{\mathcal{R}} & \lambda x.(x \times 1) \end{array}$$

This is the expected result, and we see on the first few steps how it would have worked with n instead og 1, rewriting to $x \times \cdots \times x \times 1$.