Purely Functional Structured Programming

Steven Obua

Abstract. The idea of functional programming has played a big role in shaping today's landscape of mainstream programming languages. Another concept that dominates the current programming style is Dijkstra's structured programming. Both concepts have been successfully married, for example in the programming language Scala. This paper proposes how the same can be achieved for structured programming and *purely* functional programming via the notion of *linear scope*. One advantage of this proposal is that mainstream programmers can reap the benefits of purely functional programming like easily exploitable parallelism while using familiar structured programming syntax and without knowing concepts like monads. A second advantage is that professional purely functional programmers can often avoid hard to read functional code by using structured programming syntax that is often easier to parse mentally.

1 Introduction

Purely functional programming (PFP) has a chance of becoming very popular for the simple reason that we now have laptops with four cores and more. The promise of PFP is that because there are no side-effects, no destructive updates, and no shared mutable state, partitioning a program into pieces that run in parallel becomes straightforward.

Another consequence of the freedom from impure language constructs is that reasoning about program correctness, both formally and informally, becomes much easier in PFP languages than in, say, imperative languages. Therefore it is not surprising that PFP is popular within the theorem proving community. For example, the source code of the interactive theorem proving assistant Isabelle [10] is mostly written in a purely functional style. Outside of such specialty communities though, PFP clearly has not reached the mainstream yet.

A programming paradigm that pervades today's mainstream is Dijkstra's *structured programming* [2] (SP). Most young programmers even do not know the term structured programming anymore, but anyway still construct their object-oriented programs out of building blocks like **if**-branches and **while**-loops.

Interestingly, the PFP community largely rejects SP because it smells of side-effects, destructive updates, and mutable state, just the things a purely functional programmer wants to avoid. As an example, let us examine the Isabelle (version 2009-2) source code. Discounting blank lines, it consists of about 140000 lines of Standard ML [5] (SML) code. Yet, only ten of those lines use the **while** keyword of SML! Furthermore, five out of those ten lines are part of Isabelle's system level code, and a further three lines stem from the author of

this paper trying to circumvent missing tail-recursion optimization. The reason for this sparse use of **while** is clear: in order to use **while** in SML one must also use reference cells which are the embodiment of the small amount of impurity still left in SML.

The easiest way to make PFP more mainstream might be to make SP, which already is part of the mainstream, an integral part of PFP! This is what this paper is about. Our central tool for such a unification of PFP and SP is the notion of *linear scope*. Linear scope makes heavy use of *shadowing*, therefore we first look at shadowing and its treatment in other languages that draw on functional programming, like Erlang and Scala. We then present the syntax of a toy language called *Mini Babel-17* to prepare a proper playground for the introduction of linear scope. First we concentrate on how linear scope interacts with the sequencing and nesting of statements. From there the extension to conditionals and loops is straightforward. Finally we give a formal semantics for Mini Babel-17 and hence also for linear scope.

2 Shadowing is Purely Functional

Here is how you could code in SML the function $x \mapsto x^4$:

> **fn** x => **let val** x = x * x **in** x * x **end**

There is no doubt that the above denotes a pure function. The fact that the introduction of the variable x via **val** x = x * x *shadows* the previous binding of x in **fn** x might make it look a little bit more unusual than the more common

> **fn** x => **let val** y = x * x **in** y * y **end**,

but of course both denotations are equivalent. Rewriting both functions in De Bruijn notation [3] would actually yield the exact same closed term.

Yet it seems that the conception that shadowing is somehow wrong lies at the heart of why PFP and SP do not overlap in the mind of many programmers.

An instance where shadowing is forbidden in order to obtain a notion of pure variables is the programming language Erlang which features *single-assignment* of variables. Quoting the inventor of Erlang [1, p. 29]:

> When Erlang sees a statement such as X = 1234, it binds the variable X to the value 1234. Before being bound, X could take any value: it's just an empty hole waiting to be filled. However, once it gets a value, it holds on to it forever.

