Coding a Lisp Interpreter in Shen: a Case Study

Mark Tarver

Shen is a new generation programming language utilising optional static typing based on sequent calculus. The notation and the power of the system presents a wide array of challenges and opportunities for programmers to design their programs. We show how the use of sequent calculus can create the facility for producing natural and type secure programs as well as the challenges and temptations that arise from being given this power. We present our study from a real-life example coded by a Shen programmer ***without*** *types and step-by-step analyse what is needed to add types to the program, noting how Shen can be used to formulate a natural and powerful type system for this work.*

An Introduction to Shen

Shen [19] is a functional language introduced in 2011 arising from development work on Qi [18]. The goal of the Shen project was to reproduce and extend the functionality of the Qi language within a RISC Lisp, Kλ [20]. Kλ consists of 46 primitive functions in which all Shen functions are translated and which, if defined within any host platform, will serve to port Shen to that host. The successful achievement of this aspect of the project was demonstrated within 18 months of the publication of Shen, by ports to Common Lisp, Scheme, Javascript, Clojure, Java,, the JVM, Python, C++ and Ruby.

Shen is a complete and powerful functional language incorporating a logic engine based on Prolog [15], a pattern-directed functional notation, a programmable type system based on sequent calculus, a macro package and an advanced compiler-compiler all of which were used to bootstrap the Shen kernel in less than 5000 lines of code. The language has been used to construct an entire web framework [14] as well as a large number of Github [3] and library applications [17].

Programmers working in Shen received the language well, noting that it was a clear and compact language, but struggled with the type notation based on sequent calculus. Part of the goal of this paper is to examine why this problem arises and to emphasise how Shen's powerful and optional typing system brings out certain tensions inherent in the programming community in respect of working with and without static typing.

A Mini-Lisp Interpreter in Shen

The following program was submitted by Racketnoob [12], an active member of the Shen news group, under the title 'A Mini-Lisp Interpreter in Shen'.

```
(define extend-env
  Var Val Env -> [[Var | Val] | Env])

(define get-from-env
  Var [] -> (error "Cannot find ~A in the environment!" Var)
  Var [[Var | Val] | _] -> Val
  Var [_ | Rest] -> (get-from-env Var Rest))

(define lambda?
  [lambda | _] -> true
  _ -> false)
```

```
(define closure?
  [closure | _] -> true
  _ -> false)

(define closure-var
  [closure [lambda Var | _] _] -> Var)

(define closure-body
  [closure [lambda _ Body] _] -> Body)

(define closure-env
  [closure [lambda _ _] Env] -> Env)

(define interp
  N Env -> N where (number? N)
  Lam Env -> [closure Lam Env] where (lambda? Lam)
  Var Env -> (get-from-env Var Env) where (symbol? Var)
  [Op M N] Env -> (Op (interp M Env) (interp N Env)) where (element? Op [+ - * /])
  [Op M N] Env -> (if (Op (interp M Env) (interp N Env)) 1 0)
                      where (element? Op [= < > <= >=])
  [let Var E1 E2] Env -> (interp E2 (extend-env Var (interp E1 Env Var) Env))
  [if Test ET EF] Env -> (let T (interp Test Env)
                           (if (not (= T 0)) (interp ET Env) (interp EF Env)))
  [Exp1 Exp2] Env -> (let Cl (interp Exp1 Env)
                        (if (closure? Cl) (interp (closure-body Cl)
                                                  (extend-env (closure-var Cl)
                                                              (interp Exp2 Env)
                                                              (closure-env Cl)))
                           (error "~A is not a closure!" Cl))))
```

Figure 1 The untyped Shen program for a miniLisp interpreter

The program, as shown, is slightly simplified compared to the original program, in that the author was interested in using circular lists. Nevertheless, in all other respects, this is very much the original code with very little input from the author of this paper. The main function is `interp`. The leading line

```
N Env -> N where (number? N)
```

states that given inputs `N` and `Env`, that `N` is to be returned where `N` is a number. Capitalised symbols are variables, as in Prolog. The second line

```
Lam Env -> [closure Lam Env] where (lambda? Lam)
```

States that where `Lam` is a lambda expression, a closure is to be returned. The third line

```
[Op M N] Env -> (Op (interp M Env) (interp N Env)) where (element? Op [+ - * /])
```

states that if the expression is a three element list headed by an `Op` that occurs in the list `[+ - * /]`, then the elements `M` and `N` are to be interpreted and the `Op` applied to the result.

