Strongly-Typed Multi-View Stack-Based Computations

Pieter Koopman
Radboud University
Nijmegen, Netherlands
pieter@cs.ru.nl

Mart Lubbers
Radboud University
Nijmegen, Netherlands
mart@cs.ru.nl

Abstract

High-level language languages are often implemented by transforming them to a stack-based intermediate language. To ensure correctness of the implementation, it is desirable to have a type-system for the stack-based code that ensures that the required arguments are available on the stack. This is quite challenging since the stack contains values of mixed types. Moreover, a single stack is shared by all basic stack instructions and the functions implemented with those instructions. Just like basic instructions, function calls are expected to replace their arguments by the result and to leave the rest of the stack untouched.

This paper shows a Domain-Specific Language, DSL, for stack-based computations embedded in strongly typed functional programming language. We use heterogeneous lists in the DSL to ensure that the top of the stack contains the required elements for instructions and functions. Type correctness of the composition of instructions and functions is ensured by requiring that the remainder of the stack is unchanged. However, standard typing restrictions impose that all function applications have identically typed arguments and hence an identical stack layout. We present a simple solution based on data types with universally quantified type variables. The resulting DSL supports multiple views and handles mutually recursive functions of arbitrary arities.

CCS Concepts: • **Software and its engineering** → **Functional languages**; **Domain specific languages**; **Source code generation**.

Keywords: DSL, Stack-Based Computing, Strongly-Typed

ACM Reference Format:
Pieter Koopman and Mart Lubbers. 2023. Strongly-Typed Multi-View Stack-Based Computations. In *PPDP23: 25th International Symposium on Principles and Practice of Declarative Programming*. ACM, New York, NY, USA, 13 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Strong static type systems provide widely used support to prevent runtime type errors. Such type systems are known for all kinds of programming paradigms, ranging from imperative via object-oriented to functional programming. A significant amount of work has been done to carry over the advantages of statically typed programming languages to embedded Domain-Specific Languages, DSLs, constructed inside these programming languages. This resulted in language extensions like GADTs [38] and advanced uses of language constructs like class-based embedding of DSLs [6].

Many implementations of programming languages use stack-based computations somewhere in their implementation. A stack-based language expects instruction and functions arguments to be on the stack, the result replaces those arguments. This abstraction level is a convenient step between high-level language constructs and actual machine code, but also a convenient representation in many interpreters. Famous examples in functional programming include the G-machine [13] used as a basis for the Haskell implementation and the ABC-machine [17] used in the implementation of Clean. More recently, we used a stack machine in the implementation of mTask which is our task-oriented programming interpreter on small IoT devices [21]. Stack-based machines are by no means limited to functional programming, e.g., the famous Java virtual machine is stack based [1, 37].

When we embedded a stack-based DSL in a high level language like Clean or Haskell, the stack typically becomes a list of tagged values. The tagged values are required since the stack contains values of various types, like integers and booleans. This implies that each and every operation has to check dynamically whether enough arguments of the correct type are available on the stack. This results in possible runtime type errors and has a performance penalty. For example, the addition instruction of a stack machine requires two integers values on the stack and replaces them by their sum.

```
:: Elem = I Int | B Bool
:: Stack :== [Elem]

add :: Stack → MaybeError String Stack
add stack = case stack of
  [I x:I y:s] = pure [I (x+y):s]
  stack       = fail "wrong stack in add"
```

One can also use an array to represent a stack [36]. Lists can grow and shrink by need, while arrays typically have a fixed size. All required actions modify the initial part of the list and are $O(1)$.

In this paper, we show how a variant of heterogeneous lists guarantees statically that the required arguments of the proper type are available at runtime. This eliminates the need for runtime checks and the corresponding dynamic errors.

For interpretations of our stack-based DSL other than evaluation, like pretty printing and code generation, it is required that functions are part of the DSL. Using functions of the host language to mimic recursive DSL functions produces infinite results. We use Higher-Order abstract syntax, HOAS [31], to make strongly typed functions in our DSL. A simplified example defines the increment function in some basic computations of which the details are explained later. The operator `:.` denotes sequencing of stack manipulations.

```
example1 =
  (λinc → push 5 :. inc :. push 6 :. inc :. mul)
  (push 1 :. add)
```

The type system of host languages like Clean and Haskell requires that all applications of a function argument have exactly the same type. More precisely, this is the monomorphism restriction for bound variables in Hindley-Milner type systems [10, 26]. This causes a problem when we use heterogeneous lists to type the stack. The type system correctly determines that the first application of `inc` requires a stack with one integer as argument and result. However, the type system correctly determines that there are two integers on the stack in the second application. Hence, this example is rejected by the compiler since the rest of the stack cannot be a variable `s` as well as this `s` with an additional integer on the stack.

Instantiating polymorphic type variables with polymorphic types is called impredicative polymorphism[1] citeAQuickLookAtImpredicativity. We propose a light-weight solution that is based on the universally quantified variables that are allowed in algebraic data types [18]. This requires no extensions of the Clean type system[2], nor is it required to specify types. We show how these data types allow function definitions in an evaluator for stack computations, as well as in a muli-view DSL based on type classes [6].

Section 2 introduces the type concepts in a shallowly embedded evaluator for strongly-types stack programs. In Section 3 we show that the same approach also applies to a class-based DSL that has multiple views. In Section 4 we briefly show how such type safe code can be generated. Finally, we discuss related word and draw conclusions.

[1]Haskell requires the ImpredicativeTypes extension to allow this and requires explicit types. In our embedded DSL those types can become quite elaborated and tedious to write. Moreover, Clean does not allow this.
[2]Haskell requires the ExistentialQuantification extension to allow this.

2 Strongly-Typed Stack Manipulations

To introduce our approach, we start with a single view DSL for stack manipulations. This DSL is a shallowly embedded interpreter. This interpreter is a function that takes the typed stack as argument and produces the final stack as a result.

For self-contained expressions it would be enough to type only the expressions manipulating the stack. These expressions can still evaluate a traditional stack that contains values of different types, e.g., a list of tagged values. The type of the expression guarantees that the type matches will never fail. By using a typed stack we can avoid all dynamic type checks. This is more elegant and efficient. In this paper we only used typed stacks.

2.1 The Stack

We use Clean as host language for the implementation of our DSLs [32]. A very similar implementation can be made in Haskell [22]. Some details on types, especially restrictions on quantifier for type variables and class constraints, might differ.

The implementation of a typed stack is very similar to heterogeneous collections known as HLists [16]. The type of heterogeneous lists reflects the type of the various elements of such a list. The static type system guarantees that those elements are used type safely. A similar idea is used in dependently-typed programming to distinguish empty lists from non-empty lists, or to reason at the type level, i.e. at compile time, about the length of lists [9, 14, 23, 29, 36].

The key difference between our stacks and this work is that we are only interested in the top segment of the stack and not the whole stack. However, it is important that the remainder of the stack is unchanged. For instance, the addition replaces two numbers on top of the stack by their sum. The type system ensures that these elements are available and have the desired type. The rest of the stack is unaffected. Hence, the operation does not put any other restrictions on that part of the stack. In contrast to the `HList` package, we use algebraic data types for to represent the stack instead of classes.

