Talk Notes

Rob Kleffner

February 23, 2017

1 Introduction

For those of you concerned with theorems and proofs, I apologize, today is going to be a very bad day.

To start things off, and make sure everyone is on the same page, I will give a couple brief examples of how to write code in stack languages, and show how they are evaluated. The syntax Ill be using is a variant of the stack-language Joy, with lambdas instead of what are called quotations in that community. I wont be introducing any formal syntax or semantics just yet; Ill be building those up once I start talking about type inference.

A quick bit of context about Joy: the language was invented by Manfred von Thun, a professor of philosophy at La Trobe University in Australia. It found a small following on a Yahoo mailing list called concatenative, which was their term for languages that looked like stack-based languages. There exists only one publication on Joy, a paper by von Thun presented at EuroForth 2001. It has nothing to do with type inference, but for those interested in stack languages, the paper is **Joy: Forths Functional Cousin**. I have also listed resources for learning about other stack languages for the tenaciously curious. I wont talk about Forth much because I want to focus on languages that make writing and examining higher-order functions easy, and Forth is tricky in that regard.

So, lets look at a quick example:

```
2 3 +
```

What this expression evaluates to should be intuitively clear, but we can already see a difference from more familiar languages: the plus is placed after its arguments rather than between them. This is called postfix notation, and almost all stack-based languages use it.

Now, this example clearly evaluates to five, but lets try a bigger expression:

```
2 3 + (\a.a a *) call
```

Okay, theres something that looks like a lambda in there now, but theres two other things we need to know before we get to that. First, we note that all expressions in stack languages are lists of terms. A term is data, like a number, or an operator, like +, *, and call, or even, in the case of our example, a lambda. Second, stack-based languages are so named because each term performs some operation on a stack. Lets run through an evaluation example:

At the start of evaluation, we have an empty stack that grows upward here on the board, and we have a cursor that points to the next term we need to evaluate, which is placed at the beginning of our expression. To evaluate a term, we examine the next term in the list, find that its a two, put a two on the stack, and move the cursor to the next term. We next find a three, so we put a three on the stack and then move the cursor. Now we see an add, and what add does is pops the top two elements of the stack and pushes the result of adding them together. So far so good.

Now we hit the lambda, and what do we do with it? Well, in stack languages, we dont apply functions implicitly; thats why we have the call word in there. Instead, what we do is push a closure onto the stack, and move the cursor past the lambda. Our next term tells us to apply the function on top of the stack to the rest of the stack. So we pop the function off the stack and start evaluating it. This particular function has one named argument, a, but how do we do anything with it? Well, we pop the next element off the stack and substitute it for each occurrence of a in the subexpression. Now we can evaluate the subexpression, which pushes five two times and then multiplies those two fives together. After evaluating the subexpression, we jump back to call, move our cursor past it, and see that we have no terms left to evaluate, which means were done, left with a stack containing only the number 25.

In the example we just saw, both expressions returned exactly one result, but this is not always the case. We could very well have left out the multiply in the previous example, and the result of our evaluation would have been two fives, and this is perfectly natural for a stack language to do. In fact, stack languages often encourage the use of multiple return values, although keeping too many values on the stack quickly becomes a burden on the programmers brain.

2 HPCL

Alright, now that were all on the same page, lets start formalizing these languages and examining type inference for them. Well start by specifying the syntax for a language called HPCL, for HP Calculator language. Like I said before, expressions in our language will be a list of terms, and for this language our only terms will be numbers and addition.

$$\begin{array}{rcl} e & ::= & \vec{t} \\ t & ::= & n \mid + \end{array}$$

This language hardly warrants the amount of attention were going to give it, but it is so simple that it will allow us to focus on setting up the foundations for semantics and type inference.

The semantics for our stack languages will be done on an abstract machine. Our reduction relation is on machine states to machine states, where a machine state is a pair consisting of an expression and a value stack. For now, our only values are numbers.

$$\begin{array}{rcl} R & ::= & m \to m \\ m & ::= & < e,\ s > \\ s & ::= & \vec{v} \\ v & ::= & n \end{array}$$

We have only two reduction rules for this basic machine: pushing a number onto the stack, and applying addition.

$$\begin{array}{rcl} < n : e,\ s > & \to & < e,\ n : s > \\ < + : e,\ n_1 : n_2 : s > & \to & < e,\ (n_2 + n_1) : s > \end{array}$$

When we reach a state where the expression part is empty, evaluation terminates the result is the stack part. What is interesting to note is that it is already possible to get stuck states in this language: we can underflow the stack. For instance, the expression 1 add cannot be reduced any further, but it isnt a valid value stack. In general, if we get to any state where a transition rule cannot be applied and the expression part of the state is non-empty, we say that evaluation is stuck. This will be our definition of stuckness for every language that we investigate today.

