

Typing Functional Stack-Based Languages

Christopher Diggins

Independent (no affiliation)

cdiggins@gmail.com

Abstract

Stack-based languages (e.g. Forth (Moore 1974), Postscript (Inc. 1999)) have been around for nearly four decades. They are particularly popular today for use as intermediate languages (e.g. CIL (ECMA 2002), JVM (Lindholm and Yellin 1999), (Morrisett et al. 1998)). This is for several reasons: they have good run-time performance characteristics, they resemble the machine level instructions on many computers, they are easy to implement, and they have compact representations. In these stack-based languages instructions are not first-class values.

The Joy programming language (von Thun 2001) and the Factor programming language (Pestov 2003) are examples of functional stack-based languages: they allow instructions to be treated as data and placed on the stack. These languages however are lacking a static type system.

This paper aims to bridge the gap between statically typed imperative stack-based languages and untyped functional stack-based languages by defining a type-system for a point-free functional stack-based language.

Categories and Subject Descriptors D.3.1 [Programming Languages]: Formal Definitions and Theory—Type Systems

General Terms Functional Programming, Stack-Based Languages, Type Systems

Keywords Language, Functional, Stack, Programming, Type System, Semantics

1. Introduction

A stack-based programming language has the property that function arguments and results are passed and returned on one or more shared stack objects. Every instruction can then be equated with a function that maps from a tuple of stacks to a tuple of stacks. A functional stack-based language has the property that all instructions are first class values that can be pushed and popped on a stack. This feature however is noticeably absent from many popular modern intermediate stack-based languages such as the CIL (ECMA 2002) and JVM (Lindholm and Yellin 1999).

While several papers have been written that formally describe the type system for non-functional stack-based languages (e.g. (Raffalli 1993), (Stata and Abadi 1998), (O’Callahan 1998), (Pöial 2006)) these papers don’t fully address the issue of typing higher-

order functions. Paul Levy’s paper on call by push value (CBPV) semantics (Levy 1999) describes a type system for a functional stack-based language with names. Levy’s type system isn’t appropriate for most stack-based language which are mostly point-free.

This paper’s contribution is to formally define the operational semantics and a type-system for a point-free stack-based language that allows all function or primitive instruction to be treated as data.

2. Cat Programming Language

This paper defines a kinding system for a subset of the Cat programming language (Diggins 2007). The Cat semantics and syntax are loosely based on the Joy language (von Thun 2001). For the sake of simplicity, this paper explore the semantics of a subset of the Cat language that excludes side-effects, and supports only integer, boolean, and function types.

Every term in Cat is a function which maps from one stack to another. Implicit between every two terms is the function composition operation. The Cat expression $g \ f$ is equivalent to the mathematical function $f \circ g$. Constant literals are functions that transform a stack into a copy of the original but with a constant on top. For example the Cat program `6 7 *` will transform any stack by making a copy of it with the value 42 on top. No operation in Cat allows a reference to previous stack-states, and as a result an implementation of Cat can use a single shared stack with destructive.

Like several stack-based languages, the core Cat language is point-free (i.e. it has no named variables or named parameters) and uses a postfix notation. Every term is a function which maps from a single stack to a single stack. New functions are constructed by either composing or quoting existing functions.

Even though Cat is a functional language, it is equally valid to describe Cat programs as if they manipulate a single global stack imperatively. This is possible because the Cat semantics do not permit previous stack states to be referenced from later functions. This enables a Cat implementation to use a stack data type with destructive semantics. Expressing the behavior of Cat imperatively is more succinct for informal discussion, but the functional model is preferable for formal discussion.

3. Function Application and Function Composition

Functional stack-based languages are examples of non-applicative functional programming languages, whereas the lambda calculus, and other functional programming languages (such as ML (Milner et al. 1990) and Haskell (Hudak et al. 1992)) are examples of applicative functional programming languages. This is because the implicit operation between terms is function composition instead of function application.