The next line

```
[Op M N] Env -> (if (Op (interp M Env) (interp N Env)) 1 0)
                    where (element? Op [= < > <= >=])
```

states that if the `Op` is a boolean comparison, that it is applied to the interpretation of its arguments and if the result is true then 1 is returned otherwise 0. Here 1 and 0 are being used as booleans, hinting that the interpreter is designed to be numeric.

The next line deals with local assignments.

```
[let Var E1 E2] Env -> (interp E2 (extend-env Var (interp E1 Env Var) Env))
```

Here the `let` is a constant, since it is not capitalised. The action is to extend the environment by the local assignment and interpret `E2` by the extended environment. The next line

```
[if Test ET EF] Env -> (let T (interp Test Env)
                            (if (not (= T 0)) (interp ET Env) (interp EF Env)))
```

suspends strict evaluation and forks control depending on the results of `Test`. The last rule deals with applications in the manner of lambda calculus i.e. in curried form.

The Challenge of Types

The coding is clear and reflects fairly well the clarity of the Shen notation. Having submitted the code, it was suggested that it might be a good idea to add types to the system. Racketnoob's response [12] was interesting and has been echoed by programmers working in Common Lisp who are confronted with working in ML and Haskell.

No, I didn't think about writing typed version of this program. You know, I'm not "disciplined" kind of person so type discipline is (for now) pretty foreign to me. :) I have the strange feeling that types hampers programmer's creativity.

The underlined sentence is a compact summary of the reluctance that programmers often feel in migrating to statically typed languages – that they are losing something, a degree of freedom that the writer identifies as hampering creativity.

Is this true? I will argue, to a degree – yes. A type checker for a functional language is in essence, an inference engine; that is to say, it is the machine embodiment of some formal system of proof. What we know, and have known since Godel's incompleteness proof [9] [11], is that the human ability to recognise truth transcends our ability to capture it formally. In computing terms our ability to recognise something as correct predates and can transcend our attempt to formalise the logic of our program. Type checkers are not smarter than human programmers, they are simply faster and more reliable, and our willingness to be subjugated to them arises from a motivation to ensure our programs work.

That said, not all type checkers are equal. The more rudimentary and limited our formal system, the more we may have to compromise on our natural coding impulses. A powerful type system and inference engine can mitigate the constraints placed on what Racketnoob terms our creativity. At the same time a sophisticated system makes more demands of the programmer in terms of understanding. The invitation of adding types was thus taken up by myself, and the journey to making this program type secure in Shen emphasises the conclusion in this paragraph

Working in Sequent Calculus

In Shen, types are defined in the notation of a single-conclusion sequent calculus. Sequent calculus has a long tradition in logic stemming from Gentzen's foundational work in the area. It was taken up by many authors (e.g [6], [21]) to specify formal systems including type theories. In 1988 Torkel Franzen [8] noted that there was a conceptual connection between intuitionistic sequent calculus and Prolog [15]. It is in fact feasible to regard

Prolog as an animation of a single conclusion sequent calculus with certain 'impure' features necessary for practical programming.

It was the author's insight that these two ideas could be combined together to provide a practical means for specifying the type disciplines of functional programs by compiling formalisations of types in sequent calculus into efficient logic programs which would drive the process of type checking [16].

In sequent calculus systems of logic it is usual to find that the logical connectives of the system are explained by the proof rules for reasoning about them. This proof theoretic account of the meaning of logical connectives exists as a counterpoint to the model-theoretic account and it is the task of soundness and completeness proofs to show these accounts are not antagonistic. The task of the formalisation of the inference rules is performed by giving right and left rules for each connective χ. The left rules (L rules) state what can be proved *from* assumptions whose main connective is χ. The right rules (R rules) state how to prove conclusions whose main connective is χ. Thus in the case for the proof rules for v there are two right rules.

P; Q;
(P v Q); (P v Q);

and one left rule corresponding to a proof by cases in mathematics

P >> R;
Q >> R;
(P v Q) >> R;

In certain cases the L and R rules are essentially symmetrical; the rule for dealing with a conjunction is to split it into the component parts.