Now, its not the best motivating example, but say we have some basic calculator expression that we would like to execute on an empty stack. We might like to verify ahead of time that our expression doesnt underflow the stack, without having to add more information or annotations to our expression. This gets us to our first paper:

Poial, FORML 90, Algebraic Specification of Stack Effects for Forth Programs

Stack effects were not invented by Poial; rather, they come out of the Forth community at large, where they are used as a form of documentation for function input and output, much like the contracts of the PDP course. A stack effect has two parts: an input sequence and an output sequence. These sequences denote the elements required at the top of the stack when the function begins, and the elements that will be at the top when the function ends. So, the stack effect comment for a division operator in Forth might look something like:

```
; / : i1 i2 -- div rem
```

Which is a way to tell other programmers that this division operator takes two integers, and returns both the divisor and the remainder, with the remainder at the top of the stack above the divisor. It is important to note that this stack effect implicitly promises that our division function will not touch any elements on the stack below the two integers it requires.

What Poial recognized was that, with the proper restriction and formalization, stack effects could be seen as basic type algebra. To see this clearly, lets formally define the syntax of stack effects: we have a set of base types, which for HPCL is just integer, and we have stack effects, which are constructed from (possibly empty) finite sequences of base types:

$$\begin{array}{rcl} b & ::= & \text{int} \\ f & ::= & \vec{b} \to \vec{b} \;\mid\; \mathbf{0} \end{array}$$

It is then easy to infer the types of each of our terms. We also say that the empty expression has the empty stack-effect.

$$\textsc{Empty}\ \frac{}{\epsilon : \ \to}$$

$$\textsc{Num}\ \frac{}{n : \ \to \text{int}}$$

$$\textsc{Add}\ \frac{}{+ : \text{int int} \to \text{int}}$$

The type of + looks pretty intuitive, but why do numbers have an arrow in their type? Well, every term in a stack language can be thought of as a function from a stack to a stack. The type for numbers specifies that they will not touch any element on the stack, only placing the number value they signify on top of the stack. This formulation makes it particularly easy to infer the type of our expressions, to which we turn next.

As a motivating example, consider the expression `1 add`. We know that the type of `1` is $\to$ int, and we know that the type of add is int int $\to$ int, but how do we combine those two to infer the full type of the expression? What we need is an operation on stack effects, which is called product or composition throughout the literature. We will also need a notion of an invalid stack effect, which Poial denotes **0**. We define the composition operator $\circ$ with a few rules:

$$\begin{array}{rcll} \circ & : & f \to f & \\ (i... \to o...) \circ \mathbf{0} & = & \mathbf{0} & \\ \vec{0} \circ (i... \to o...) & = & \mathbf{0} & \\ (i_1... \to \ ) \circ (i_2... \to o_2...) & = & (i_2...\ i_1... \to o_2...) & \\ (i_1... \to o_1...) \circ (\ \to o_2...) & = & (i_1... \to o_1...\ o_2...) & \\ (i_1... \to o_1...\ b_1) \circ (i_2...\ b_2 \to o_2...) & = & (i_1... \to o_1...) \circ (i_2... \to o_2...) & \text{where } b_1 = b_2 \\ (i_1... \to o_1...\ b_1) \circ (i_2...\ b_2 \to o_2...) & = & \mathbf{0} & \text{where } b_1 \neq b_2 \end{array}$$

The first two rules are simple: attempting to composition a stack-effect with the invalid stack-effect yields the invalid stack-effect.

The next two rules are our valid terminating conditions. For rule 3, if the LHS stack effect generates no output, then composing the two types together means that RHS will expect its arguments to be on the stack after the term on the LHS has been executed. Hence, we place the input sequence of the RHS stack-effect to the left of the input required by the LHS, since this means they must be lower on the stack.

For rule 4, if the RHS consumes no input, and only produces output, then we know that the output of the RHS will be above the output produced by the LHS. Hence, we place the output sequence of the RHS to the right of the output sequence of the LHS, since they must appear higher in the stack.

