Cassowary PureScript Proposal

Contact: Athan Clark
Email: athan.clark@gmail.com
Date: June 4, 2015

Pure Functional Programming for Total Correctness

The benefits of functional languages like PureScript for implementing complex, mathematically dense concepts, like constraint solvers or compilers.

1 Overview of Functional Programming

It has been recently discovered that computer science is the heart of mathematics and logic. However, this is true only for functional programming languages like Haskell and PureScript - they do away with the concept of a machine, and direct all attention to the concept of a mathematical function. Programming then becomes the creation of logical specifications as types, and functions as values, rather than sequentially manipulating physical data. I argue that this model of programming is superior to the imperative style for the development and implementation of complex mathematical algorithms, like constraint solvers and compilers.

1.1 Correctness

Through functional programming, you can be **guaranteed** software correctness. The insurance comes from precise specifications as *types*, bounding the derivable behavior of stateless functions. For dependently typed languages (like Idris, Agda and Coq) if an inconsistency is found, the program will simply not compile. In Hindley-Milner type systems (like Haskell and PureScript), we can get a similar guarantee by *generating test cases* by the hundreds, to establish properties and conformance to logical propositions. These insurance techniques safeguard codebases far more effectively than unit tests. By establishing *algebraic properties* (such as associativity and commutativity of expressions), we can **eliminate entire classes of bugs** from being possible.

In the domain of implementing a constraint solver, for instance, we can be sure that our constraint set is commutative - we can move constraints around in our set. Likewise, we can be sure that evaluating and reducing our constraint set is also associative - we can evaluate in any direction. With entailment often as a first-class object, we are free to prove logical propositions about our code, without much overhead or headache.

1.2 Performance

Functional languages like Haskell or PureScript find an elegant balance between formal correctness and performance. By relying on purity - the lack of affecting reality - compilers are free to optimize at a **macroscopic level**. This is evident in the premier Haskell compiler, GHC, where you can get better SIMD performance than hand and machine-optimized C. Likewise, Warp (a web server written in Haskell) can achieve over 20,000 responses per second on a $200 laptop (without optimizations enabled) compared to Node.js getting only 600 rps on the same machine.

This is the remarkable effect of algebraic programming languages. By not relying on the real world, we can condense our imaginary definition (soundly) to a much smaller idea, before giving it reigns over reality.

1.3 Clarity

Purely functional languages produce more correct software and casually have greater performance, but in addition to this, the are often smaller and more concise than their imperative counterparts. With less room for human error and more ability to express ideas, writing algorithms like quicksort and factorial can be a brief couple lines of code.

1.4 Examples

The benefits of using such a language are obvious for any software project, *especially* when problems get complex.

CompCert is a C compiler (and compiler verification initiative), similar to GCC, implemented in a functional programming language. It's performance is almost as good as GCC's - about 80%. But remarkably, the size of the source code is minuscule: CompCert is at about 60,000 lines of code - 50,000 dedicated to exhaustive proofs (verifying correctness of binary translations on all supported architectures), with about 8000 lines actually implementing the compiler. This is hardly a large codebase, especially compared to the 15 million lines of code behind GCC.

Type inference algorithms are inherently constraint-oriented. For Hindley-Milner type systems, the traditional approach is to use Algorithm W - recurse down the abstract syntax tree of an expression, populating a c̈ontextẅith assumptions, then merging the assumptions to a conclusion. In the GHC Haskell compiler, the act of instantiating polymorphic type variables with actual instances (a mix of ad-hoc and parametric polymorphism), *is* constraint solving - in fact, we are given utilities to manipulate the notion of a constraint from within the language.

In my λtext command-line utility, I've taken a variant of Algorithm W, directly oriented at constraint solving to implement type checking and type inference. There are approximately 1000 lines of code in the entire project.

Less code *may* mean less room for bugs, but the real reason for functional programming's elegance is its underlying theory - one can directly translate category theory specifications to programs, or formal proofs in propositional logic to types, in an identical syntax. Cryptol is a great example of how someone can define cryptographic algorithms that mirror their algebraic specifications, one-for-one.

1.5 FP Conclusion

In essence, if you are doing anything complicated, it is best to have a mathematical concept of your problem. And if you have a mathematical definition of a problem, it is **trivial** to implement the algorithm in a functional language, because of how closely it's tied to the field. There have been many successful projects implemented in pure functional programming languages - automated trading systems for the stock market, compilers and interpreters, constraint solvers, and a wide array of mathematically dense algorithms.

Cassowary is a great example of an algorithm that has rich mathematical heritage, but has grown rotten from neglect - it is extremely difficult to take a high-level, abstract concept and implement it in an imperative setting. But, with the rise of pure functional programming, I am confident that I can make a smaller, more correct, and more efficient implementation of the Cassowary algorithm in PureScript for client-side browsers.