The stack types are defined by two algebraic data types. The type `Bot` denotes the empty stack, bottom. We use `Push` to add an element of type `a` to a stack `s`.

```
:: Push a s = Push a s
:: Bot = Bot
```

For example, the operation `f` pushes a Boolean and two integers on a stack.

```
f :: s → Push Int (Push Int (Push Bool s))
f s = Push 42 (Push 7 (Push False s))
```

This function works for any stack `s`. Apart from the three new elements, the stack is unchanged.

In the same style, we can make instructions to push and pop an element from the stack and to swap the two elements on top of the stack[3].

```
push :: a s → Push a s
push a s = Push a s

pop :: (Push a s) → s
pop (Push a s) = s

swap :: (Push x (Push y s)) → Push y (Push x s)
swap (Push x (Push y s)) = Push y (Push x s)
```

2.2 Arithmetic Instructions

A limited set of instructions contains addition, subtraction, multiplication, and equality.

```
add :: ((Push a (Push a s)) → Push a s) | + a
add = binop (+)

sub :: ((Push a (Push a s)) → Push a s) | - a
sub = binop (-)

mul :: ((Push a (Push a s)) → Push a s) | * a
mul = binop (*)

equ :: ((Push v (Push v s)) → Push Bool s) | == v
equ = binop (==)

binop :: (x y→z) (Push y (Push x s)) → Push z s
binop op (Push y (Push x s)) = Push (op y x) s
```

It is obvious how to add more of those instructions to our DSL. Note that these stack instructions are overloaded by design. By choosing appropriate types for our instructions, we specify whether they are overloaded or work only for a single type. For instance, when we replace the type variable `a` by `Int` in the definition of `add`, it can only add integers. Such restricted types are relevant when we want to generate assembly code with our DSL.

2.3 Moving Stack Elements

Apart from the real computations, we typically need a set of instructions to copy elements from the stack to the top. Some typical instructions are:

```
copy0 :: (Push x s) → (Push x (Push x s))
copy0 (Push x s) = (Push x (Push x s))

copy1 :: (Push x (Push y s)) → Push y (Push x (Push y s))
copy1 (Push x (Push y s)) = Push y (Push x (Push y s))

copy2 :: (Push x (Push y (Push z s)))
        → Push z (Push x (Push y (Push z s)))
copy2 (Push x (Push y (Push z s)))
  = Push z (Push x (Push y (Push z s)))
```

We can generalize those instructions by using a typed counter to indicate the depth in the stack. This would work similar to H-lists.

Other instructions keep the top element and remove some elements after that value. This is very useful to keep the result of a function body and remove the function arguments that are below this result on the stack.

```
trim1 :: (Push x (Push y s)) → Push x s
trim1 (Push x (Push y s)) = Push x s

trim2 :: (Push x (Push y (Push z s))) → Push y (Push x s)
trim2 (Push x (Push y (Push z s))) = Push y (Push x s)
```

Depending on the calling conventions, clearing of the stack is done by the caller or the callee. The given instructions can handle both conventions.

It is obvious how more instructions to manipulate the items on the stack can be added by need. The type of the manipulations closely follows the implementation. Hence, the type system of the host language is able to derive those types.

2.4 Composition of Stack Manipulations

We introduce the operator `:.` to denote sequential composition of stack instructions. The name of the operator resembles the semicolon used in imperative languages. This is just function composition, where the function on the left-hand side is executed first. The listed priority and binding direction limits the amount of parentheses needed in our stack-based programs.

```
(:.) infixr 2 :: (a→b) (b→c) → a→c
(:.) s t = t o s
```

The example function `f` from Section 2.1 is now reformulated as

```
f :: (s → Push Int (Push Int (Push Bool s)))
f = push False :. push 0 :. push 42
```

2.5 Conditional and Repetition

In the tradition of stack machines, the conditional statement `cond` expects a Boolean value on top of the stack. The conditional operator has two arguments. When the boolean value on top of the stack is `True` the first argument is executed. Otherwise, the else-part is executed.

```
cond :: (s→t) (s→t) → (Push Bool s)→s
cond then else = λ(Push b s) → (if b then else) s
```

A conditional statement parametrized by three functions is more common in high-level programming languages. The

[3]The type notation in Clean differs slightly from Haskell [32]. In Clean the number of function arguments is reflected in the type. Information about the number of arguments is used by the compiler for optimization. There is no arrow separating the function arguments. An arrow in a function type separates the arguments from the result. Finally, class constraints (like `| + a`) are written at the end of the type. The Haskell equivalent of the first type is `add :: Num a => Push a (Push a stack) → Push a stack`.

The additional parenthesis in the type of add indicate that it has zero arguments. The function is used like any other curried function. The function `binop` has two arguments.

first argument computes the condition and leaves the result as a boolean on the stack. The other arguments are the then- and else-part.

```
If :: (s→Push Bool s) (s→t) (s→t) → s→t
If pred then else = pred :. cond then else
```

In the same spirit, we define the repetition statement `While`. The condition has to be evaluated here before each evaluation of the body. This makes it convenient to specify it only once as a parameter of `While`.

```
While :: (s→Push Bool s) (s→s) → s→s
While pred body = pred :. cond (body :. While pred body) id
```

An imperative version of the famous factorial function is defined as follows using the tooling above. It expects the argument as an integer on the stack and leaves its result, also as an integer, on the stack.

```
facWhile :: ((Push Int s) → Push Int s)
facWhile =
  push 1 :.                             // result r
  While (push 1 :. copy2 :. equ :. new) ( // n <> 1
    copy1 :. mul :.                     // r = r * n
    copy1 :. push 1 :. sub :. trim2     // n = n - 1
  ) :.
  trim1                                 // clean up stack
```

A simple application applies this statement to a stack containing the integer 5.

```
Start = facWhile (Push 5 Bot)
```

Execution yields the desired result: `Push 120 Bot`.

2.6 Function Calls

Using the lazy evaluation of the host language, we can also use functions in the host language to define recursive functions in the DSL, like a recursive version of the factorial.

```
fac :: ((Push Int s) → Push Int s)
fac =
  If (copy0 :. push 0 :. equ)                  // n == 0
    (pop :. push 1)                            // replace n by 1
    (copy1 :. push 1 :. sub :. fac :. mul)  // else n * fac (n-1)
```

Note that polymorphism on the stack `s` is used here actively. In the recursive application of `fac`, there is one integer more on the stack than in the initial function invocation. So, if we execute `fac (Push 5 Bot)` the type of `fac` is `Push Int Bot→Push Int Bot`. In the recursive call, this is `Push Int (Push Int Bot) → Push Int (Push Int Bot)`. Since `fac` is a top level function, this works flawlessly even when we do not specify the type. However, when such a function is used as an argument, the type system requires that all type variables have a single value. For the factorial function and its first recursive call, the required argument stack type is respectively `Push Int s` and `Push Int (Push Int s)`. Hence, the type checker will reject the use of such a function as argument. We discuss how to circumvent this restriction in Section 2.9 below.

Although this approach works correctly thanks to lazy evaluation, it is not a very satisfactory solution. When we would like to add another interpretation of our DSL, like pretty printing or low-level code generation, all recursive calls would be unfolded. It is basically old-fashioned macro expansion. This results in infinite printed versions or an unbounded amount of generated code.

