A self-applicable partial evaluator for a subset of Haskell

Silvano Dal-Zilio
Ecole Normale Supérieure de Lyon
(dalzilio@ens.ens-lyon.fr)

director: **John Hughes**
Departement of Computing Science,
Chalmers Tekniska Hogskola
(rjmh@cs.chalmers.se)

August 16, 1993

Abstract

Partial evaluation is becoming very promising as a programming tool, as its practice is now well developed. But the theorical foundation are not equally well understood.

In this paper, we report on the making of a partial evaluator for a functionnal language deriving from Haskell. And we discuss on the many problems arised from its self-application, theorical as much as practical.

Contents

1 Introduction.

Functionnal programing provides an elegant solution to the increase of software complexity, allowing high-structured programing and providing powerful tools, such as lazyness and higher-order function definition [Hug84]. But this advantage in comfort of programmation is, like often, counterbalanced by a loss of efficiency.

Use of partial-evaluation can remedy to this drawback, program's specialization given automatically more efficient and always faithfull results. And this for a wide range of problems, such as neural net training, scientific computing or computer graphics.

The most developed area for the use of partial-evaluation is probably the design of programming language compilers, which requires application of partial evaluation to partial evaluator themselves.

But self-application arises many problems. Especially when a strongly-typed language, such as Haskell [Hask90], is used.

For example, whereas Futamura produced his "projection" in 1971 [Fut71] , the first partial evaluator having been succesfully self-applicated (produced at DIKU, Copenhagen) was made only in the middle of the eighties. And the first written in a strongly-typed language, PEL, a subset of Lazy ML (LML), only in 1989 [Lau89].

In this report we defined section 3, after an overview of partial-evaluation principles in section 2, a subset of the functionnal programming language Haskell. This language will be usefull to our study of partial evaluation, section 4. Section 5 described the final partial evaluator, with an emphasis on the compiler generator . Whereas section 6 contains a discussion on possible extensions. Section 7 concludes.

2 About partial evaluation.

2.1 Principle of partial evaluation.

Partial evaluation can be seen as an extension of the principle of projection, well-known in geometry or in analysis.

Partial evaluation consist of specializing a program to a part of its arguments. For example, if we consider a program f, with two arguments x and y, we can compute or reduce expression in f if we know that x will take a static value a. This computation carries out a new program, f_a, such that $f_a\ y =\ f\ a\ y$, that can possibly run much faster than f.

The first formulation of the idea of partial evaluation is the works of S C Kleene, published in 1952. Kleene demonstrates his s-m-n theorem, that can be interpreted as:

For a general $m+n$-argument computable function, and given values for the first m arguments $x_1, \ldots, x_m$, there exist a program for the specialized function $f_{x_1,\ldots,x_m}$. Moreover, there is a program (a computable function) which effectively constructs the specialized program from every computable function f and a set of values (in fact Kleene argue with recursive function on the integers, and Turing machine.) [HML, p. 705-707]

A partial evaluator, given the program and the values of the static parameters, construct a new program which, when given the remainings input, yield the same result that the program would have produced [1].

We could defined this property using a equational definition:

$$[\![[\![mix]\!]\ (f, x)]\!]\ y = [\![f]\!]\ x\ y$$

Where $[\![f]\!]$ is the "function" that the programs f define, and mix is the partial evaluator.

The goal is to generate an efficient program automatically. This is done, intuitively, by performing all the calculation depending only on statics parameters (*evaluation*), and by generating code for calculations depending on both type of parameters (*reduction*: for example unfolding function calls, or reducing conditionnal branch with static-test expressions). It's a mixture of computation and code generation, hence the name *mix* for our partial-evaluator.

An ideal partial evaluator will performed all computations that can be done without the dynamic values.

For example, if we define the function $power\ n\ x$, that computes x^n:

$$\begin{array}{rcl} power\ n\ x & = & if\ (n == 0) \\ & & then\ 1 \\ & & else\ x * power\ (n-1)\ x \end{array}$$

We can specialize $power$ for static values of n or x:

$$\begin{array}{rclcrcl} power_{n=3}\ x & = & x * (x * (x * 1)) & \quad & power_{x=4}\ n & = & if\ (n == 0) \\ & & & & & & then\ 1 \\ & & & & & & else\ 4 * power_{x=4}\ (n-1) \end{array}$$

We could see with this little example some aspects of partial evaluation:

- We must be able to compute, for efficiency, that, if n is static, the test $n == 0$ is a static expression. Which means that we must be able to differientate static from dynamic expression. This process, also named *Binding-Time Analysis*, is one of the main problem. Indeed one can demonstrate that, resolve the problem of binding times analysis is resolve the halting problem [2].