For rule 5, we check to see whether the top element produced by the LHS is the same as the top element consumed by the RHS. If they are the same, we continue composition recursively, without the specified elements. Otherwise, if they are not the same, then we know that the RHS function will get stuck, and we signal a type-error by yielding the invalid stack-effect $\mathbf{0}$.

We can now finish our example, inferring the type of `1 add`, by adding another rule.

$$\textsc{Expr}\ \frac{e_1 : f_1 \qquad e_2 : f_2}{e_1\ e_2 : f_1 \circ f_2}$$

To be perfectly concrete, we have $(\to \text{int}) * (\text{int int} \to \text{int})$. We notice that we can only apply rule 5, and since int = int, we get the subproblem $(\to) * (\text{int} \to \text{int})$. Now we notice that we can apply rule 3, giving us the final stack effect $(\text{int} \to \text{int})$. This corresponds with our intuition that the expression 1 add will consume an integer from its input stack, and leave that integer incremented by one on top of the stack.

It is interesting to note that stack effects together with the composition operator form a monoid, since the composition operator is associative and has the identity element $(\to)$. It is also an inverse semigroup, a useful property that Poial claims allows to generate basic Forth programs from stack effects themselves.

For our HPCL language, it is impossible to have a type clash, that is, we cannot generate the invalid stack effect $\mathbf{0}$. However, say we add booleans to our base types, we can then imagine trying to compose two stack effects $(\to \text{int}) * (\text{bool} \to)$, which does gives us $\mathbf{0}$. This corresponds to our intuition that an expression that produces an int on top of the stack cannot be composed with an expression that expects a boolean on top of the stack; it is a type error.

However, it is not yet clear how this system enables us to prevent stack underflow. Indeed, by itself, it does not: it simply looks at the expression and tells us what will be required at the top of the stack for that expression to evaluate, in addition to which values will be left at the top of the stack by the expression. What Poial does is restrict the type of the final expression to consume no input and produce no output, that is, the final type must be $(\to)$.

This works for Forth, which is very low level and must explicitly specify how to get a return value to the operating system. However, it is not particularly useful for us, since the only valid expression in our language would be the empty expression. Instead, we specify that the final expression type must be of the form $(\to o...)$.

3 CBL

Well, its nice that we can verify ahead of time that our calculator expression wont underflow the stack, but boy is this a boring language! Lets level up and add booleans and conditional statements to our language. We call this language CBL for, Conditional Branching Language. Heres our updated syntax and semantics. Notice that our if-term only has the two branches; it expects the conditional to have already been performed previously.

$$\begin{array}{rcl} t & ::= & n| + |lt|eq|ifee \\ v & ::= & n|true|false \end{array}$$

HPCL Reduction rules...

$$\begin{array}{rcl} < \text{lt} : e,\ n_1 : n_2 : s > & \rightarrow & < e, (n_2 < n_1) :\ s > \\ < \text{eq} : e,\ n_1 : n_2 : s > & \rightarrow & < e, (n_2 = n_1) :\ s > \\ < \text{if } e_1\ e_2 : e_3,\ true : s > & \rightarrow & < e_1.e_3,\ s > \\ < \text{if } e_1\ e_2 : e_3,\ false : s > & \rightarrow & < e_2.e_3,\ s > \end{array}$$

For the straightforward extension to this language, we only need to extend our set of base types to include booleans, and add three more inference rules for our three new terms.

$$b \ := \ \text{int} \ | \ \text{bool}$$

$$\textsc{Lt}\ \frac{}{lt : \text{int int} \rightarrow \text{bool}}$$

$$\textsc{Eq}\ \frac{}{eq : \text{int int} \rightarrow \text{bool}}$$

$$\textsc{If}\ \frac{e_1 : (i... \rightarrow o...) \qquad e_2 : (i... \rightarrow o...)}{\text{if } e_1\ e_2 : (i... \text{ bool} \rightarrow o...)}$$

In this formulation, both branches of the if-term must consume exactly the same input sequences and produces exactly the same output sequences. However, in many typical Forth programs, this is not the case, as the elements expected by one branch could be different than those expected by another. This caveat brings us to our next paper:

Poial, FORML 91, Multiple Stack Effects of Forth Programs

In this paper, Poial notices that many Forth definitions do not have a single stack effect, but rather a finite set of stack effects. The question he attempts to solve is how to type expressions which have multiple stack effects. How do they compose together? His solution is to use a restricted form of intersection types, although he does not use that term. Specifically, he only allows intersections on stack effects; an intersection may not appear inside a stack effect. We end up with a type syntax that looks like:

$$\begin{array}{rcl} b & := & \text{int} \ | \ \text{bool} \\ f & := & \vec{b} \rightarrow \vec{b} \ | \ \mathbf{0} \\ \iota & := & \{\ \vec{f}\ \} \end{array}$$

