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
  - Scalable to running code, written by millions of programmers who do not know each other, that supports billions of users
  - 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.

- 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
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 subset of Verse is a fairly ordinary functional language.

- **Integers**
  - 3
  - 3+7

- **Tuples/arrays**
  - (3,4)
  - ((92,2),3,4)
  - `fst(3,4)`
  - `a[7]`
  - `array{3,4}`
  - `array{3}`

“array{..}” is long-form syntax.

- Singleton tuple
- Indexing
Syntax: ":=" and ";" 

For now, think "letrec-binding"

Order does not matter
Functions and lambda

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

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

Verse uses infix "=>" for lambda

Arguments on the LHS...

..or use lambda
fac(x:int):int :=
  if (x=0) then 1 else n * fac(n-1)
Verse 2: choice
A Haskell expression denotes one value.

A Verse expression denotes a sequence of zero or more values.

- Choice operator

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>One value</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>false?</td>
<td>Zero values</td>
</tr>
<tr>
<td>1..10</td>
<td>Ten values</td>
</tr>
</tbody>
</table>

A quirky notation for “fail”
Binding and choices

- A bit like Haskell list comprehension

- 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)

\[ x := (1|7|2); \ x+1 \]

Denotes sequence of three values: 2, 8, 3

\[ [x+1 \mid x <- [1,7,2]] \]
What sequence of values does this denote?

\[ x := (1 \mid 2); \quad y := (7 \mid 8); \quad (x, y) \]

- Answer: \((1, 7), (1, 8), (2, 7), (2, 8)\)
- Like Haskell list comprehension: \([ (x, y) \mid x \leftarrow [1, 2]; \ y \leftarrow [7, 8] ]\)
- But more fundamentally built in
- Key point again: a variable is always bound to a single value, not to a sequence of values
You can also write

\[(x, y) = ((1|2), (7|8))\]

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 \[(77+1) | (77+3)\]

\[77 + \text{false?}\] means the same as \[\text{false?}\]
What sequence of values does this denote?

\[ x := (y | 2); \quad y := (7 | 8); \quad (x, y) \]

- Answer: \((7, 7), \quad (8, 8), \quad (2, 7), \quad (2, 8)\)
- Order of results is still left-to-right
- But data dependencies can be “backwards”
- Haskell

\[
[(x, y) \mid x <- [y, 2]; y <- [7, 8]]
\]

-- Rejected!
Conditionals

- No Booleans!

```plaintext
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 \(x \geq 20\)
  - 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

\[
\text{if } (0<x<20) \text{ then e1 else e2}
\]

c.f. Haskell

\[
\text{if } (0<x \&\& x<20) \text{ then ... else ...}
\]
Conjunction and disjunction

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

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 \(x\)

- "If \(x\) is 2 or 3 then..."

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

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

c.f. Haskell

if \((x==2 \, || \, x==3)\) then ... else...
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

  \[
  \text{for}\{3 \mid 4\} \quad \text{The tuple (3,4)}
  \]

  \[
  \text{for}\{4 \mid 3\} \quad \text{The tuple (4,3)}
  \]

- 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
Generalising for

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)

\[
\text{for } (i:=1..3) \text{ do } i*i = (1*1), (2*2), (3*3) = (1, 4, 9)
\]
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
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

\[
\text{for (i:=1..4) do (i<3)} = (1<3, 2<3, 3<3, 4<3) \\
= (1, 2, \text{false?}, \text{false?}) \\
= \text{false?}
\]
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 an array/tuple, as[i], fails on bad indices

- 1..n is (1 | 2 | ... | n)

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

- Returns (4,8,5)

- Indexing an array/tuple, as[i], fails on bad indices

- \text{as}:= (3,7,4)

- \text{as}[0]\quad\text{Denotes one value, 3}

- \text{as}[2]\quad\text{Denotes one value, 4}

- \text{as}[7]\quad\text{Fails: denotes zero values}

- \text{if (x:=as}[i]\text{)}\ then\ x+1\ else\ 0

- Returns 0 if i is out of range
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

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

= 

\[
\begin{align*}
\text{as:=(3,7,4); for}\{i: \text{int}; ((i=0; 3+1) | (i=1; 7+1) | (i=2; 4+1)) \} \\
\end{align*}
\]

Haskell

\[
\text{array (bounds a) [ (i,a!i + 1) | i<-indices a ]}
\]
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])}
Verse 3: functional logic
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”

\[ x := 7; \quad x + 1 > 3; \quad y = x \times 2 \]

means the same as

\[ x : \text{int}; \quad x = 7; \quad x + 1 > 3; \quad y = x \times 2 \]

means the same as

\[ x = 7; \quad x + 1 > 3; \quad y = (x : \text{int}) \times 2 \]

\[ x + 1 > 3; \quad y = (x := 7) \times 2 \]

Think:
- “:” brings the variable into scope.
- Scope extends to the left as well as right.
Towards functional logic programming

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

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

  - Bring y,z into scope
  - Give them values
Towards functional logic programming

- Partial values

\[ x: \text{tuple}(\text{int, int}); \]
\[ x = (2, y: \text{int}); \]
\[ x = (z: \text{int}, 3); \]
\[ x \]

- x’s first component is 2
- y is a fresh unbound variable

- x’s second component is 3
- z is a fresh unbound variable
You can even pass those in-scope-but-unbound variables to a function

