# The Verse Calculus: a Core Calculus for Functional Logic Programming

LENNART AUGUSTSSON, Epic Games, Sweden
JOACHIM BREITNER and KOEN CLAESSEN, Epic Games, Sweden
RANJIT JHALA, Epic Games, USA
SIMON PEYTON JONES, Epic Games, United Kingdom
OLIN SHIVERS, Epic Games, USA
TIM SWEENEY, Epic Games, USA

Functional logic languages have a rich literature, but it is tricky to give them a satisfying semantics. In this paper we describe the Verse calculus, $\mathcal{VC}$, a new core calculus for functional logical programming. Our main contribution is to equip $\mathcal{VC}$ with a small-step rewrite semantics, so that we can reason about a $\mathcal{VC}$ program in the same way as one does with lambda calculus; that is, by applying successive rewrites to it. This semantics elegantly fills the gap between very high level "magical" semantics, and very low-level operational semantics.

*This draft paper describes our current thinking about Verse. It is very much a work in progress, not a finished product. The broad outlines of the design are stable. However, the details of the rewrite rules may well change; we think that the current rules are not confluent, in tiresome ways. (If you are knowledgeable about confluence proofs, please talk to us!)*

*We are eager to enagage in a dialogue with the community. Please do write to us.*

## 1 INTRODUCTION

Functional logic programming languages add expressiveness to functional programming by introducing logical variables, equality constraints among those variables, and choice to allow multiple alternatives to be explored. Here is a tiny example:

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

This expression introduces three logical (or existential) variables $x$, $y$, $z$, constrains them with two equalities ($x = \langle y, 3\rangle$ and ($x = \langle 2, z\rangle$), and finally returns $y$. The only solution to the two equalities is $y = 2$, $z = 3$, and $x = \langle 2, 3\rangle$; so the result of the whole expression is 2.

Functional logic programming has a long history and a rich literature. But it is somewhat tricky for programmers to *reason* about functional logic programs: they must think about logical variables, narrowing, backtracking, Horn clauses, resolution, and the like. This contrasts with functional programming, where one can say "just apply rewrite rules, such as beta reduction, let-inlining, and case-of-known-constructor." We therefore seek *a precise expression of functional logic programming as a term-rewriting system*, to give us both a formal semantics (via small-step reductions), and a powerful set of equivalences that programmers can use to reason about their programs, and that compilers can use to optimize them.

We make the following contributions in this paper. First, we describe a new core calculus for functional logic programming, the Verse calculus or $\mathcal{VC}$ for short (Section 2 and 2.8). Like any functional logic language, $\mathcal{VC}$ supports logical variables, equalities, and choice, but it is distinctive in several ways:

Authors' addresses: Lennart Augustsson, Epic Games, Sweden, lennart.augustsson@epicgames.com; Joachim Breitner, mail@joachim-breitner.de; Koen Claessen, Epic Games, Sweden, koen.claessen@epicgames.com; Ranjit Jhala, Epic Games, USA, ranjit.jhala@epicgames.com; Simon Peyton Jones, Epic Games, United Kingdom, simonpj@epicgames.com; Olin Shivers, Epic Games, USA, olin.shivers@epicgames.com; Tim Sweeney, Epic Games, USA, tim.sweeney@epicgames.com.

2023. 2475-1421/2023/1-ART1 $15.00
https://doi.org/

- $\mathcal{VC}$ natively supports *higher-order programming*, just like the lambda calculus. Indeed, *every lambda calculus program is a $\mathcal{VC}$ program.* In contrast, most of the functional-logic literature is rooted in a first-order world, and addresses higher-order features via an encoding called defunctionalisation [Hanus 2013, 3.3].
- All functional logic languages have some notion of "flexible" *vs.* "rigid" variables, or "suspending" *vs.* "narrowing" operations. $\mathcal{VC}$ offers a new way to address these notions, namely the operators **one** (Section 2.5) and **all** (Section 2.6). This enables an elegant economy of concepts: for example, there is just one equality (other languages have a suspending equality and a narrowing equality), and conditional expressions are driven by failure rather than booleans (Section 2.5).
- $\mathcal{VC}$ uses *spatial choice*, meaning that the choice operator behaves a bit like a data constructor: it appears in normal forms (Section 3.5). This makes $\mathcal{VC}$ *deterministic*, unlike most functional logic languages which are non-deterministic (Section 5.1). In $\mathcal{VC}$ choices are laid out in *space*, in the syntax of the term, rather than in *time*.

As always with a calculus, the idea is that $\mathcal{VC}$ distills the essence of functional logic programming. Each construct does just one thing, and $\mathcal{VC}$ cannot be made smaller without losing key features. We believe that it is possible to use $\mathcal{VC}$ as the compilation target for a variety of functional logic languages such as Curry [Hanus 2016] (although see Appendix B.4). We are ourselves working on Verse, a new general purpose programming language, built directly on $\mathcal{VC}$; indeed, our motivation for developing $\mathcal{VC}$ is practical rather than theoretical. No single aspect of $\mathcal{VC}$ is unique, but we believe that their combination is particularly harmonious and orthogonal. We discuss the rich related work in Section 5, and design alternatives in Appendix B.

Our second contribution is to equip $\mathcal{VC}$ with a *small-step term-rewriting semantics* (Section 3). We said that the lambda calculus is a subset of $\mathcal{VC}$, so it is natural to give its semantics using rewrite rules, just like the lambda calculus. That seems problematical, however, because logical variables and unification involve sharing and non-local communication that seems hard to express in a rewrite system.

Exactly the same difficulty arises with call-by-need. For a long time, the only semantics of call-by-need that was faithful to its sharing semantics (in which thunks are evaluated at most once) was an operational semantics that sequentially threads a global heap through execution [Launchbury 1993]. But then Ariola *et al.*, in a seminal paper, showed how to *reify the heap into the term itself*, and thereby build a rewrite system that is completely faithful to lazy evaluation [Ariola et al. 1995]. Inspired by their idea, we present a new rewrite system for functional logic programs, that reifies logical variables and unification into the term itself, and exploits our notion of spatial choice to replace non-deterministic search with a (deterministic) tree of successful results. For example, the expression above can be rewritten thus[1]:

$$
\begin{array}{ll}
\longrightarrow\{\textsc{deref-h}\} & \exists x\, y\, z.\ x = \langle y, 3\rangle;\ \langle y, 3\rangle = \langle 2, z\rangle;\ y \\
\longrightarrow\{\textsc{u-tup}\} & \exists x\, y\, z.\ x = \langle y, 3\rangle;\ (y = 2;\ 3 = z;\ \langle y, 3\rangle);\ y \\
\longrightarrow\{\textsc{deref-s} \times 2\} & \exists x\, y\, z.\ x = \langle 2, 3\rangle;\ (y = 2;\ 3 = z;\ \langle 2, 3\rangle);\ 2 \\
\longrightarrow\{\textsc{norm-seq-assoc,norm-swap-eq}\} & \exists x\, y\, z.\ x = \langle 2, 3\rangle;\ y = 2;\ z = 3;\ \langle 2, 3\rangle;\ 2 \\
\longrightarrow\{\textsc{norm-val,elim-def}\} & 2
\end{array}
$$

Rules may be applied anywhere they match, again just like the lambda calculus. The question of confluence arises, as we discuss in Section 4.