Clearly in Erlang shadowing is a victim of the idea that variables are not just names bound to values, but that *the variables themselves are the state*.

Something similar can be observed in the programming language Scala [6]. Scala combines functional and structured programming in an elegant fashion. But when it comes to integrate *purely* functional programming and SP, Scala does not go all the way: it also forbids shadowing. For example, the following Scala function implements $x \mapsto x^8$,

> (x : Int) => { **val** y = x*x; **val** z = y*y; z*z },

but both of the following expressions are illegal in Scala:

(x : Int) => { **val** y = x∗x; **val** y = y∗y; y∗y },
(x : Int) => { **val** y = x∗x; y = y∗y; y∗y }.

The last expression can be turned into a legal Scala expression by replacing the keyword **val**, which introduces immutable variables, with the keyword **var**, which introduces mutable variables:

(x : Int) => { **var** y = x∗x; y = y∗y; y∗y }.

It might seem that after all, shadowing in Scala is possible! But this is not the case. That **var** behaves differently than shadowing can easily be checked:

(x : Int) => { **var** y = x∗x
                   val h = () => y
                   y = y∗y
                   h() ∗ y }

also implements $x \mapsto x^8$. With shadowing, we would expect above function to implement $x \mapsto x^6$.

3 Syntax of Mini Babel-17

Babel-17 [8] is a new dynamically-typed programming language in the making which is being developed by the author of this paper. One of its main features is that it combines purely functional programming and structured programming, building on the key observation that shadowing is purely functional. For illustration purposes we use a simplified version of a subset of Babel-17, which we call *Mini Babel-17*, as a proposal of how a purely functional structured programming language could look like. An implementation of Mini Babel-17 is available at [9].

A *block* in Mini Babel-17 is a sequence of *statements*:

block $\longrightarrow$ *statement*$_1$
                  $\vdots$
           statement$_n$

Several statements within a single line are separated via semicolons. There are seven kinds of statements:

statement $\longrightarrow$ *val-statement*
              | *assign-statement*
              | *yield-statement*
              | *if-statement*
              | *while-statement*
              | *for-statement*
              | *block-statement*

val-statement $\longrightarrow$ **val** *identifier* = *expression*
val-assign-statement $\longrightarrow$ *identifier* = *expression*
yield-statement $\longrightarrow$ **yield** *expression*
if-statement $\longrightarrow$ **if** *simple-expression* **then** *block* **else** *block* **end**

while-statement $\longrightarrow$ **while** *simple-expression* **do** *block* **end**
for-statement $\longrightarrow$ **for** *identifier* **in** *simple-expression* **do** *block* **end**
block-statement $\longrightarrow$ **begin** *block* **end**

If the last statement of a *block* is a *yield-statement*, then the **yield** keyword may be dropped in that statement.

A *simple-expression* is an expression like it can be found in most other functional languages, i.e. it can be an integer, a boolean, an identifier, an anonymous function, function application, or some operation on *expressions* like function application, multiplication or comparison:

simple-expression $\longrightarrow$
    integer | *boolean* | *identifier*
    | *identifier* => *expression*
    | *expression*$_1$ *expression*$_2$
    | *expression*$_1$ $*$ *expression*$_2$
    | *expression*$_1$ == *expression*$_2$
    $\vdots$

An *expression* is either a *simple-expression* or a statement:

expression $\longrightarrow$ *simple-expression*
          | *if-statement*
          | *while-statement*
          | *for-statement*
          | *block-statement*

4 Linear Scope

Let us gain a first intuitive understanding of Mini Babel-17 before formally introducing its semantics. Here is how you could denote $x \mapsto x^8$ in Mini Babel-17:

    x => **begin val** y = x$*$x; **val** z = y$*$y; z$*$z **end**

This looks pretty much like the Scala denotation of $x \mapsto x^8$ from Section 2. But because Mini Babel-17 is designed so that shadowing of variables is allowed, an equivalent notation is:

    x => **begin val** x = x$*$x; **val** x = x$*$x; x$*$x **end**

The central idea of Mini Babel-17 is the notion of *linear scope*. Whenever an identifier x is in linear scope, it is allowed to rebind x to a new value, and *that rebinding will affect all other* later *lookups of x that happen within its normal lexical scope*. The rebinding is done via a *val-assign-statement*.