P; Q; P, Q >> R;
(P & Q); (P & Q) >> R;

Reeves and Clarke [13] introduce a notational abbreviation for this case; the double underline.

P; Q;
═══
(P & Q);

Such a rule is referred to in Shen as an LR rule.

The First Type Error

An environment in Racketnoob's program is an association list comprised of variables and their bindings and the function `get-from-env` is a simple lookup function. We can identify an environment then with a list of bindings. In defining datatypes in Shen it is common to follow the pattern set down in classical logic. An environment is defined in Shen as a datatype fixed by an LR rule.

```
(datatype env

  ____________
  [] : env;

  Var : symbol; Val : expr; Env : env;
  ====================================
  [[Var | Val] | Env] : env;)
```

The structure of this rule reflects the recursive nature of the datatype; a base case (embodied in the first rule) states that an empty list is an environment and a recursive case (embodied in an LR rule) states the recursive condition. The `====` is the keyboard substitute for the LR notation of Reeves and Clarke.

Shen is an explicitly typed language in the manner of Hope [5]; it requires functions to be annotated with their types.[1] The initial attempt to sign the `get-env` function with the type `symbol --> env --> expr` resulted in a type error. This is what was entered

```
(define get-from-env
  {symbol --> env --> expr}
  Var [] -> (error "Cannot find ~A in the environment!" Var)
  Var [[Var | Val] | _] -> Val
  Var [_ | Rest] -> (get-from-env Var Rest))
```

It was assumed that variables were symbols (a primitive type in Shen) and these variables were associated with expressions (`expr`). No definition was supplied of an `expr,` but the information given was sufficient to be able to deal with this function in isolation from the rest of the program. Shen returned a type error

```
type error in rule 3 of get-from-env
```

It was not clear what was wrong with the line `Var [_ | Rest] -> (get-from-env Var Rest)` and so the type checker was traced. The trace package for the type checker prints the process of type checking as a proof in sequent calculus driven by backward chaining. The type checker faltered on the following sequent.

```
?- (cons &&Parse_Y612 &&Rest) : env

1. &&Parse_Y612 : Var115
2. &&Rest : Var118
```

It is clear that the problem arises over `[_ | Rest].` Understanding the source of this error requires understanding some of the inner essentials of Shen and the reasoning embodied in the type checker.

---

[1] The avoidance of implicit typing in the manner of ML [24] and Haskell [22] arises because Shen allows far more freedom in the formulation of type theories than either of these two languages. Without the constraint of user direction, the search spaces involved in type checking would often get too big.

An ML type checker requires that the programmer use type constructors in defining types, and these constructors act as flags for the type checker to find its way through the code. The view that Hindley-Milner languages do not depend on any programmer guidance is misleading. The ML programmer provides that information in the way he sets up his types. The Shen programmer uses the ubiquitous list without using special tags and the guidance is provided in signing the function. Arguably this facilitates the transition from untyped to typed code because the actual body of code does not have to be materially changed to incorporate types.

The Isomorphism Requirement and Mnemonic Recognition

Between the formal definition of the types in sequent calculus and the developer's program there has to exist some form of 'glue' that connects the two together algorithmically. To the novice Shen programmer this glue often appears invisible and Shen automagically certifies the program or draws attention to places where there is an error. However viewing Shen automagically (in the way that novice Prolog programmers sometimes view Prolog automagically) will lay up trouble for programmers who spend time in typed Shen without understanding the mechanism of the Shen type checker. This algorithm T*, along with associated correctness proofs, is described in detail in [18] and [19]. What follows here is a precis.

Given to prove f : A_1 `-->` ... A_n for a function f; for each rule R in f of the form P_1 ...P_{n-1} `->` E Shen attempts to show two properties.

1. That the patterns P_1,...P_{n-1} to the left of the `->` in R fit the types A_1 ...A_n-$_1$ described in the sequent calculus formalisation. This is the *integrity condition*.
2. That assuming $P_1 : A_1$ ... $P_{n-1} : A_{n-1}$ that $E : A_n$ can be proved. This is the *correctness condition*.