- A results following from the "undecidability" of the binding-time analysis, is that we cannot decide if we can unfold a function call. For example we cannot unfold the function $power_x$ without entering an infinite loop

$$\begin{array}{rclllll} power_4\ n & = & if & (n == 0) & & & \\ & & then & 1 & & & \\ & & else & if & ((n-1) == 0) & & \\ & & & then & 4 * 1 & & \\ & & & else & if & (((n-1)-1) == 0) \\ & & & & then & 4 * 4 * 1 \\ & & & & else & \ldots \end{array}$$

[1] *Static parameters* are parameters whose value are known in advance, as opposed to *dynamic parameters* whose values are known at computation times.

[2] If you know that a variable is static or not, then you are able to say if this variable is "visited" during the computation. And then you can decide if your program end.

This example demonstrate that **specialization faces problems of termination**.

- The specialization can increase size and number of call of the program. For example, if we do not unfold calls to *power* in the first example, we have:

$$\begin{array}{rcl} power_3\ x & = & x * power_2 x \\ power_2\ x & = & x * power_1 x \\ power_1\ x & = & x * power_0 x \\ power_0\ x & = & 1 \end{array}$$

  Then **we can't predict the speedup and the size from the program before specialization**.

2.2 Futamura projections.

Futamura projection shows the capabilities of partial evaluation for generating compiler generator.

We have already seen that a partial evaluator takes two argument, a program $f : A \times B \to C$, to specialize , and a value a from A [3].

In particular we have the relation

$$(1)\ :\ \forall b \in B, [\![f]\!]\ a\ b\ =\ [\![f_a]\!]\ b\ , where\ f_a\ =\ [\![mix]\!]\ f\ a.$$

A compilator from a language $\mathcal{S}$ (*source*) to a language $\mathcal{T}$ (*target*) is also defined by such an equation (2): $target\ =\ [\![compiler]\!]\ source$.

Besides, the result of mix is not necesseraly written in its input language, and, like for compilers, a partial evaluator has an input and an output language.

So, if we define an interpreter *int*, from $\mathcal{S}$ to a target language $\mathcal{T}$, and if mix is a partial evaluator "from $\mathcal{S}$ to $\mathcal{T}$", it follows that:

$$\begin{array}{rcll} result & = & [\![f]\!]\ input & f\ is\ written\ in \mathcal{S} \\ & = & [\![int]\!]\ f\ input & \\ & = & [\![[\![mix]\!]\ int\ f]\!]\ input & (1) \\ & = & [\![target]\!]\ input & target\ is\ written\ in \mathcal{T} \end{array}$$

And then we prove the first Futamura projection:

$$F_1 :\ \ \mathcal{P}_{target} = [\![mix]\!]\ int\ \mathcal{P}_{source}$$

This equation means that, specializing an interpreter to a particular program, we obtain a program in the "target" language of *int* (the language in which *mix* has been written).

Using this first result we prove the second Futamura projection:

$$\begin{array}{rcll} target & = & [\![mix]\!]\ int\ source & \\ & = & [\![[\![mix]\!] mix\ int]\!]\ source & (1) \\ & = & [\![compiler]\!]\ source & (2) \end{array}$$

[3] As we will use typed-functionnal-languages latter, we consider that programm are function and that, at each data, can be given a type

$$F_2 : \quad compiler \ = \ [\![mix]\!] \ mix \ int$$

Then we are able to generate automatically a compiler given an interpreter. The interest lies in the fact that interpreters are easier to write, and that we are sure to produce compilator always "correct" with respect to the interpreter. i.e. semantically correct compiler.

The only restriction is that *mix* must be written in the language used for its input, as the "text" of mix is the first argument.

Remark. We say that *mix* must be self-applicable. Obviously we do not apply mix to himself. Self-application, like in demonstration of the Halting-Problem, is used for invalidate result. We only use the text of mix [4].

These two previous formulas bring us to examine the meaning of $[\![mix]\!] mixmix$

The mix-equation (1) gives:

$$[\![[\![mix]\!] \ mix \ mix]\!] \ int = [\![mix]\!] \ mix \ int$$

and, from F_2, we know that $[\![mix]\!] \ mix \ int$ is a compilator

Then

$$F_3 : \ [\![mix]\!] \ mix \ mix \text{ is a compiler-generator.}$$

A program that, given an interpreter, produce a compiler.

3 TinyHaskell, or a subset of Haskell.

3.1 Introduction.

The subset of Haskell, TinyHaskell, used for the implementation of mix, is the result of many contradictory imperatives.