The linear scope of a variable is contained in the usual lexical scope of that variable. The linear scope of a variable x starts

- in the statement after the *val-statement* that defines x, or
- in the first statement of an anonymous function that binds x as a function argument, if the body of that function is a block, or

– in the first statement of the block of a *for-loop* where x is the identifier bound by that loop.

It continues throughout the rest of the block unless a new linear scope for x starts. It does extend into nested blocks and statements, but not into *simple-expressions*. The reason for this is that blocks and statements are ordered sequentially, but there is no natural order for the evaluation of the components of a *simple-expression*.

Using the linear scope rules of Mini Babel-17, the above function can also be encoded as

x => **begin** x = x∗x; x = x∗x; x∗x **end**

If there are no nested blocks involved, then linear scope is no big deal. It is just a fancy way of saying that when in a *val-statement* the variable being defined shadows a previously defined variable, often it is ok to drop the *val* keyword, effectively turning a *val-statement* into a *val-assign-statement*.

But with nested blocks, linear scope becomes important:

val x = 2
begin
    val y = x∗x
    val x = y
end
x+x
	val x = 2
	begin
	    val y = x∗x
	    x = y
	end
	x+x
		val x = 2
		begin
		    val y = x∗x
		    val x = 0
		    x = y
		end
		x+x

evaluates to 4	evaluates to 8	evaluates to 4

The left and right programs both evaluate to 4 because the **begin** ... **end** block is superfluous as none of its statements have any effect in the outer scope. The middle program evaluates to 8, though, because the rebinding x = y effects all later lookups in the lexical scope of that x that has been introduced via **val** x = 2, and x+x certainly is such a later lookup.

Maybe the rules of linear scope sound confusing at first. But they really are not. Just replace in your mind in the above three programs the **val**s by **var**s and view them as imperative programs. What value would you assign now to each program?

Let us also recode the last Scala expression of Section 2 as a Mini Babel-17 expression:

```
x => begin
        val y = x∗x
        val h = dummy => y
        y = y∗y
        h 0 ∗ y
    end
```

Mini Babel-17 is purely functional, therefore the value of h is of course not changed by the rebinding y = y∗y which affects only *later* lookups of y. Thus the above expression implements $x \mapsto x^6$, not $x \mapsto x^8$.

5　Conditionals and Loops

Conditionals and especially loops are the meat of structured programming. With linear scope, they are easily seen also as part of purely functional programming. All we need to do is to apply linear scoping rules to the nested blocks that the *if-*, *while-* and *for-statements* consist of. For example, this is how you can encode the subtraction based Euclidean algorithm for two non-negative integers in Mini Babel-17:

```
a => b =>
    if a == 0 then
      b
    else
      val a = a
      while b != 0 do
        if a > b then
          a = a − b
        else
          b = b − a
        end
      end
      a
    end
```

Note the line **val** a = a which on first sight seems to be superfluous. But while the linear scope of b encompasses the whole function body, the linear scope of a does not, because linear scope does not extend into *simple-expressions*. If Mini Babel-17 had pattern matching, the line **val** a = a could be avoided by starting the function definition with

```
[a,  b] =>
    ⋮
```

instead.

6　Semantics of Mini Babel-17

In this section we define an operational semantics for Mini Babel-17 by building a Mini Babel-17 interpreter written in Standard ML[1].