**Abstract syntax**

| | | | |
|---|---|---|---|
| *Integers* | $k$ | | |
| *Variables* | $x, y, z, f, g$ | | |
| *Primops* | $op$ | ::= | **gt** \| **add** |
| *Scalar Values* | $s$ | ::= | $x \mid k \mid op$ |
| *Heap Values* | $h$ | ::= | $\langle s_1, \cdots, s_n \rangle \mid \lambda x.\, e$ |
| *Head Values* | $hnf$ | ::= | $h \mid k$ |
| *Values* | $v$ | ::= | $s \mid h$ |
| *Expressions* | $e$ | ::= | $v \mid eu;\, e \mid \exists x.\, e \mid \mathbf{fail} \mid e_1 \,\mathbf{|}\, e_2 \mid v_1\, v_2 \mid \mathbf{one}\{e\} \mid \mathbf{all}\{e\}$ |
| | $eu$ | ::= | $e \mid v = e$ |
| *Programs* | $p$ | ::= | $\mathbf{one}\{e\}$ where $\mathrm{fvs}(e) = \emptyset$ |
| *Bindings* | $c$ | ::= | $x = v$ |

**Concrete syntax:** Infix operators "|", ";", "=", and ">" are all right-associative.
"=" binds more tightly than ";".
Function application $(v_1\ v_2)$ is left-associatve, as usual.
"$\lambda$", "$\exists$" scope as far to the right as possible.
*e.g.*, $(\lambda y.\, \exists x.\, x = 1;\, x + y)$ means $(\lambda y.\, (\exists x.\, ((x = 1);\, (x + y))))$.

**Desugaring**

| | | | |
|---|---|---|---|
| $v_1 + v_2$ | means | $\mathbf{add}\langle v_1, v_2 \rangle$ | |
| $v_1 > v_2$ | means | $\mathbf{gt}\langle v_1, v_2 \rangle$ | |
| $\exists x_1\, x_2\ \cdots\ x_n.\, e$ | means | $\exists x_1.\, \exists x_2.\, \cdots \exists x_n.\, e$ | |
| $e_1(e_2)$ | means | $\exists f\ a.\, f = e_1;\, a = e_2;\, f(a)$ | $f, a$ fresh |
| $\langle e_1, \cdots, e_n \rangle$ | means | $\exists x_1\, x_2\ \cdots\ x_n.\, x_1 = e_1;\, \cdots;\, x_n = e_n;\, \langle x_1, \cdots, x_n \rangle$ | $x_i$ fresh |
| $e_1 = e_2$ | means | $\exists x.\, x = e_1;\, x = e_2;\, x$ | $x$ fresh |
| **if** $e_1$ **then** $e_2$ **else** $e_3$ | means | $\exists y.\, y = \mathbf{one}\{(e_1;\, \lambda x.\, e_2) \,\mathbf{|}\, (\lambda x.\, e_3)\};\, y\langle\rangle$ | $x, y$ fresh |
| $x := e_1;\, e_2$ | means | $\exists x.\, x = e_1;\, e_2$ | |

$\mathrm{fvs}(e)$ means the free variable of $e$; in $\mathcal{VC}$, $\lambda$ and $\exists$ are the only binders.

Fig. 1. **The Verse Calculus: Syntax**

## 2 THE VERSE CALCULUS, INFORMALLY

We begin by presenting the Verse calculus, $\mathcal{VC}$, informally. We will give its rewrite rules precisely in Section 3. The syntax of $\mathcal{VC}$ is given in Fig. 1. It has a very conventional sub-language that is just the lambda calculus with some built-in operations and tuples as data constructors:

- *Values.* A value $v$ is either a *scalar value* $s$, which can be freely duplicated, or a *heap value* $h$. A heap value is a lambda or a tuple; and tuples only have value components. In $\mathcal{VC}$ a variable counts as a value, because in a functional logic language an expression may evaluate to an as-yet-unknown logical variable.
- *Built-in functions.* Our tiny calculus offers only integer constants $k$ and two illustrative operators $op$, namely **gt** and **add**

[1]The rule names come from Fig. 3, to be discussed in Section 3; they are given here just for reference.

- *Expressions* $e$ includes values $v$, and applications $v_1\ v_2$; we will introduce the other constructs as we go. For clarity we sometimes write $v_1(v_2)$ rather than $v_1\ v_2$ when $v_2$ is not a tuple.
- *A program*, $p$, is a closed expression from which we extract one result using **one** (see Section 2.5).

The formal syntax for $e$ allows only applications of *values*, $(v_1\ v_2)$, but the desugaring rules in Fig. 1 show how to desugar general applications $(e_1\ e_2)$. This ANF-like normalisation is not fundamental; it simply reduces the number of rewrite rules we need. Modulo this desugaring, every lambda calculus term is a $\mathcal{VC}$ term, and has the same semantics. Just like the lambda calculus, $\mathcal{VC}$ is untyped; adding a type system is an excellent goal, but is the subject of another paper.

Expressions also include two other key collections of constructs: logical variables and unification (Section 2.1), and choice (Section 2.2). The details of choice and unification, and especially their interaction, are rather tricky, so this section will do a lot of arm-waving. But fear not: Section 3 will make all this precise. We only have space to describe one incarnation of $\mathcal{VC}$; Appendix B explores some possible alternative design choices.

## 2.1 Logical variables and unification

The Verse calculus includes first class logical variables and unification: you can bring a fresh logical variable into scope with $\exists$; equate a value with an expression $v = e$; and sequence two expressions with $e_1;\ e_2$ (see Fig. 1). As an example, what might be written **let** $x = e_1$ **in** $\{e_2\}$ in a conventional functional language can be written $\exists x.\ x = e_1;\ e_2$ in $\mathcal{VC}$. A unification $(v = e)$ always equates a *value* to an expression, and can only appear to the left of a ";" (see *ue* in Fig. 1). Again the deugaring rules rewrite a general equality $e_1 = e_2$ into this simpler form.

*A program executes by solving its equations.* For example,

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

is solved by unifying $x$ with $\langle y, 3\rangle$ and with $\langle 2, z\rangle$; that in turn unifies $\langle y, 3\rangle$ with $\langle 2, z\rangle$, which unifies $y$ with 2 and $z$ with 3. Finally 2 is returned as the result. Note carefully that, like any declarative language, *logical variables are not mutable*; a logical variable stands for a single, immutable value. We use "$\exists$" to bring a fresh logical variable into scope, because we really mean "there exists an $x$ such that ...". Logical variables *are* existential variables.

High-level functional languages usually provide some kind of pattern matching; in such a language, we might define *first* by $first\langle a, b\rangle = a$. Such pattern matching is typically desugared to more primitive **case** expressions, but in $\mathcal{VC}$ we do not need **case** expressions: unification does the job. For example we can define *first* like this:

$$first = \lambda pr.\ \exists a\, b.\ pr = \langle a, b\rangle;\ a$$

For convenience, in this presentation we allow ourselves to write a term like $first\langle 2, 5\rangle$, where we define *first* separately. Formally, you can imagine each example $e$ being wrapped with a binding for *first*, thus $\exists first.\ first = \ldots;\ e$; and similarly for all other library functions.