We have first to reconcile in our language,

- power of expression: as we must write mix in TinyHaskell to permit later self-application. [5]
- simplicity: in order to better understand the behaviour of mix, we do not handle difficult programming paradigms, like higher-order function for example.

We also have to keep, as much as possible, Haskell semantics. For example, TinyHaskell and its "big brother", share the same layouts rules and module's definition[Hask90].

To avoid the difficult problem of binding-time analysis, we entrust programmers with taking care to put annotation that describe the nature of each expression. Which makes the problem much simpler and allows experiment.

Those annotations are used as specifications to decide which conditional should be reduced, which expression may be completely evaluated or which function call should be

[4] remenber the difference between f and $[\![f]\!]$

[5] see 2^{nd} Futamura projection.

unfolded. It is of course possible to compute a part of those informations on-the-fly, given the static variables, but it is both more efficient [6] and easier to understand when presented in the form of annotation [Lau91].

So this language must provide facilities to represent those annotations. We have reserved special character to define

‘	static variables or expression.
$	recursive call (to avoid infinite unfolding) [7]
˜	for partially-static structures

Here an example of an annotated function definition in TinyHaskell:

```
power ‘n x = if ‘(n == 0)
               then 1
               else x * $power (n - 1)  x
```

Figure 1: The function power annotated.

those annotations defined n as a static parameters of the function, and the conditionnal $(n \;==\; 0)$ as static (i.e. computable as compile-time). The dollar-sign in front of the function call ($\$power\ (n-1)\ x$) avoid the unfolding, and leads to the computation of a specialized version of $power$ for the value $n-1$.

3.2 TinyHaskell Context-Free Syntax.

program	$\rightarrow$	*decls*	
decls	$\rightarrow$	$decl_1$ `;` ... `;` $decl_n$	$(n \geq 1)$
decl	$\rightarrow$	*lhs* `=` *exp* [`where` { *decls*}]	
lhs	$\rightarrow$	[`$`]*identifier* ([`‘`\|`˜`]*identifier*)*	
exp	$\rightarrow$	$exp^f\ exp'$	
exp'	$\rightarrow$	`:` *exp*	
	|	*reservedop exp*	
	|	ϵ	
exp^f	$\rightarrow$	`if` *exp* `then` *exp* `else` *exp*	(*conditional*)
	|	`case` *exp* `of` *alt* (; *alt*)*	(*case expression*)
	|	*fexp*	
fexp	$\rightarrow$	$-$ *exp*	
	|	*reservedfun* (*aexp*)*	
	|	[`$`]*identifier* (*aexp*)*	
	|	*con* (*aexp*)*	
	|	*aexp*	
aexp	$\rightarrow$	*identifier*	
	|	*con*	(*constructor*)
	|	*integer*	

[6] see discussion on off-line partial evaluator section 4.2.

[7] see section 5

	$\mid$	*string*	
	$\mid$	(*exp*)	(*parenthesised expression*)
	$\mid$	(exp_1 , ... , exp_k)	(*tuple*, $k \geq 2$)
	$\mid$	[exp_1 , ... , exp_k]	(*list*, $k \geq 0$)
	$\mid$	[‘|˜]*exp*	(*annotated expression*)
con	$\rightarrow$	**True**	
	$\mid$	**False**	
	$\mid$	$uppercase\{\ char\ \}^*$	
identifier	$\rightarrow$	$lowercase\{\ char\ \}^*_{<reservedop,reservedfun>}$	
alt	$\rightarrow$	$pattern \rightarrow exp$	
pattern	$\rightarrow$	_	(*wildcard*)
	$\mid$	*identifier*	
	$\mid$	$con\ (identifier)^*$	
	$\mid$	[ ]	
	$\mid$	*identifier* : *identifier*	
	$\mid$	(exp_1 , ... , exp_k)	($2 \leq k \leq 6$)
	$\mid$	(*pattern*)	

Remark. The case expression defined in TinyHaskell is not the same than this defined in Haskell.

As you could see in the context-free grammar, we only match pattern result of the application of a constructor on identifier, or identifier themselves. List and tuples might be seen as a special case. ":" being the constructor for the lists, and #n the constructor for tuples which length is n.

4 The specializer.

4.1 Reduction and evaluation.

4.1.1 Algorithm.

The structure of a specializer, and then of mix, is very closed to that of an interpreter. And in our case of a TinyHaskell-self-interpreter, as both mix input and output are TinyHaskell programs.

Mix specialize its first argument, an annotated program, according to a list of its static parameters. The result, a specialized program, is a list of specialized function definition.

We could describe this algorithm defining the specializing loop:

1. We define a *Pending list* of all the function definition yet to be specialized, paired with their static-parameters-values. This list is initialized with the the arguments of mix [8], and represent what specialization is still to be performed.

