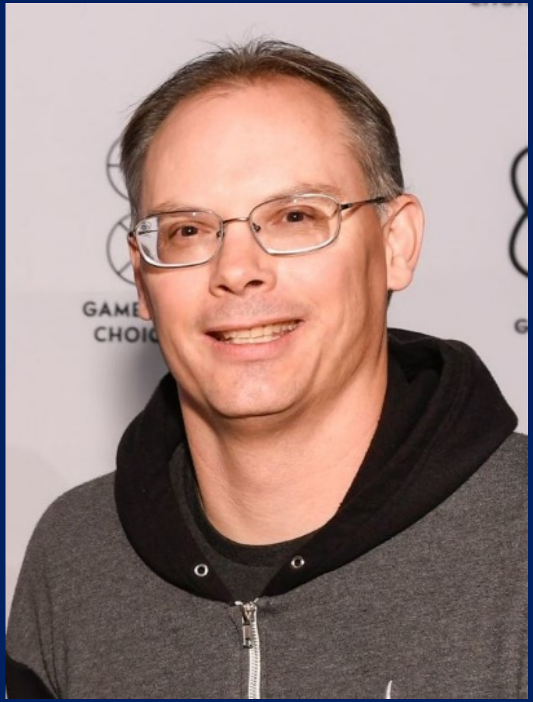
BEYOND FUNCTIONAL PROGRAMMING: THE VERSE PROGRAMMING LANGUAGE



Simon Peyton Jones, Tim Sweeney
Lennart Augustsson, Koen Claessen, Ranjit Jhala, Olin Shivers
Epic Games

December 2022

Verse: a language for the metaverse

Tim's vision of the metaverse

- Social interaction in a shared real-time 3D simulation
- An open economy with rules but no corporate overlord
- A creation platform open to all programmers, artists, and designers, not a walled garden
- Much more than a collection of separately compiled, statically-linked apps: everyone's code and content must interoperate dynamically, with live updates of running code
- Pervasive open standards. Not just Unreal, but any other game/simulation engine e.g. Unity.

Verse is open

Like the metaverse vision, Verse itself is open

- We will publish papers, specification for anyone to implement
- We will offer compiler, verifier, runtime under permissive open-source license with no IP encumbrances.

Goal: engage in a rich dialogue with the community that will make Verse better.

Do we really need a new language?

- Objectively: no. All languages are Turing-complete.
- But we think we can do better with a new language
  - Transactional from the get-go; the only plausible way to manage concurrence across 1M+ programmers
  - Strong interop guarantees over time: compile time guarantees that a module subsumes the API of the previous version.
  - Scalable to running code, written by millions of programmers who do not know each other, that supports billions of users
- And ...
  - Learnable as a first language (c.f. Javascript yes, C++ no)
  - Extensible: mechanisms for the language to grow over time, without breaking code.

A taste of Verse

- Verse 1: a familiar FP subset
- Verse 2: choice
- Verse 3: functional logic

View from 100,000 feet

- Verse is a functional logic language (like Curry or Mercury).
- Verse is a declarative language: a variable names a single value, not a cell whose value changes over time.
- Verse is lenient but not strict:
  - Like strict:, everything gets evaluated in the end
  - Like lazy: functions can be called before the argument has a value
- Verse has an unusual static type system: types are first-class values.
- Verse has an effect system rather than using monads.

A taste of Verse

- A subset of Verse is a fairly ordinary functional language
- Integers `3` `3+7`
- Tuples/arrays `(3,4)` `((92,2),3, 4)`

`fst(3,4)`

"array{..}" is long-form syntax

`array{3,4}`

`array{3}`

Singleton tuple

Bindings

```
x:=3; x+x
```

Syntax: ":=" and ";"

```
x:=3; y:=x+1; x*y
```

For now, think "letrec-binding"

```
y:=x+1; x:=3; x*y
```

Order does not matter

Functions and lambda

```
f(x:int):int := x+1;  f(3)
```

Arguments on the LHS…

```
f:=(x:int=>x+1); f(3)
```

..or use lambda

Verse uses infix "=>" for lambda

Conditionals and recursion

- A subset of Verse is a fairly ordinary functional langauge

```
fac(x:int):int :=
  if (x=0) then 1 else n * fac(n-1)
```