This way of desugaring pattern matching means that the input to *first* is not required to be fully determined when the function is called. For example:

$$\exists x\, y.\ x = \langle y, 5\rangle;\ first(x) = 2;\ y$$

Here $first(x)$ evaluates to $y$, which we then unify with 2. Another way to say this is that, as usual in logic programming, we may constrain the *output* of a function (here $first(x) = 2$), and thereby affect its *input* (here $\langle y, 5\rangle$).

Although ";" is called "sequencing", the order of that sequence is immaterial for equations. For example consider $(\exists x\, y.\ x = 3 + y;\ y = 7;\ x)$. In $\mathcal{VC}$ we can only unify $x$ with a *value*; we will

see why in Section 2.2. So the equation $x = 3 + y$ is stuck. No matter! We simply leave it and try some other equation. In this case, we can make progress with $y = 7$; and that in turn unlocks $x = 3 + y$ because now we know that $y$ is 7, so we can evaluate $3 + 7$ to 10 and unify $x$ with that. The idea of leaving stuck expressions aside, and executing other parts of the program is called *residuation* [Hanus 2013][2], and is at the heart of our mantra "just solve the equations."

## 2.2 Choice

In conventional functional programming, an expression evaluates to a single value. In contrast, a $\mathcal{VC}$ expression evaluates to a choice of zero, one, or many values (or it can get stuck, which is different from producing zero values). The expression **fail** yields no values; a value $v$ yields one value; and the choice $e_1 \mid e_2$ yields all the values yielded by $e_1$ and all the values yielded by $e_2$. Duplicates are not eliminated and, as we shall see in Section 2.7, order is maintained; in short, an expression yields a *sequence* of values, not a bag, and certainly not a set.

The equations we saw in Section 2.1 can fail, if the arguments are not equal, yielding no results. Thus $3 = 3$ succeeds, returns a single result, namely 3, while $3 = 4$ fails, returning no results. In general, we use "fail" and "returns no results" synonymously.

What if the choice was not at the top level of an expression? For example, what does $\langle 3, (7 \mid 5)\rangle$ mean? In $\mathcal{VC}$ it does *not* mean a pair with some kind of multi-value in its second component. Indeed, as you can see from Fig. 1, this expression is syntactically ill-formed. We must instead give a name to that choice, and then we can put it in the pair, thus: $\exists x.\, x = (7 \mid 5);\ \langle 3, x\rangle$. This is syntactically legal, but what does it mean? In $\mathcal{VC}$ a variable is never bound to a multi-value. Instead, $x$ is successively bound to 7, and then to 5, like this:

$$\exists x.\, x = (7 \mid 5);\ \langle 3, x\rangle \quad \longrightarrow \quad (\exists x.\, x = 7;\ \langle 3, x\rangle) \mid (\exists x.\, x = 5;\ \langle 3, x\rangle))$$

We duplicate the context surrounding the choice, and "float the choice outwards."

## 2.3 Mixing choice and unification

We saw in Section 2.1 that *equations* are insensitive to sequencing—but *choice* is not. Consider $\exists x\, y.\, x = (3 \mid 4);\ y = (20 \mid 30);\ \langle x, y\rangle$. The choices are made *left-to-right*, so that the result is $(\langle 3, 20\rangle \mid \langle 3, 30\rangle \mid \langle 4, 20\rangle \mid \langle 4, 30\rangle)$.

So much for choice under unification. What if we have unification under choice? For example:

$$\exists x.\, (x = 3;\ x + 1) \mid (x = 4;\ x * 2)$$

Intuitively, either unify $x$ with 3 and return $x + 1$, or unify $x$ with 4 and return $x * 2$. But so far we have said only "a program executes by solving its equations" (Section 2.1). Well, we can see two equations here, $(x = 3)$ and $(x = 4)$, which are mutually contradictory, so clearly we need to refine our notion of "solving." The answer is pretty clear: in a branch of a choice, solve the equations in that branch to get the value for some logical variables, *and propagate those values to occurrences in that branch (only).* Occurrences of that variable outside the choice are unaffected. We call this *local propagation.* This local-propagation rule would allow us to reason thus:

$$\exists x.\, (x = 3;\ x + 1) \mid (x = 4;\ x * 2) \longrightarrow \exists x.\, (x = 3;\ 4) \mid (x = 4;\ 8)$$

Are we stuck now? No, we can float the choice out as before[3],

$$\exists x.\, (x = 3;\ 4) \mid (x = 4;\ 8) \longrightarrow (\exists x.\, x = 3;\ 4) \mid (\exists x.\, x = 4;\ 8)$$

[2]Hanus did not invent the terms "residuation" and"narrowing", but his survey is an excellent introduction and bibliography.
[3]Indeed we could have done so first, had we wished.

and now it is apparent that the sole occurrence of $x$ in each $\exists$ is the equation $(x = 3)$, or $(x = 4)$ respectively; so we can drop the $\exists$ and the equation, yielding $(4 \mid 8)$.

### 2.4 Pattern matching and narrowing

We remarked in Section 2.1 that we can desugar the pattern matching of a high-level language into unification. But what about multi-equation pattern matching, such as this definition in Haskell:

$$append\ [\,]\ ys = ys$$
$$append\ (x : xs)\ ys = x : append\ xs\ ys$$

If pattern matching on the first equation fails, we want to fall through to the second. Fortunately, choice allows us to express this idea directly[4]:

$$append = \lambda\langle xs, ys\rangle.\ (xs = \langle\rangle;\ ys) \quad | \quad (\exists x\, xrest.\ xs = \langle x, xrest\rangle;\ \langle x, append\langle xrest, ys\rangle\rangle)$$

If $xs$ is $\langle\rangle$, the left-hand choice succeeds, returning $ys$; and the right-hand choice fails (by attempting to unify $\langle\rangle$ with $\langle x, xrest\rangle$). If $xs$ is of the form $\langle x, xrest\rangle$, the right-hand choice succeeds, and we make a recursive call to *append*. Finally if $xs$ is built with head-normal forms other than the empty tuple and pairs, both choices fail, and *append* returns no results at all.

This approach to pattern matching is akin to *narrowing* [Hanus 2013]. Suppose $single = \langle 1, \langle\rangle\rangle$, a singleton list whose only element is 1. Consider the call $\exists zs.\ append\langle zs, single\rangle = single;\ zs$. The call to *append* expands into a choice

$$(zs = \langle\rangle;\ single) \quad | \quad (\exists x\, xrest.\ zs = \langle x, xrest\rangle;\ \langle x, append\langle xrest, single\rangle\rangle)$$

which amounts to exploring the possibility that $zs$ is headed by $\langle\rangle$ or a pair—the essence of narrowing. It should not take long to reassure yourself that the program evaluates to $\langle\rangle$, effectively running *append* backwards in the classic logic-programming manner.

This example also illustrates that $\mathcal{VC}$ allows an equality (for *append*) that is recursive. As in any functional language with recursive bindings, you can go into an infinite loop if you keep fruitlessly inlining the function in its own right-hand side. It is the business of an *evaluation strategy* to do only rewrites that make progress towards a solution (Section 3.7).