First we represent the grammar of Mini Babel-17 as SML datatypes:

```
datatype block = Block of statement list
and statement =
        SVal of identifier  ∗ expression
      | SAssign of identifier  ∗ expression
      | SYield of expression
      | SIf of simple_expression ∗ block ∗ block
```

---

[1] The original SML sources of the interpreter and all Mini Babel-17 programs of this paper are available online [8].

```
            | SWhile of simple_expression * block
            | SFor of  identifier  * simple_expression * block
            | SBlock of block
   and expression =
            ESimple of simple_expression
            | EBlock of statement
   and simple_expression =
            EInt of int   | EBool of bool | EId of  identifier
            | EFun of identifier * expression
            | EBinOp of (value * value −> value) *
                        expression * expression
   and identifier  = Id of string
```

Note that function application, multiplication, comparison and so on are all described via the EBinOp constructor by providing a suitable parameter of type value * value −> value. The type value represents all values that can be the result of evaluating a Mini Babel-17 program:

```
datatype value = VBool of bool | VInt of int
                | VFun of value −> value
                | VList of value  list
```

Mini Babel-17 wants to be both purely functional and structured; the most important ingredients of a purely functional program are expressions; the most important ingredients of an SP program are blocks and statements. This dilemma is resolved by treating statements as special expressions.

The interpreter defines the following evaluation functions:

```
eval_b  : environment −> block −> environment * value list
eval_st  : environment −> statement −> environment * value list
eval_e  : environment −> expression −> value
eval_se  : environment −> simple_expression −> value
```

The evaluation of blocks and statements yields lists of values instead of single values, the block

begin yield 1; **yield** 2; 3 **end**

for example evaluates to $[1, \ 2, \ 3]$.

Consider the following Mini Babel-17 program:

val x = 0
begin x = 1; x **end** * **begin val** x = x + 2; x **end**

It does not obey the linear scoping rules of Mini Babel-17 because x is not in linear scope in the *val-assign-statement* x = 1. In such a situation, the exception Illformed is raised during evaluation. Furthermore, an exception TypeError is raised when for example the condition of an if-statement evaluates to a list instead of a boolean. Note by the way that the program

val x = 0
begin val x = 1; x **end** * **begin val** x = x + 2; x **end**

is perfectly fine and evaluates to 2.

What does the environment look like? It is actually split into two parts, one part for those identifiers that have linear scope, and one part for identifiers that don't. The nonlinear part is a mapping from identifiers to values, the linear part a mapping from identifiers to reference cells of values. Both parts can be described by the polymorphic type 'a idmap:

```
type idmap = (string * 'a) list
fun lookup [] _ = raise Illformed
  | lookup ((t, x)::r) (Id s) =
      if t = s then x else lookup r (Id s)
fun remove [] _ = []
  | remove ((t,x)::r) (Id s) =
      if t = s then r else remove r (Id s)
fun insert m ((Id s),x) = (s,x)::(remove m (Id s))
```

The type of environments is then introduced as follows:

```
type environment = value idmap * (value ref) idmap
fun deref [] = [] | deref ((s, vr)::m) = ((s,!vr)::(deref m))
fun freeze (nonlinear, linear) = (nonlinear@(deref linear), [])
fun bind (nonlinear, linear) (id,value) =
  (remove nonlinear id, insert linear (id, ref value))
fun rebind (env as (_, linear)) (id, value) =
  (lookup linear id := value; env)
```

Note that bind returns a new environment, and rebind returns the same environment with a mutated linear part. The function freeze turns all mutable linear bindings into immutable nonlinear ones.