```plaintext
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

\[
f(p:int,q:int):int :=
\begin{align*}
\text{if } (x=0) \text{ then } & \{ p=3; \quad q=4 \} \\
\text{else } & \{ p=232; \quad q=913 \};
\end{align*}
\]
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

Sets the value of y

Reads the value of x

- 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
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);
...

Residuation

‘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.
MaxVerse is a big language

To give it precise semantics, we use a small Core Verse language:
- Desugar MaxVerse into CoreVerse
- 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 \( \text{op} ::= \text{gt} \mid \text{add} \)
Values \( \nu ::= x \mid k \mid \text{op} \mid \langle s_1, \ldots, s_n \rangle \mid \lambda x. e \)
Expressions \( e ::= \nu \mid \text{eu} \mid e \mid \exists x. e \mid \text{fail} \mid e_1 \mid e_2 \mid \nu_1 \nu_2 \mid \text{one}\{e\} \mid \text{all}\{e\} \)
\( \text{eu} ::= e \mid \nu = e \)

- “=” is a language construct, not a primop (like gt)
- \(<v_1,\ldots,v_n>\) for tuples to avoid ambiguity with \((x)\)
- \(\exists x\) is what we previously wrote “\(x:ty\)” (except I’m not telling you about types)
- \text{fail} is a language construct, alongside “|”
- Core Verse is untyped (like lambda calculus)
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?

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

\[ \text{foo } x = x \times x + 1 \]

- \( \text{foo } (3+2) \)
- \( \text{let } x = 3+2 \text{ in } x \times x + 1 \)
- \( (3+2) \times (3+2) + 1 \)
- \( 5 \times (3+2) + 1 \)
- \( 5 \times 5 + 1 \)
- \( 25 + 1 \)
- \( 26 \)
- \( \text{let } x = 5 \text{ in } x \times x + 1 \)
- \( (3+2) \times 5 + 1 \)
To answer "what does this program do, or what does it mean?" just apply the rewrite rules

Rewrite rules are things 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
x:tuple(int,int);
x = (2,y:int);
x = (z:int,3);
x

\[ \exists x. \ x = (\exists y. \ (2,y)) \]
\[ \ x = (\exists z. \ (z,3)) \]
\[ x : \text{tuple}(\text{int}, \text{int}); \]
\[ x = (2, y : \text{int}); \]
\[ x = (z : \text{int}, 3); \]

\[ \exists x. \exists y. \exists z. x = (2, y); \]
\[ x = (z, 3); \]
\[ x \]

\[ \exists x. x = (\exists y. \langle 2, y \rangle); \]
\[ x = (\exists z. \langle z, 3 \rangle); \]
\[ x \]
\( x: \text{tuple}(\text{int}, \text{int}); \)  
\( x = (2, y: \text{int}); \)  
\( x = (z: \text{int}, 3); \)  
\( x \)  

\( \exists x. \exists y. \exists z. x = \langle 2, y \rangle; \)  
\( x = \langle z, 3 \rangle; \)  
\( x \)  

Desugar  
\( \exists x. x = (\exists y. \langle 2, y \rangle); \)  
\( x = (\exists z. \langle z, 3 \rangle); \)  
\( x \)  

Float \( \exists \)  
\( \exists x y z. x = \langle 2, y \rangle; \langle 2, y \rangle = \langle z, 3 \rangle; \)  
\( x \)  

Substitute for (one occurrence of) \( x \)
\( x : \text{tuple}(\text{int}, \text{int}); \)
\( x = (2, y: \text{int}); \)
\( x = (z: \text{int}, 3); \)

\( \exists x. \exists y. \exists z. x = (2, y); \)
\( x = (z, 3); \)