### 2.5 Conditionals and one

Every source language will provide a conditional, such as **if** $(x = 0)$ **then** $e_2$ **else** $e_3$. But what is the equality operator in $(x = 0)$? One possibility, adopted by Curry, is this: there is one "=" for equations (as in Section 2.1), and another, say "==", for testing equality (returning a boolean with constructors *True* and *False*). $\mathcal{VC}$ takes a different, more minimalist position. In $\mathcal{VC}$ *there is just one equality operator*, written "=" just as in Section 2.1. The expression **if** $(x = 0)$ **then** $e_2$ **else** $e_3$ tries to unify $x$ with 0. If that succeeds (returns one or more values) the **if** returns $e_2$; otherwise it returns $e_3$. There are no data constructors *True* and *False*; instead failure plays the role of falsity.

But something is terribly wrong here. Consider $\exists x\, y.\ y = (\textbf{if}\ (x = 0)\ \textbf{then}\ 3\ \textbf{else}\ 4);\ x = 7$. Presumably this is meant to set $x$ to 7, test if it is equal to 0 (it is not), and unify $y$ with 4. But what is to stop us instead unifying $x$ with 0 (via $(x = 0)$), unifying $y$ with 3, and then failing when we try to unify $x$ with 7? Not only is that not what we intended, but it also looks very non-deterministic: the result is affected by the order in which we did unifications!

To address this, we give **if** a special property: in the expression **if** $e_1$ **then** $e_2$ **else** $e_3$, unifications inside $e_1$ (the condition of the **if**) can only unify variables bound inside $e_1$; variables bound outside $e_1$ are called "rigid." So in our example, the $x$ in $(x = 0)$ is rigid and cannot be unified. Instead, the **if**

[4] We use the empty tuple $\langle\rangle$ to represent the empty list and pairs to represent cons cells; and we allow ourselves to write $\lambda\langle x, y\rangle.\ body$ rather than $\lambda p.\ \exists x\, y.\ p = \langle x, y\rangle;\ body$

is stuck, and we move on to unify $x = 7$. That unblocks the **if** and all is well. This special property is precisely the local propagation rule that we sketched for choice (Section 2.3).

In fact, $\mathcal{VC}$ distills the three-part **if** into something simpler, the unary construct **one**$\{e\}$. Its specification is this: if $e$ fails, **one**$\{e\}$ fails; otherwise **one**$\{e\}$ returns the first of the values yielded by $e$. Now, **if** $e_1$ **then** $e_2$ **else** $e_3$ can (nearly) be re-expressed like this:

$$\mathbf{one}\{(e_1;\ e_2) \mid e_3\}$$

If $e_1$ fails, the first branch of the choice fails, so we get $e_3$; if $e_1$ succeeds, we get $e_2$, and the outer **one** will select it from the choice. But this isn't right: what if $e_2$ or $e_3$ themselves fail or return multiple results? Here is a better translation, given in Fig. 1, which wraps the then and else branches in a thunk:

$$(\mathbf{one}\{(e_1;\ (\lambda x.\ e_2)) \mid (\lambda x.\ e_3)\})\langle\rangle$$

The argument of **one** evaluates to either $((\lambda x.\ e_2) \mid \cdots)$ or $(\lambda x.\ e_3)$ depending on whether $e_1$ succeeds or fails, respectively, and **one** then picks that lambda and applies it to $\langle\rangle$. As a bonus, provided we do no evaluation under a lambda, then $e_2$ and $e_3$ will remain un-evaluated until the choice is made, just as we expect.

We use the same local-propagation rule for **one** that we do for choice (Section 2.3); together with the desugaring for **if** into **one**, we get the "special property" of **if** described above.

### 2.6 Tuples and all

The main data structure in $\mathcal{VC}$ is the tuple. A tuple is a finite sequence of values, $\langle v_1, \cdots, v_n\rangle$. It can be used like a function: indexing is simply function application with the argument being integers from 0 and up. Indexing out of range is **fail**. For example, $\exists t.\ t = \langle 10, 27, 32\rangle;\ t(1)$ reduces to 27 and $t(3)$ reduces to **fail**. The reduction rule for indexing in tuples admits multi-valued index expressions. For instance, $\exists t.\ t = \langle 10, 27, 32\rangle;\ t(1 \mid 0 \mid 1)$ reduces to $(27 \mid 10 \mid 27)$.

Tuples can be constructed by collecting all the results from a multi-valued expression, using the **all** construct: if $e$ reduces to $(v_1 \mid \cdots \mid v_n)$ then **all**$\{e\}$ reduces to the tuple $\langle v_1, \cdots, v_n\rangle$; as a consequence, if $e$ fails, **all** produces the empty tuple. Note that $\mid$ is associative, which means that we can think of a sequence or tree of binary choices as really being a single $n$-way choice.

You might think that tuple indexing would be stuck until we know the index, but $\mathcal{VC}$ uses narrowing to make progress. The expression $\exists t.\ t = \langle 10, 27, 32\rangle;\ \exists i.\ t(i)$ looks stuck because we have no value for $i$, but in fact it rewrites to

$$\exists i.\ (i = 0;\ 10) \mid (i = 1;\ 27) \mid (i = 2;\ 32)$$

which (as we will see in Section 3) simplifies to just $(10 \mid 27 \mid 32)$. So **all** allows a choice to be reified into a tuple; and $(\exists i.\ t(i))$ allows a tuple to be turned back into a choice.

Do we even need **one** as a primitive construct, given that we have **all**? Can we not use $(\mathbf{all}\{e\})(0)$ instead of **one**$\{e\}$? Indeed they behave the same if $e$ fully reduces to finitely many choices of values. But **all** really requires the evaluation of *all* choices before proceeding, while **one** only needs to evaluate the *first* choice. So, supposing that *loop* is a non-terminating function, **one**$\{1 \mid loop\langle\rangle\}$ reduces to 1, while $(\mathbf{all}\{1 \mid loop\langle\rangle\})(0)$ loops.

### 2.7 for loops

The expression **for**$(e_1)$ **do** $e_2$ will evaluate $e_2$ for each of the choices in $e_1$, rather like a list comprehension in languages like Haskell or Python. The scoping is peculiar[5] in that variables bound in $e_1$ also scope over $e_2$. So, *e.g.*, **for**$(\exists x.\ x = 2 \mid 3 \mid 5)$ **do** $(x + 1)$ will reduce to the tuple $\langle 3, 4, 6\rangle$.

[5]But similar to C++.

