A Call-by-push-value Scheme 

Max S. New

University of Michigan

Scheme 2026

Some Historical Context

The year was 2018...

Some Historical Context

The year was 2018...

We had just written the paper that was the core of my thesis

Gradual Type Theory

MAX S. NEW, Northeastern University, USA
DANIEL R. LICATA, Wesleyan University, USA
AMAL AHMED, Northeastern University, USA and Inria Paris, France

Some Historical Context

The year was 2018...

We had just written the paper that was the core of my thesis

Gradual Type Theory

MAX S. NEW, Northeastern University, USA
DANIEL R. LICATA, Wesleyan University, USA
AMAL AHMED, Northeastern University, USA and Inria Paris, France

A paper with a *very* long extended version full of proofs

Some Historical Context

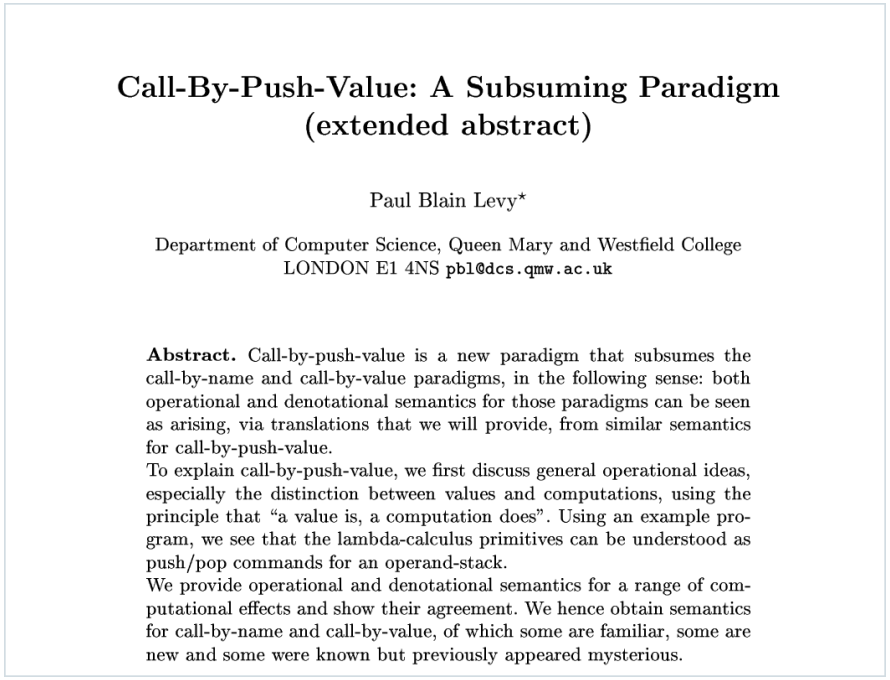
Dynamically Typed Call-by-push-value. Our interpretation of the dynamic types in CBPV suggests a design for a Scheme-like language with a value and computation distinction. This may be of interest for designing an extension of Typed Racket that efficiently supports CBN or a Scheme-like language with codata types. While the definition of the dynamic computation type by a lazy product may look strange, we argue that it is no stranger than the use of its dual, the sum type, in the definition of the dynamic value type. That is, in a truly dynamically typed language, we would not think of the dynamic type as being built out of some sum type construction, but rather that it

is the *union* of all of the ground value types, and the union happens to be a *disjoint* union and so we can model it as a sum type. In the dual, we don't think of the computation dynamic type as a *product*, but instead as the *intersection* of the ground computation types. Thinking of the type as unfolding:

$$\underline{¿} = \underline{F}\underline{¿} \wedge (? \to \underline{F}?) \wedge (? \to ? \to \underline{F}?) \wedge \cdots$$

This says that a dynamically typed computation is one that can be invoked with any finite number of arguments on the stack, a fairly accurate model of implementations of Scheme that pass multiple arguments on the stack.

Call-by-push-value and Me



Call-By-Push-Value: A Subsuming Paradigm
(extended abstract)

Paul Blain Levy*

Department of Computer Science, Queen Mary and Westfield College
LONDON E1 4NS pbl@dcs.qmw.ac.uk

Abstract. Call-by-push-value is a new paradigm that subsumes the call-by-name and call-by-value paradigms, in the following sense: both operational and denotational semantics for those paradigms can be seen as arising, via translations that we will provide, from similar semantics for call-by-push-value.
To explain call-by-push-value, we first discuss general operational ideas, especially the distinction between values and computations, using the principle that "a value is, a computation does". Using an example program, we see that the lambda-calculus primitives can be understood as push/pop commands for an operand-stack.
We provide operational and denotational semantics for a range of computational effects and show their agreement. We hence obtain semantics for call-by-name and call-by-value, of which some are familiar, some are new and some were known but previously appeared mysterious.

In our paper, we heavily used the CBPV calculus.

It simplified the semantics and proofs.

But what would a programming language directly based on CBPV look like?

A language-design experiment

Scheme + **CBPV** = **Fiddle**

What if Scheme were based on CBPV?

Minimalist Philosophy

Scheme 🤝 CBPV

Use composition to build complex features
from a small core.

