Free Variable as Effect, in Practice

Oleg Kiselyov
Tohoku University
Japan
oleg@okmij.org

Abstract

Variable environment is the time-honored way of making sense of free variables, used in programming language theory as well when writing interpreters and some compilers. Algebraic effects give another way, as was pointed already at HOPE 2017. Although a theoretical curiosity, it may have surprising practical benefits: a new way of writing compilers, with the incremental type-checking, with easy variable usage, leaf function analyses. This work-in-progress report prototypes and illustrates the idea.

1 Introduction

Whenever one writes an interpreter or a compiler, or studies logic, when writing interpreters and some compilers. Algebraic effects Variable environment is the time-honored way of making sense of free variables, used in programming language theory as well when writing interpreters and some compilers. Algebraic effects give another way, as was pointed already at HOPE 2017. Although a theoretical curiosity, it may have surprising practical benefits: a new way of writing compilers, with the incremental type-checking, with easy variable usage, leaf function analyses. This work-in-progress report prototypes and illustrates the idea.

2 Interpreting Languages with Variables

To explain the idea, let’s write an interpreter, which we later turn into a compiler by changing the domain of interpretation. In rough strokes the development, however simplified, actually follows the compiler class.

We start with the simplest interpreter: the source language has only integers and addition. The language should hopefully be clear from the grammar, in the (ocaml) yacc form. We borrowed this example from the ocamlex/ocamlyacc chapter of the OCaml Reference [4, §15.6].

```
exp: INT { int $1 }
    | exp PLUS exp { add $1 $3 }
    | LPAREN exp RPAREN { $2 };
```

The grammar defines the concrete syntax of the language. The semantic actions int and add are arranged in a separate module

```
module type EvalInt = sig
  type repr
  val int : int -> repr
  val add : repr -> repr -> repr
  val observe : repr -> obs
end
```

2.1 Values

The value domain is int (OCaml integers), which is also the domain of interpretation repr. The function observe is invoked after the parsing is finished; it observes the repr value representing the result of the whole program, by printing it. We also made the interpreter to print the (intermediate) results, of each addition expression. As we shall see, it is a good diagnostic for the evaluation order. Let’s add variables. We add the productions

```
| IDENT { var $1 } | LET IDENT EQ exp IN exp { let_ ($2,$4) $6 }
```

The first benefit is the ability to evaluate intermediary expressions and report errors soon, before the whole program is parsed – hence reducing the amount of memory for intermediary data and improving latency. The approach also facilities variable usage and leaf function analyses, indispensable in compilation.

With the following signature, which in effect, defines the abstract syntax.

```
module type LangInt = sig
  type dom
  val int : int * dom
  val rep : repr -> dom
  val observe : dom -> obs
end
```

Free Variable as Effect, in Practice

The complete code accompanying the paper is available at https://okmij.org/ftp/Computation/var-effect/

1The complete code accompanying the paper is available at https://okmij.org/ftp/Computation/var-effect/

2as well as memory requirements: deferring a computation needs memory to store what is to compute later.

3Here $>$ is left-to-right function composition.
2.1 Variable as an Effect

When dealing with expressions like \((1+2)+x\), we need to know what value corresponds to \(x\). We can just ask. The meaning of an expression is then either an answer \(A(v)\), or a question \(Q(n,k)\) about the value of the variable \(n\), to be continued as \(k\), perhaps asking further questions until the final answer. We hence introduce the following variable effect (which is the Free monad implementation of the Reader effect, and entirely standard):

```haskell
module VarEff = struct
  type name = string
  type δ t = A of δ | Q of name * (δ → δ t)
  let ans : δ → δ t = fun v → A v
  let var : name → δ t = fun n → Q(n,ans)

  let lift : (δ → δ t) → (δ t → δ t) = fun e → fun e2 →
    match e with
    | (A v1, A v2) → A (op v1 v2)
    | (Q (n,k), e2) → Q (n, (fun v → lift2 (op (k v) e2))
    | (e1, Q (n,k)) → Q (n, (fun v → lift2 op e1 (k)))

  let rec lift2 : (δ → δ t → δ t) → (δ t → δ t → δ t) = fun e1 e2 →
    match (e1,e2) with
    | (A v1, A v2) → A (op v1 v2)
    | (Q (n,k), e2) → Q (n, (fun v → lift2 (op (k v) e2))
    | (e1, Q (n,k)) → Q (n, (fun v → lift2 op e1 (k)))

let handle_var : (δ → δ t → (name → δ option) → δ t → δ t = ...
let letv : name * δ → δ t = fun n → ((fun v → handle_var ans (function n' when n=m → Some v | _ → None)
let top_hand : δ t → δ = function A v → v
```