Unlike applicative functional languages Cat does not have an abstract syntax node for representing the operation of function application; instead function application is implemented as a primitive operation (`eval`) which has the type $(A \ (A \rightarrow B) \rightarrow$

[Copyright notice will appear here once ‘preprint’ option is removed.]

'B), where 'A and 'B are stack variable kinds. This is explained in depth further on.

The fundamental operation in Cat is the function composition operation. The Cat expression $a\ b\ c$ is equivalent to the mathematical function: $c(b(a(x)))$ whereas in an applicative language it would have been equivalent to $c(b)(a)$.

4. Cat Syntax and Semantics

The syntax and semantics of Cat are defined using small-step operational semantics. This consists of a presentation of the abstract syntactic form in figure 1, the computation rules in figure 2, the congruence rules in figure 3, and the kind system in figure 5. The core Cat primitive instructions and their types are defined in figure 4.

The only syntactic construct with any predefined semantic meaning in Cat is the quoting operation which is defined using the square brackets '[' and ']'. When a set of terms are surrounded by square brackets, they are pushed on to the stack as a single anonymous function. For example the expression $[2\ *]$ pushes an anonymous function on the stack that when evaluated multiplies the value on the top of stack by two.

Functions can be evaluated or dequoted using the application operation 'eval'. The 'eval' operation is a primitive function, that applies the function on the top of the stack to the rest of the stack, yielding a new stack. For example the following expression $21\ [2\ *]\ eval$ takes an input stack as input and yields a copy of the stack but with the value 42 pushed on top.

5. Kinding System

The Cat type system is defined in figure 5 using an elegant polymorphic kinding system due to Andreas Rossberg (Rossberg 2006). The Cat type system distinguishes between type kinds, which correspond to the category of individual values, and stack kinds, which correspond to the category of stacks of values.

The core set of typing rules for Cat are expressed in natural deduction in figure 6 and are also due to Andreas Rossberg (Rossberg 2006).

Type annotations in Cat are, for the most part, optional because in most cases the types can be inferred by the compiler using Hindley-Milner type inference (Hindley 1969), (Milner 1978) or a generalized form thereof (Aiken and Wimmers 1993). Cat types express the requirements of a function on the input stack and the relation of the input stack to the output stack. The constituent parts of a function type are called the consumption and production. The consumption is the configuration of types required on the top of the input stack and the production is the configuration of types expected on the top of the output stack.

The consumption of a function corresponds very closely to the notion of function arguments in mainstream functional and imperative languages. The production of a function corresponds to the results of a function. Unlike most imperative and functional programming languages, stack-based languages have the ability to return true multiple results without having to package them in a list or tuple.

Function types have the form: $(a_0, a_1, \dots, a_n \rightarrow b_0, b_1, \dots, b_m)$. The types to the left of the arrow represent the consumption, and the types to the right represent the production. In this case the type represents a function which can only be executed when the stack has the types $a_0 \dots a_n$ on top, where the topmost item has type a_n . Once executed the top n values will have been replaced with new values of type $b_0 \dots b_m$, where the top item has type b_m .

Types starting with a single forward quote character followed by a lower-case letter (e.g. 'a and 'b) are type variables. Types starting with an upper-case letter (e.g. 'A and 'B) are examples of

exp ::=	expressions (a.k.a. terms)
\emptyset	the null operation
[exp]	quotation
exp exp	concatenation
v	constant value
prim ::=	primitive functions
succ	replaces top value on the stack with its successor
pred	replaces top value on the stack with its predecessor
lteq	tests if the top value is greater than the next value
pop	removes top value from stack
dup	duplicate top value on stack
swap	swaps top two items on stack
eval	evaluates function on the top of the stack
dip	evaluate function temporarily removing top value
if	conditional evaluation
constantly	constant generating function
compose	function composition
bool ::=	boolean values
true	true value
false	false value
num ::=	numeric values
0	zero value
num succ	successor value
num pred	predecessor value
fun ::=	function values
[exp]	quotation
val ::=	values
bool	boolean value
num	numeric value
fun	function value