Conditionals

Recursion

Verse 2: choice

Choice

- A Haskell expression denotes **one** value
- A Verse expression denotes a **sequence of zero or more** values

`3`	One value
`3 | 4`	Two values
`false?`	Zero values
`1..10`	Ten values

Choice operator (pointing to `|`)

A notation for "fail" (pointing to `false?`)

Binding and choices

```
x:=(1|7|2); x+1
```

Denotes sequence of three values: 2, 8, 3

- A bit like Haskell [x+1 | x <- [1,7,2]]
- Key point: a variable is always bound to a single value, not to a sequence of values. I.e.
  - We execute the (x+1) with x bound to 1, then with x bound to 7, then with x bound to 2.
  - Not with x bound to (1|7|2)

Nested choices

- What sequence of values does this denote?

```
x:=(1|2); y:=(7|8); (x,y)
```

- Answer: `(1,7), (1,8), (2,7), (2,8)`
- Like a nested for-loop
- Like Haskell list comprehension [(x,y) | x<-[1,2], y<-[7,8]]
- But more fundamentally built in
- Key point: a variable is always bound to a single value, not to a sequence of values

Nested choices

```
x:=(1|2); y:=(7|8); (x,y)
```

- You can also write `((1|2), (7|8))`
  - This still produces the same *sequence of pairs*, *not* a single pair containing two sequences!
- Same for all operations

`77 + (1|3)` means the same as `(77+1) | (77+3)`

`77 + false?` means the same as `false?`

Nested choices and funky order

- What sequence of values does this denote?

```
x:=(y|2); y:=(7|8); (x,y)
```

- Answer: `(7,7), (8,8), (2,7), (2,8)`
- Order of results is still left-to-right
- But data dependencies can be "backwards"
- (Not like Haskell list comprehensions!)

Conditionals

- No Booleans!

```
if (e) then e1 else e2
```

- Returns e1 if e succeeds
  - "Succeeds" = returns one or more values
- Returns e2 if e fails
  - "Fails" = returns zero values

Comparisons

```
if (x<20) then e1 else e2
```

- (x<20)
  - fails if x >= 20
  - succeeds if x < 20, *returning the left operand*
- Example: (3 + (x<20))
  - Succeeds if x=7, returning 10
  - Fails if x=25
- Example: (0 < x < 20)
  - Succeeds if x is between 0 and 20, returning 0
  - Fails if x is out of range
  - (<) is right-associative

```
if (0<x<20) then e1 else e2
```

c.f. Haskell

```
if (0<x && x<20) then … else …
```

Conjunction and disjunction

```
if (x<20, y>0) then e1 else e2
```

- The tuple expression (x<20,y>0) fails if either (x<20) or (y>0) fails

```
if (x<20 | y>0) then e1 else e2
```

- Choice succeeds if either branch succeeds

Equality

```
if (x=0) then e1 else e2
```

- (x=0)
  - fails if x is not zero
  - succeeds if x is zero, *returning x*

As we will see, "=" is a super-important operator

- *"If x is 2 or 3 then…"*

```
if (x=(2|3)) then e1 else e2
```

c.f. Haskell

```
if (x==2 || x==3) then … else…
```

From choice to tuples

- `for` turns a choice into a tuple/array

`for{ 3 }`	The singleton tuple, array(3)
`for{ 3 | 4 }`	The tuple (3,4)
`for{ false? }`	The empty tuple ()
`for{ 1..10 }`	The tuple (1,2,…, 10)

Order is important

- `for` turns a choice into a tuple/array

```
for{ 3 | 4 }
```

The tuple (3,4)

```
for{ 4 | 3 }
```

The tuple (4,3)

- That's why we say that an expression denotes a sequence of values, not a bag of values, and definitely not a set.
- So "|" is associative but *not* commutative

From tuples to choice

- ? turns a tuple/array into a choice

```
(3,4)?
```

The choice (3 | 4)

```
for{ e }?
```

Same as e

- false := (), the empty tuple
  so false? always fails.