2.7 First-Class Functions in the DSL

To tackle the problem of automatic function expansion of host language functions in the DSL, we need to make function definitions part of our DSL. This allows us to expand a function definition as well as to do other things with the function definition such as printing the name of that function or generating code for a function call. To ensure type safety, we use host language functions to define the needed variables [8, 31]. The basic idea is to have a definition primitive like `def :: a (a→b) → b`. The first argument is the body of the function, and the second argument is the expression where the function is applied. For evaluation, the definition is just `def body app = app body`.

This is a bit too simple for recursive functions; the body is outside the scope of the application. To solve this problem, we type definitions as `def :: (a→(a,b)) → b`. We replace the normal tuple by the infix version `In`, to make the syntax more appealing,

```
:: In a b = In infixr 1 a b

def :: (a → (In a b)) → b
def f = let (a In b) = f a in b
```

As illustration we show a recursive factorial function with an accumulator. The function `acc` expects two integers on the stack, the factorial argument and the accumulator on top of that. When the argument is zero, it replaces the argument by the accumulator using `trim1`. Otherwise, it multiplies the accumulator by the argument and decrements the argument before the recursive call. The factorial function `fac` just pushes the initial accumulator on the stack and calls `acc`. The main expression pushes the argument of factorial on the stack and calls `fac`.

```
facAcc :: Int → (s→Push Int s)
facAcc n =
  def λacc =
   If (copy1 :. push 0 :. equ)
    trim1
    (copy1 :. mul :. copy1 :. push 1 :. sub :. upd2 :. acc)
  In def λfac = push 1 :. acc
  In push n :. fac
```

Evaluating the expression `facAcc 4 Bot` produces the required result `Push 24 Bot`.

2.8 Mutual Recursive Functions

The facAcc example above shows that we can define nested functions. However, for mutual recursion, this is not good enough. The second function is outside the scope of the first function. A solution is to define the mutual recursive functions as tuple.

As illustration, we define the functions even and odd that call each other to determine if the argument of even is a positive even number.

```
evenDef :: Int →  (s→Push Bool s)
evenDef n =
  def λ(even, odd) =
    (If (copy0 :. push 0 :. equ)
      (pop :. push True)
      (push 1 :. sub :. odd)
    ,If (copy0 :. push 0 :. equ)
      (pop :. push False)
      (push 1 :. sub :. even)
    )
  In push n :. even
```

The additional strictness induced by the pattern match on the tuple (even,odd) cause a cycle in spine error; the value of the body tuple is needed before it can be calculated. We compensate for this additional strictness by introducing an extra tuple calculation in the situation that a tuple of functions is defined by def. To distinguish this situation, we make def a class with two (overlapping) instances.

```
class def a :: (a → (In a  b)) → b
instance def a where def f = let (a In b) = f a in b
instance def (x,y)
  where def f = let (a In b) = f (fst a, snd a) in b
```

With this new def the expression evenDef 7 Bot correctly produces the desired result Push False Bot.

Whenever we would need three or more mutual recursive functions, the corresponding instances of def should be provided.

2.9 Calling Functions in Different Contexts

Although the examples above work correctly, there is in general a serious problem. The type system of the host language derives the correct type for the functions in our DSL, but requires that this type is identical in all applications of the DSL function. The Hindley-Milner rules require that a function argument can only have a monomorphic type. This unfortunately implies that a DSL program is rejected as soon as a DSL function created with def is applied in situations with a different stack layout. Nevertheless, we need such a def in all views of the DSL apart from evaluation to ensure termination of that view for recursive functions.

A very simple example is a DSL program that defines a function p1 that pushes the integers one on the stack. This function is applied twice before we add those integers.

```
sumDef :: (s → Push Int s)
sumDef = def λp1 = push 1 In p1 :. p1 :. add
```

The type system correctly determines that the first application of p1 has type s → Push Int s while the second one has type s → Push Int (Push Int s)). These types cannot be coerced. Hence, the program is rejected. This is a well known problem in Hindley-Milner type systems for functional programming languages; polymorphic types are themselves not first class. The polymorphic type variables inside function arguments are monomorphic inside a function application.

There are at least two solutions for this polymorphism problem. The first solution is to pack the function in an algebraic data type that has a universal quantified type variable for the variable that needs several instances. Each time we take the function from this data type, we get a fresh variable for this universal quantified type variable. These fresh variables solve our polymorphism problem.

The second solution for the polymorphism problem is a special language extension that deviates from the normal Hindley-Milner types. Since version 9.2.1, GHC offers an alternative for these constructor-call pairs in the form of the ImpredicativeTypes extension, allowing for impredicative polymorphism. It requires that the user of the DSL fully specifies the types of the function, including a forall quantifier. See the paper of Serrano et al. for details [33]. The sumDef example from above becomes:

```
sumHaskell :: s → Push Int s
sumHaskell
 = def ((λf→(push 1, f .: f .: add))
   :: (forall s1.s1 → Push Int s1) →
         (forall s0.s0 → Push Int s0, s2 → Push Int s2)
      )
```

Le Botlan and Didier give an overview of other extensions of type systems to solve this problem [5].

The first solution with data types has the advantage that the compiler can derive all types when we indicate the proper data types and unpack functions. This implies that the user can indicate types as usual, but is not forced to write the somewhat intimidating types. Clean has no extension to allow impredicative polymorphism, this makes the choice obvious as long as we want to use that host language for our DSL. We will discuss the solution based on special data types in detail.

The idea of the solution is to give every occurrence of the DSL-function call its own type. This is done by an explicit quantifier for the rest of the stack. We define algebraic data types to hold those functions. These types are necessary to allow the explicit quantification. We need a type for each number of arguments and results on the stack. The first digit in the name indicates the number of function arguments, the last digit the number of results on the stack.

```
:: Fun11 a b   = Fun11 (∀ s:(Push a s)→Push b s)
:: Fun10 a     = Fun10 (∀ s:(Push a s)→s)
:: Fun01   b   = Fun01 (∀ s:s→Push b s)
:: Fun21 a b c = Fun21 (∀ s:(Push a (Push b s))→Push c s)
```

These types are sufficient for all our examples. In the same style, we can make instances with other numbers of arguments or results.

For all applications of the function, we extract it from the data type. The various `call` functions just do that. Since we extract a fresh function for each application, every instance `s` of the rest of the stack is fresh.

```
call11 :: (Fun11 a b) → (Push a s)→Push b s
call11 (Fun11 f) = f

call10 :: (Fun10 a) → (Push a s)→s
call10 (Fun10 f) = f

call01 :: (Fun01 b) → s→Push b s
call01 (Fun01 f) = f

call21 :: (Fun21 a b c) → (Push a (Push b s))→Push c s
call21 (Fun21 f) = f
```

The packing of functions in the data types $\texttt{Fun}_{nm}$ and the unpacking by $\texttt{call}_{nm}$ is designed such that the required elements on the stack are checked. The rest of the stack can be anything, but is unaffected by applying the function.