**Notation**

$f(x) := e$ means $f := \lambda x.\, e$

$f\langle x, y\rangle := e$ means $f := \lambda p.\, \exists x\, y.\, p = \langle x, y\rangle;\ e \quad p$ fresh

$$
\begin{array}{rcl}
head(xs) & := & xs(0) \\
tail(xs) & := & \mathbf{all}\{\exists i.\, i > 0;\ xs(i)\} \\
cons\langle x, xs\rangle & := & \mathbf{all}\{x \mid \exists i.\, xs(i)\} \\
append\langle xs, ys\rangle & := & \mathbf{all}\{(\exists i.\, xs(i)) \mid (\exists i.\, ys(i))\} \\
flatMap\langle f, xs\rangle & := & \mathbf{all}\{\exists i.\, f(xs(i))\} \\
map\langle f, xs\rangle & := & \mathbf{if}\ x := head(xs)\ \mathbf{then}\ cons\langle f(x), map\langle f, tail(xs)\rangle\rangle\ \mathbf{else}\ \langle\rangle \\
filter\langle p, xs\rangle & := & \mathbf{all}\{\exists i.\, x := xs(i);\ \mathbf{one}\{p(x)\};\ x\}) \\
find\langle p, xs\rangle & := & \mathbf{one}\{\exists i.\, x := xs(i);\ \mathbf{one}\{p(x)\};\ x\}) \\
every\langle p, xs\rangle & := & map\langle p, xs\rangle \\
fromTo\langle lo, hi\rangle & := & \mathbf{all}\{\ \mathbf{if}\ lo > hi\ \mathbf{then\ fail\ else}\ (lo \mid fromTo\langle lo + 1, hi\rangle)\}
\end{array}
$$

Fig. 2. Common list functions

Like list comprehension, **for** supports filtering; in $\mathcal{VC}$ this falls out naturally by just using a possibly failing expression in $e_1$. So, $\mathbf{for}(x := 2 \mid 3 \mid 5;\ x > 2)\ \mathbf{do}\ (x + 1)$ reduces to $\langle 4, 6\rangle$. Nested iteration in a **for** works as expected, and requires nothing special. So, $\mathbf{for}(x := 10 \mid 20;\ y := 1 \mid 2 \mid 3)\ \mathbf{do}\ (x + y)$ reduces to $\langle 11, 12, 13, 21, 22, 23\rangle$

Just as **if** is defined in terms of the primitive **one** (Section 2.5), we can define **for** in terms of the primitive **all**. Again, we have to be careful when $e_2$ itself fails or produces multiple results; simply writing $\mathbf{all}\{e1;\ e2\}$ would give the wrong semantics. So we put $e_2$ under a lambda, and apply each element of the tuple to $\langle\rangle$ afterwards, using the *map* function defined in Fig. 2. The full desugaring is

$$\mathbf{for}(e_1)\ \mathbf{do}\ e_2 \ \equiv\ \exists v.\, v = \mathbf{all}\{e_1;\ \lambda x.\, e_2\};\ map\langle \lambda z.\, z\langle\rangle, v\rangle$$

for a fresh variable $v$. Note how this achieves that peculiar scoping rule: variables defined in $e_1$ are in scope in $e_2$. Any effects (like being multivalued) in $e_2$ will not affect the choices defined by $e_1$ since they are in a thunk. So, *e.g.*, $\mathbf{for}(x := 10 \mid 20)\ \mathbf{do}\ \{x \mid x + 1\}$ will reduce to $\langle 10, 20\rangle \mid \langle 10, 21\rangle \mid \langle 11, 20\rangle \mid \langle 11, 21\rangle$. At this point it is crucial to use *map*, not *flatMap*.

Given that tuple indexing expands into choices, we can iterate over tuple indices and elements using **for**. For example $\mathbf{for}(\exists i\ x.\, x = t(i))\ \mathbf{do}\ (x + i)$ produces a tuple with the elements of $t$, increased by their index in $t$.

## 2.8 Programming in Verse

$\mathcal{VC}$ is a fairly small language, but it is quite expressive. For example, we can define the typical list functions one would expect from functional programming by using the duality between tuples and choices, as seen in Fig. 2. A tuple can be turned into choices by indexing with a logical variable *i*. Conversely, choices can be turned into a tuple using **all**. The choice operator, |, serves as both *cons* and *append* for choices.

Pattern matching for function definitions is simply done by unification of ordinary expressions. This means that we can use ordinary abstraction mechanisms for patterns. For example, here is a function that should be called like $fcn\langle 88, 1, 99, 2\rangle$.

$$fcn(t) := \exists x\, y.\, t = \langle x, 1, y, 2\rangle;\ x + y$$

If we want to give a name to the pattern, it is simple to do so:

$$pat\langle v, w\rangle \coloneqq \langle v, 1, w, 2\rangle; \qquad fcn(t) \coloneqq \exists x\, y.\ t = pat\langle x, y\rangle;\ x + y$$

## 3 REWRITE RULES

How can we give a precise semantics to a non-strict functional programming language? Here are some possibilities:

- *A denotational semantics* is the classical approach, but it is tricky to give a (perspicuous) denotational semantics to a functional logic language, because of the logical variables. We have such a denotational semantics under development, which we offer for completeness in Appendix C, but that is the subject of another paper.
- *A big-step operational semantics* typically involves explaining how a (heap, expression) starting point evaluates to a (heap, value) pair; Launchbury's natural semantics for lazy evaluation [Launchbury 1993] is the classic paper. The heap, threaded through the semantics, accounts for updating thunks as they are evaluated.
- *A small-step operational semantics.* Despite its "operational semantics" title, the big-step approach does not convey accurate operational intuition, because it goes all the way to a value in one step. So-called "small-step" operational semantics are therefore widely used; they typically describe how a (heap, expression, stack) configuration evolves, one small step at a time (*e.g.*, [Peyton Jones 1992]). The difficulty is that the description is now so low level that it is again hard to explain to programmers.
- *A rewrite semantics* steers between these two extremes. For example, Ariola *et al.*'s "A call by need lambda calculus" [Ariola et al. 1995] shows how to give the semantics of a call-by-need language as a set of rewrite rules. The great advantage of this approach is that it is readily explicable to programmers. Indeed teachers almost always explain the execution of Haskell or ML programs as a succession of rewrites of the program (*e.g.*, inline this call, simplify this **case** expression, *etc.*).

Up to this point there has been no satisfying rewrite semantics for functional logic languages (see Section 5 for previous work). Our main technical contribution is to fill this gap with a rewrite semantics for $\mathcal{VC}$, one that has the following properties:

- The semantics is expressed as a set of rewrite rules (Fig. 3 and 4).
- Any rule can be applied, in either direction, anywhere in the program term (including under lambdas) to obtain an equivalent program.
- The rules are oriented, with the intent that using them left-to-right makes progress.
- Despite this orientation, the rules do not say which rule should be applied where; that is the task of a separate *evaluation strategy* (Section 3.7).
- The rules can be applied by programmers, to reason about what their program does; and by compilers, to transform (and hopefully optimise) the program.
- There is no "magical rewriting" (Section 5.3): all the variables on the right-hand side of a rule are bound on the left.