   We also define a *Done list* of all the specialized call already computed to avoid duplicated work (and then, sometimes, endless specialization of the same call). This list represent what specialization have already been performed.

[8] remenber that a program is a function

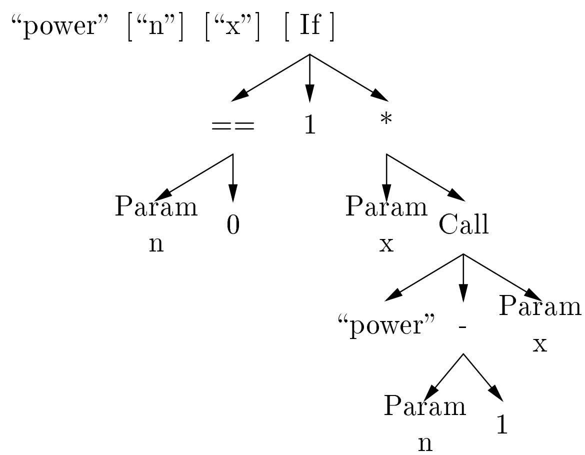
2. While **Pending** is non-empty, we take a menber of the list and, if it is not in **Done**, we construct a new specialized function using *eval* on the function body.

   The function *eval* [9], given the names of the parameters, their values (the value of the dynamic parameter x is `Parm` "x") and the status of an expression, reduce or evaluate this expression. Indeed these two action are very similar. we can notice the function *static* that, given an expression, compute its status according to the annotation.

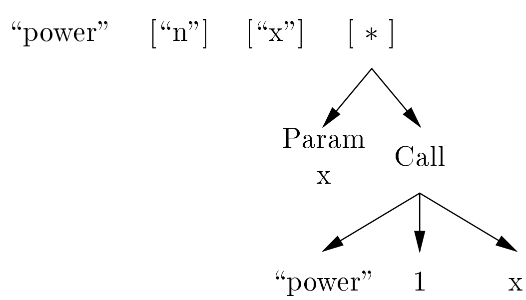
   We could see *eval* as an algorithm of reduction on graph (Indeed we work with the program-parse result). For example, if we study the specialization of the function *power n x* for the value 2 of n:

```
power 'n x = if '(n == 0) then 1
                          else x * power (n - 1) x
```

   We evaluate the body of *power* (given below), knowing that $n == 0$ is a static-expression[10].



   The result of the evaluation is a graph with less state, as “static conditional branches” have been pruned and n has been replaced by its value:



   The residual call *power* 1 x can itself be evaluated in two different way. It can be unfolded, or it can lead to the creation of a new specialized function $power_{n=1}$

[9] see figure in the appendix

[10] a function call is represented by the name of the function, a list of static and a list of dynamic parameters, and the body of the function

3. The residual expression is scanned for any residual function call that may need further specialisation.
4. When **Pending** is empty, we rename the function of the **Done**-list, using their static values, to obtain the final result.

4.1.2 Type of expressions.

We define a type for each expressions and value of the TinyHaskell language, using the Haskell type system[Hask90]. [11]:

```
data Term = Num Int   |
            Bl Bool   |                       -- boolean
            Str String   |
            Term Term   |
            Lst [Term]  |
            Constr String [ Term ]  |
            Parm String  |
            Case Term [ (String,[String],Term) ]  |
            If Term Term Term  |
            Prim String [Term]  |
            Call String [Term] [Term]  |
            RCall String [Term] [Term]  |
            Ann Int Term
```

universal type definition for expression.

`Term` is the general typed for expression.

`Constr s list` is used to represent application of the constructor `s` upon the list of term `list`. For example a list can be expressed using the list constructor ":":

$$Lst[Num1, Num2, Num3] = Constr" : "[Num1, Constr" : "[Num2, Constr" : "[Num3, [\ ]]]]$$

`Parm` is used to represent a parameter of the function.

The first argument to the `CASE`-constructor is the expression over which the case-expression proceed. Each clauses being a triple: (Constructor , List of variable , Expression).

`Prim` is used to represent primitive function calls, like $+, -, \ldots$

`Call` is used to represent user-defined functions that could be unfolded, and `RCall` to represent recursive call that should not be unfolded (tagged by a \$ -sign). Their parameters are split under static and dynamic parameters.

`Ann` is used to represent annotated expression.

Others type constructor are explicit.