With this tooling, we can write a well typed version of our `sumDef` example from above. The function `p1` expects no arguments on the stack and leaves just one element on the stack. Hence, we use `Fun01`.

```
sumDef :: (s → Push Int s)
sumDef =
  def λp1 = Fun01 (push 1)
  In call01 p1 :. call01 p1 :. add
```

The normal recursive factorial function in the DSL suffers from the same problem as in recursive calls, the stack contains an extra element. This is spotted by the compiler and hence the program is rejected. The `Fun11` type solves the problem. It indicates that the call expects one element on the stack and yields one result. All other items on the stack are ignored for this function call.

```
facDef :: ((Push Int s)→Push Int s)
facDef =
  def λfac =
    Fun11 (If (copy0 :. push 0 :. equ)
      (pop :. push 1)
      (copy0 :. push 1 :. sub :. call11 fac :. mul)) In
    call11 fac
```

This function can be applied to any stack containing one integer at the top. For example:

```
Start = facDef (Push 5 Bot)
```

The execution of this program produces the desired stack `Push 120 Bot`.

Using the constructors to indicate the number of arguments and results is always allowed. However, it is only required when the defined function is used in contexts with a different stack layout. The type system determines the types of these stack elements in both situations. When the user forgets to use such a constructor and the associated call or uses the wrong pair, the type system of the host language will reject the embedded DSL program.

In contrast to the definition of `fac` in Section 2.6, this definition is completely part of the DSL and hence we can control both the function definition and the function call.

3 A Multi-View Strongly-Typed Stack DSL

The previous section shows that it is possible to construct type-safe DSLs to execute stack-based computations. The type system ensures that those manipulations cannot fail, the required elements are always available on the stack.

In many situations we want to do more with a DSL program than just executing it. For instance, it might be very convenient to print a DSL program. Especially when such a term is generated such a print option is convenient. Another required use of a DSL-program might be the generation of code for some particular hardware platform. Typically, we would like to produce a file containing the generated code. Hence, this it is just a special form of printing.

Making specific DSLs for those various purposes is an unsatisfactory solution. It is much better to have a single DSL that can serve all existing and future uses. The holy grail in DSL definitions is to make strongly-typed DSL with multiple interpretations, views. A well-known approach for deep embedding uses Generalized Algebraic data types, GADTs [38]. This is a variant of algebraic data types where one can specify the type of constructor by a function type. These function types are sufficient to specify the intended stack manipulations. The various views of the DSL are functions that take this GADT as argument.

An alternative is class-based shallow embedding, also known as tagless final [6]. Here, the idea it to replace each function from the evaluation DSL, as introduced in the previous section, by a type (constructor) class. Each instance of these type classes provides its own view of the DSL. We port our function-based approach to a class-based multi-view version since it is much easier to build such a DSL and its views incrementally[4].

3.1 DSL definition

The DSL consists of a set of type constructor classes. As language designers we have to choose how many functions we put in one class. For fine grain control we can give each language element its own class. To reduce the number of classes needed as much as possible we can collect all class members in a single class. We have chosen an intermediate design that uses separate classes for arithmetic operations, stack manipulations and control structures. For convenience,

[4]The code is available at https://gitlab.science.ru.nl/mlubbers/typed-stack.

we define the type class stack that is the union of all class members in those classes.

```
class stack v | arith, plumbing, control v
```

The multi-view function definition deserve its own class.

3.1.1 Arithmetic Instructions.

Arithmetic stack operations are treated first. We reuse the types Push and Bot from Section 2.1 to type the segment of the stack that is needed for the operations. Apart from the new type variable v these types are similar to the corresponding instructions in the previous DSL. There are two design decisions made in these types. Firstly, we only lift the stack manipulations to the view. The argument of push remains a plain value in the host language, rather than becoming a member of v a. Lifting the argument inside the instruction to the required view eliminates the need for explicitly lifting the values pushed to v a. Secondly, we create a view on the changed stack rather than a view on the initial and final stack. A version with a view on the given and produced stack is typed as push :: a → (v s) → v (Push a s). Both versions work fine, but our choice yields more concise code.

```
class arith v where
  push :: a → v (s→Push a s) | toString a
  pop  :: v ((Push a s) → s)
  add  :: v ((Push a (Push a s)) → Push a s) | + a
  sub  :: v ((Push a (Push a s)) → Push a s) | - a
  mul  :: v ((Push a (Push a s)) → Push a s) | * a
  equ  :: v ((Push a (Push a s)) → Push Bool s) | == a
  neg  :: v ((Push a s) → Push a s) | ~ a
```

The type a pushed on the stack and manipulated in the other operations is no type parameter of the class. Hence, we add the required class constraints, toString a and + a, to the member functions. Apart from the add type class variable v, these types are identical to the corresponding stack manipulation in Section 2.1.

The class plumbing defines operations to move and copy stack elements. Like above, the types tell exactly what these operations do. The type system ensures that they exactly perform the desired action.

```
class plumbing v where
  pop   :: (v ((Push a s) → s))
  copy0 :: (v ((Push x s) → Push x (Push x s)))
  copy1 :: (v ((Push x (Push y s)) →
                Push y (Push x (Push y s))))
  copy2 :: (v ((Push x (Push y (Push z s))) →
                Push z (Push x (Push y (Push z s)))))
  swap  :: (v (Push x (Push y stack)) →
                Push y (Push x stack))
  trim1 :: (v ((Push x (Push y s)) → Push x s))
  trim2 :: (v ((Push x (Push y (Push z s))) →
                Push y (Push x s)))
  no_op :: (v (s→s))
```

3.1.2 Control Structures.

The class control contains the control structures of our language and the sequencing operator for sequencing. They are the direct lifting of the control structures from the evaluator to the multi-view class. In contrast to the previous class, the members of this class have multiple arguments with a view v. Each of the stack manipulation functions provided here as argument is lifted to a view on the stack operations.

```
class control v where
  If    :: (v (s→Push Bool u)) (v (u→t)) (v (u→t)) →
             v (s→t)
  cond  :: (v (s→t)) (v (s→t)) → v ((Push Bool s)→t)
  While :: (v (s→Push Bool s)) (v (s→s)) → v (s→s)
  (:.) infixr 1 :: (v (a→b)) (v (b→c)) → v (a→c)
```

3.1.3 Function Calls.

This uses the data type In from Section 2. We extend the function definition with a type variable v to allow different views on the function definition.

```
class def v a :: (a → (In a (v b))) → v b
```

To allow function calls in different stack contexts we reuse the approach with a data type to allow universal quantification over the rest of the stack s. The only difference between the data types here and in Section 2.9 is the added type variable v to denote the view.

```
:: Fun11 v a b   = Fun11 (∀ s:v ((Push a s)→Push b s))
:: Fun10 v a     = Fun10 (∀ s:v ((Push a s)→s))
:: Fun01 v   b   = Fun01 (∀ s:v (s→Push b s))
:: Fun21 v a b c = Fun21 (∀ s:v ((Push a (Push b s))→
                                  Push c s))

call11 :: (Fun11 v a b) → v ((Push a s)→Push b s)
call11 (Fun11 f) = f
```

We have skipped the other call functions for brevity. They follow exactly the same pattern as call11.