Generalising for

```
for e1 do e2
```

Iterate over the N (non-failing) choices in the **domain** e1

Form the N-tuple from the value(s) of **range** e2
(variables bound in e1 scope over e2)

```
for (i:=1..3) do i*i
```

=

```
( (1*1), (2*2), (3*3))
```

=

```
(1,4,9)
```

Generalising for

```
for e1 do e2
```

Iterate over the N (non-failing) choices in the **domain** e1

Form the N-tuple from the value(s) of **range** e2
(variables bound in e1 scope over e2)

- Range expression can yield **multiple values**

```
for (i:=1..3) do (i|i+7)
```

=

```
( (1|8), (2|9), (3|10) )
```

=

```
(1,2,3) | (1,2,10) |
(1,9,3) | (1,9,10) |
..
```

And we can use that choice to iterate:

```
xs := for(1..5) do (0|1|2); ...xs...
```

xs is successively bound to all 5-digit numbers in base 3

Generalising for

```
for e1 do e2
```

Iterate over the N (non-failing) choices in the **domain** e1

Form the N-tuple from the value(s) of **range** e2 (variables bound in e1 scope over e2)

- Range expression can **fail**

```
for (i:=1..4) do (i<3)
```

=

```
(1<3, 2<3, 3<3, 4<3)
```

=

```
(1, 2, false?, false?)
```

=

```
false?
```

Generalising for

```
for e1 do e2
```

Iterate over the N (non-failing) choices in the **domain** e1

Form the N-tuple from the value(s) of **range** e2
(variables bound in e1 scope over e2)

- Domain expression can **fail**

```
for (i:=1..4, isEven(i)) do (i*i)
```

=
```
(2*2, 4*4)
```

=
```
(4,16)
```

Indexing arrays

- Indexing an array/tuple *fails on bad indices*

```
as:=(3,7,4)
```

`as[0]`	Denotes one value, 3
`as[2]`	Denotes one value, 4
`as[7]`	Fails: denotes zero values

1..n is (1 | 2 | … | n)

```
for{i:=1..Length(as); as[i]+1}
```

Returns (4,8,5)

```
if (x:=as[i]) then x+1 else 0
```

Returns 0 if i is out of range

Narrowing

```
as:=(3,7,4);
for{i:int; as[i]+1}
```

- What values can i take? Clearly just 0,1,2!
- So expand as[i] to those three choices
- This is called "narrowing" in the functional logic literature

```
as:=(3,7,4);
for{i:int; as[i] + 1}
```

=

```
as:=(3,7,4);
for{i:int; ((i=0; 3+1) |
            (i=1; 7+1) |
            (i=2; 4+1)) }
```

Haskell

```
array (bounds a) [ (i,a!i + 1) | i<-indices a ]
```

Some functions

```
head(xs)        := xs(0)
tail(xs)        := for{i:int; i>0; xs[i]}
cons(x,xs)      := for{x | xs[i:int]}
snoc(xs,x)      := for{xs[i:int] | x}
append(xs,ys)   := for{xs[i:int] | ys[j:int]}
map(f,xs)       := for{f(xs[i:int])}
```

Fails on empty tuple

Verse 3: functional logic

Separating "bring into scope" from "give value"

```
x:=7; x+1>3; y=x*2
```

means the same as

```
x:int; x=7; x+1>3; y=x*2
```

Bring x into scope.
I'm not telling you what its value is yet

By the way, x must be 7 (or else fail)

The very same "=" as before

Separating "bring into scope" from "give value"

```
x:=7; x+1>3; y=x*2
```

means the same as

```
x:int; x=7; x+1>3; y=x*2
```

means the same as

```
      x=7; x+1>3; y=(x:int)*2
```

```
           x+1>3; y=(x:=7)*2
```

Think:
- ":" brings the variable into scope.
- Scope extends to the left as well as right

Towards functional logic programming

- Haskell

```
let (y,z) = if (x=0)  then (3,4)
                      else (232, 913)
in y+z
```

- Verse

Bring y,z into scope

```
y:int; z:int;
if (x=0) then { y=3;   z=4 }
          else { y=232; z=913 };
y+z
```