Whereas condition 2. is clear, the concept of a pattern 'fitting' a sequent calculus formalisation is an intuitive one. What is required is some way of fleshing out this intuitive notion of 'fitting'. The situation here is comparable to the vague intuitive notion of computability that existed prior to Turing's [23] definition of computability in 1936. We cannot *prove* formally that Turing's account of computability meets our intuitive concept because formal proof begins only when our intuitions have been given shape. However the formal account should not go contrary to our intuitions.

The concept of 'fitting' can be fleshed out more clearly by saying that it has to be *possible* for the P_i to inhabit A_i and Shen interprets the modal auxiliary by requiring that there be some assignment of types to the variables in P_i from which $P_i : A_i$ can be proved. In turn this requirement is posed as a sequent problem - where V_1,..,V_m are all the variables in P_i, and T_1 ...T_m are fresh type variables, prove $V_1 : T_1$,.. ,$V_m : T_m$ >> $P_i : A_i$. When this interpretation is played out with a unification-driven logic engine, it works out as a good model for 'fitting'.

One consequence of this interpretation is that there has to be a structural isomorphism between the patterns in the body of the program and the patterns used to define types. If for instance, a *k*-element list is used to define a type *A* in sequent calculus and a pattern *P* is arraigned under *A*, then *P* should be a *k*-element pattern. The type error in Racketnoob's program arises because the isomorphism condition is broken. The `[_ | Rest]` of his program does not reflect the `[[Var | Val] | Env]` in the type definition. The solution is easy; restore the missing structure by using `[[_ | _] | Rest]`. An alternative is to put a layer of abstraction into the type theory.

```
(datatype env

  ____________
  [] : env;

  Binding : binding; Env : env;
  ============================
  [Binding | Env] : env;)

(datatype binding

  Var : symbol; Val : expr;
  =========================
  [Var | Val] : binding;)
```

The process of adding explicit structure in order to meet the constraints of the type checker runs contrary to unfettered programming practice. Programmers who are left to run free generally use what can be called *mnemonic recognition*. Data structures are queried in mnemonic recognition only to the level needed for the programmer to determine what to do. The result is less code and a faster performance. The downside is that mnemonic recognition can be based on boundary assumptions about user input or even erroneous models of program execution where the recognition goes wrong. The Shen type checker avoids mnemonic recognition and incurs a performance overhead in the course of making certain that this sort of error does not occur.

Defining `exprs`

Our type definition avoids defining `expr`. The nature of this type can be gleaned by examining the main loop of the program – `interp`. A number is certainly an expression

```
(datatype expr

  N : number;
  ___________
  N : expr;
```

and certainly a symbol must be since variables are expressions.

```
  S : symbol;
  ___________
  S : expr;
```

... as are local assignments, ...

```
 Var : symbol; E1 : expr; E2 : expr;
 ===================================
 [let Var E1 E2] : expr;
```

... lambda expressions.

```
  X : symbol; Y : expr;
  =====================
  [lambda X Y] : expr;
```

... closures

```
  [lambda X Y] : expr; E : env;
  ================================
  [closure [lambda X Y] E] : expr;
```

... conditionals

```
   X : expr; Y : expr; Z : expr;
   ============================
   [if X Y Z] : expr;
```

... special dyadic operators. These are described in a side-condition.

```
  if (element? Op [+ - * / > < <= >= =])
  X : expr; Y : expr;
  ===================
  [Op X Y] : expr;
```

... and finally applications in the style of lambda calculus

```
X : expr; Y : expr;
==================
[X Y] : expr;)
```

The `interp` function plainly returns a normal form `(expr),` given an expression `(expr)` and an environment `(env)` as inputs; the type of `interp` is therefore `expr --> env --> expr.` Typechecking the program reveals that the first type error arises with `Lam Env -> [closure Lam Env] where (lambda? Lam)`. The problem again is the lack of significant structure. Racketnoob's program uses the SICP [1] strategy of establishing barriers of abstraction by creating recognisers for his datatype. An approach based on abstract datatypes would allow us to follow his example[2], but since we have decided to code the program using concrete datatypes, we must replace this code by an explicit appeal to lambda expressions as defined. The guard is otiose and the new line is `[lambda X Y] Env -> [closure [lambda X Y] Env]`

Dynamic type checking and the `verified` type

The type checker flags the next line `Var Env -> (get-from-env Var Env) where (symbol? Var)` with a type error. The problem is easy to see; the inference engine is armed with the assumption that `Var` is an `expr`; but the type of `get-from-env` requires that `Var` have the type `symbol`. Though all symbols are `expr`s, not all `expr`s are symbols. Racketnoob's program compensates for this by placing a guard on the rule; effectively a kind of dynamic type checking.