Now we can give the definition of all evaluation functions:

```
fun eval_b env (Block []) = (env, [])
  | eval_b env (Block (s::r)) =
    let
      val (env', values_s) = eval_st env s
      val (env'', values_r) = eval_b env' (Block r)
    in (env'', values_s @ values_r) end
and eval_nestedb env b =
    let
      val (_, values) = eval_b env b
    in (env, values) end
and eval_st env (SVal (id, e)) =
    let
      val value = eval_e env e
    in (bind env (id, value), []) end
  | eval_st env (SAssign (id, e)) =
    let
      val value = eval_e env e
    in (rebind env (id, value), []) end
  | eval_st env (SYield e) =
    let
      val value = eval_e env e
    in (env, [value]) end
```

```
    | eval_st  env (SBlock b) = eval_nestedb env b
    | eval_st  env (SIf (cond, yes,  no)) =
       (case eval_se env cond of
          VBool true => eval_nestedb env yes
        | VBool false => eval_nestedb env no
        | _ => raise TypeError)
    | eval_st  env (loop as SWhile (cond, body)) =
       (case eval_se env cond of
          VBool true =>
            let
              val (_, values_1) = eval_b env body
              val (_, values_2) = eval_st env loop
            in (env, values_1 @ values_2) end
        | VBool false =>
            (env, [])
        | _ => raise TypeError)
    | eval_st  env (SFor (id,  list , body)) =
       (case eval_se env list  of
          VList L => eval_for env id body L
        | _ => raise TypeError)
and eval_for env id body [] = (env, [])
  | eval_for  env id body (x::xs) =
     let
       val (_, values_1) = eval_b (bind env (id,x))  body
       val (_, values_2) = eval_for  env id body xs
     in (env, values_1@values_2) end
and eval_e env (ESimple se) = eval_se env se
  | eval_e  env (EBlock s) =
      (case eval_b env (Block [s])  of
          (_, [a])  => a
        | (_, L) => VList L)
and eval_se env se  = eval_simple (freeze  env) se
and eval_simple env (EInt i) = VInt i
  | eval_simple  env (EBool b) = VBool b
  | eval_simple  env (EBinOp (f, a, b)) =
      f (eval_e env a,  eval_e  env b)
  | eval_simple (nonlinear,  _) (EId id) =
      lookup nonlinear id
  | eval_simple  env (EFun (id, body)) =
      VFun (fn value =>
              eval_e (bind env (id,  value)) body)
```

Here is the evaluation function that computes the meaning of a Mini Babel-17
program, i.e. of a block:

```
eval : block -> value
fun eval prog = snd (eval_e  ([],   [])  (EBlock (SBlock prog)))
```

It is straightforward how to extract from above evaluation functions a wellformedness-
criterion such that if a Mini Babel-17 program is statically checked to be well-

formed according to that criterion, no Illformed exception will be raised during the evaluation of the program:

```
val VALUE = VInt 0
fun check_b env (Block []) = env
  | check_b env (Block (s :: r)) =
        check_b (check_st env s) (Block r)
and check_st env (SVal (id, e)) =
      (check_e env e; bind env (id, VALUE))
  | check_st env (SAssign (id, e)) =
          (check_e env e; rebind env (id, VALUE))
  | check_st env (SYield e) = (check_e env e; env)
  | check_st env (SBlock b) = (check_b env b; env)
  | check_st env (SIf (cond, yes, no)) =
      (check_se env cond;
       check_b env yes; check_b env no; env)
  | check_st env (loop as SWhile (cond, body)) =
      (check_se env cond; check_b env body; env)
  | check_st env (SFor (id, list, body)) =
      (check_se env list;
       check_b (bind env (id, VALUE)) body; env)
and check_e env (ESimple se) = check_se env se
  | check_e env (EBlock s) =
          (check_b env (Block [s]); ())
and check_se env se = check_simple (freeze env) se
and check_simple env (EInt i) = ()
  | check_simple env (EBool b) = ()
  | check_simple env (EBinOp (f, a, b)) =
      (check_e env a; check_e env b)
  | check_simple (nonlinear, _) (EId id) =
      (lookup nonlinear id; ())
  | check_simple env (EFun (id, body)) =
          check_e (bind env (id, VALUE)) body
fun check prog = check_e ([], []) (EBlock (SBlock prog))
```

The function *check* terminates because it is basically defined via primitive recursion on the structure of the program. Furthermore, the set of calls to *lookup* generated during an execution of *check prog* is clearly a superset of the set of calls to *lookup* generated during the execution of *eval prog*. Therefore, if *check prog* does not raise an exception *Illformed*, then neither will *eval prog*.