A binary operation on two expressions lift2 op checks to see if both operands have the answer. If so, the operation op can be performed right away. Otherwise, lift2 propagates operand’s questions. Eventually, the questions have to be answered, which is the job of a handler. The handler handle_var is the mapping/fold over the denotation (δ t tree). Its particular instance letv replies to questions only about the given name, propagating all others. The domain of interpretation is now dom VarEff.t, to which the semantic functions are lifted:

```haskell
module EvalEff = struct
module V = VarEff
type dom = EvalInt.dom
type repr = dom V.t

let int = EvalInt.int ⊳ V.ans
let add = V.lift2 EvalInt.add
let var = V.var
let rec lift2 : (DOM → DOM → DOM) → (DOM → DOM) → (DOM → DOM) →
  let repr : name → repr → repr → repr = fun (n,b) body →
    V.lift (fun v → V.letv (n,v) body)

  let let_ : name → repr → repr → repr = fun (n,b) body →
    V.lift (fun v → V.letv (n,v) body)
```

As expected, let_ acts as a handler, answering questions about its bound variable, and propagating all other questions up. One may show, using the technique in [3], that EvalEff.repr has the same equational theory as EvalInt.repr – that is, EvalEff is extensionally equivalent to EvalInt. Still, ",
"(1+2)+x/n" and "x+(1+2)+3" now print the result of interpreting 1+2 right away, without waiting for the whole program to be parsed. Furthermore, when we enter the program
The ability of EvalEff offers further opportunities for optimization: if the body of
a let-expression has A v as its interpretation (denotation) – that
is, not a question – the body has not needed the value of the bound
variable. We have hence come upon an easy way to determine the
usage of bound variables, which is valuable in compilation, as we
shall see in the next section.

The variable-as-effect approach scales up to functions – as was
in effect shown already in [1]. Here are two sample programs:

```
let y = let x = 1 + 2
     in x + x + 3 in
y + 1;
```

we not only see 1+2 being evaluated right away, but also x+x+3 be-
ing evaluated as soon as it has been parsed, at the end of the second
line. Questions about local variables can therefore be answered
quickly, without waiting for the whole program be parsed.

EvalEff offers further opportunities for optimization: if the body
of a let-expression has A v as its interpretation (denotation) – that
is, not a question – the body has not needed the value of the bound
variable. We have hence come upon an easy way to determine the
usage of bound variables, which is valuable in compilation, as we
shall see in the next section.

The variable-as-effect approach scales up to functions – as was
in effect shown already in [1]. Here are two sample programs:

```
let x = 1 in
let fun f(y) = x + y in
let x = 2 in f(2)
```

Since a variable dereference is an effect, to be handled by a dynami-
cally enclosed handler, one may wonder if we are really implement-
ing lexical binding, with some work. Generally, a mechanism to
capture the current dynamic environment is needed. The current
implementation uses a simpler approach: handling the body of a
function in the handling environment of its definition rather than
of its invocation. Therefore, both sample programs evaluate to 3.

3 Compilation

The ability of EvalEff to evaluate as soon as possible, without wait-
ing for the whole program to be parsed is especially valuable in
compilation, where it translates to reporting type and other errors
early and reducing memory footprint. There is another benefit,
hinted earlier: the ease of variable use analyses, which are needed
for memory/register allocation. This section demonstrates both
benefits.