[11] in this system, which could be seen has a powerfull extension of an Hindley-Milner type system, the type defining the list of element of A is represented by $[A]$

```
eval prog names values stat e = case e of

     Num i -> i
     Str s -> s
     Term e -> e
                           ....
     Parm x -> lookup_envt names values x
     If b e1 e2  -> if   ((static b) == "static") || (stat == "static"))
                           then
                                if   (eval prog names values 2 b)
                                then (eval prog names values stat e1)
                                else (eval prog names values stat e2)
                           else (
                                If (eval prog names values stat b)
                                    (eval prog names values stat e1)
                                    (eval prog names values stat e2)
                                )

     RCall f st dy -> Call f (map (eval prog names values "static") st)
                                    (map (eval prog names values "dynamic") dy)
                            ....
```

Figure 2: an oversimplified *eval* -definition.

4.2 Reducing the size of datas structure.

4.2.1 Theorical aspect.

We can inferred the type of mix using the same notation than in the introduction. But we need first to introduce a new notation to distinguish between the type of a function, and the type of the "program implementing this function". Like previously, with the convention upon the meanings of f and $[\![f]\!]$, this notation is not correct. Many different program implement the same function, and program written in many different languages. But we use it for convenience.

So if we define $\overline{f}$ as the "type of a program " implementing f, and the type of mix's first argument as : $A \times B \to C$, mix must be of type:

$$mix\colon \overline{(A \times B \to C)} \times \overline{A} \to \overline{B \to C}$$

We can notice that mix's second argument is also an encoding of the static value. Indeed, even if in untyped language the second solution: $mix\colon \overline{(A \times B \to C)} \times A \to \overline{B \to C}$ is used, it is inapplicable to our purpose because we cannot cannot express it. The type of the second argument varies according to the "value" of the first argument, which is only element of a simple fixed type (the type `Term` as it happens).

So, if we want to apply mix to $[\![mix]\!]$, we obtain the right type instantiating A to $\overline{A \times B \to C}$, B to $\overline{A}$ and C to $\overline{B \to C}$, which gives:

$$mix\colon \overline{(\overline{A \times B \to C} \times \overline{A} \to \overline{B \to C})} \times \overline{\overline{A \times B \to C}} \to \overline{\overline{A}} \to \overline{\overline{B \to C}}$$

Then the second argument of mix must be a double encoding of the program to which $[\![mix]\!]$ is being specialized. We will see the repercussion of this double-encoding in the next section.

Remark: Exactly the same feature arise in the definition of the compiler-generator, which is defined by $[\![mix]\!]\ mix\ mix$, where the compiler-generator has the type:

$$cogen\colon \overline{\overline{(A \times B \to C)}} \to \overline{\overline{A} \to \overline{B \to C}}$$

4.2.2 Practical consequence.

The need of a double encoding is not without consequence. That means that the parameter of mix, when we make a compiler, is represented by a huge data structure.

For example, the simple expression $l1 ++ l2$ (++ is the concatenation-operator) will be represented, with our convention, by:

$$Prim\ \text{“}++\text{”}\ [Parm\ \text{“}l1\text{”}, Parm\ \text{“}l2\text{”}]$$

but is represented as a value by:

$$(*)\quad Constr\ \text{“}Prim\text{”}\ [\ Str\ \ \text{“}++\text{”}, Lst\ [\ Constr\ \text{“}Parm\text{”}\ [\ Str\ \text{“}l1\text{”}], Constr\ \text{“}Parm\text{”}\ [\ Str\ \text{“}l2\text{”}]]\ ]$$

The size is more than double.

And the loss in efficiency is even worse. For example, the size of the heap corresponding to that gigantic data-structure, increased frequency of garbage collection or memory paging. And more, before specialization, we have to test if the function call to be specialized is already recorded in the Done-list. So we have to compare the values of the static parameters of this call, which necessite the used of an equality test, test directly proportionnal to the size of the objects being compared. This latter argument also demonstrate that the lazyness of our languages is no help in this problem, as the equality test force the full computation of each object.

The solution used in our study, and already succesfully employed in the making of a self-applicable partial evaluator for LML [Lau89], is to replace lazyness by delaying the expansion ourselves.

We used the type-constructor `Term` to help us in this way. We could see Term as an equivalent of "quote" in other functionnal programming languages. The value and "type" of $Term\ (Prim\ \text{"}++\text{"}\ [Parm\ \text{"}l1\text{"}, Parm\ \text{"}l2\text{"}])$ is equal to $(*)$, for example.

We could then compress or expand expression-encoding according to the situation and then gain almost an order of magnitude in the size needed for encoding all our datas. In practice this means that the times taken to compute $mix\ \overline{mix}\ \overline{\overline{mix}}$, for example, is reduce from hours to minutes[Lau91]. Even if, with this strategy, some terms might be expanded several times (for example a static value examined by several case expressions).