Give them values

Towards functional logic programming

- Partial values

```
x:tuple(int,int);
x = (2,y:int);
x = (z:int,3);
x
```

x's first component is 2
y is a fresh unbound variable

x's second component is 3
z is a fresh unbound variable

Towards functional logic programming

- You can even pass those in-scope-but-unbound variables to a function

```
f(p:int,q:int):int
   := if (x=0) then { p=3;   q=4 }
                else { p=232; q=913 };
y:int; z:int;
f(y,z);
y+z
```

Pass y,z to f, which binds each of them to a value

...and add up the results

Towards functional logic programming

```
f(p:int,q:int):int :=
  if (x=0) then { p=3;    q=4 }
               else { p=232; q=913 };
y:int; z:int;
f(y,z);
y+z
```

- y,z look very like logical variables in Prolog, aka "unification variables".
- And "=" looks very like unification.

Towards functional logic programming

- We can do the usual "run functions backwards" thing

```
swap(x:int, y:int) := (y,x)
```

```
swap(3,4)
```

Run swap "forward": returns (4,3)

```
w:tuple(int,int);
swap(w) = (3,4);
w
```

Run swap "backward": Also returns (4,3)

Flexible and rigid variables

- What does this do?

```
x:int; y:int;
if (x=0) then y=1 else y=2;
x=7;
y
```

Sets the value of x

Reads the value of x

Sets the value of y

- One plan (Curry): two different equality operators
- Verse plan:
  - inside a conditional scrutinee, variables bound outside (e.g. x) are "rigid" and can only be read, not unified
  - outside, x is "flexible" and can be unified

Lenience

- Clearly Verse cannot be strict
  - call-by-value
  - with a defined evaluation order

  because earlier bindings may refer to later ones;
  and functions can take as-yet-unbound logical variables as arguments
- And it cannot be lazy, because all those "=" unifications must happen, to give values to variables.
- So Verse is lenient
  - Everything is eventually evaluated
  - But only when it is "ready"
  - Like dataflow

```
x:int;
if (x=0) …;
f(x);
…
```

'if' is stuck until x gets a value

Let's hope f gives x its value

Making it all precise

Designing the aeroplane during take-off

- MaxVerse: the glorious vision.
  A significant research project in its own right.
- ShipVerse: a conservative subset we will ship to users in 2023.

Core Verse

- MaxVerse is a big language

```
MaxVerse code
```

- To give it precise semantics, we use a small Core Verse language:
  - Desugar MaxVerse into CoreVerse

```
CoreVerse code
```

  - Give precise semantics to CoreVerse
  - CoreVerse might well be a good compiler intermediate language
- Analogy:
  - MaxVerse = Haskell
  - CoreVerse = Lambda calculus

Core Verse

Integers	k		
Variables	x, y, z, f, g		
Primops	op	$::=$	$\mathbf{gt} \mid \mathbf{add}$
Values	v	$::=$	$x \mid k \mid op \mid \langle s_1, \cdots, s_n \rangle \mid \lambda x.\, e$
Expressions	e	$::=$	$v \mid eu;\, e \mid \exists x.\, e \mid \mathbf{fail} \mid e_1 \,\mathbf{\vert}\, e_2 \mid v_1\, v_2 \mid \mathbf{one}\{e\} \mid \mathbf{all}\{e\}$
	eu	$::=$	$e \mid v = e$

- "=" is a language construct, not a primop (like gt)
- <v1,..,vn> for tuples to avoid ambiguity with (x)
- "∃x" is what we previously wrote "x:any" (except I'm not telling you about types)
- **fail** is a language construct, alongside "|"
- Core Verse is untyped

```
x:tuple(int,int);
x = (2,y:int);
x = (z:int,3);
x
```

"Exists"

Desugar

```
∃x. x = (∃y. <2,y>);
    x = (∃z. <z,3>);
    x
```

- Main constructs
  - exists ∃ brings a variable into scope
  - unification = says that two expressions have the same value
  - sequencing ; sequences unifications
  - choice |, fail
  - conditional one return first success
  - for-loops all return all successes