Figure 1. Syntactic Forms

$exp_0\ 0\ pred \mapsto exp_0\ -1$
$exp_0\ 0\ succ \mapsto exp_0\ 1$
$exp_0\ num_0\ pred\ succ \mapsto exp_0\ num_0$
$exp_0\ num_0\ num_0\ lteq \mapsto exp_0\ true$
$exp_0\ num_0\ num_0\ succ\ lteq \mapsto exp_0\ true$
$exp_0\ num_0\ num_0\ pred\ lteq \mapsto exp_0\ false$
$exp_0\ val_0\ pop \mapsto exp_0$
$exp_0\ val_0\ dup \mapsto exp_0\ val_0\ val_0$
$exp_0\ val_0\ val_1\ swap \mapsto exp_0\ val_1\ val_0$
$exp_0\ val_0\ constantly \mapsto exp_0\ [val_0]$
$exp_0\ [exp_1]\ [exp_2]\ compose \mapsto exp_0\ [exp_1\ exp_2]$
$exp_0\ [exp_1]\ eval \mapsto exp_0\ exp_1$
$exp_0\ val_0\ [exp_1]\ dip \mapsto exp_0\ exp_1\ val_0$
$exp_0\ true\ [exp_1]\ [exp_2]\ if \mapsto exp_0\ exp_1$
$exp_0\ false\ [exp_1]\ [exp_2]\ if \mapsto exp_0\ exp_2$
$exp_0\ [exp_1]\ [exp_2]\ while \mapsto exp_0\ exp_1\ []\ [[exp_1]\ [exp_2]\ while]\ if$

Figure 2. Computation Rules

$t1 \mapsto t1' \Rightarrow [t1] \mapsto [t1']$
$t1 \mapsto t1' \Rightarrow t1\ t2 \mapsto t1'\ t2$
$t2 \mapsto t2' \Rightarrow t1\ t2 \mapsto t1\ t2'$

Figure 3. Congruence Rules

```

0 : ( → int)
succ : (int → int)
pred : (int → int)
lteq : (int int → bool)
pop : ('a → )
dup : ('a → 'a 'a)
swap : ('a 'b → 'b 'a)
constantly : ('a → ( → 'a))
compose : (('A → 'B) ('B → 'C) → ('A → 'C))
eval : ('A ('A → 'B) → 'B)
dip : ('A 'b ('A → 'C) → 'C 'b)
if : ('A bool ('A → 'B) ('A → 'B) → 'B)
while : ('A ('A → 'A) ('A → 'A bool) → 'A)

```

Figure 4. The primitive operations and their types

t ::=	types
'a	type variable
(r → r)	function type
int	integer type
bool	boolean type
s ::=	stacks
'A	stack variable
s t	stack s on bottom and type t on top
∅	empty stack

Figure 5. Kinding system

$$\frac{}{T \vdash \emptyset : (\rho \rightarrow \rho)} \text{ (T-EMPTY)}$$

$$\frac{}{T \vdash \text{val} : (\rho \rightarrow r \rho)} \text{ (T-CONST)}$$

$$\frac{T \vdash \text{exp} : t}{T \vdash [\text{exp}] : (r \rightarrow r t)} \text{ (T-QUOTE)}$$

$$\frac{T \vdash \text{exp}_1 : (\rho_1 \rightarrow \rho_2), T \vdash \text{exp}_2 : (\rho_2 \rightarrow \rho_3)}{T \vdash \text{exp}_1 \text{ exp}_2 : (\rho_1 \rightarrow \rho_3)} \text{ (T-COMP)}$$

Figure 6. Typing rules

stack variables. A type variable corresponds to the type of precisely one value, whereas a stack variable corresponds to the types of zero or more values on the stack.

It is important to note that in Cat the types ('a → ('b → 'c)) and ('a 'b → 'c) are not equivalent. This differs from the functional languages that are based on function application (i.e. Standard ML (Milner et al. 1990)) where the two types are equivalent and can therefore be written using the shorthand: ('a → 'b → 'c).

Since every function in Cat maps from a single stack to another stack, the production and consumption components of the function type used in the type notation represents only the configuration of types on the top of the stack. From a type-theoretic standpoint, it is more accurate to include in any description of function types an extra stack variable, representing the rest of the stack.