Desugar

\( \exists x. x = (\exists y. \langle 2, y \rangle); \)
\( x = (\exists z. \langle z, 3 \rangle); \)
\( x \)

Float \( \exists \)

\( \exists x y z. x = (2, y); \) \( (2, y) = (z, 3); \)
\( x \)

Decompose equality of pairs (unification)

\( \exists x y z. x = (2, y); \) \( z = 2; \) \( y = 3; \)
\( x \)
\( x : \text{tuple}(\text{int}, \text{int}) ; \)
\( x = (2, y : \text{int}) ; \)
\( x = (z : \text{int}, 3) ; \)
\( x \)

\[ \exists x. x = (\exists y. \langle 2, y \rangle) ; \]
\[ x = (\exists z. \langle z, 3 \rangle) ; \]
\[ x \]

**Substitute for another occurrence of** \( x \):

\[ \exists x y z. x = \langle 2, y \rangle ; y = 3 ; z = 2 ; \langle 2, y \rangle \]

**Substitute for** \( y \):

\[ \exists x y z. x = \langle 2, y \rangle ; y = 3 ; z = 2 ; \langle 2, y \rangle \]

**Garbage collect**:

\[ \exists x y z. x = \langle 2, y \rangle ; y = 3 ; z = 2 ; x \]

\[ \langle 2, 3 \rangle \]
\( x: \text{tuple(int,int)}; \)
\( x = (2, y: \text{int}); \)
\( x = (z: \text{int}, 3); \)
\( x \)

\( \exists x. \exists y. \exists z. x = \langle 2, y \rangle; \)
\( x = \langle z, 3 \rangle; \)
\( x \)

\( \exists xyz. x = \langle 2, y \rangle; \langle 2, y \rangle = \langle z, 3 \rangle; \langle z, 3 \rangle \)

\( \exists xyz. x = \langle 2, y \rangle; \)
\( z = 2; \)
\( y = 3; \)
\( \langle z, 3 \rangle \)

\( \langle 2, 3 \rangle \)
Unification rewrite rules

U-SCALAR \quad s = s; \ e \quad \rightarrow \quad e

U-TUP \quad \langle v_1, \cdots, v_n \rangle = \langle v'_1, \cdots, v'_n \rangle; \ e \quad \rightarrow \quad v_1 = v'_1; \cdots; v_n = v'_n; \ e

U-FAIL \quad hnf_1 = hnf_2 \quad \rightarrow \quad \text{fail} \quad \text{if neither U-SCALAR nor U-TUP match}

Scalar Values \quad s ::=} x \mid k \mid op

Heap Values \quad h ::=} \langle v_1, \cdots, v_n \rangle \mid \lambda x. e

Head Values \quad hnf ::=} h \mid k

Values \quad v ::=} s \mid h

Expressions \quad e ::=} v \mid eu; \ e \mid \exists x. e \mid \text{fail} \mid e_1 \mid e_2 \mid v_1 v_2 \mid \text{one}\{e\} \mid \text{all}\{e\}

eu ::=} e \mid v = e
### Primitive operations

**Application:** \( \mathcal{A} \)

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>APP-BETA</td>
<td>((\lambda x. e) v \rightarrow \exists x. x = v; e)</td>
<td>if ( x \notin \text{fvs}(v) )</td>
</tr>
<tr>
<td>APP-TUP0</td>
<td>(\langle \rangle v \rightarrow \text{fail})</td>
<td></td>
</tr>
<tr>
<td>APP-TUP</td>
<td>(\langle v_0 \ldots v_n \rangle v \rightarrow \exists x. x = v; (x = 0; v_0 \mid \ldots \mid x = n; v_n))</td>
<td>if ( x \notin \text{fvs}(v), n \geq 0 )</td>
</tr>
<tr>
<td>APP-ADD</td>
<td>(\text{add}\langle k_1, k_2 \rangle \rightarrow k_1 + k_2)</td>
<td></td>
</tr>
<tr>
<td>APP-GT</td>
<td>(\text{gt}\langle k_1, k_2 \rangle \rightarrow k_1)</td>
<td>if ( k_1 &gt; k_2 )</td>
</tr>
<tr>
<td>APP-GT-FAIL</td>
<td>(\text{gt}\langle k_1, k_2 \rangle \rightarrow \text{fail})</td>
<td>if ( k_1 \leq k_2 )</td>
</tr>
</tbody>
</table>
**Normalization rewrite rules**