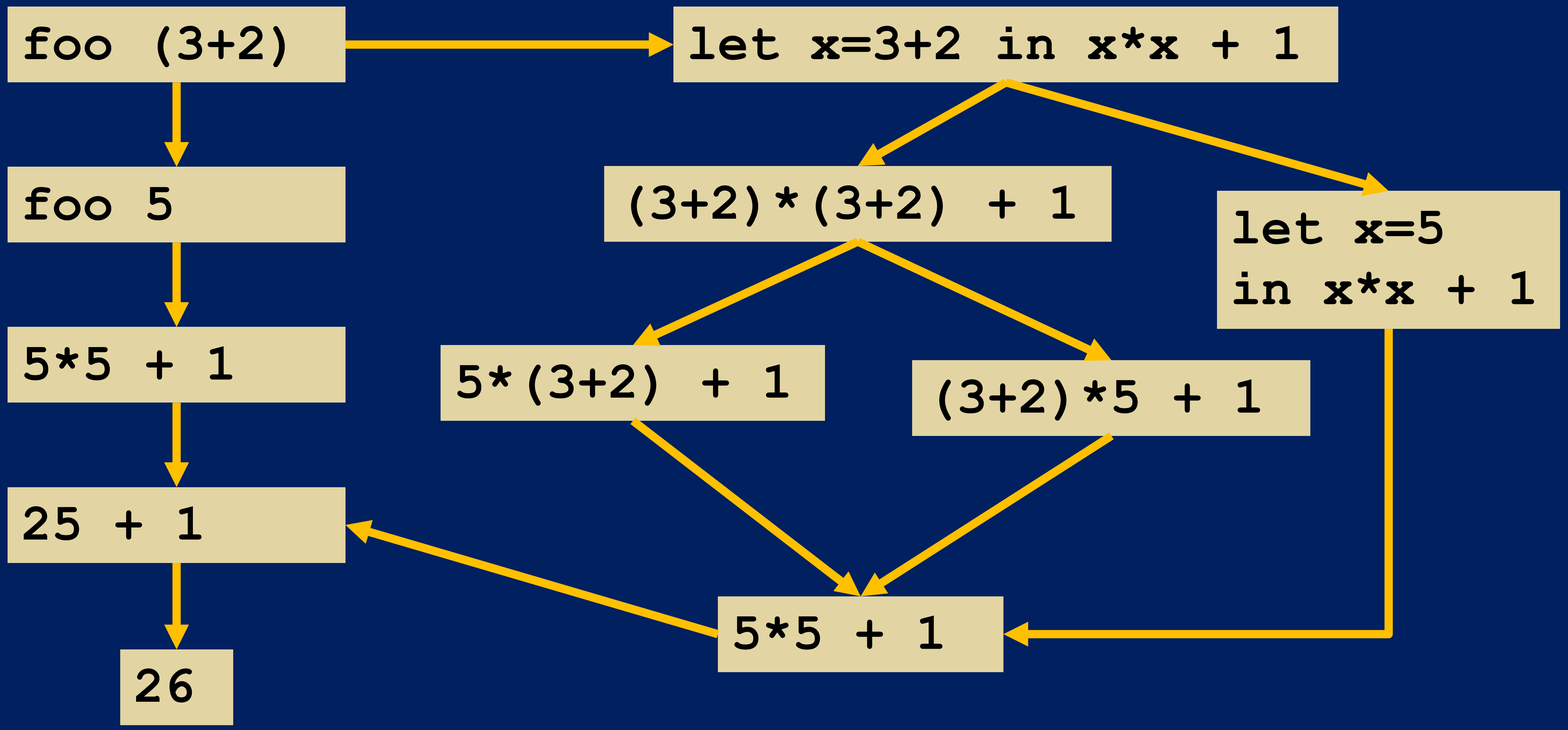
What is execution?

```
∃x. x = (∃y. <2,y>);
    x = (∃z. <z,3>);
    x
```

- Execution = "solve the equations"
  - Find values for the exists variables that make all the equations true.
- In this example:
  - `x=<2,3>, z=2, y=3`
- Operationally: unification.
- But unification is hard for programmers
  - backtracking, choice points, undoing, rigid variables, …