Such manoeuvres are common in programming and Shen evolved a technique for dealing with them based on the use of guards. Guards are booleans which in the event of the rule firing, must evaluate to `true`. Therefore Shen assumes when evaluating the action part of the rule, that any guard has the type `verified.` We can consider the type `verified` to be inhabited by all expressions whose normal form is `true`. Shen includes no inbuilt axiomatisation for this type and it is left to the programmer to formalise as much of the theory of this type as is required. Here we want to say that we can conclude `Var : symbol` if we have `(symbol? Var) : verified` as an assumption i.e. if it has been verified that `Var` is a symbol.

```
_____________________________________
(symbol? Var) : verified >> Var : symbol;
```

Coping with Special Operators

The type checker now breaks down on the next line

```
[Op M N] Env -> (Op (interp M Env) (interp N Env)) where (element? Op [+ - * /])
```

There are several problems here. First the integrity condition fails; `[Op M N]` is not isomorphic to `[+ M N]` (`Op` and `+` differ). This can be avoided by citing the cases individually `[+ M N]` ..., `[- M N]` etc. at the cost of turning one line into four. But even if this is done, there is a problem in that the program simply applies `Op` to the normal forms of `M` and `N`, even though there is no formal guarantee that these normal forms are numbers. Shen fails the program at that point too.

Dealing with each of these problems in turn; we could avoid triggering an isomorphism error by introducing a layer of abstraction into the type theory by introducing a type `sysop` into the system.

```
Op : sysop; X : expr; Y : expr;
==============================
[Op X Y] : expr;
```

[2] Such an approach is sketched in [19] p 234-236.

```
(datatype sysop

  if (element? Op [+ / - * > < = <= >= =])
  ____________________________________
  Op : sysop;)
```

But actually this approach is wrong. It is an example of **poisoning**; the introduction of false type assumptions into the type system in pursuit of the goal of validating the program. The LR rule is wrong because the L part is wrong

```
Op : sysop; X : expr; Y : expr;
_______________________________
[Op X Y] : expr >> P;
```

If we have a 3 element list [*x y z*] as an `expr` we cannot conclude that *x* is a `sysop` (e.g. `[lambda x x]` is such a list, but `lambda` is not a `sysop`). An alternative is to use a special tag – say `@` - to indicate that a system operator is used.

```
Op : sysop; X : expr; Y : expr;
===============================
[@ Op X Y] : expr;
```

The rule is now

```
[@ Op M N] Env -> (Op (interp M Env) (interp N Env)) where (element? Op [+ - * /])
```

But there is still no formal guarantee that the normal forms of `M` and `N` are numbers. We make some impression on this problem by inserting a number test for these cases.

```
[@ Op M N] Env -> (let IM (interp M Env)
                       IN (interp N Env)
                       (if (and (number? IM) (number? IN))
                           (Op IM IN)
                           (error "arithop applied to non-numeric argument")))
                      where (element? Op [+ - * /])
```

For Shen to take advantage of this test, we need to explain the logic of the conditional and conjunctive forms.

```
  P : boolean;
  P : verified >> Q : A;
  (not P) : verified >> R : A;
  ___________________________
  (if P Q R) : A;

  P : verified, Q : verified >> R;
  ________________________________
  (and P Q) : verified >> R;
```

In addition we have to say that the `number`? test establishes that an object is a number and that an object has a type A when it is an element of a list of objects of type A.

```
  ____________________________________
  (number? N) : verified >> N : number;

  L : (list A);
  _____________________________________
  (element? X L) : verified >> X : A;
```

By parity of reasoning we have to change the next rule to

```
[@ Op M N] Env -> (let IM (interp M Env)
                       IN (interp N Env)
                       (if (and (number? IM) (number? IN))
                           (if (Op IM IN) 1 0)
                           (error "arithop applied to non-numeric argument")))
                     where (element? Op [= < > <= >=])
```