5 MIX, a partial evaluator written with TinyHaskell.

5.1 Improving partial evaluation.

The chief motivation for doing partial evaluation is speed. Then, an estimate of the obtainable speedup before the specialization is done, would be a valuable information. When we use mix as a compiler, for example, It could be interesting to know wether specialization plus specialized program run time is greater or not than program interpretion, if the program has to be run once or if the time to run the specializer itself is a significant factor. ie:

$$t_{mix}(interpreter, program) + t_{program}(input) \geq t_{interpreter}(program, input)$$

We will consider the speedup obtain with partial evaluation as a measure of partial evaluation efficiency. We could defined the speedup using a straightforward mathematical definitions[NDJ]:

`Definition:` for a fixed two-arguments program int and a static input st, we define the speedup by:

$$su_{st}(input) = \frac{t_{int}(st, input)}{t_{mix(int,st)}(input)}$$

Then, if we represent the (finite) computation of a program using a "weakened" operational semantic:

$$prog\ s_0\ d_0 \equiv (p_0, (s_0, d_0)) \rightarrow (p_1, (s_1, d_1)) \rightarrow \ldots \rightarrow (p_n, (s_n, d_n))$$

where s represent static values (that means values depending only on s_0) and d dynamic values, and $(p_i, (s_i, d_i)) \rightarrow (p_{i+1}, (s_{i+1}, d_{i+1}))$ represent a "derivation" between control points of the flow chart of $prog$. We could interpreted partial evaluation as a "derivation compressor".

Indeed, variable values depending only on p_i and s_i can be evaluated at specialization time, and if a state $(p_i, (s_i, d_i))$ only depends on static inputs, the specialization "can shift control" to p_{i+1} (unfolding).

So, if we call t_{s_0} (resp. t_{d_0}) the time spent in static (resp. dynamic) computation during *prog*'s-execution, we obtain:

$$su_{s_0}(d_0) = \frac{t_{s_0} + t_{d_0}}{t_{d_0}}$$

Assume that partial evaluation of *prog* on s_0 terminates in K derivation. Then in the standard computation there can be at most $K - 1$ "static steps" since mix is no faster than direct execution. This means in particular that $t_{s_0} \leq K.t_{d_0}$. And then we have an upper-bound for the speedup, that is K.

This bound being independent of the dynamic input (d_0), it follows that no superlinear speedup can be achieve using partial evaluation. We could at most expect linear speedup. And this bound is clearly far larger than what is usually seen in practice. But experiment shows a linear speedup of approximatively 5 when we use the compiler, and 25 when we use the compiler-generator on the self-interpreter.

The speedup obtain with partial evaluation on an interpreter can be explain using the same approach. Differences between execution of a program an interpretation lies in the many overhead arised by "manipulation on syntax", recursive calls of the evaluation, command execution functions and variable access. The cost of interpretation-overheads can be expressed by the empirical law:

for a typical interpreter int's running time on inputs *prog* and *input* we have:

$$\exists \alpha_{prog},\ \forall\ input, \alpha_{prog}.t_{prog}(input) \leq t_{int}(prog, input)$$

(in experiments α_{prog} is often superior to 10 and grow as a function of *prog*'s size)

The speedup that we can expect, that is $\frac{1}{\alpha_{prog}}$, is then bounded by 1. We could then defined an "optimal" partial evaluator as a partial evaluator able to `remove a complete layer of interpretation`, ie a partial evaluator responding to the formula $su = 1$:

$$t_{mix(int,prog)}(input) = t_{prog}(input)$$

The rest of this report is dedicated to the study of the relationship between partial evaluation and compilation, and more precisely, on the different methods to obtain a compiler that remove the maximum of interpretation's overhead.

5.2 The TRICK.

We can first remark that mix's "efficiency" depend mainly on its first argument.

For example, it often happens that a parameter, *var*, takes only a bounded number of values (for example values in a list *names*). If the function to be specialized implement a dynamic scoping for *var* it is unlikely that the variable name will disappear from the mixed-program. Indeed as the list is dynamic, *var* must be tagged as dynamic and call to this variable can't be specialized.

But it is not the case if the function implement a static-scoping. Intuitively, we could specialize the call if mix compares *var* with all the possible values and produce specialized code for them, which is possible as they are in a finite number and as the values are known at compile-time.

This trick: specializing "statically bounded parameters" using all their possible values, is so common that it has been named **The Trick**. And is necessary to avoid trivial self-application of mix.

5.3 A new trick.

We have defined, section 3.1, an annotation for partially static structures (˜). Wich could be define by dynamic structures with known "properties".