3.2 Evaluation View

To make an instance of the classes of our DSL we need a type. The newtype E is defined for the evaluation view.

```
:: E a =: E a

deE :: (E a) → a
deE (E a) = a
```

This type is just a wrapper for its argument. The evaluation view differs only from the plain evaluation defined in Section 2 by the adding and removing of the constructor E. There are at least two implementation strategies for this. Tailor made monad instances can handle the packing and unpacking of the stack, or we can explicitly handle these constructors in our code. The monadic approach does not yield more concise code for the evaluation view. Hence, we explicitly add and remove the E constructors. The print view defined below in Section 3.4 does use a monadic approach, since there is a more complex state passed around.

3.2.1 Arithmetic Instructions.

The evaluation of the arithmetic instructions is basically equal to the DSL in Section 2. The use of an updated `binop` function hides the constructor `E` for many class members.

```
instance arith E where
  push a = E λs → Push a s
  pop    = E λ(Push a s) → s
  add    = binop (+)
  sub    = binop (-)
  mul    = binop (*)
  equ    = binop (==)
  neg    = E λ(Push x s) → Push (~ x) s

binop :: (a b→c) → E ((Push b (Push a s)) → Push c s)
binop f = E λ(Push b (Push a s)) → Push (f a b) s
```

Like above, we assume that the first argument is pushed first on the stack and, hence becomes the deepest element for operations.

3.3 Evaluation of Plumbing Operations

Like earlier, the operations to copy and move elements on the stack are completely specified by their type.

```
instance plumbing E where
 copy0 = E λ(Push x s)→Push x (Push x s)
 copy1 = E λ(Push x (Push y s))→Push y (Push x (Push y s))
 copy2 = E λ(Push x (Push y (Push z s))→
             Push z (Push x (Push y (Push z s)))
 trim1 = E λ(Push x (Push y s))→Push x s
 trim2 = E λ(Push x (Push y (Push z s)))→Push y (Push x s)
 no_op = E id
```

3.3.1 Evaluating Control Structures.

The control structures also follow the patterns from Section 2.

```
instance control E where
  If c t e = c :. cond t e
  cond t e = E λ(Push b s) → deE (if b t e) s
  While c b = If c (b :. While c b) no_op
  (:.) f g = E (deE g o deE f)
```

Due to lazy evaluation, the recursive expansion for `While` works fine in this view. In the printing view, such an implementation would lead to infinite recursion. See Section 3.4.2 for our print view of the control structures. Hence, `While` must be part of the DSL and cannot be defined as the given macro expansion in the host language. Function definitions must be part of the DSL for exactly the same reason.

3.3.2 Evaluating Function Definitions and Calls.

For the function definitions, we use the same cyclic definition as above.

```
instance def E a where def f = let (a In b) = f a in b
instance def E (x,y) | def E x & def E y
  where def f = let (a In b) = f (fst a, snd a) in b
```

Like above we add another computation for the instance for tuple definitions which is needed to define mutual recursive functions.

3.3.3 Evaluation.

The actual evaluator just takes an `E` view of the DSL as argument and applies that function to the empty stack, `Bot`.

```
eval :: (E (Bot→a)) → a
eval (E f) = f Bot
```

All examples from above can be directly ported to this multi-view DSL. When we skip the type definitions, the host language compiler derives the correct types. When we specify the types manually, they should include the class variable `v` and the required class constraints. The factorial function becomes:

```
facDef :: Int → (v (s→Push Int s))
        | stack v & def v (Fun11 v Int Int)
facDef n =
  def λfac =
    Fun11 (If (copy0 :. push 0 :. equ)
      (pop :. push 1)
      (copy0 :. push 1 :. sub :. call11 fac :. mul))
  In push n :. call11 fac
```

Elements from the Hofstader female sequences are computed by the mutual recursive functions `male` and `female` [11, 40]. Since the calls to these mutually recursive functions are nested, the implementation of these functions in our DSL will be called with various stack layouts. The data type `Fun11` ensures that the type system considers only the top most element as argument and result for each call.

$$
\begin{aligned}
F(0) &= 1 \\
F(n) &= n - M(F(n-1)), n > 0 \\
M(0) &= 0 \\
M(n) &= n - F(M(n-1)), n > 0
\end{aligned}
$$

Element `n` of the female sequence is computed in our DSL by `hofstadter n`. This sequence is known as A005378 in the online encyclopedia of integer sequences [34].

```
hofstadter :: Int → (v (s→Push Int s))
  | stack v & def v (Fun11 v Int Int,Fun11 v Int Int)
hofstadter n =
 def λ(male,female) =
  (Fun11 (If (copy0 :. push 0 :. equ) (pop :. push 0)
    (copy0 :. push 1 :. sub :. call11 male :.
     call11 female :. sub))
  ,Fun11 (If (copy0 :. push 0 :. equ)
    (pop :. push 1)
    (copy0 :. push 1 :. sub :. call11 female :.
     call11 male :. sub)))
 In push n :. call11 female
```

We use this as `Start = eval (hofstadter 6)`. It yields the desired result `Push 4 Bot`.

These functions provide an additional example of how our approach leverages data types with universally quantified type variables. By using data types for the types of functions in the DSL, the notation becomes more concise.

3.4 Print View

Printing the DSL is the process of transforming the terms in the language to a string representation. The printing state `PS` contains an integer, `cnt`, for generating fresh variables, `fun` representing the current function, and a difference list of strings [12]. In Clean, strings are unboxed arrays of characters. Difference lists are used instead of single strings to have constant time concatenation without having to allocate intermediate strings.

```
:: PS = {cnt :: Int, fun :: Int, out :: [String]→[String]}
```

The printing itself occurs in the `PrintMonad` data type, a type synonym for a state monad of `PS` [39]. The pretty printing view, `Print`, is a data type wrapping a `PrintMonad`. The type of the term is not used in the data type, it is a phantom type [7, 19].

```
:: PrintMonad a :== State PS a
:: Print a = P (PrintMonad ())
```

To simplify the implementation of the type classes, three helper functions are introduced. The function `print` prints the given string as a `Print` type, `printM` has the same functionality but then as a `PrintMonad` type. Finally, >>! sequences two print operations using the monadic sequence operator from the state monad.

```
print :: (a → Print b) | toString a
print = P o printM

printM :: a → PrintMonad () | toString a
printM a = modify λs→{s & out =λl→s.out [toString a:l]}

(>>!) infixl 1 :: (Print a) (Print b) → Print b
(>>!) (P a) (P b) = P (a ≫| b)
```

3.4.1 Arithmetic Instructions and Plumbing.

With this tooling, the instances of ordinary instructions become very simple. We list only the instance of the arithmetic instructions for brevity, the plumbing instructions just differ in name.

```
instance arith Print where
  push a = print "push " >>! print a
  pop = print "pop"
  add = print "add"
  sub = print "sub"
  mul = print "mul"
  equ = print "equ"
  neg = print "neg"
```

3.4.2 Control Structures.

The instances for the control structures are similar. Printing visits each branch of the DSL construct once and lards the output with the proper keywords and brackets.

```
instance control Print where
  If c t e = print "If (" >>! c >>! print ") ("
    >>! t >>! print ") (" >>! e >>! print ")"
  cond t e = print "cond (" >>! t >>! print ") ("
    >>! e >>! print ")"
  (:.) f g = f >>! print " :. " >>! g >>! print ""
  While c b = print "While (" >>! c >>! print ") ("
    >>! b >>! print ")"
```

3.4.3 Function Definition.

The function definition implementation for the printer differs greatly from the evaluator. Using the same cyclic definition as the evaluator would result in an infinitely large printer output in the case of recursive functions. In order to circumvent this, we provide a custom argument to the `def` function that does not contain the actual function, but a printer that prints the name of the function when called. In this way, we break the cycle and control both the calls to the function and the definition of the function.