### 3.1 Functions and function application

Looking at Fig. 3, rule APP-ADD should be familiar: it simply rewrites an application of **add** to integer constants. For example $\mathbf{add}\langle 3, 4\rangle \longrightarrow 7$. Rules APP-GT and APP-GT-FAIL are more interesting: $\mathbf{gt}\langle k_1, k_2\rangle$ fails if $k_1 \leqslant k_2$ (rather than returning *False* as is more conventional), and returns $k_1$ otherwise (rather than returning *True*). An amusing consequence is that $(10 > x > 0)$ succeeds iff $x$ is between 10 and 0 (comparison is right-associative).

$$
\begin{array}{lll}
\textit{Expression context} & E ::= \Box \mid \langle s_1, \cdots, \Box, \cdots, s_n \rangle \mid \lambda x.\, E \mid \exists x.\, E \mid E = e \mid e = E \\
& \quad \mid E;\, e \mid e;\, E \mid E\, v \mid v\, E \mid E \mid e \mid e \mid E \mid \mathbf{all}\{E\} \mid \mathbf{one}\{E\} \\
\textit{Application context} & A ::= \Box\, v \mid op\,\Box \mid \Box = hnf \mid v = A \mid \exists x.\, A \mid A;\, e \mid e;\, A \\
& \quad \mid A \mid e \mid e \mid A \mid \mathbf{all}\{A\} \mid \mathbf{one}\{A\} \\
\textit{Scope context} & SX ::= \Box \mid e \mid e \mid \Box \mid \mathbf{one}\{\Box\} \mid \mathbf{all}\{\Box\} \\
\textit{Choice context} & CX ::= \Box \mid v = CX \mid CX;\, e \mid ce;\, CX \mid \exists x.\, CX \\
\textit{Choice-free expr} & ce ::= v \mid v = ce \mid ce_1;\, ce_2 \mid \mathbf{one}\{e\} \mid \mathbf{all}\{e\} \mid op(v) \mid \exists x.\, ce \\
\textit{Bound variables} & \mathrm{bvs}(E) = \text{The variables that are bound by } E \text{ at the hole} \\
& \qquad \text{e.g. } \mathrm{bvs}((\exists x.\, x = 3) \mid (\exists y.\, \Box = 4))) = \{y\}
\end{array}
$$

*Unification:* $\mathcal{U}$

| | | | | |
|---|---|---|---|---|
| DEREF-S | $x = s;\, E[x]$ | $\longrightarrow$ | $x = s;\, E[s]$ | $x \not\equiv s, x \notin \mathrm{bvs}(E), s \notin \mathrm{bvs}(E)$ |
| DEREF-H | $x = h;\, A[x]$ | $\longrightarrow$ | $x = h;\, A[h]$ | $x \notin \mathrm{bvs}(A), \mathrm{fvs}(h) \notin \mathrm{bvs}(A)$ |
| 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 |

*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 $k_1 \leqslant k_2$ |

*Speculation:* $\mathcal{S}$

| | | | | |
|---|---|---|---|---|
| CHOOSE | $SX[CX[e_1 \mid e_2]]$ | $\longrightarrow$ | $SX[CX[e_1] \mid CX[e_2]]$ | if $CX \neq \Box$ |
| CHOOSE-ASSOC | $SX[(e_1 \mid e_2) \mid e_3]$ | $\longrightarrow$ | $SX[e_1 \mid (e_2 \mid e_3)]$ | |
| CHOOSE-R | $SX[\mathbf{fail} \mid e]$ | $\longrightarrow$ | $SX[e]$ | |
| CHOOSE-L | $SX[e \mid \mathbf{fail}]$ | $\longrightarrow$ | $SX[e]$ | |
| ONE-FAIL | $\mathbf{one}\{\mathbf{fail}\}$ | $\longrightarrow$ | **fail** | |
| ONE-CHOICE | $\mathbf{one}\{e_1 \mid e_2\}$ | $\longrightarrow$ | $e_1$ | if $\emptyset \vdash e_1 \rightsquigarrow (\overline{x} \mid \overline{c} \mid v)$ |
| ONE-VALUE | $\mathbf{one}\{e\}$ | $\longrightarrow$ | $e$ | if $\emptyset \vdash e \rightsquigarrow (\overline{x} \mid \overline{c} \mid v)$ |
| ALL-FAIL | $\mathbf{all}\{\mathbf{fail}\}$ | $\longrightarrow$ | $\langle \rangle$ | |
| ALL-CHOICE | $\mathbf{all}\{e_1 \mid \cdots \mid e_n\}$ | $\longrightarrow$ | $\exists \overline{x}.\, \overline{c};\, \langle \overline{v} \rangle$ | if $\vdash_* \overline{e} \rightsquigarrow (\overline{x} \mid \overline{c} \mid \overline{v})$, $n \geqslant 1$ |

Fig. 3. **The Verse Calculus: Rewrite Rules**

Beta-reduction is performed quite conventionally by APP-BETA; the only unusual feature is that on the RHS of the rule we use a $\exists$ to bind $x$, together with $(x = v)$ to equate $x$ to the argument. The rule may appear to use call-by-value, because the argument is a value $v$, but remember that values include variables, which may be bound to an as-yet-unevaluated expression. For example:

$$\exists y.\, y = 3 + 4;\, (\lambda x.\, x + 1)(y) \longrightarrow \exists y.\, y = 3 + 4;\, \exists x.\, x = y;\, x + 1$$

Finally, the side condition $x \notin \mathrm{fvs}(v)$ in APP-BETA ensures that the $\exists\, x$ does not capture any variables free in $v$. If $x$ appears free in $v$, just use $\alpha$-conversion to rename $x$ to $x' \notin \mathrm{fvs}(v)$.

In $\mathcal{VC}$, tuples behave like (finite) functions, in which application is indexing. Rule APP-TUP describes how tuple application works. Notice that APP-TUP does not require the argument to be evaluated to an integer $k$; instead the rule works by narrowing. So the expression $\exists x.\, \langle 2,3,2,7,9\rangle(x) = 2$; $x$ does not suspend awaiting a value for $x$; instead it explores all the alternatives, returning $(0 \mid 2)$. This is a free design decision: a suspending semantics would be equally easy to express.

### 3.2 Unification

Next we study unification, again in Fig. 3. Rules U-SCALAR and U-TUP are the standard rules for unification, going back nearly 60 years [Robinson 1965]. Note that when unification succeeds it yields the common value; hence $s = s$ rewrites to $s$[6]. Rule U-FAIL makes unification fail on two different head-normal forms (see Fig. 1 for the syntax of *hnf*). Note in particular that unification fails if you attempt to unify a lambda with any other value (including itself) – see Section 4.2.

The key innovation in $\mathcal{VC}$ is the way bindings (that is, just ordinary equalities) of logical variables are propagated. The key rules are:

$$\begin{array}{lllll} \textsc{deref-s} & x = s;\, E[\,x\,] & \longrightarrow & x = s;\, E[\,s\,] & x \not\equiv s, x \notin \mathrm{bvs}(E), s \notin \mathrm{bvs}(E) \\ \textsc{deref-h} & x = h;\, A[\,x\,] & \longrightarrow & x = h;\, A[\,h\,] & x \notin \mathrm{bvs}(A), \mathrm{fvs}(h) \notin \mathrm{bvs}(A) \end{array}$$

These rules make use of so-called *contexts*, $E$ and $A$, whose syntax is given in Fig. 3 [Felleisen and Friedman 1986; Felleisen et al. 1987]. In general, a context is an expression containing a single hole, written $\Box$. The notation $E[\,s\,]$ is the expression obtained by filling the hole in $E$ with $s$.