For our study, for example, we consider static-list, which are dynamic list of known length. In fact, such static-list are frequent in programs. For example the environment of a program, as used by the partial evaluator, which is a list of all the parameters and their values, is a static list (we know the name of each argument but not their values).

We could then specialize a static-list $\mathcal{L}$ of length n, by replacing all occurences of $\mathcal{L}$ by a list of n new variables $l_1, \ldots, l_n$. And also specialize call to $\mathcal{L}$ using those new variable. For example we could specialize the function:

```
head ~l = case ~l of
       []      -> fail ''too small''
       x : xs  -> x
```

into

```
head_[n] l1 l2 ... ln = l1
```

and then, when unfolding a specialized call to the function head with a static-list argument, we obtain directly the result.

This "new trick" can improve mix's self-application. Indeed, mix use a static list to handle the environment of the program being specialized. Then, without static-list specialization, when mix is self-applied, we obtain for each call to a parameter's value, a sequence of call to head and tail[12] in the residual program.

6 Extension of the programming language.

6.1 *Let* expression.

We will begin the extension of our tiny functional language by adding let-expression.

Let-expressions interest lies in their use to avoid "re-computation". For example it is more efficient to compute the value (20! , 20!) with the expression: *let a_lot = factorial 20 in (a_lot , a_lot)* than with the expression *(factorial 20 , factorial 20)*.

We could use this interesting property in the partial evaluator to avoid the problem of code duplication, which leads at run-time to duplication of computation. For example, if we define:

[12] *car* and *cdr* in Scheme.

```
double x = (x + x)

power_of_2 n = if n == 0 then 1 else (double (power_of_2 (n - 1)))
```

then unfolding the function call gives:

```
power_of_2' n = if n == 0 then 1 else ((power_of_2 (n - 1)) + (power_of_2 (n - 1)))
```

transforming a linear program in an exponential one's.

But if during a preprocessing phases we insert let-binding for each duplicate variables[13], we could get rid of those duplication.

```
double n = let x = n in (x + x)

power_of_2 n = let x = n in
                    if x == 0
                    then 1
                    else (double (power_of_2 (x - 1)))

power_of_2' n = let x = n in
                    if x == 0
                    then 1
                    else let x' = (power_of_2 (x - 1)) in (y + y)
```

Evaluation of let-expressions in mix is given in figure (3). Where we could see the new type constructor introduced to represent let-expressions: `Let s e1 e2`, which represent the definition *let s = e1 in e2* (s is a variable). If *e1* is static, then we unfold the let definition in a new environment where the variable *s* is bounded to the value of e1. If not, we created a specialized let-expression.

6.2 Higher-order functions.

Before to introduce higher order function, we will first discuss on the representation of lambda-expression in TinyHaskell.

Lambda-expression are represented by the type-constructor: `Lam stat x e`, where x is the parameter, e the expression, and *stat* is the status of the lambda-expression: static ($\lambda`\ x.e$) or dynamic ($\lambda\ x.e$) (a dynamic lambda-expression is: $\lambda\ x.\ (y + x)$ where y is dynamic for example, whereas $\lambda\ x.\ power\ 3\ x$ is static).

We assume in TinyHaskell that we only apply lambda-expression. ($\lambda\ x.\ power\ 3\ x$) 5 is correct, but ($power$ 3) 5 is not. And we provide the operator `@` to "annotate" static application of a lambda-expression to an expression. An application $e1\ e2$ is represented with our type system by `Appl i e1 e2`, where i is the status of the application and $e1$ and $e2$ are Term.

We could then define, for example, the function *map_stat* which map a static function to a known list:

[13] we must then be able to count variable's occurences.

```
map_stat ‘f ‘l = case ‘l of
        [] -> []
        x : xs -> (f @ x) : (map_stat f xs)
```

The evaluation of lambda expression and application is given figure (4):

6.3 Specializing types definition.

We have defined, previously, an efficient partial evaluator as a partial evaluator able to remove a “complet layer of interpretation”

We will see in this section that mix is unable to achieve this goal. Especially when applied to himself, mix is unable to remove all the interpretion overhead.

Imagine that we specialize a program which uses some type not represented in `Term`. For example the addition in an algebraic type for numbers

```
add m n = case m of
             ZERO   -> n
             SUCC x -> add x (SUCC n)
```

the result of the specialization of the TinyHaskell self-interpreter to this program gives:

```
int_0 values_2  values_1
    =   (eval_1  values_2 values_1)
eval_1 vs_2  vs_1
    =   case vs_2  of
         Constr n  p  -> if  ( "ZERO"  ==  n )
                             then   vs_1
                             else  if  ( "SUCC"  ==  n )
                                     then   eval_1  (head  p) (AConstr "SUCC" [vs_1])
                                     else   fail "no match in case expression!!"
```

We could see that the type used in the program is coded using the “universal”-datatype of mix with `Constr`. Then the case expression is translated into a nested conditionnal, whereas it would have been more efficient to compile it into a case expression with a multiway jump. Moreover, the test of each conditionnal is based on an expansive matching upon string, and as the second element of a “Constr” is a list, we have multiple call to the primitives function head and tail.