ι is our top-level intersection type. Composition of stack effects is unchanged, but now we need a way to compose intersections of effects. We do that by composing each element of one intersection with each element in the other, and taking only those that do not result in 0. The empty intersection type becomes our new signifier for a type error, as every valid program has at least one stack effect.

$$\begin{array}{rcl} & \circ & : \quad \iota \to \iota \\ \{\ \vec{f_i}\ \} \circ \{\ \vec{f_j}\ \} & = & \{\ f_i \circ f_j\ \} \quad \text{where } f_i \circ f_j \neq \mathbf{0} \end{array}$$

We can now give inference rules for the new language. The rules for numbers and the basic operators are very similar, the only difference is that now they are an intersection of a single stack effect. But the rule for if is much different:

$$\textsc{If}\ \frac{\begin{array}{cc} e_1 : \{\ (i_l... \to o_l...)...\ \} & e_2 : \{\ (i_r... \to o_r...)...\ \} \\ \text{len}(i_l) = \text{len}(i_r) & \text{len}(o_l) = \text{len}(o_r) \end{array}}{\text{if } e_1\ e_2 : \{\ (i_l... \text{ bool} \to o_l...)...\ \} \cup \{\ (i_r... \text{ bool} \to o_r...)...\ \}}$$

This rule allows us to construct intersections of stack effects. However, we make the restriction that each all input sequences be the same length, and likewise for output sequences. This restriction keeps the system sound, while enabling a simple form of function overloading. For instance, in this system, `add` could be given the type (int int $\to$ int) $\wedge$ (float float $\to$ float) if we had floating point types.

While this is certainly an interesting addition, and a useful mechanism for type inference in the Forth setting, for our next language we will build off of the first variant of CBL, so that we dont have to bother with the intersection types. Know that for stack-effect based type systems, this is a perfectly reasonable extension we can use to allow for more expressive programs.

4 MiniForth

We still seem to be missing a vital part of most stack programming languages? Wheres dup? Wheres swap? Where are all our friendly neighborhood stack shuffle words? Lets add them in next. The language we will be studying here Ive called MiniForth, and it is just the non-intersection variant of CBL with a couple basic stack shuffling terms added. The system we develop will be able to support arbitrary stack shuffle terms, but we stick with only a few here for simplicity.

The three words we will add to the language are dup, swap, and zap:

$$\begin{array}{rcl} t & := & n \mid + \mid \text{lt} \mid \text{eq} \mid \text{if } e\ e \\ & \mid & \text{dup} \mid \text{swap} \mid \text{pop} \end{array}$$

CBLi reduction rules...

$$\begin{array}{rcl} < \text{dup} : e,\ v : s > & \rightarrow & < e,\ v : v : s > \\ < \text{swap} : e,\ v_1 : v_2 : s > & \rightarrow & < e,\ v_2 : v_1 : s > \\ < \text{pop} : e,\ v : s > & \rightarrow & < e,\ s > \end{array}$$

Now, say we have an expression `1 dup`. Its pretty clear what the inferred type of this expression should be: its ($\rightarrow$ int int). But what about the expression `zap swap dup`? We have two base types possible in our large, but its not clear from the expression which types are being shuffled. Since we have a finite set of base types, we could make use of the intersection types from our second variant of CBL, but this is a suboptimal solution. Instead, we introduce the type variables and the notion of parametric polymorphism.

This leads us to our third paper:

Stoddart & Knaggs, 93, Type Inference in Stack-based Languages

Stoddart and Knaggs introduce both parametric polymorphism and subtyping into their inference algorithm. However, their treatment of subtyping consists of exactly 5 sentences, so Im also not going to bother with it here. If you want to see more about subtyping with type inference in stack languages, haha, good luck.

Their key contribution is to tie type inference in stack languages halfway back into the existing literature on type inference by including type variables. We extend our set of base types with type variables α.

$$b := \text{int} \mid \text{bool} \mid \alpha$$

This tiny change to our type syntax forces us to make a major modification to our stack effect composition rules. Now, we must handle matching on variables, and substituting for those variables in the rest of the stack effect.