Because stack variables represent sets of types, they need to be distinguished from types themselves. This is accomplished through the use of a kinding system. Stack variables are of a separate kind

than type variables, and are closely related to Wand's notion of row variables (Wand 1987).

If we were to be explicit in our notation we would explicitly use stack variables to represent the part of the stack that is unchanged in a function. For example the type of the function 'dup', which duplicates the top item on the stack, would be expressed as dup : (ρ 'a → ρ 'a 'a). Where ρ (the Greek letter rho) is a stack variable representing the rest of the stack.

6. Type Unification

According to the T-COMP typing rule, in order for two consecutive terms to be well typed, the stack produced by the first term must satisfy the consumption requirements of the second term. This requirement results in a set of constraints which must be satisfied in order for the expression to be well-typed.

If the expression is well-typed, then there exists a principal unifier for the set of constraints which can be found using the Hindley-Milner type unification algorithm (Hindley 1969), (Milner 1978), (Robinson 1971).

An important part of type unification for Cat is to assure that all variables appear for the first time on the left side of the arrow. Any term which is unable to do so is ill-typed, and is not a valid program.

7. Currying

Currying is the process of binding a function argument to a constant value thus generating a new function which requires one fewer arguments. This is one of the most fundamental and common functional programming techniques. In the Cat standard library a generic curry function is provided called 'rcurry' (for reverse curry), which will bind the top value on the stack to the value below it. In other words given a value 'x' on the top of the stack followed by a function 'f' evaluating the 'rcurry' function will create a new function 'g' which first places 'x' on the stack and then calls 'f'. The 'rcurry' function in Cat can be defined from the primitive operations as 'constantly swap compose'. Since currying is such a fundamental operation in functional programming, it is a useful example to demonstrate how the type of 'rcurry' can be derived from the typing rules. The step by step type derivation of curry is shown in figure 7. Keep in mind that lower-case variable names are type variables, upper-case letters are explicit stack variables, and greek variable names are implicit stack variables.

$$\frac{\text{constantly} : (\rho_1 a_1 \rightarrow \rho_1(\sigma_1 \rightarrow \sigma_1 a_1))}{\text{swap} : (\rho_2 a_2 b_2 \rightarrow \rho_2 b_2 a_2)} \text{--- COMP}$$

$$\frac{\text{constantly swap} : (\rho_1 a_1 \rightarrow \rho_2 b_2 a_2)}{(\rho_1(\sigma_1 \rightarrow \sigma_1 a_1)) = (\rho_2 a_2 b_2)} \text{--- UNIFY}$$

$$\frac{b_2 = (\sigma_1 \rightarrow \sigma_1 a_1), \rho_1 = \rho_2 a_2}{\text{let } a_3 = a_2, \text{let } b_3 = a_1, \text{let } \rho_3 = \rho_2, \text{let } \sigma_3 = \sigma_1}{\text{constantly swap} : (\rho_3 a_3 b_3 \rightarrow \rho_3(\sigma_3 \rightarrow \sigma_3 b_3) a_3)} \text{--- UNIFY}$$

$$\frac{\text{compose} : (\rho_4(A_4 \rightarrow B_4)(B_4 \rightarrow C_4) \rightarrow \rho_4(A_4 \rightarrow C_4))}{\text{constantly swap compose} : (\rho_3 a_3 b_3 \rightarrow \rho_4(A_4 \rightarrow C_4))} \text{--- COMP}$$

$$\frac{(\rho_3(\sigma_3 \rightarrow \sigma_3 b_3) a_3) = (\rho_4(A_4 \rightarrow B_4)(B_4 \rightarrow C_4))}{a_3 = (B_4 \rightarrow C_4), \sigma_3 b_3 = B_4, \sigma_3 = A_4, \rho_3 = \rho_4} \text{--- UNIFY}$$

$$\frac{\text{let } A_5 = A_4, \text{let } b_5 = b_3, \text{let } C_5 = C_4, \text{let } \rho_5 = \rho_4}{\text{constantly swap compose}} \text{--- UNIFY}$$

$$\text{ : } (\rho_5(A_5 b_5 \rightarrow C_5) b_5 \rightarrow \rho_5(A_5 \rightarrow C_5))$$