In order to do so, some helper functions and type classes are needed. The `nfresh` helper function generates a fresh number for function names, it reserves `n` names. Reserving multiple names is needed when we define multiple functions at once to allow mutual recursion. The printing machinery can handle any number of simultaneous definitions. The `cnt` field in the print state is used to count the number of fresh names generated.

```
nfresh :: Int → PrintMonad Int
nfresh n =
  getState ≫= λs→put {s & cnt = s.cnt + n} ≫| pure s.cnt
```

The `toPrint` type class allows us to convert the type of the function, the `a` in the `def` class, to a printer. The `toPrint` function is used when we have an `a` and want to print it, i.e. the function definition. The `fromPrint` function is used when we have the printer but want to produce a value of the type, i.e. in the function application. Finally, there is the `componentSize` function that yields the number of functions in the component. It is not always trivial to get a value of the argument type since high-order abstract syntax is used, instead of a value, this function therefore only requires a witness of that type in the form of a `Proxy` value. Usually this number is one, but in the case of mutual recursion, it is higher.

```
:: Proxy a = Proxy
class toPrint a where
  toPrint :: a → Print a
  fromPrint :: (Print a) → a
  componentSize :: (Proxy a) → Int
```

Instances of `toPrint` are given for `Print a` and also for the `Fun*` data types. In the regular cases, the helper function `printAFun` is used to retrieve the current function number from the state and print the correct function name accordingly.

```
instance toPrint (Print a) where
  toPrint a = a >>! print ""
  fromPrint a = printAFun a
  componentSize _ = 1
instance toPrint (Fun11 Print a b  ) where
  toPrint (Fun11 a) = print "Fun11 (" >>! a >>! print ")"
  fromPrint a = Fun11 (print "call11 " >>! printAFun a)
  componentSize _ = 1
instance toPrint (Fun10 Print a    ) where ...
instance toPrint (Fun01 Print   b  ) where ...
instance toPrint (Fun21 Print a b c) where ...

printFun :: Int → PrintMonad ()
printFun n = printM ("f" +++ toString n)

printAFun :: (Print a) → Print b
printAFun a = a >>! P (getState >>= λs→printFun s.fun)
```

When a strongly connected component of functions is defined, e.g. in the case of mutual recursion, the `fun` field must be adjusted before calling the `fromPrint` instances of the children.

```
instance toPrint (a, b) | toPrint a & toPrint b where
  toPrint (a, b) =
    print "\n    (" >>! toPrint a >>! print "\n    ," >>!
    toPrint b >>! print ")"
  fromPrint a =
    (fromPrint (P (dePrint a))
    ,fromPrint (a >>! P (modify (λs→{s & fun=s.fun+1}))))
  componentSize _ = 2
```

The final helper function is `printAFuns`. This prints the names of the function definitions. This means just printing the function itself when it is the only function in the component, and printing the functions as a tuple when there are more.

```
printAFuns :: Int Int → PrintMonad ()
printAFuns a 1 = printFun a
printAFuns a csize =
  printM "(" >>| sequence (intersperse (printM ", ")
  [printFun (a + i) \\ i←[0..csize-1]]) >>| printM ")"
```

Finally, the implementation of printing `def` is created. This printer first generates fresh variables for every function in the component. The size of the component is determined with some type trickery using `componentSize`, see `csize` and `farity`. In Haskell, inline type ascription or visible type application can be used to circumvent this trickery.

To prevent a cycle in the `def` function, we pass it a printer that sets the `fun` record field to the correct fresh function number. The `fromPrint` function will retrieve it and prints the correct function name. Once `f` is called, the definition (`a`) is printed using `toPrint` and the rest of the program (`b`) is printed directly.

```
instance def Print a | toPrint a where
  def f = P $
    nfresh csize >>= λv→
    let (a In b) = f (fromPrint (P (modify λs→{s&fun=v})))
    in printM "\ndef \\" >>| printAFuns v csize >>|
    printM " = " >>| dePrint (toPrint a) >>|
    printM "\n  In " >>| dePrint b >>| printM ""
  where
    csize = componentSize (farity f)
    farity :: (a → (In a (Print b))) → Proxy a
    farity _ = Proxy
```

For convenience, we define a function `prnt` that selects the `Print` view of a DSL expression and creates a single string as output by evaluating the state monad computation and concatenating the result.

```
prnt :: (Print a) → String
prnt (P f) =
  concat ((execState f {cnt=0,out=id,fun=0}).out ["\n"])
```

3.5 Example

The function `answer` is a definition in our DSL that computes a number. Specifying a type for this function is not required.

```
answer =
  def λ(inc,arg) =
    (Fun11 (push 1 :. add)
    ,Fun01 (push 5 :. call11 inc))
  In call01 arg :. call11 inc :. call01 arg :. mul
```

The functions `inc` and `arg` are not actually mutually recursive, although their combined definition allows this. Nested definitions would have been sufficient. We have used the combined definition to illustrate how the printer view handles this more complex DSL construct.

Using rank-n polymorphism we can combine various views of our DSL in a single function [30]. The function `views` evaluates the given expression `e` and prints it. The class constraints ensure that the given expression is limited to our DSL. The class `funs` enumerates the allowed instances of definitions.

```
views :: (∀ v: v (Bot→a) | stack v & funs v) → (a,String)
views e = (eval e, prnt e)

class funs v | funs1 v Int & funs1 v Bool & funs2 v Int Bool
class funs1 v a
  | def v (Fun21 v a a a) & def v (Fun11 v a a)
  & def v (Fun01 v a) & def v (Fun10 v a)
class funs2 v a b
  | def v (Fun21 v a a b) & def v (Fun21 v a b a) & ...
```

This can be used to evaluate and print `answer`.

```
Start = views answer
```

This produces a correct result.

```
"def λ(f0, f1) =
  (Fun11 (push 1 :. add)
  ,Fun01 (push 5 :. call11 f0))
In call01 f1 :. call11 f0 :. call01 f1 :. mul
",Push 42 Bot)
```

4 Code Generation

It is perfectly possible to generate code for this DSL by hand. In many application one generates such code instead of manually writing it. To show that this is perfectly possible, we introduce a code generator for a class based DSL for ordinary infix expressions.

The expression DSL consists of literals, `lit`, a conditional expression, `IF`, an equality operator, `==.` as well as instances of the operators `+`, `-`, and `*`.

```
class lit v :: a → v a | toString a
class IF v :: (v Bool) (v a) (v a) → v a
class (==.) infix 4 v a :: (v a) (v a) → v Bool
```

Some exprssions in this DSL are `e42` and `c42`.

```
e42 :: Code v Int | stack v
e42 = (lit 6 + lit 5 - lit 4) * (lit 3 * lit 2)

c42 :: Code v Int | stack v
c42 = IF (e42 ==. lit 42) (lit 7) (lit -1)
```

It seems possible to extend this DSL with functions like the definitions in our stack language. This requires at least bookkeeping in the code generation about the position of arguments on the stack and the current stack height. In order to avoid the problems we have seen here with function types, it seems attractive to start with a similar solution for functions in the DSL we want to compile to the stack based DSL. This requires further research.