The rules Stoddart and Knaggs use are somewhat tricky and needlessly verbose, possibly due to their being unfamiliar with the existing techniques for polymorphic type inference (they do not cite any of the widely available literature on Hindley-Milner, for example). The modifications to

their system that I present here are both a generalization and a simplification of their inference rules for polymorphic stack shuffle terms.

First, lets examine the inference rules for our stack shuffle words. We assume the existence of a function fresh that generates a fresh type variable.

$$\textsc{Dup}\ \frac{\alpha = \text{fresh}}{\text{dup} : \alpha \to \alpha\ \alpha}$$

$$\textsc{Swap}\ \frac{\alpha, \beta = \text{fresh}}{\text{dup} : \alpha\ \beta \to \beta\ \alpha}$$

$$\textsc{Pop}\ \frac{\alpha = \text{fresh}}{\text{dup} : \alpha \to}$$

These are pretty intuitive, but we dont see unification anywhere. Indeed, we only do unification during stack-effect composition. We only need update the last rule of our composition operator:

$$\begin{array}{lcll} (i_1... \to o_1...b_1) \circ (i_2...b_2 \to o_2...) & = & \Phi(i_1... \to o_1...) \circ \Phi(i_2... \to o_2...) & \text{where } b_1 \sim b_2 = \Phi \\ (i_1... \to o_1...b_1) \circ (i_2...b_2 \to o_2...) & = & \mathbf{0} & \text{where } b_1 \nsim b_2 \end{array}$$

So we attempt to unify the top elements of the LHS output sequence and the RHS input sequence, and if we get a valid unifier, we apply it to all the rest of the elements in both stack effects, then continue performing composition recursively. If we cannot unify the two types, we signal a type error as before by simply returning the invalid stack effect. We use standard Robinson unification as our unifier, which means we can extend our type syntax with type operators like polymorphic lists and product types quite while maintaining our inference algorithm.

It is also easy to imagine adding other control operators, such as for or while loops, to this language, without needing to fundamentally change the type system. This type system thus has all the power we need to do type inference on a significant subset of the Forth programming language.

5 HiForth

We now turn our attention to a significant problem with the overall idea of stack-effects: they make type inference in a higher-order stack-based language very cumbersome and almost unusable. To examine this, we will create a variant of Forth with lambdas, called HiForth since our lambdas will be able to be used as values. As we saw in the first part of the presentation, we will need an explicit call term in order to apply our lambdas to the stack as well. To keep the presentation of our problem simple, we leave VariForth behind and make HiForth an extension of MiniForth. We no longer need our primitive stack shuffle words, as we can now write them using lambdas and call. To keep the semantic presentation simple, we only support lambdas with one or zero parameters. Our new terms and values are:

$$\begin{array}{rcl} t & := & n \mid + \mid \text{lt} \mid \text{eq} \mid \text{if } e\ e \\ & \mid & x \mid \lambda x^?.e \mid \text{call} \\ v & := & n \mid \text{true} \mid \text{false} \mid (\lambda x^?.e) \end{array}$$

And the semantics for our new operations are:

$$\begin{array}{rcl} < (\lambda x^?.e_1) : e_2,\ s > & \rightarrow & < e_2,\ (\lambda x^?.e_1) : s > \\ < \text{call} : e_1, (\lambda\ .e_2) : s > & \rightarrow & < e_2 \cdot e_1,\ s > \\ < \text{call} : e_1, (\lambda x.e_2) : v : s > & \rightarrow & < e_2[x/v] \cdot e_1,\ s > \end{array}$$

Can we use the existing MiniForth type system to do general inference here? Lets examine a simple lambda which, when called, returns the composition of two functions on top of the stack, such that the lower functions is called before the top one:

```
(\f.(\g.(\ . g call f call)))
```

What should be the type of this lambda? For that matter, what should be the type of `call`?

The problem with this example is that, in our existing type system, call needs to know exactly how many elements of the stack g and f consume and produce. But this information isnt given anywhere in our term, and for good reason: we would like this function composition lambda to work for functions of disparate arities.

Essentially, the problem is that we cannot represent the types of functions that are polymorphic with regard to the stack. Both g and f can consume and receive any number of arguments, the only restriction we would like to make is that the output of g does not conflict with the input of f. We only want to write this composition function once, and it have a valid principal type.