Figure 7. Type Derivation of the 'rcurry' Function

8. Type Safety

By definition a Cat program (P) is well-typed if the stack (S) produced by the program (P) is well-defined given an empty stack as input. A program (P_n) consisting of a sequence of terms $t_0 \dots t_{n-1}$ is well-typed iff $\forall i : i \leq n$ the program $P_i = t_0 \dots t_i$ is well-typed (t_i represent a term). The empty program (P_0) is well-typed and the empty stack (S_0) is well-defined.

$$S t : (\rho \sigma_0 \rightarrow \rho \sigma_1) \mapsto S' \Rightarrow S = \theta \sigma_0, S' = \theta \sigma_1$$

Figure 8. Type progression

The type progression rule in figure 8 shows that when a well-typed term (t) of type $(\rho \sigma_0 \rightarrow \rho \sigma_1)$ (where ρ , σ_0 , and σ_1 are stack kinds, representing sequences of types on the stack) is applied to a well-defined stack (S) the resulting stack (S') is well-defined iff S has the types contained in the stack σ_0 at the top. The type progression rule goes on to state that the resulting stack will contain the same types as the consumed stack, but with the types in σ_0 replaced with the types in σ_1 .

Quotations are held to a slightly weaker requirement of type soundness. A quotation ($Q = [t_0 \dots t_n]$), is well-typed iff $\exists S_q$ such that $S_q t_0 t_1 \dots t_n$ is a well-typed program.

9. Related Work

Stack-based application programming languages, such as Forth (Moore 1974), Desktop Calculator (DC) (Morris and Cherry 1978), Postscript (Inc. 1999), Joy (von Thun 2001) and others, have been around for nearly four decades. At the same time research interest in stack-based languages has largely been focused on either imperative stack-based languages (e.g. Java Virtual Machine Language (JVML) (Lindholm and Yellin 1999) and Common Intermediate Language (CIL) (ECMA 2002)), or functional languages with some stack facilities (e.g. Stack-based Typed Assembly Language (STAL) (Morrisett et al. 1998)).

Historically, stack-based languages have been mostly imperative in nature, with little or no support for functional programming. Manfred von Thun's programming language Joy made a significant contribution by combining the stack-based semantics of languages like Forth and Postscript and ideas from both the Functional Programming (FP) system described by John Backus (Backus 1978) and combinatorial logic (Curry and Feys 1958). Joy in turn has inspired several other functional stack-based languages, one of the most active projects being the Factor programming language (Pestov 2003). Recently Lynas and Stoddart also proposed extending Forth with Lambda expressions (Lynas and Stoddart 2006).

10. Conclusion and Future Directions

The type system presented here for the Cat language could be applied to other functional stack-based languages (such as Joy or Factor) to design a static typed system for those language. Similarly the idea of type-safe quotations and evaluation functions could be easily integrated into existing statically-typed stack-based languages (such as MSIL or JVML). Future planned extensions for the type system presented in this paper include typed lists, record types, recursive types, and algebraic types.

Acknowledgments

This work would not be possible without the generous contributions, patience, and inspiration of members of the Lambda-the-Ultimate.org and Artima.com online community, the concatenative discussion group, and the Types forum.

I wish to also acknowledge the following people for their generous help and inspiration: Andreas Rossberg, Peter Grogono, Martin Odersky, Markus Lumpe, Reno Camb, Mario B., Matt M., Sandro Magi, William Tanksley Jr., Manfred von Thun, Dan McCabe, Michael Stone, Robbert van Dalen, Brent Kerby.

Special thanks to Kris Unger, Matías Giovannini, Peter Selinger, and Francois Pottier for taking the time to provide very detailed reviews and discussions of this paper and previous drafts.