CBPV separates values from computations

value types $\quad A ::= \quad U\underline{B} \mid \sum_{i \in I} A_i \mid 1 \mid A \times A$

computation types $\quad \underline{B} ::= \quad FA \mid \prod_{i \in I} \underline{B}_i \mid A \to \underline{B}$

Value types classify data.

Computation types classify the stacks that programs run against.

Source: Paul Blain Levy, Adjunction Semantics for Call-By-Push-Value (2004), slide 13.

Application pushes; lambda pops

$$\begin{array}{lrl} V`M & K & \rightsquigarrow \\ M & V :: K & \end{array}$$

$$\begin{array}{lrl} \lambda \mathtt{x}.N & V :: K & \rightsquigarrow \\ N[V/\mathtt{x}] & K & \end{array}$$

So $V`$ means push V and $\lambda \mathtt{x}$ means pop $\mathtt{x}$.

A computation in $A \to \underline{B}$ pops a value in A, then behaves in $\underline{B}$.

Source: Paul Blain Levy, Adjunction Semantics for Call-By-Push-Value (2004), slide 27.

CBV Function Decomposition

A CBV type translates into a value type.

$$A \to B \quad \mapsto \quad U(A \to FB)$$

CBV Function Decomposition

A CBV type translates into a value type.

$$A \to B \quad \mapsto \quad U(A \to FB)$$

U	Closure: suspended code.
A $\to$	Argument: a value supplied on the stack.
F B	Return continuation: what consumes the result.

Source: Paul Blain Levy, A Tutorial on Call-by-Push-Value (2016), slide 37.

Scheme Minimalism

A tiny set of primitive ingredients:

`atoms` + `cons cells` + `closures`

More complex data (lists, trees) built up from just a single binary data constructor

Scheme Minimalism?

Closures come with a richer argument protocol:

`(lambda (x y) ...)`	fixed arity
`(lambda (x y . rest) ...)`	collect extras as a list
`(case-lambda [(x) ...] [(x y) ...])`	dispatch on arity
`(apply f args)`	supply arguments from a list

Compound, arbitrary size constructs are baked into the core of procedures.

Fiddle: A CBPV Scheme

Treat the stack the way Scheme treats data.

	BASE CASE	EXTENSION	INSPECTION
VALUES	`atomic data`	`cons`	`car/cdr/cons?/nil?`
STACKS	`continuations`	`argument`	`case-lambda`

pushing an argument is the cons of the stack.

LIVE DEMO

```
#lang fiddle
```

Minimalistic Untyped CBPV Core Language

CODE	`(~ e)` ⇄ `(! v)`	thunk / force
ARGUMENT	`(^ e v)` ⇄ `(λ (x) e)`	push / pop
RESULT	`(ret v)` ⇄ `(bind [x e1] e2)`	return / bind

case-λ: minimalistic pattern match primitive for the stack

Fiddle Implementation

TURNSTILE	Implemented using the Turnstile library checks values vs. computations
VIRTUALIZED STACK	global mutable list threaded through the Racket program
INTEROP	Racket ⇄ Fiddle wrappers to mediate calling conventions

Thanks to my labmate Stephen Chang
for helping me to use Turnstile.

Fiddle Implementation and Libraries

CORE LANGUAGE	457
RUNTIME + PRELUDE	1,171
STANDARD LIBRARY	955
ADVENT OF CODE, 2018–19 · 29 DAYS	≈4,100

Advent of Code solutions were quite slow compared to direct Racket implementations.

Value constructions have stack duals

VALUE	STACK
cons	argument
patterns	copatterns

Value constructions have stack duals

VALUE	STACK
cons	argument
patterns	copatterns
atomic values	atomic stacks?

Value constructions have stack duals

VALUE	STACK
cons	argument
patterns	copatterns
atomic values	atomic stacks?
structs	methods

Nominal Stack Constructors: Methods

STRUCTS / DATA

named fields

opaque structs provide
data abstraction

METHODS / STACK

named stack slots

opaque methods provide
stack abstraction?

Can mechanisms like these help encode our sophisticated Scheme marks and delimiters smoothly?

Fiddle's Successor: Zydeco

FIDDLE 🎻

untyped prototype

Racket

→

ZYDECO 🪗

typed CBPV

System Fω polymorphism

Yuchen Jiang

Built with Yuchen Jiang and Michigan undergraduates.

THEORY

Notions of Stack-Manipulating Computation and Relative Monads

YUCHEN JIANG, University of Michigan, USA
RUNZE XUE, University of Michigan, USA and University of Cambridge, UK
MAX S. NEW, University of Michigan, USA

IMPLEMENTATION

efficient CBPV intermediate representations

ongoing work

Takeaways from Fiddle

RACKET

Racket and Turnstile made prototyping fast.

Starting untyped made it easy to play with CBPV code.

CBPV

CBPV brings stack manipulation into a functional language.

Methods and copatterns came from treating it like data.

SELF CARE

Successful Procrastiworking

It helped me not burn out and play with side research ideas that I picked up again years later.

Conclusion

Scheme + CBPV = Fiddle

Pushing an argument is the cons of the stack.

Fiddle

A call-by-push-value Scheme

https://github.com/zydeco-lang/fiddle

Zydeco

A typed call-by-push-value language

https://github.com/zydeco-lang/zydeco