First, we turn our interpreter into a compiler, to Wasm. We change
the interpretation domain from int to a sequence of Wasm
instructions that leave the int result on the stack.

```
let x = 1
let fun f(y) = x + y
let x = 2 in f(2)
```

```
module EvalInt_wasm = struct
  type dom = Wasm.instr
  type repr = dom

  let int = Wasm.I32.const
  let add x y = Wasm.(exp [x; y; I32.add])

  type obs = unit
  let observe x =
    let open Wasm in
```

wasm_module [func <result:32 [x]] >> observe end

We rely on the module Wasm: tagless-final embedding of Wasm.

The new EvalInt_wasm is quite like EvalInt, structurally. It interpre-
ts "1+2+3" as:

```
i32.const 1
i32.const 2
i32.add
```

Just as we lifted EvalInt to EvalEff in §2.1, we lift Eval_wasm; the
result, to be called Eval_var, is EvalEff with EvalInt replaced with
Eval_wasm. One may now compile programs with local variables;
for example,

```
let x=10+11 in 1+x+x+3
```

produces:

```
i32.const 1
i32.const 10
i32.const 11
i32.add
i32.add
i32.add
i32.add
i32.add
```

The variable x turns out substituted with its bound expression: the
let-binding got inlined. One should not be too surprised: after all,
variables are like named ‘holes’ in the domain, with let-expressions
telling how to fill the holes. Such behavior of let-expressions –
effecting sharing in the compiler rather than in the object code – is
well-known in code generation [5].

To properly compile let-expressions, allocating storage (Wasm
locals) for bound variables, we lift Eval_var one more time. In
other words, we generate Wasm with ‘holes’, to be filled with the
names of the allocated Wasm locals. The allocation is performed
after a let-expression is compiled and the variable usage in its body
is determined. Strictly speaking, the compilation becomes two-pass.
However, the first pass generates as much Wasm code as possible.
Local let-expressions can even be compiled entirely before the end
of parsing of the whole program.

The let-handler is particularly notable:

```
let ltv : name * dom -> repr -> repr = fun (n,v) b ->
  let cnt = ref 0 in
  let vars = ref [] in
  let lkp = function
    | n' when n = n' -> incr cnt; Some (V.var n)
    | n' -> if List.mem n' vars then () else vars := n' :: !vars; None
  in
  let ret res =
    if !cnt = 0 then V.ans res (- no need to allocate anything -)
    else if !cnt = 1 then V.ans (Eval_var(let_ (n,v) res) (- inline -)
else
    (request allocation, reporting n and the list of alive,
    hence conflicted variables -)
  in
  V.handle ret lkp b
```

As the handler answers questions about its bound variable, it counts
them. At the end, it knows how many times the bound variable has
been accessed. If zero, there is no need to allocate storage for the
variable. (If the source language has no side effects, as ours currently,
we may even skip compiling the bound expression). If the variable
was used only once, we substitute it with the bound expression,
using Eval_var’s let-machinery to do the substitution. Again, no
storage allocation is needed. The ltv-handler also watches for other
variable requests, and learns of all free variables in its managed
expression. Their list is reported to the allocator: these are conflicts,
I.e., their storage must be disjoint. We thus obtain all the information

---

1 One can see that for themselves by compiling and running the code in the directories
step2 and step3 in the accompanying code. The former implements the environment
and the latter effect semantics for variables.

2 However, straightened-out let-expressions are right-associated. Therefore, their pars-
ing finishes only at the end of the program.

3 See step4 in the accompanying code.

4 See the directory wasm in the accompanying code.

5 See step7, in particular, eval.ml in that directory.
(variable usage and conflicts) needed for storage allocation; see the source code for details.

For example, the program

\[
\text{let } x = 1 + 2 \text{ in let } y = x + 1 \text{ in let } z = y + x \text{ in } z + z + y
\]

compiles to the following Wasm module

\[
\text{(module (func (export "start" ) (result i32 ) (local $t_1 i32) (local $t_2 i32 )
(i32.const 1) (i32.const 2) i32.add local.set $t_1 local.get $t_1
(i32.const 1) i32.add local.set $t_2 local.get $t_2 local.get $t_1 i32.add
local.set $t_1 local.get $t_1 local.get $t_1 add local.set $t_2 i32.add local.get $t_2 i32.add))}
\]

The variables \(x\) and \(z\) share the same Wasm local \(t_1\).