The code generation yields an element of the `stack` class introduced above. We use a type variable `v` to prevent that the view of this class is fixed in the code generation. The type of that expression is given by the type variable `a`. We use again a universally qualified variable `s` for the rest of the stack. This `s` can be anything, but the type guarantees that the code generation pushes exactly one value of type `a` and leaves the rest of the stack untouched.

```
:: Code v a = Code (∀ s:v (s→Push a s)) & stack v

deCode :: (Code v a) → v (s→Push a s) | stack v
deCode (Code c) = c
```

The code generation strategy itself is straightforward. The code for every expression generates code that leaves the value of that expression on the stack. In correspondence with the conventions of the `stack` operations we first push the first argument of binary operators followed by the second argument.

In line with the evaluation view of the `stack` DSL, introducing a set of monadic operations for `Code v` is considered overkill of this simple type. We add and remove the constructor `Code` explicitly.

```
instance lit (Code v) | stack v where lit a = Code (push a)
instance IF  (Code v) | stack v where
  IF (Code c) (Code t) (Code e) = Code (If c t e)
instance ==. (Code v) a | stack v & == a where
  (==.) (Code x) (Code y) = Code (x :. y :. equ)
instance +   (Code v a) | stack v & + a where
  (+) (Code x) (Code y) = Code (x :. y :. add)
instance -   (Code v a) | stack v & - a where
  (-) (Code x) (Code y) = Code (x :. y :. sub)
instance *   (Code v a) | stack v & * a where
  (*) (Code x) (Code y) = Code (x :. y :. mul)
```

The code mixes well with the `print` and `eval` of our `stack` DSL.

```
Start =
  (prnt (deCode e42), eval (deCode e42), eval (deCode c42))
```

This yields (with a manually added line break):

```
"push 6 :. push 5 :. add :. push 4 :. sub :. push 3 :.
 push 2 :. mul :. mul",(Push 42 Bot),(Push 7 Bot))
```

5 Related Work

It is recognized long ago that it is desirable to type stack-based computations. Earlier attempts to make a type-safe stack language are based on standalone languages with a tailor-made type system. Some famous examples are [15, 27, 28, 35]. In this paper we use a fairly standard type system to achieve type safety for an embedded stack-based language instead of a standalone language for stack manipulations.

There is a long tradition in proving compilers correct [4, 25]. McKinna and Wright used Epigram to construct a provable correct compiler to a stack-based language [24]. Here there is a separate proof showing the correctness of the stack manipulations. This work has been generalized by Atkey and others [2, 3]. In our work we use the type system of the host language to enforce type correctness of the stack manipulations. Proof-based approaches are used also in explicit imperative contexts [20].

Several papers use dependent types to ensure correctness of the stack-based manipulations. These approaches are closely related to implementations of heterogeneous lists. The idea of heterogeneous lists has been explored in various ways in Haskell and Clean. The default implementation today in the `HList` package in Haskell.[5] This is firmly based on a paper by Kiselyov et al. [16]. Section 10 of this paper gives a good overview of alternative implementations of heterogeneous lists like [9, 23, 29]. Later work extends the research on functional dependencies, e.g., Jones et al. [14]. Our work does not require functional dependencies. In contrast to the `HList` approach our handling of stacks is not primarily based on type classes to type the stack manipulations, we employ simple algebraic data types to ensure type safety.

6 Discussion

Stack-based computations are used in many versions of low-level code generation. In such a language, functions and operators expect their arguments on the stack and replace them by the result. Developing such programs is error-prone since the usual type systems do not guarantee that the required arguments are available on the stack. It is very easy to forget to push an argument at the right time or leave an element that is no longer needed on the stack. This results in runtime errors that are in general hard to find.

One can argue that those stack-based programs are typically generated by a compiler. Once the compiler is correct,

[5] https://hackage.haskell.org/package/HList

runtime type errors do not occur. This reasoning puts a high demands on the compiler and excludes changes of the compiler, the runtime system as well as handwritten pieces of code for optimization purposes [2–4, 24, 25, 36].

In this work we present a way to construct an embedded stack-based DSL where the type system of the host language ensures type correctness. The type system guarantees that each operation finds the required elements on the stack and leaves the expected results on the stack. This brings the custom correctness assurances of a strong type system to the world of stack-based computations. In writing the examples for this paper and the code generator we experienced that the type system indeed signals our errors in such programs.

We achieve this by a stack representation that is similar to heterogeneous lists. The key difference is that we do not want to handle stacks of finite and known length. The untouched part of stack can contain any number of elements of arbitrary types. The type system just ensures that this is untouched. We introduced an approach to allow that functions in the DSL can be applied to stacks with different untouched parts by using the universally quantified type variables allowed in algebraic data types. The Hindley-Milner based type system of will enforce that all instances used for the type variable indicating the rest of the stack have the same type, and hence the same number of arguments. This type system does include extensions like universally quantified type variables and overlapping class instances. Our approach avoids the need for a language extension like impredicative polymorphism. However, we need some data types to pack functions and boilerplate code to unpack the functions.

As future work we want to check if this scales to a full compiler. A good candidate is our dynamic translation of task-oriented programs for to byte code for restricted internet of things devices [21]. It is worthwhile to investigate whether a type synonyms and other GHC extensions such as visible type application can make the impredicative types simple enough for use in our DSL. In combination with the previous point, this requires an extension of the Clean compiler similar to impredicative types in Haskell or porting the mTask system to Haskell.

References

[1] 2002. *Proceedings of the 2nd Java Virtual Machine Research and Technology Symposium.* USENIX Association, USA.

[2] Robert Atkey. 2009. Parameterised Notions of Computation. *J. Funct. Program.* 19, 3–4 (jul 2009), 335–376. https://doi.org/10.1017/S095679680900728X

[3] Casper Bach Poulsen, Arjen Rouvoet, Andrew Tolmach, Robbert Krebbers, and Eelco Visser. 2017. Intrinsically-Typed Definitional Interpreters for Imperative Languages. *Proc. ACM Program. Lang.* 2, POPL, Article 16 (dec 2017), 34 pages. https://doi.org/10.1145/3158104

[4] Patrick Bahr and Graham Hutton. 2015. Calculating correct compilers. *Journal of Functional Programming* 25 (2015), e14. https://doi.org/10.1017/S0956796815000180

[5] Didier Le Botlan and Didier Rémy. 2009. Recasting MLF. *Information and Computation* 207, 6 (2009), 726–785. https://doi.org/10.1016/j.ic.2008.12.006

[6] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. 2009. Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages. *J. Funct. Program.* 19, 5 (Sept. 2009), 509–543. http://dx.doi.org/10.1017/S0956796809007205

[7] James Cheney and Ralf Hinze. 2003. *First-class phantom types.* Technical Report CUCIS TR2003-1901. Cornell University. https://hdl.handle.net/1813/5614

[8] Adam Chlipala. [n. d.]. Parametric Higher-Order Abstract Syntax for Mechanized Semantics. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2008) *(ICFP '08)*. ACM, 143–156. https://doi.org/10.1145/1411204.1411226 event-place: Victoria, BC, Canada.