Perhaps if we illustrate what we can do, both the problem and the solution will become more clear. To satisfy the type system from MiniForth, we need a schema of call terms, one for each arity of input-output arity pairing. More concretely, we need:

$$
\begin{array}{lcl}
\text{call}_{1,1} & :: & (\alpha\ (\alpha \to \beta) \to \beta) \\
\text{call}_{1,2} & :: & (\alpha\ (\alpha \to \beta\ \gamma) \to \beta\ \gamma) \\
\text{call}_{2,1} & :: & (\alpha\ \beta\ (\alpha\ \beta \to \gamma) \to \gamma) \\
\text{call}_{2,2} & :: & (\alpha\ \beta\ (\alpha\ \beta \to \gamma\ \delta) \to \gamma\ \delta)
\end{array}
$$

Having this schema of call terms means we, in fact, cannot write a general compose function with a valid type. We could say that call has an infinite intersection of these types, but then every higher order function we write would have a similar infinite intersection type, which is clearly less than ideal.

6 Okasaki

So whats the solution? Well, I must now ask you to take a journey with me into the dark and dangerous kingdom of Haskell, and well find out.

After the publication by Stoddart and Knaggs, there was a period of about 10 years with essentially no publications about type inference for stack languages. And then, out of the blue, we get Chris Okasaki.

Okasaki, 2002, Techniques for Embedding Postfix Languages in Haskell

The key contribution of this paper is our first way to give types function arguments which are polymorphic with regard to the stack. Okasaki is able to do this by representing heterogeneous stack as left-nested pairs. I will give here a slightly modified but equivalent presentation of Okasakis embedding, so that we wont get too bogged down in fancy Haskell types.

Well start by examining how to translate MiniForth into Okasakis embedded Haskell language. Our convention in the type signatures is that variables after 'q' will range over stacks, and all other variables will refer to some value that can be placed on the stack:

```
push :: a -> s -> (s,a) pop :: (s,a) -> s dup :: (s,a) -> ((s,a),a) swap :: ((s,a),b) -> ((s
```

```
add :: ((s,Int),Int) -> (s,Int) mult :: ((s,Int),Int) -> (s,Int) lt :: ((s,Int),Int) -> (s,I
eq :: ((s,Int),Int) -> (s,Bool)
```

```
if :: (s -> q) -> (s -> q) -> (s,Bool) -> q
```

These types are pretty simple, and GHC can easily infer these same types from the implementation of each of these terms; this is just basic Hindley-Milner. Now lets see how we can use them, taking the empty tuple as our stack:

```
snd (mult (dup (add (push 5 (push 6 ()))))
```

We use the snd function to retrieve the top of the stack and bring it back into Haskell land. And we find that Haskell has no trouble infer the type of this expression. However, the downside of this current embedding is that it is not postfix, but rather prefix. Okasaki offers two solutions to transform these prefix expressions into postfix form. The first, which we will use here in the interest of brevity, is an infix operator that swaps the positions of the function and the argument:

```
x # f = f x
```

For those familiar with Haskell, this just the dollar-sign operator inverted. We can now write the above expression as:

```
snd $ () # push 6 # push 5 # add
```

Which, although ugly, certainly looks a lot more like a stack based language. Okasaki does present a way for us to eliminate the hash operator, but it is somewhat technical and introduces higher-rank polymorphism, which breaks our goal of global type inference. Indeed, Okasaki must include type annotations for presentation without the hash operator. However, if we just use the hash operator, the problem is reduced to standard Hindley-Milner type inference extended with pairs.

The question now becomes, can we write the example from the first part of the presentation in this embedding, and have it infer the correct type?

```
2 3 + (\a.a a *) call
```

Well, we dont have lambdas, but we do have dup, which is enough to get us something similar:

```
() # push 2 # push 3 # add # push (dup # mult) # call
```

Notice that we can put a function value on the stack in Haskell simply by wrapping a stack expression in parentheses, thanks to Haskells automatic currying. This is very similar to the quotation mechanism found in the Joy language mentioned earlier. The inferred type of this miniature expression is:

```
push (dup # mult) :: s -> (s,(q,Int) -> (q,Int))
```

This corresponds with our intuition that we can push function values onto the stack. Notice that our function value represents the rest of the stack with a variable s. This is exactly what we want: function type that are polymorphic with regard to the stack. We can call this function on any stack that at least has one integer on top of the stack, and expect the same stack back with only the top integer changed. Now we would like to apply this function, and fortunately the type of call is now very easy to represent:

```
call :: (s,(s -> q)) -> q
```

That is, call expects a function value on top of the stack, and the stack required as input to that function just below it. The result of call is simply the stack that results from applying the function to the rest of the stack.