Let us add functions – for simplicity, second-class top-level functions whose bodies have no free variables aside from the arguments (since functions are second class, their names are distinct from ordinary variable names).\(^\text{10}\) Here is an example:

\[
\text{let fun } f(x) = x + 2 \text{ in let fun } g(x, y) = f(y) + x \text{ in } f(g(1,2))
\]

Since functions may take several arguments, there comes the possibility of applying a function to a wrong number of arguments – which is a type error. We should report it at the compilation time.

The language with top-level second-class functions \(\text{Lang2Fun}\) is the extension of \(\text{LangLet}\) with function calls and function declarations:

\[
\text{module type } \text{Lang2Fun} = \text{sig}
\]

\[
\text{include } \text{LangLet}
\]

\[
\begin{align*}
\text{val} \ \text{call :& name -> repl list -> repr} \\
\text{val} \ \text{fundecl} :& \text{function declaration} \\
\text{val} \ \text{defun} :& \text{name * name list * repr -> fundecl} \\
\text{val} \ \text{defns} :& \text{a sequence of fundecl} \\
\text{val} \ \text{defn_empty} :& \text{defns} \\
\text{val} \ \text{defn_add} :& \text{defns -> fundecl -> defns} \\
\text{val} \ \text{topform} :& \text{topform -> obstr}
\end{align*}
\]

Here, \text{defun} interprets a declaration (the function name, the list of argument names and the function body) as \text{fundecl}. Since functions may only be declared at top-level and may not refer to outside variables, all function declarations have to appear at the beginning of the program, followed by the top-level expression (main program body) – which is what topform signifies. The compilation for function bindings and function calls is not much different from what we have seen for integer-type let-expressions. A question about a function name is answered with its type (i.e., arity) and the Wasm name\(^\text{11}\) (needed to generate the Wasm call instruction). We refer to the accompanying code for details (see the directory step8).

We have claimed that the effect semantics for variable and function names enables incremental type checking and the early reporting of errors. Let us see. First, consider the OCaml code:

\[
\begin{align*}
\text{let } f(x) &= x + 2 \\
\text{let } g(y) &= f(y,1) + y \\
\text{f}(g(1XXX)
\end{align*}
\]

with two problems. On line 2 the function \(f\) is invoked with a wrong number of arguments. Then there is a parse error on line 3. Although it occurs later in the code, it and only it is reported by the OCaml compiler:

\[
3 \mid \text{f}(g(1XXX}
\]

Indeed, an OCaml program must first be completely parsed, and only then type-checked. When writing or refactoring code, however, one would have liked to type check fragments (definitions) as soon as they are finished, before the whole program is completed.

In contrast, if we submit the similar code

\[
\begin{align*}
\text{let fun } f(x) &= x + 2 \\
\text{let fun } g(y) &= f(y,1) + y \\
\text{f}(g(1XXX)
\end{align*}
\]

to our compiler, we get the compilation error about the first problem:

- **Function \(f\) requires 1 arguments but was invoked with 2**

In fact, if we feed the code into the compiler line-by-line, we notice that the error is reported right after the second line is entered – before the third, ill-formed, line is even input.

\[4\] Conclusions

In the environment semantics the meaning of an expression is a function from the environment, which is opaque and cannot be examined. We cannot tell which variables in the environment have actually been used, and how many times. Algebraic effects make the denotation more observable: a handler can watch questions and find out which variables have been asked about, and how many times. Thus we obtain the variable usage analysis in the ordinary course of compilation, almost for free, so to speak.

It remains to be seen how this promise holds for a real compiler for a realistic programming language. I intend to find it out by trying this technique out in the new installment of the compiler class (which is underway).

\[5\] Acknowledgments

I thank Chung-Chieh Shan for helpful discussions, and the reviewers and participants of the HOPE 2023 workshop for many insightful comments.

\[6\] References