So DEREF-S says that if we have an equality $(x = s)$ to the left of a term $E[\,x\,]$ that mentions $x$, we can replace that (single) occurrence of $x$ with $s$, yielding $E[\,s\,]$ instead. There are several things to notice:

- DEREF-S fires only when the right-hand side of the unification is a *scalar value* $s$; that is, a variable or integer literal. That is because $E$ allows the occurrence of $x$ to be in places that only syntactically allow scalars. [LA: Is it really true now that $E$ can have places where only scalars are allowed?] Rule DEFREF-H allows substitution of heap values, but again only in places that syntactically allow such expressions; also see Section 4.2.
- Both rules fire only when the RHS is a value, so that the substitution does not risk duplicating either work or choices. This restriction is precisely the same as the LET-V rule of [Ariola et al. 1995], and (by not duplicating choices) it neatly implements so-called *call-time choice* [Hanus 2013]. We do not need a heap, or thunks, or updates; the equalities of the program elegantly suffice to express the necessary sharing.
- Both DEREF rules replace a single occurrence of $x$, *leaving the original* $(x = v)$ *undisturbed*. For example, we can rewrite $(x = 3;\ y = x + 1;\ z = x + 3)$ to $(x = 3;\ y = 3 + 1;\ z = x + 3)$, using $E = (y = \Box + 1;\ z = x + 3)$. We must not drop the $(x = v)$ because there may be other occurrences of $x$, such as the $x + 3$ in this example. When there are no remaining occurrences of $x$ we may garbage collect the binding: see Section 3.4.
- Both rules substitute only to the *right* of a binding. How can we rewrite $(y = x + 1;\ x = 3)$, where the occurrence of $x$ is to the left of its binding? Answer, by moving the $x = 3$ binding to the left, a process we call normalization, discussed in Section 3.4.
- The $x \not\equiv s$ in DEREF-S prevents a binding $x = x$ from substituting infinitely often, doing nothing each time. The guard $x \notin \mathrm{bvs}(E)$ ensures that $x$ is actually free in $E[\,x\,]$, while $s \notin \mathrm{bvs}(E)$ ensures that $s$ is not captured by $E$ in $E[\,s\,]$.

[6] An alternative choice would for unification to yield $\langle\rangle$ on success. It does not make much difference either way.

- DEREF-S substitutes a scalar anywhere, but DEREF-H is much more parsimonous: *it never substitutes a heap value h under a lambda or inside a tuple*, as can be seen by examining the syntax of application contexts $A$. This is a tricky point: see Section 4.2.
- Rather unconventionally, there is no "occurs check", leading to **fail**. It is very important to allow allow bindings like $(f = \lambda x.\ \cdots\ (f(x-1))\ \cdots)$ to substitute, because that is how we define a recursive function! We even allow $(x = \langle 1, x\rangle)$. Of course, recursive bindings can lead to infinite rewriting sequences; it is up to the evaluation strategy to avoid this (Section 3.7).

### 3.3 Local substitution

Consider this (extremely) tricky term: $\exists x.\ x = \textbf{if}\ (x = 0;\ x > 1)\ \textbf{then}\ 33\ \textbf{else}\ 55$. What should this do? At first you might think it was stuck; how can we simplify the **if** when its condition mentions $x$ which is not yet defined? But in fact, rule DEREF-S allows us to substitute *locally* in any X-context surrounding the equality $(x = 0)$ thus:

$$\begin{array}{ll} & \exists x.\ x = \textbf{if}\ (x = 0;\ x > 1)\ \textbf{then}\ 33\ \textbf{else}\ 55 \\ \longrightarrow\{\text{DEREF-S}\} & \exists x.\ x = \textbf{if}\ (x = 0;\ 0 > 1)\ \textbf{then}\ 33\ \textbf{else}\ 55 \\ \longrightarrow\{\text{U-FAIL,FAIL-SEQL}\} & \exists x.\ x = \textbf{if fail then}\ 33\ \textbf{else}\ 55 \\ \longrightarrow\{\text{simplify }\textbf{if}\} & \exists x.\ x = 55 \\ \longrightarrow\{\text{ELIM-DEF}\} & 55 \end{array}$$

Minor variants of the same example get stuck instead of reducing. For example, if you replace the $(x = 0)$ with $(x = 100)$ then rewriting gets stuck, as the reader may verify; and yet there is a solution to the equations, namely $x = 55$. And if you replace $(x = 0)$ with $(x = 55)$ then rewriting again gets stuck, and reasonably so, since in this case there are *no* valid solutions to the equations. Perhaps this is not surprising: we cannot reasonably expect the program to solve arbitrary equations. For example, $\exists x.\ x * x = x$ has two solutions but discovering that involves solving a quadratic equation.

### 3.4 Normalization rules

The syntax of Fig. 1 allows $(\exists x.\ e)$, $(v = e)$, and $(e_1;\ e_2)$ to occur anywhere in an expression. But to make other rules more applicable, it may be necessary to "float" these expression upward. For example, we can't use DEREF-H to substitute for $x$ in $(x = (e;\ 3);\ x + 2)$, because the RHS of the $x$-equality is not a value. But if we were to float the semicolon outwards to give $(e;\ x = 3;\ x + 2)$, we could then substitute for $x$.

Thus motivated, Fig. 4 gives a collection of rules that systematically move existentials and unifications upward and to the left. The net effect is to normalise the term to a form with existentials at the top, then scalar equalities, and then heap equalities, thus

$$\exists x_1, \cdots, x_n.\ x_1 = s_1;\ \cdots;\ x_i = h_i;\ x_n = h_n;\ e$$

You can think of this form as "an expression $e$ wrapped in some heap bindings $x_i = v_i$". The heap bindings express, as a term, the possibly-recursive values of the $x_i$, but the right-hand sides $v_i$ are all values, so there is no computation left in the heap. This decomposition is so important that we define a judgement $\Gamma \vdash e_1 \rightsquigarrow (\overline{x} \mid \overline{c} \mid e_2)$ in Fig. 5, which decomposes an expression $e_1$ into its heap, specified by $\overline{x}$ and $\overline{c}$, and the expression wrapped in that heap, $e_2$. (The non-terminal $c$ is just short for $x = v$; Fig. 1). Notice that, if invoked with $\Gamma = \emptyset$, this judgement checks that that the equalities $\overline{c}$ fix only variables bound by one of the existentials $\overline{x}$; and moreover that there is only one such equality for any particular $x_i$.

One very useful application of this decomposition is ELIM-DEF in Fig. 4, which allows an entire heap of possibly-recursive (but computation-free) bindings to be discarded if none of its variables

*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 | $v = (eu;\ e_1);\ e_2$ | $\longrightarrow$ | $eu;\ v = e_1;\ e_2$ | |

*Fail Propagation: $\mathcal{F}$*

| | | | | |
|---|---|---|---|---|
| FAIL-SEQL | $\mathbf{fail};\ e$ | $\longrightarrow$ | $\mathbf{fail}$ | |
| FAIL-SEQR | $e;\ \mathbf{fail}$ | $\longrightarrow$ | $\mathbf{fail}$ | |
| FAIL-EQ | $v = \mathbf{fail}$ | $\longrightarrow$ | $\mathbf{fail}$ | |

*Garbage Collection: $\mathcal{G}$*

| | | | | |
|---|---|---|---|---|
| ELIM-DEF | $e_1$ | $\longrightarrow$ | $e_2$ | if $\emptyset \vdash e_1 \rightsquigarrow (\overline{x} \mid \overline{c} \mid e_2)$ and $\overline{x} \notin \mathrm{fvs}(e_2)$ |

*Structural rules*

| | | | | |
|---|---|---|---|---|
| SWAP-D | $\exists x.\ \exists y.\ e$ | $\equiv$ | $\exists y.\ \exists x.\ e$ | |
| SWAP-C | $x_1 = v_1;\ x_2 = v_2;\ e$ | $\equiv$ | $x_2 = v_2;\ x_1 = v_1;\ e$ | |