[9] Thomas Hallgren. 2001. Fun with functional dependencies. In *Proc Joint CS/CE Winter Meeting, Chalmers Univerity, Varberg, Sweden.*

[10] R. Hindley. 1969. The Principal Type-Scheme of an Object in Combinatory Logic. *Trans. Amer. Math. Soc.* 146 (1969), 29–60. http://www.jstor.org/stable/1995158

[11] Douglas R. Hofstadter. 1979. *Godel, Escher, Bach: An Eternal Golden Braid.* Basic Books, Inc., USA.

[12] R.John Muir Hughes. 1986. A novel representation of lists and its application to the function "reverse". *Inform. Process. Lett.* 22, 3 (1986), 141–144. https://doi.org/10.1016/0020-0190(86)90059-1

[13] Thomas Johnsson. 2004. Efficient Compilation of Lazy Evaluation. *SIGPLAN Not.* 39, 4 (apr 2004), 125–138. https://doi.org/10.1145/989393.989409

[14] Mark P. Jones and Iavor S. Diatchki. 2008. Language and Program Design for Functional Dependencies. *SIGPLAN Not.* 44, 2 (sep 2008), 87–98. https://doi.org/10.1145/1543134.1411298

[15] Maarten Keijzer. 2013. Push-Forth: A Light-Weight, Strongly-Typed, Stack-Based Genetic Programming Language. In *Proceedings of the 15th Annual Conference Companion on Genetic and Evolutionary Computation* (Amsterdam, The Netherlands) *(GECCO '13 Companion)*. Association for Computing Machinery, New York, NY, USA, 1635–1640. https://doi.org/10.1145/2464576.2482742

[16] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. 2004. Strongly Typed Heterogeneous Collections. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell* (Snowbird, Utah, USA) *(Haskell '04)*. Association for Computing Machinery, New York, NY, USA, 96–107. https://doi.org/10.1145/1017472.1017488

[17] Pieter Koopman, Marko Van Eekelen, and Rinus Plasmeijer. 1995. Operational machine specification in a functional programming language. *Software: Practice and Experience* 25, 5 (1995), 463–499. https://doi.org/10.1002/spe.4380250502 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.4380250502

[18] Konstantin Läufer and Martin Odersky. 1994. Polymorphic Type Inference and Abstract Data Types. *ACM Trans. Program. Lang. Syst.* 16, 5 (sep 1994), 1411–1430. https://doi.org/10.1145/186025.186031

[19] Daan Leijen and Erik Meijer. 2000. Domain Specific Embedded Compilers. In *Proceedings of the 2nd Conference on Domain-Specific Languages* (Austin, Texas, USA) *(DSL '99)*. Association for Computing Machinery, New York, NY, USA, 109–122. https://doi.org/10.1145/331960.331977

[20] Xavier Leroy. 2009. A Formally Verified Compiler Back-End. *J. Autom. Reason.* 43, 4 (dec 2009), 363–446. https://doi.org/10.1007/s10817-009-9155-4

[21] Mart Lubbers, Pieter Koopman, and Rinus Plasmeijer. 2019. Interpreting Task Oriented Programs on Tiny Computers. In *Proceedings of the 31st Symposium on Implementation and Application of Functional Languages (IFL '19)*, Jurriën Stutterheim and Wei Ngan Chin (Eds.). Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3412932.3412936 event-place: Singapore, Singapore.

[22] Simon Marlow et al. 2010. Haskell 2010 language report. *Available online http://www. haskell. org/(May 2011)* (2010).

[23] Conor McBride. 2002. Faking it: Simulating dependent types in Haskell. *J. Funct. Program.* 12, 4&5 (2002), 375–392. https://doi.org/10.1017/S0956796802004355

[24] James McKinna and Joel Wright. 2006. A type-correct, stack-safe, provably correct, expression compiler in Epigram. *J. Funct. Program.* Submitted (2006).

[25] Henricus Johannes Maria Meijer. 1992. *Calculating compilers.* PhD Thesis.

[26] Robin Milner. 1978. A theory of type polymorphism in programming. *J. Comput. System Sci.* 17, 3 (1978), 348–375. https://doi.org/10.1016/0022-0000(78)90014-4

[27] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. 2002. Stack-Based Typed Assembly Language. *J. Funct. Program.* 12, 1 (jan 2002), 43–88. https://doi.org/10.1017/S0956796801004178

[28] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. 1999. From System F to Typed Assembly Language. *ACM Trans. Program. Lang. Syst.* 21, 3 (may 1999), 527–568. https://doi.org/10.1145/319301.319345

[29] Matthias Neubauer, Peter Thiemann, Martin Gasbichler, and Michael Sperber. 2001. A functional notation for functional dependencies. In *Proceedings of 2001 Haskell Workshop.* 101–120.

[30] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. 2007. Practical Type Inference for Arbitrary-Rank Types. *J. Funct. Program.* 17, 1 (jan 2007), 1–82. https://doi.org/10.1017/S0956796806006034

[31] F. Pfenning and C. Elliott. [n. d.]. Higher-Order Abstract Syntax. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation* (New York, NY, USA, 1988) *(PLDI '88)*. ACM, 199–208. https://doi.org/10.1145/53990.54010 event-place: Atlanta, Georgia, USA.

[32] Rinus Plasmeijer, Marko van Eekelen, and John van Groningen. 2012. Clean Language Report. https://wiki.clean.cs.ru.nl/download/html_report/CleanRep.2.2_1.htm [Online; accessed 5-December-2022].

[33] Alejandro Serrano, Jurriaan Hage, Simon Peyton Jones, and Dimitrios Vytiniotis. 2020. A Quick Look at Impredicativity. *Proc. ACM Program. Lang.* 4, ICFP, Article 89 (aug 2020), 29 pages. https://doi.org/10.1145/3408971

[34] Slone. 2023. The female of a pair of recurrences. https://oeis.org/A005378. [Online; accessed 23-May-2023].

[35] Christopher A. Stone, David Tarditi, Greg Morrisett, Perry Cheng, Peter Lee, and Robert Harper. 1996. The TIL/ML Compiler: Performance and Safety through Types. In *ACM SIGPLAN'96 Conference on Programming Language Design and Implementation (PLDI'96).*

[36] Wouter Swierstra. 2010. More Dependent Types for Distributed Arrays. *Higher Order Symbol. Comput.* 23, 4 (nov 2010), 489–506. https://doi.org/10.1007/s10990-011-9075-y

[37] Bill Venners. 2000. *Inside the Java 2 Virtual Machine* (2 ed.). McGraw-Hill Companies.

[38] Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. 2006. Simple unification-based type inference for GADTs. In *International Conference on Functional Programming (ICFP'06).* ACM SIGPLAN. https://www.microsoft.com/en-us/research/publication/simple-unification-based-type-inference-for-gadts/ 2016 ACM SIGPLAN Most Influential ICFP Paper Award.

[39] Philip Wadler. [n. d.]. Comprehending monads. In *Proceedings of the 1990 ACM conference on LISP and functional programming* (1990). ACM, 61–78.

[40] Wikipedia contributors. 2022. Hofstadter sequence — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Hofstadter_sequence&oldid=1115357514. [Online; accessed 23-May-2023].