The remaining rules present little challenge. The `extend-env` function is replaced by a simple construction and the final rule by a simple function call. Many of the auxiliary recognisers defined in the program are dropped. The entire program exists in the standard library [17] and is shown below, with the changes annotated

```
(define interp
  {expr --> env --> expr}
  N Env -> N where (number? N)
  [lambda X Y] Env -> [closure [lambda X Y] Env] \\ <--- explicit structure introduced
  Var Env -> (interp (get-from-env Var Env) Env)
                                  where (symbol? Var) \\ <--- verified type
  [@ Op M N] Env -> (let IM (interp M Env)  \\ <--- verified type; test and branch introduced
                       IN (interp N Env)
                       (if (and (number? IM) (number? IN))
                           (Op IM IN)
                           (error "arithop applied to non-numeric argument")))
                     where (element? Op [+ - * /])
  [@ Op M N] Env -> (let IM (interp M Env)  \\ <--- verified type; test and branch introduced
                       IN (interp N Env)
                       (if (and (number? IM) (number? IN))
                           (if (Op IM IN) 1 0)
                           (error "arithop applied to non-numeric argument")))
                     where (element? Op [= < > <= >=])
  [let Var E1 E2] Env
   -> (interp E2 [[Var | (interp E1 Env)] | Env]) \\ <-- simplified
  [if Test ET EF] Env -> (let T (interp Test Env)
                           (if (not (= T 0))
                               (interp ET Env)
                               (interp EF Env)))
  [Exp1 Exp2] Env -> (let Cl (interp Exp1 Env)   \\ <--- help function introduced
                            (handle-closure Cl Exp2)))

 (define get-from-env
  {symbol --> env --> expr}
  Var [] -> (error "Cannot find ~A in the environment!" Var)
  Var [[Var | Val] | _] -> Val
  Var [[_ | _] | Rest] -> (get-from-env Var Rest)) \\ <--- explicit structure introduced

(define handle-closure
  {expr --> expr --> expr}
  [closure [lambda Var Body] Env] Exp2
  -> (interp Body [[Var | (interp Exp2 Env)] | Env]) \\ <--- explicit structure introduced
  X _ -> (error "~A is not a closure!" X))
```

The entire type theory is given in figure 2.

```
(datatype expr

 X : number;
 ___________
 X : expr;

 X : symbol;
 ___________
 X : expr;

 X : symbol; Y : expr;
 =====================
 [lambda X Y] : expr;

 X : expr; Y : expr; Z : expr;
 =============================
 [if X Y Z] : expr;

 [lambda X Y] : expr; E : env;
 ==============================
 [closure [lambda X Y] E] : expr;

 X : symbol; Y : expr; Z : expr;
 ===============================
 [let X Y Z] : expr;

 Op : sysop; X : expr; Y : expr;
 ===============================
 [@ Op X Y] : expr;

 X : expr; Y : expr;
 ===================
 [X Y] : expr;)
```

```
(datatype sysop

  if (element? Op [+ / - * > < = <= >= =])
  _______________________________________
  Op : sysop;)

 (datatype verified-types

  ____________________________________________
  (symbol? Var) : verified >> Var : symbol;

  ____________________________________
  (number? N) : verified >> N : number;

  P : boolean;
  P : verified >> Q : A;
  (not P) : verified >> R : A;
  ___________________________________
  (if P Q R) : A;

  P : verified, Q : verified >> R;
  ________________________________
  (and P Q) : verified >> R;

  L : (list A);
  ____________________________________
  (element? X L) : verified >> X : A;)

(datatype env

  ___________
  [] : env;

  Var : symbol; Val : expr; Env : env;
  ====================================
  [[Var | Val] | Env] : env;)
```

Figure 2 The type theory for the mini-Lisp interpreter

Conclusion

We have presented Shen and the challenges and opportunities in using sequent calculus as a programming medium through a worked example provided by a third party. Though the program is short, it has a fairly rich and interesting type structure which requires some thought to excavate. The example shows why many programmers who get into Shen, attracted by the concise notation, often fail to mount the learning curve to mastering static typing in Shen. The latter aspect demands a good grasp of sequent calculus, logic programming and the T* algorithm and the trace tool in Shen cannot be used without understanding these concepts.