2 Plans for New Cassowary

Currently, all focus is being directed toward implementing the underlying engine - the augmented simplex method, pivoting, etc. until everything logically works as expected. I already have inequality expressions defined as an abstract syntax tree, and luckily the paper defines the steps so clearly that it should be straightforward to manipulate these constructs to get the correct result. When it is first implemented, it will be a PureScript DSL - we'll use $.+.$ as the addition operator, and $.*.$ as the multiplication operator. Likewise, we'll also have :==: and :<=: as operators to create inequalities. From there, it's just a matter of taking the expressions and adding them to our constraint set. As a rough outline for how the code would look from PureScript's perspective:

```
mainConstraints :: Cassowary Subst
mainConstraints = do
  addConstr 0 $ var "x" .*. 5 :<=: var "y"
  addConstr 1 $ var "y" :=>: var "z" .+. var "x"
  addConstr 1 $ var "x" :==: 10
  subst ← inform [(x, 10), (y, 30)]
  ...
```

Where *Cassowary* is a stateful monad, *addConstr* takes a syntax tree of our expressions (and reduces them), and *inform* populates some of the variables with data. I'm still not comfortable with deciding this as how the api will look, but it is a rough estimate. Lastly, *subst* would be the resulting optimized substitution, given the data we include. DSLs are an iconic feature of functional programming, and the style I've presented is somewhat similar to the bindings to CVC4 and Z3.

In the example, the first parameter to $addConstr$ is the weight associated with the constraint. Also, $inform$ is an alias for:

$$inform \;=\; last \;<\$>\; zipWithM\; \big(\lambda key\; val \rightarrow addConstr\; 1\; \$\; key\; :==:\; val\big)\; \circ\; unzip$$

I'm unsure whether or not the structure will also form a comonad, for extraction. Many monads also form comonads - including $Reader$, $Writer$ and $State$, so it would not surprise me. Along the same line, Walder et al.'s thesis on monadic constraint solving appears to also be a comonad. David Overton has also made a trivial constraint solver with a different method to encoding the constraints themselves, but will also be advised for inspiration. I will need to do more formal specification before I can be confident in this estimate.

$addConstr$ itself needs to reduce our expressions. Internally, the abstract syntax tree for our DSL will look something like this:

```
data LinAst =
  EVar String
  ELit Double
  ECoeff LinAst Double
  ESum LinAst LinAst
```

Where $. + .$ is an alias for $ESum$ and $. * .$ for $ECoeff$. Note that the right side of $ECoeff$ is a $Double$ literal - this makes it impossible to construct quadratic & higher expressions.

From here, we would like to translate our user-level AST to standard form, to make the translation to *basic feasible solved form* easier - having a data type similar to:

```
data LinVar = LinVar
  { varName  :: String
  , varCoeff :: Double }

data LinExpr = LinExpr
  { exprVars  :: [LinVar]
  , exprConst :: Double }
```

Where the list represents the summation of each variable, with it's coefficient. We will need a reduction / translation function that turns our AST to a $LinExpr$ - I'll refer to this as $astToExpr$. I'll ommit the definition for brevity - it's fairly routine; distribute $ECoeff$ multiplications, combine $ELit$ constants, then fold over the AST to build a *unique* $LinExpr$. Constraint equations themselves would look something like this:

$$\begin{aligned}&\textbf{data}\ Constraint\ =\\ &\quad ConstrEqv\ \ LinExpr\ \ LinExpr\\ &\quad ConstrLte\ \ LinExpr\ \ LinExor\end{aligned}$$

Where :==: is an alias for $\lambda xy \to ConstrEqv(astToExprx)(astToExpry)$, and likewise for :<=:. :=>: is just $flip$:<=:. From here, we can build the actual constraint solver - $pivot$ does sound refactoring of $LinExpr$'s, then quasi-linear optimization would follow a similar technique as augmented simplex form, annotated with error metrics, for each weight. I haven't implemented this yet, so I can't give a definition.

2.1 Backward Compatability

I am very familiar with building raw JavaScript apis (I built a runtime typechecker for javascript that uses chaining to implement partial application - see frankenscript). To make a cassowary.js-compliant api would be possible, but for me that is less important than making a PureScript implementation that works well. Along this line of reasoning, I could also cater your desires for an end-user api to your liking, but again this will have to be post-implementation.

2.2 Expected Time

I expect to have a working implementation in less than two weeks - I already have syntax data structures implemented, including some basic reduction algorithms. I would like to estimate at two weeks, though, to give myself the time to properly polish, test and benchmark the codebase.

I also see this library as beloning to you, my employer, therefore publication rights and licensing is up to you. I would prefer to make the system open source, to allow other people to improve on it, but I am not against making the product privately owned.

# # #