**Getting stuff “out of the way”**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Normalization: $N$</th>
</tr>
</thead>
<tbody>
<tr>
<td>NORM-VAL</td>
<td>$\nu; \ e \quad \rightarrow \quad e$</td>
</tr>
<tr>
<td>NORM-SEQ-ASSOC</td>
<td>$(\text{eu}; \ e_1; \ e_2) \quad \rightarrow \quad \text{eu}; \ (e_1; \ e_2)$</td>
</tr>
<tr>
<td>NORM-SEQ-SWAP1</td>
<td>$\text{eu}; \ (x = \nu; \ e) \quad \rightarrow \quad x = \nu; \ (\text{eu}; \ e)$</td>
</tr>
<tr>
<td></td>
<td>(if $\text{eu}$ not of form $x' = \nu'$)</td>
</tr>
<tr>
<td>NORM-SEQ-SWAP2</td>
<td>$\text{eu}; \ (x = \text{s}; \ e) \quad \rightarrow \quad x = \text{s}; \ (\text{eu}; \ e)$</td>
</tr>
<tr>
<td></td>
<td>(if $\text{eu}$ not of form $x' = \text{s}'$)</td>
</tr>
<tr>
<td>NORM-EQ-SWAP</td>
<td>$\text{hnf} = \nu \quad \rightarrow \quad \nu = \text{hnf}$</td>
</tr>
<tr>
<td>NORM-SEQ-DEFR</td>
<td>$(\exists \nu. \ e_1); \ e_2 \quad \rightarrow \quad \exists \nu. \ (e_1; \ e_2)$</td>
</tr>
<tr>
<td></td>
<td>(if $\nu \notin \text{fvs}(e_2)$)</td>
</tr>
<tr>
<td>NORM-SEQ-DEFL</td>
<td>$\text{eu}; \ (\exists \nu. \ e) \quad \rightarrow \quad \exists \nu. \ \text{eu}; \ e$</td>
</tr>
<tr>
<td></td>
<td>(if $\nu \notin \text{fvs}(\text{eu})$)</td>
</tr>
<tr>
<td>NORM-DEFR</td>
<td>$\nu = (\exists \nu. \ e_1); \ e_2 \quad \rightarrow \quad \exists \nu. \ \nu = \ e_1; \ e_2$</td>
</tr>
<tr>
<td></td>
<td>(if $\nu \notin \text{fvs}(\nu, \ e_2)$)</td>
</tr>
<tr>
<td>NORM-SEQQR</td>
<td>$\nu = (\text{eu}; \ e_1); \ e_2 \quad \rightarrow \quad \text{eu}; \ \nu = \ e_1; \ e_2$</td>
</tr>
<tr>
<td></td>
<td>(if $\nu \notin \text{fvs}(\nu, \ e_2)$)</td>
</tr>
</tbody>
</table>
Desugar conditionals like this:

\[\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \text{ means } \exists y. y = \text{one}(e_1; \lambda x. e_2) \mid (\lambda x. e_3); y\langle\rangle\]

Variables bound in \(e_1\) can scope over \(e_2\)

Rewrite rules for one

- **ONE-FAIL**
  \[\text{one}\{\text{fail}\} \rightarrow \text{fail}\]

- **ONE-CHOICE**
  \[\text{one}\{v_1 \mid e_2\} \rightarrow v_1\]

- **ONE-VALUE**
  \[\text{one}\{v\} \rightarrow v\]
Loops

- Desugar for-loops like this:

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

- Rewrite rules for ‘all’

  \[
  \text{ALL-FAIL} \quad \text{for}\{\text{fail}\} \rightarrow \langle \rangle \\
  \text{ALL-CHOICE} \quad \text{for}\{v_1 \mid \cdots \mid v_n\} \rightarrow \langle v_1, \cdots, v_n \rangle
  \]
How to rewrite \((e_1 \mid e_2)\)?

\[
\text{CHOOSE} \quad CX[e_1 \mid e_2] \rightarrow CX[e_1] \mid CX[e_2] \quad \text{if } CX \neq \square
\]

E.g. \((x + (y \mid z) \times 2)\) \(\rightarrow\) \((x + y \times 2) \mid (x + z \times 2)\)

**Choice context**

\[CX ::= \Box \mid v = CX \mid CX; e \mid ce; CX \mid \exists x. CX\]

**Choice-free expr**

\[ce ::= v \mid v = ce \mid ce_1; ce_2 \mid \text{one}\{e\} \mid \text{all}\{e\} \mid \text{op}(v) \mid \exists x. ce\]
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.

More in the paper...
https://simon.peytonjones.org/verse-calculus
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
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