7 Loops or Functionals?

With Mini Babel-17, you can freely choose between a programming style that uses loops and a programming style that puts its emphasis on the use of higher-order functionals. If you have an imperative background, you might start out with using loops everywhere, and then migrate slowly to the use of functionals like *map* or *fold* as your understanding of functional programming increases.

But even after your functional programming skills have matured, you might still choose to use loops in appropriate situations. Let us for example look at a function that takes a list of integers $[a_0, \ldots, a_n]$ and an integer x as arguments and returns the list

$$[q_0, \ldots, q_n] \quad \text{where} \quad q_k = \sum_{i=0}^{k} a_i \, x^i$$

The implementation in Mini Babel-17 via a loop is straightforward, efficient and even elegant:

```
m => x => begin
            val y = 0
            val p = 1
            for a in m do
              y = y + a*p
              p = p * x
              yield y
            end
          end
```

8 Related Work

We have already mentioned how Scala also combines structured programming with functional programming, but fails to deliver a combination of structured programming and *purely* functional programming. Actually, it should be possible to conservatively extend Scala so that linear scope for variables defined via **val** is supported.

The work done on monads in the purely functional programming language Haskell [4] has a superficial similarity with the work done in this paper. With monads it is possible to formulate sequences of (possibly shadowing) assignments, and with the help of monad transformers even loops can be modeled. But in order to understand and effectively use monads a solid background in functional programming is useful, if not even required; linear scope on the other hand is understood intuitively by programmers with a mostly imperative background, because Mini Babel-17 programs can look just like imperative programs and do not introduce additional clutter like the need for lifting. Actually, in Haskell monads are used to limit the influence of mutable state to a confined region of the code that can be recognized by its type; the work in this paper has the entirely different focus of trying to merge the structured and purely functional programming style as seamlessly as possible.

This work is not directly connected to work done on linear or uniqueness types [7]. Of course one might think about applying uniqueness typing to Mini Babel-17, but Mini Babel-17 itself is dynamically-typed and its values are persistent and can be passed around without any restrictions.

9   Conclusion

The current separation between SP and PFP is an artificial one. There is no good reason anymore why SP should not be used where appropiate for the sequential parts of a purely functional program except the personal preference of the programmer. The purpose of Mini Babel-17 is to show the importance of linear scope for unifying SP and PFP. Babel-17 incorporates further important features for purely functional and structured programming like mutually recursive functions, pattern matching, exceptions, objects, memoization, concurrency, laziness, and more syntactic sugar.

References

1. Armstrong. *Programming Erlang: Software for a Concurrent World.* Pragmatic Bookshelf, 2007.
2. Dahl, Dijkstra, Hoare. *Structured Programming.* Academic Press, London, 1972.
3. De Bruijn, Govert. *A survey of the project AUTOMATH.* In Hindley and Seldin (editors), *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, Academic Press, 1980.
4. Jones, Wadler. Imperative Functional Programming. *ACM Symposium on Principles of Programming Languages*, 1993, pp 71-74.
5. Milner; Tofte, Harper, MacQueen. *The Definition of Standard ML.* MIT Press, 1997.
6. Odersky, Spoon, Venners. *Programming in Scala.* Artima Press, 2008.
7. Wadler. Linear types can change the world! *In M. Broy and C. Jones, editors, Programming Concepts and Methods.* North Holland, Amsterdam, 1990.
8. Babel-17. `http://www.babel-17.com`
9. Mini Babel-17. `http://www.babel-17.com/Mini-Babel-17`
10. Isabelle. `http://isabelle.in.tum.de`
11. Delany. *Babel-17.* Ace Books, 1966.