One possibility to handle more efficiently users datatypes, is to use a postprocessing phases to transform nested conditionnals and to represent each constructor by integers (the equality test being cheaper).

But a more promising approach is to specialize types definition themselves. The idea is to create specialized constructor from users type definition. For example we can create two specialized version of `Constr "ZERO" [ ]` and `Constr "SUCC" [n]`, that is `Constr_ZERO` and `Constr_SUCC n`. The nested conditionnal being replaced by a case expression.

We could see with this little example the numerous advantages of this technique. Using `Constr_SUCC`, we save a call to the primitive head. And, in each case, we save interpretation overheads created by the application of the type constructor `Constr`.

7 Discussion.

This exercise in making a self applicable partial evaluator, has given me the opportunity to learn an exciting and promising method for both optimizing interpretive programs, and for understanding the theorical relationship between interpreters and compilers.

It has also been the occasion to discover the field of functionnal programming and lazyness.

Although partial evaluation is a universal paradigm. Partial evaluator for C and Prolog have been succesfully self-applied. Specificities of functional programming gives it an other dimension, in particular by its facility to work on program transformation.

But there are not only advantages. Partial evaluation of lazy languages faces, by essence, many problems. With efficiency: it is sometimes more efficient to preserve the lazyness of a program than to specialize it. And with termination: how should we handle an infinite structure[14]?

That's why a better understanding of the partial evaluation's process is needed. And in particular of the theoricals underpinnings. And I hope that this brief overview of partial evaluation, based on the study of a self-applicable partial evaluator for a "tiny" functionnal language, would have help you to foresee some of those problems.

acknowledgments.

Thanks to Graham Hutton for all the precious advices given to me during my stay in Chalmers, and for my discover of the joy of combinators and categories. Thanks to Jan Ekman for his courses always instructive. Thanks to Carsten Keller for his precious advices and to K.V.S. Prasad for his experiment and his kindness. Thanks to John Hughes, finally, who has still been attentive and who has been a constant source of inspiration during those two months.

[14]in lazy language one can define the list of all the integers for example

References

[Lau91] John Launchbury *A Strongly-Typed Self-Applicable Partial Evaluator.*
FPCA 1991 (Springer LNCS).

[Lau89] John Launchbury *Projection Factorisations in Partial Evaluation.*
Ph.D. Thesis, University of Glasgow (Nov. 1989) .

[Fut71] Y. Futamura *Partial Evaluation of Computation Process - An approach to a Compiler-Compiler.*
Systems, Computers, Controls 1971 (Vol. 2 No 5).

[Hug84] John Hughes *Why Functionnal Programming Matters.*
Report 16, Programming Methodology Group .

[Hask90] P. Hudak *Report on the Programming Language Haskell* .
Glasgow University 1990 .

[NDJ] Neil D. Jones, Carsten K. Gomard, Peter Sestoft *Partial Evaluation and Automatic Program Generation.*
Prentice-Hall, C.A.R. Hoare series editor.

[HML] *Handbook of Mathematical Logic.*
Barwise editor, volume 90 .

```
eval prog names values static e =
 case e of
        ....

   Let s e1 e2   -> if (static e1) == ‘‘static’’
       then let
              ss = eval prog ns vs ‘‘static’’  e1
            in
              eval prog (s : ns) (ss : vs) (max stat (static e2))  e2
       else let
              ss = eval prog ns vs  (max stat (static e1))  e1
            in
             ALet s ss (eval prog (s : ns) ((Parm s) : vs) (max stat (static e2)) e2)

        ....
```

Figure 3: evaluation of let expression in the function *eval* of mix.

```
eval prog names values status e =
 case e of
        ....

   Lam i x t    ->  Lam i x (eval prog (x:ns) ((Parm x): vs) (max i status) t)

   Appl i e1 e2 ->  if (i == ‘‘static’’)
         then
            case e1 of
               Lam i s t -> (\ x -> (eval prog (s:ns) (x:vs) static t))
                                  (eval prog ns vs status e2)
                _         -> fail "fail in static lambda-appl. "
         else
            Appl i (eval prog ns vs status e1) (eval prog ns vs status e2)

         ....
```

Figure 4: evaluation of lambda expression and application in the function *eval* of mix.