Idea! Use rewriting

```
foo x = x*x + 1
```



Rewriting: key ideas

- To answer "what does this program do, or what does it mean?" just apply the rewrite rules
- Rewrite rules are like
  - Add/multiply constants
  - Replace a function call with a copy of the function's RHS, making substitutions
  - Substitute for a let-binding
- You can apply any rewrite rule, anywhere, anytime
  - They should all lead to the same answer ("confluence")
- Good as a way to explain to a programmer: just source-to-source rewrites
- Good for compilers, when optimising/transforming the program
- Not good as a final execution mechanism

Execution = rewriting

```
x:tuple(int,int);
x = (2,y:int);
x = (z:int,3);
x
```

Desugar

```
∃x. x = (∃y. ⟨2,y⟩);
    x = (∃z. ⟨z,3⟩);
    x
```

Execution = rewriting

```
x:tuple(int,int);
x = (2,y:int);
x = (z:int,3);
x
```

Desugar

```
∃x. x = (∃y. ⟨2,y⟩);
    x = (∃z. ⟨z,3⟩);
    x
```

Float ∃

```
∃x. ∃y. ∃z. x = ⟨2,y⟩;
            x = ⟨z,3⟩;
            x
```

Execution = rewriting

```
x:tuple(int,int);
x = (2,y:int);
x = (z:int,3);
x
```

Desugar

```
∃x. x = (∃y. ⟨2,y⟩);
    x = (∃z. ⟨z,3⟩);
    x
```

Float ∃

```
∃x. ∃y. ∃z. x = ⟨2,y⟩;
            x = ⟨z,3⟩;
            x
```

```
∃xyz. x = ⟨2,y⟩; ⟨2,y⟩=⟨z,3⟩; x
```

Substitute for
(one occurrence of) x

Execution = rewriting

```
x:tuple(int,int);
x = (2,y:int);
x = (z:int,3);
x
```

Desugar

```
∃x. x = (∃y. ⟨2,y⟩);
    x = (∃z. ⟨z,3⟩);
    x
```

Float ∃

```
∃x. ∃y. ∃z. x = ⟨2,y⟩;
            x = ⟨z,3⟩;
            x
```

```
∃xyz. x = ⟨2,y⟩; ⟨2,y⟩=⟨z,3⟩; x
```

```
∃xyz. x = ⟨2,y⟩; z=2; y=3; x
```

Decompose equality of pairs (unification)

Execution = rewriting

```
x:tuple(int,int);
x = (2,y:int);
x = (z:int,3);
x
```

Desugar

```
∃x. x = (∃y. ⟨2,y⟩);
    x = (∃z. ⟨z,3⟩);
    x
```

Substitute for y

```
⟨2,y⟩;
⟨z,3⟩;
```

Substitute for another occurrence of x

```
x = ⟨2,y⟩;
```

Garbage collect

```
∃xyz. x = ... y⟩; y=3; z=2; x
```

```
∃xyz. x = ⟨2,y⟩; y=3; z=2; ⟨2,y⟩
```

```
∃xyz. x = ⟨2,y⟩; y=3; z=2; ⟨2,3⟩
```

```
⟨2,3⟩
```

An alternative sequence

```
x:tuple(int,int);
x = (2,y:int);
x = (z:int,3);
x
```

Desugar

```
∃x. x = (∃y. ⟨2,y⟩);
    x = (∃z. ⟨z,3⟩);
    x
```

Float ∃

```
∃x. ∃y. ∃z. x = ⟨2,y⟩;
            x = ⟨z,3⟩;
            x
```

```
∃x. ∃y. ∃z. x = ⟨2,y⟩;
            x = ⟨z,3⟩;
            ⟨z,3⟩
```

```
∃xyz. x = ⟨2,y⟩; ⟨2,y⟩=⟨z,3⟩; ⟨z,3⟩
```

```
∃xyz. x = ⟨2,y⟩; z=2; y=3; ⟨z,3⟩
```

```
∃xyz. x = ⟨2,y⟩; z=2; y=3; ⟨2,3⟩
```

```
⟨2,3⟩
```

Unification rewrite rules

U-SCALAR	$s = s;\ e$	$\longrightarrow$	e	
U-TUP	$\langle v_1, \cdots, v_n \rangle = \langle v'_1, \cdots, v'_n \rangle;\ e$	$\longrightarrow$	$v_1 = v'_1; \cdots; v_n = v'_n;\ e$	
U-FAIL	$hnf_1 = hnf_2$	$\longrightarrow$	**fail**	if neither U-SCALAR nor U-TUP match