To finalize our understanding of Okasakis system, lets examine how we infer the type of:

```
push (dup # mult) # call
```

We already know the type of both parts of this expression, so lets see what unification gets us as a result. We unify the type of `push (dup # mult)` with call by taking the output type of the first term, turning it into a function with a fresh variable in the result, and unifying that with the type of the latter term.

```
(s,(u,Int) -> (u,Int)) -> r =?= (q,(q -> t)) -> t
```

This gives us the unifier: `{t -> (u,Int), q -> (u,Int), s -> (u,Int), r -> (u,Int)}`

We then take the input stack of the first term and the new type variable that represents our new output stack, create a function type built from those, and apply the unifier:

```
apply uni (s -> r) = (u,Int) -> (u,Int)
```

And this is exactly the type we want. We now know we infer the types of higher order functions in stack languages!

7 MiniJoy

Of course, a representation based on nested tuples is neither pretty nor particularly efficient, and we would like to be able to write our expressions without a backwards function application operator. Can we solve both of these problems?

In a subtle way, we already have. The clue was in my suggestions that we only let variables after 'q' range over stack types, and variables at the front of the alphabet range over individual elements on the stack. There is a fundamental distinction between these variables that is hidden within the tuple representation: the q variables are substituted with a sequence of types, and the other variables are substituted by an individual type.

This insight brings us to our last paper in the development of type inference for stack-based programming languages.

Diggins, 2008, Simple Type Inference for Higher-Order Stack Languages

This paper is actually a technical report, not published in a journal, and contains no theorems or proofs. It does not even specify the semantics of the core language. However, Diggins does spell out the general idea, which we will do here.

Diggins' idea is to create two sorts of type variable: individual type variables, and stack type variables. Individual type variables are substituted with a single type, whereas stack type variables are substituted with a possibly-empty sequence of types. He also makes the restriction that stack variables can only occur at the left-most position of any sequence of types.

To formally introduce Diggins system, well use the same terms and semantics from HiForth, and then extend the type syntax. Our new language will be called MiniJoy.

The set of base types is extended with function types by allowing stack effects to be nested inside a pair of parentheses:

$$b \;:=\; \text{int} \;\mid\; \text{bool} \;\mid\; \alpha \;\mid\; f$$

However, we now introduce a new construction, which we call a type stack. Type stacks are a sequence of types optionally terminated by a stack variable. Diggins represents stack variables with capital letters, but I think it is clearer and more standard to represent them as lower case variables followed by three dots. Indeed, stack variables are a restriction on Rackets dotted type variables. Then, our stack-effects are modified to have both an input and an output stack.

$$f \;:=\; \alpha... \;\vec{b} \rightarrow \alpha... \;\vec{b}$$

It is useful to note that a stack without a leftmost variable now treats its leftmost element as the guaranteed bottom element of the stack; that is, a stack without a dotted type variable must contain exactly the elements specified by the type sequence.

We turn our attention now to the type inference rules, which must all be updated to reflect the fact that our terms are explicitly polymorphic with regard to the stack. We also introduce a gamma, since we have lambdas and need our variables to have types as well.

$$\textsc{Empty}\ \frac{\alpha... = \text{fresh}}{\Gamma \vdash \epsilon : \alpha... \to \alpha...}$$

$$\textsc{Num}\ \frac{\alpha... = \text{fresh}}{\Gamma \vdash n : \alpha... \to \alpha...\ \text{int}}$$

$$\textsc{Add}\ \frac{\alpha... = \text{fresh}}{\Gamma \vdash + : \alpha...\ \text{int}\ \text{int} \to \alpha...\ \text{int}}$$

$$\textsc{Lt}\ \frac{\alpha... = \text{fresh}}{\Gamma \vdash \text{lt} : \alpha...\ \text{int}\ \text{int} \to \alpha...\ \text{bool}}$$

$$\textsc{Eq}\ \frac{\alpha... = \text{fresh}}{\Gamma \vdash \text{eq} : \alpha...\ \text{int}\ \text{int} \to \alpha...\ \text{bool}}$$

$$\textsc{If}\ \frac{\Gamma \vdash e_1 : i... \to o... \qquad \Gamma \vdash e_2 : i... \to o...}{\Gamma \vdash \text{if}\ e_1\ e_2 : i...\ \text{bool} \to o...}$$

$$\textsc{Call}\ \frac{\alpha..., \beta... = \text{fresh}}{\Gamma \vdash \text{call} : \alpha...\ (\alpha... \to \beta...) \to \beta...}$$