Most Shen programmers elect to work with type checking disabled. But more seriously, students who do elect to use it may try to circumvent the type checker by 'force feeding' the system in an attempt to beat the type checker – viewing it as an antagonist rather than an aid. It is fairly easy then for the programmer to poison, either deliberately or non-deliberately, the logic engine and thus produce a spurious verification.[3]

Shen has been used successfully to encode an entire web framework [14]; so in itself the language is a commercially viable tool. But most commercial programmers, even those with university degrees at doctoral level, report struggling with this programming in sequent calculus despite the fact that the notation is nearly 70

[3]Writing over 30 years ago, De Millo, Lipton and Perlis made the very similar point in a famous paper [2] that belaboured formal methods. In practice these methods have survived that paper and have proved themselves many times over in industry (e.g. [4], [10]).

years old. The formal background to Shen is not taught to undergraduates in computer science and often not emphasised even in university mathematics and is mostly found in theoretical computer science papers. Hence for social reasons, most programmers raised with Java, or even more recent languages like Clojure [7], fail to wield the full power of the Shen language. The Shen development path can be quite forked because the power of the language devolves the power of choice to the programmer even to the extent of offering macros to develop his own notation. In a commercial setting, this can be a management challenge, but at the same time, the power and productivity of the language and its vast portability offer great scope for software development.

References

1. Abelson H., Sussman J. — *The Structure and Interpretation of Computer Programs*, MIT Press, 1984.
2. De Millo R. A, Lipton R. J., Perlis A.J. — *Social Processes and Proofs of Theorems and Programs*, Communications of the ACM, 1979.
3. Diadov V. — Shen github repository, https://github.com/vasil-sd/shen-libs
4. Baumann C., Beckert B., Blasum H., Bormer T. — *Ingredients of Operating System Correctness? Lessons Learned in the Formal Verification of PikeOS* Embedded World 2010 Conference.
5. Burstall R.M, MacQueen D.B, Sannella D.T — *Hope: An Experimental Applicative Language*, 1980 LISP Conference, Stanford University.
6. Cardelli L. — *Basic Polymorphic Type Checking*, Science of Computer Programming, 1987.
7. Fogus M. — *The Joy of Clojure: Thinking the Clojure Way*, Manning Publications, 2011.
8. Franzen T. — *Logic Programming and Intuitionistic Sequent Calculus*, SICS Report, 1988.
9. Heijenoort J. editor — *From Frege to Gödel: A Source Book in Mathematical Logic, 1879-1931*, Harvard University Press
10. Hunt., W. A. — *FM8501: A Verified Microprocessor*, Technical Report 47, University of Texas at Austin, 1986.
11. Nagel E., Newman J. , Hofstadter D. — *Gödel's Proof*, NYU Press, 2008.
12. 'Racketnoob' — *A Mini-Lisp Interpreter in Shen* https://groups.google.com/forum/?hl=en#!topic/qilang/tv5qrYaiav0
13. Reeves. S., Clarke M. — *Logic for Computer Science*, Addison-Wesley, 1990.
14. Siram A. — *Shen: a sufficiently advanced Lisp*, talk given to Strangeloop, September, 2014.
15. Sterling L., Shapiro E. — *The Art of Prolog*, MIT Press, 1994.
16. Tarver M. — *A Language for Implementing Arbitrary Logics*, IJCAI, 1993.
17. Tarver M. — Shen standard library, http://www.shenlanguage.org/library.html
18. Tarver M. — *Functional Programming in Qi*, Fastprint Publications, 2008
19. Tarver M. — *The Book of Shen*, Fastprint Publications, 2014
20. Tarver M. — *Shendoc 15, the official Shen standard* http://www.shenlanguage.org/learn-shen/shendoc.htm
21. Thompson S. — *Type Theory and Functional Programming*, Addison-Wesley, 1991.
22. Thompson S. — *Haskell: the Craft of Functional Programming*, Addison-Wesley, 2011.
23. Turing A. — *On Computable Numbers, with an Application to the Entscheidungs problem.* Proceedings of the London Mathematical Society. 1937.
24. Wikstrom A. — *Functional Programming Using Standard ML*, Prentice-Hall, 1988.