Scalar Values	s	$::=$	$x \mid k \mid op$
Heap Values	h	$::=$	$\langle v_1, \cdots, v_n \rangle \mid \lambda x.\ e$
Head Values	hnf	$::=$	$h \mid k$
Values	v	$::=$	$s \mid h$
Expressions	e	$::=$	$v \mid eu;\ e \mid \exists x.\ e \mid \textbf{fail} \mid e_1 \,\|\, e_2 \mid v_1\ v_2 \mid \textbf{one}\{e\} \mid \textbf{all}\{e\}$
	eu	$::=$	$e \mid v = e$

Primitive operations

Application: $\mathcal{A}$

APP-BETA	$(\lambda x.\, e)\ v$	$\longrightarrow$	$\exists x.\, x = v;\ e$	if $x \notin \mathrm{fvs}(v)$
APP-TUP0	$\langle\rangle\ v$	$\longrightarrow$	**fail**	
APP-TUP	$\langle v_0\ \cdots\ v_n\rangle\ v$	$\longrightarrow$	$\exists x.\, x = v;\ (x = 0;\ v_0 \mid \cdots \mid x = n;\ v_n)$	if $x \notin \mathrm{fvs}(v), n \geqslant 0$
APP-ADD	$\mathbf{add}\langle k_1, k_2\rangle$	$\longrightarrow$	$k_1 + k_2$	
APP-GT	$\mathbf{gt}\langle k_1, k_2\rangle$	$\longrightarrow$	k_1	if $k_1 > k_2$
APP-GT-FAIL	$\mathbf{gt}\langle k_1, k_2\rangle$	$\longrightarrow$	**fail**	if $k_1 \leqslant k_2$

Normalisation rewrite rules getting stuff "out of the way"

Normalization: $\mathcal{N}$

NORM-VAL	$v;\ e$	$\longrightarrow$	e	
NORM-SEQ-ASSOC	$(eu;\ e_1);\ e_2$	$\longrightarrow$	$eu;\ (e_1;\ e_2)$	
NORM-SEQ-SWAP1	$eu;\ (x = v;\ e)$	$\longrightarrow$	$x = v;\ (eu;\ e)$	if eu not of form $x' = v'$
NORM-SEQ-SWAP2	$eu;\ (x = s;\ e)$	$\longrightarrow$	$x = s;\ (eu;\ e)$	if eu not of form $x' = s'$
NORM-EQ-SWAP	$hnf = x$	$\longrightarrow$	$x = hnf$	
NORM-SEQ-DEFR	$(\exists x.\ e_1);\ e_2$	$\longrightarrow$	$\exists x.\ (e_1;\ e_2)$	if $x \notin \mathrm{fvs}(e_2)$
NORM-SEQ-DEFL	$eu;\ (\exists x.\ e)$	$\longrightarrow$	$\exists x.\ eu;\ e$	if $x \notin \mathrm{fvs}(eu)$
NORM-DEFR	$v = (\exists y.\ e_1);\ e_2$	$\longrightarrow$	$\exists y.\ v = e_1;\ e_2$	if $y \notin \mathrm{fvs}(v, e_2)$
NORM-SEQR	$v = (eu;\ e_1);\ e_2$	$\longrightarrow$	$eu;\ v = e_1;\ e_2$	

Conditionals

Scalar Values	s	$::=$	$x \mid k \mid op$
Heap Values	h	$::=$	$\langle v_1, \cdots, v_n \rangle \mid \lambda x.\, e$
Head Values	hnf	$::=$	$h \mid k$
Values	v	$::=$	$s \mid h$
Expressions	e	$::=$	$v \mid eu;\, e \mid \exists x.\, e \mid \textbf{fail} \mid e_1 \mathbin{\textbf{\|}} e_2 \mid v_1\, v_2 \mid \textbf{one}\{e\} \mid \textbf{all}\{e\}$
	eu	$::=$	$e \mid v = e$

- Desugar conditionals like this:

> one: a new, simpler construct

$$\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \quad \text{means} \quad \exists y.\, y = \textbf{one}\{(e_1;\, \lambda x.\, e_2) \mathbin{\textbf{|}} (\lambda x.\, e_3)\};\, y\langle\rangle$$

> Variables bound in e1 can scope over e2

- Rewrite rules for one

ONE-FAIL	$\textbf{one}\{\textbf{fail}\}$	$\longrightarrow$	$\textbf{fail}$
ONE-CHOICE	$\textbf{one}\{v_1 \mathbin{\textbf{\|}} e_2\}$	$\longrightarrow$	v_1
ONE-VALUE	$\textbf{one}\{v\}$	$\longrightarrow$	v

Loops

Scalar Values	s	$::=$	$x \mid k \mid op$
Heap Values	h	$::=$	$\langle v_1, \cdots, v_n \rangle \mid \lambda x.\, e$
Head Values	hnf	$::=$	$h \mid k$
Values	v	$::=$	$s \mid h$
Expressions	e	$::=$	$v \mid eu;\, e \mid \exists x.\, e \mid \textbf{fail} \mid e_1 \,\|\, e_2 \mid v_1\, v_2 \mid \textbf{one}\{e\} \mid \textbf{all}\{e\}$
	eu	$::=$	$e \mid v = e$

- Desugar for-loops like this:

$\textbf{for}\ e$	means	$\textbf{all}\{e\}$
$\textbf{for}(e_1)\ \textbf{do}\ e_2$	means	$\exists y.\, y = \textbf{all}\{e_1;\, \lambda x.\, e_2\};\ map\langle \lambda z.\, z\langle\rangle, y\rangle$

Variables bound in e1 can scope over e2

- Rewrite rules for 'all'

ALL-FAIL	$\textbf{for}\{\textbf{fail}\}$	$\longrightarrow$	$\langle\rangle$
ALL-CHOICE	$\textbf{for}\{v_1 \,\|\, \cdots \,\|\, v_n\}$	$\longrightarrow$	$\langle v_1, \cdots, v_n \rangle$

Choice

- How to rewrite (e1 | e2)?

$$\text{CHOOSE} \quad CX[\, e_1 \,|\, e_2 \,] \longrightarrow CX[\, e_1 \,] \,|\, CX[\, e_2 \,] \quad \text{if } CX \neq \square$$

Duplicate surrounding context

E.g. (x + (y | z) *2) → (x + y*2) | (x + z*2)

Choice context	$CX ::= \square \mid v = CX \mid CX;\ e \mid ce;\ CX \mid \exists x.\, CX$
Choice-free expr	$ce ::= v \mid v = ce \mid ce_1;\ ce_2 \mid \mathbf{one}\{e\} \mid \mathbf{all}\{e\} \mid op(v) \mid \exists x.\, ce$

More in the paper...

https://simon.peytonjones.org/verse-calculus

- First attempt to give a deterministic rewrite semantics to a functional logic language.
- Much more detail, lots of examples
- Sad lack of a confluence proof. It's tricky. Details may change.

There is more. A *lot* more.

- Mutable state, I/O, and other effects.
  - An effect system, not a monadic setup
- Pervasive transactional memory
- Structs, classes, inheritance
- The type system and the verifier – lots of cool stuff here

Types

- In Verse, a "type" is simply a function
  - that fails on values outside the type
  - and succeeds on values inside the type
- So `int` is the identity function on integers, and fails otherwise
- `isEven` (which succeeds on even numbers and fails otherwise) is a type
- `array int` succeeds on arrays, all of whose elements are integers... hmm, scratch head... 'array' is simply 'map'!
- $(\lambda x. \exists p, q. x = \langle p, q \rangle; p < q)$ is the type of pairs whose first component is smaller than the second
- The Verifier rejects programs that might go wrong. This is wildly undecidable in general, but the Verifier does its best.

Take-aways

- Verse is extremely ambitious
  - Kick functional logic programming out the lab and into the mainstream
  - Stretches from end users to professional developers
  - Transactional memory at scale
  - Very strong stability guarantees
  - A radical new approach to types
- Verse is open
  - Open spec, open-source compiler, published papers (I hope!)

Before long: a conversation to which you can contribute