The only two rules that touch the variable environment are refering to a variable and the lambda introduction. As stated before, we handle lambdas of both one and zero arguments.

$$\textsc{Var}\ \frac{x : b \in \Gamma \qquad \alpha... = \text{fresh}}{\Gamma \vdash x : \alpha... \to \alpha...\ b}$$

$$\textsc{LamNoArg}\ \frac{\Gamma \vdash e : i... \to o... \qquad \alpha... = \text{fresh}}{\Gamma \vdash \lambda\ .e : \alpha... \to \alpha...\ (i... \to o...)}$$

$$\textsc{LamArg}\ \frac{\Gamma, x : b \vdash e : i... \to o... \qquad \alpha... = \text{fresh}}{\Gamma \vdash \lambda x.e : \alpha... \to \alpha...\ (i...\ b \to o...)}$$

This is a complete set of inference rules for our terms. The last order of business is to give the rule for inferring the type of an expression:

$$\textsc{Expr}\ \frac{\begin{array}{c}\Gamma \vdash e_1 : i_1... \to o_1... \qquad \Gamma \vdash e_2 : i_2... \to o_2... \\ o_1... \sim i_2... = \Phi\end{array}}{\Gamma \vdash e_1\ e_2 : \Phi(i_1... \to o_2...)}$$

Of course, with the addition of the dotted type variables, we can no longer use standard Robinson unification. Instead, we use what is called sequence unification in the presence of flexible arity functions. When all the dotted type variables are guaranteed to occur only in the left-most position of any sequence, it is known that sequence unification is unitary (Kutsia, 2002).

8 Conclusion

This completes our investigation of the existing research on type inference in stack-based languages. I would like to leave you with two possible avenues that could extend type inference for these languages further.

The first is a question that I myself have investigated inconclusively: can we lift the restriction that sequence variables must only occur in the left-most position of every sequence? We will almost certainly lose unitary unification in the process, but if we use Poials intersection types, would we be able to get an finite set of most-general types?

Lets illustrate one of the problems that arises. Suppose we just lift the restriction that the sequence variable must occur in the left-most position. We still restrict ourselves to one sequence variable per sequence, but now it can occur anywhere in that sequence. Lets try to unify these two sequences:

```
int x... =?= x... int
```

Without even knowing the specific sequence unification algorithm, can we guess what the unifiers for these two sequences are? Well, its clear that substituting x with the empty sequence would work, so we add that to our set of unifiers. But if we substitute x with a sequence of one integer, thats also a unifier, and its not any less general than the previous one, so we add it to our set. Of course, two integers also works as a unifier for x, as does three, and four, and five, ad infinitum. This is a problem: we have an infinite set of most general unifiers, so we would also get an infinite set of most-general types! This problem is doubly hard because some sequence unification problems do not have nice patterns in the infinite unifiers they generate, as our example does.

The question then becomes: is there a restrict we can place on sequence variables appearing in sequences such that we never get an infinite unifier, but such that the resulting syntax is still useful for specifying types in a stack-based language? The resulting restriction might be very subtle and possibly too complex for programmers to want to bother with it, so is it even worth finding it? Besides, restricting ourselves to the left-most sequence variable already gives us a reasonable amount of expressiveness.

Another possible extension comes from ideas in a paper by Nigel-Horspool and Jan.

Static Analysis of PostScript Code, 93

They describe an algorithm that is related to type inference, and make use of a regular expression formalism for identifying the shape of the stack. This enables them to infer type-like constructs for recursive functions that consume arbitrary numbers of stack elements. This is slightly different than dotted type variables: it is closer in spirit to Rackets starred types.

As an example, perhaps we want a summation function. Using the ideas from the PostScript paper, it might be possible to write a type for the sum function that looks something like this:

sum :: $int^* \rightarrow int$

That is, sum expects any number of integers at the top of the stack, and returns only one integer. The question is, are types like these useable, and can we infer them? It is a possbility: in 2015,

Kutsia and Marin published an algorithm for Regular Expression Order Sorted Unification, which is a generalization of sequence unification. If we extend our type syntax with regular expression types, could we get an inference algorithm simply by replacing sequence unification with regular expression unification? Again, an open problem.

Thats all!