References

- Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *FPCA '93: Proceedings of the conference on Functional programming languages and computer architecture*, pages 31–41, New York, NY, USA, 1993. ACM Press. ISBN 0-89791-595-X.
- John Backus. Can programming be liberated from the von neumann style?: a functional style and its algebra of programs. *Source Communications of the ACM archive*, 21(8):613–641, August 1978. URL <http://portal.acm.org/citation.cfm?id=359579&coll=portal&>
- H. B. Curry and R. Feys. *Combinatory Logic*. North-Holland Publishing Company, 1958.
- Christopher Diggins. Cat programming language. URL <http://www.cat-language.com/>. Programming language implementation and documentation., 2007.
- ECMA. Standard ecma-335: Common language infrastructure (cli). Technical report, ECMA, December 2002.
- J. Roger Hindley. The principal type-scheme of an object in combinatory logic. (146):29–60, 1969.
- Paul Hudak et al. Report on the programming language Haskell: a non-strict, purely functional language version 1.2. *SIGPLAN Not.*, 27(5):1–164, 1992. ISSN 0362-1340.
- Adobe Systems Inc. *Postscript Language Reference*. Addison-Wesley, 3rd edition, 1999. URL <http://www.adobe.com/products/postscript/pdfs/PLRM.pdf>.
- Paul Blain Levy. Call-by-push-value: A subsuming paradigm. In *Typed Lambda Calculus and Applications*, pages 228–242, 1999. URL citeseer.ist.psu.edu/article/levy99callbypushvalue.html.
- Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Sun Microsystems Inc., 2nd edition, 1999. URL <http://java.sun.com/docs/books/vmspec/2nd-edition/html/VM>
- Angel Robert Lynas and Bill Stoddart. Adding Lambda expressions to Forth. In *Proceedings from the 22nd EuroForth Conference*, pages 27–39, 2006. URL <http://www.complang.tuwien.ac.at/anton/euroforth2006/pape>
- R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.
- Robin Milner. A theory of type polymorphism in programming. (17):348–375, August 1978.
- Charles Moore. Forth: a new way to program a mini-computer. *Astronomy and Astrophysics Supplement*, (15), 1974.
- Robert Morris and Lorinda Cherry. Dc - an interactive desk calculator. Technical report, AT&T Bell Laboratories, Murray Hill, New Jersey 07974, 1978. URL <http://citeseer.csail.mit.edu/83333.html>.
- Greg Morrisett, Karl Cray, Neal Glew, and David Walker. Stack-based typed assembly language. In *Work-*

- shop on Types in Compilation*, March 1998. URL <http://citeseer.ist.psu.edu/morrisett01stackbased.html>.
- Robert O’Callahan. A simple comprehensive type system for java bytecode subroutines. In *ACM Principles of Programming Languages*, 1998.
- Slava Pestov. Factor programming language. URL <http://www.factorcode.org/>. Programming language implementation and documentation., 2003.
- Jaanus Pöial. Typing tools for typeless stack languages. In *Proceedings from the 23rd EuroForth Conference*, pages 40–46, 2006. URL <http://www.complang.tuwien.ac.at/anton/euroforth2006/papers/poial.pdf>.
- Christophe Raffalli. Machine deduction, types for proofs and programs. *Lecture Notes in Computer Science*, 806:333–351, 1993.
- J. Alan Robinson. Computational logic: The unification computation. (6):63–72, 1971.
- Andreas Rossberg. Public discussions at lambda-the-ultimate.org. <http://lambda-the-ultimate.org/node/1899>, 2006.
- Raymie Stata and Martín Abadi. A type system for Java bytecode subroutines. In *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, pages 149–160, New York, NY, 1998. URL citeseer.ist.psu.edu/stata98type.html.
- Manfred von Thun. Joy: Forth’s functional cousin. In *Proceedings from the 17th EuroForth Conference*, 2001. URL <http://www.latrobe.edu.au/philosophy/phimvt/joy/forth-joy.html>.
- M. Wand. Complete type inference for simple objects. In *In Symposium on Logic in Computer Science*, June 1987. A Corrigenda appeared in LICS 88.