Fig. 4. **The Verse Calculus: Normalization Rules**

$$\boxed{\Gamma \vdash e_1 \rightsquigarrow (\overline{x} \mid \overline{c} \mid e_2)}$$

$$\frac{}{\Gamma \vdash e \rightsquigarrow (\emptyset \mid \emptyset \mid e)}\ \text{WF-Exp}$$

$$\frac{\Gamma, x \vdash e_1 \rightsquigarrow (\overline{x} \mid \overline{c} \mid e_2) \qquad x \notin \overline{x}}{\Gamma \vdash \exists x.\ e_1 \rightsquigarrow (x, \overline{x} \mid \overline{c} \mid e_2)}\ \text{WF-Def}$$

$$\frac{x \in \Gamma \quad v \neq x \qquad \text{if } v = s \text{ then } x \notin \mathrm{fvs}(e_1) \qquad \Gamma - x \vdash e_1 \rightsquigarrow (\overline{x} \mid \overline{c} \mid e_2) \qquad \mathrm{fvs}(h) \notin \overline{x}}{\Gamma \vdash x = v;\ e_1 \rightsquigarrow (\overline{x} \mid x = v, \overline{c} \mid e_2)}\ \text{WF-Eq}$$

$$\frac{\emptyset \vdash r_1 \rightsquigarrow (\overline{x}_1 \mid \overline{c}_1 \mid e_1) \quad \cdots \quad \emptyset \vdash r_n \rightsquigarrow (\overline{x}_n \mid \overline{c}_n \mid e_n) \qquad \text{all } x_i \text{ distinct}}{\vdash_* r_1, \cdots, r_n \rightsquigarrow (\overline{x}_1, \cdots, \overline{x}_n \mid \overline{c}_1, \cdots, \overline{c}_n \mid e_1, \cdots, e_n)}\ \text{WF-many}$$

Fig. 5. **Well-formedness of Results**

are used. ELIM-DEF allows you to tidy up an expression, but it is not necessary for progress, and you can omit it entirely if you want. The normalization rules of Fig. 4 also

- Associate "; " to the right (rule NORM-SEQ-ASSOC).
- Drop a value to the left of a "; " (rule NORM-VAL).
- Propagate **fail** (rules FAIL-SEQL, FAIL-SEQR, and FAIL-EQ).
- Put a variable on the LHS of an equality, where possible (rule NORM-SWAP-EQ).

Note that the normalization rules preserve the left-to-right sequencing of expressions, which matters because choices are made left-to-right as we saw in Section 2.3. Moreover, note that *the normalisation rules do not float equalities or existentials out of choices.* That restriction is the key to

localizing unification (Section 2.3), and the flexible/rigid distinction of Section 2.5. For example, consider the expression $(y = ((x = 3;\ x * 2) \mid (x = 4));\ \langle x + 1, y\rangle)$. We must not propagate the binding $(x = 3)$ to the expression $(x + 1)$, because the latter is outside the choice, and a different branch of the choice binds $x$ to 4. But rule DEREF-S *can* propagate it locally within the first arm of the choice, thus[7]:

$$y = ((x = 3;\ x * 2) \mid (x = 4));\ \langle x + 1, y\rangle \longrightarrow y = ((x = 3;\ 3 * 2) \mid (x = 4));\ \langle x + 1, y\rangle$$

To make further progress, we need a rule for choice; see Section 3.5.

[LA: Somewhere we should mention that the result of a (non-stuck) reduction with be an expression $e$ with $\emptyset \vdash e \rightsquigarrow (\overline{x} \mid \overline{x = h} \mid v)$, i.e., a value with a set of bindings for heap values.]

### 3.5 Rules for choice

The rules for choice are given in Fig. 3. Rules ONE-VALUE, ONE-CHOICE and ONE-FAIL describe the semantics for **one**, just as in Section 2.5. Similarly ALL-FAIL and ALL-CHOICE describe the semantics of **all** (Section 2.6). These rules use the well-formed-result judgement, introduced in Section 3.4 and defined in Fig. 5, to ensure that each arm of the choice(s) consists of a value wrapped in a heap.

The most interesting rule is CHOOSE which, just as described in Section 2.2, "floats the choice outwards", duplicating the surrounding context. But what "surrounding context" precisely? We use two new contexts, $SX$ and $CX$, both defined in Fig. 1. A *choice context* $CX$ is like an execution context $X$, *but with no possible choices to the left of the hole*:

$$CX ::= \Box \mid v = CX \mid CX;\ e \mid ce;\ CX \mid \exists x.\, CX$$

Here, $ce$ is guaranteed-choice-free expression (syntax in Fig. 1). This syntactic condition is necessarily conservative; for example, a call $f(x)$ is considered not guaranteed-choice-free, because it depends on what function $f$ does. We must guarantee not to have choices to the left so that we preserve order—see Section 2.3.

The context $SX$ (Fig. 3) is a *scope context*; it ensures that $CX$ is as large as possible. This is a subtle point: without this restriction we lose confluence. To see this, consider[8]:

$$\begin{array}{ll} & \exists x.\ (\textbf{if}\ (x > 0)\ \textbf{then}\ 55\ \textbf{else}\ 44);\ x = 1;\ (77 \mid 99) \\ \longrightarrow\{\text{NORM-SEQ-SWAP2}\} & \exists x.\ x = 1;\ (\textbf{if}\ (x > 0)\ \textbf{then}\ 55\ \textbf{else}\ 44);\ (77 \mid 99) \\ \longrightarrow\{\text{DEREF-S}\} & \exists x.\ x = 1;\ (\textbf{if}\ (1 > 0)\ \textbf{then}\ 55\ \textbf{else}\ 44);\ (77 \mid 99) \\ \longrightarrow\{\text{simplify } \textbf{if}\} & \exists x.\ x = 1;\ 55;\ (77 \mid 99) \\ \longrightarrow\{\text{SEQ, ELIM-DEF}\} & 77 \mid 99 \end{array}$$

But suppose instead we floated the choice out, *part-way*, like this:

$$\begin{array}{ll} & \exists x.\ (\textbf{if}\ (x > 0)\ \textbf{then}\ 55\ \textbf{else}\ 44);\ x = 1;\ (77 \mid 99) \\ \longrightarrow\{\text{Bogus CHOOSE}\} & \exists x.\ (\textbf{if}\ (x > 0)\ \textbf{then}\ 55\ \textbf{else}\ 44);\ (x = 1;\ 77) \mid (x = 1;\ 99) \end{array}$$

Now the $(x = 1)$ is inside the choice branches, so we cannot use NORM-SEQ-SWAP2 to move it to the left of the **if**. Nor can we use CHOOSE again to float the choice further out, because the **if** is not guaranteed choice-free (for example, the branches might contains choices). So, alas, we are stuck! Our not-entirely-satisfying solution is to force CHOOSE to float the choice all the way to the innermost enclosing scope construct; hence the $SX$ in the rule.

Rule CHOOSE moves choices around; only ONE-CHOICE and ALL-CHOICE *decompose* choices. So choice behaves a bit like a data constructor, or normal form, of the language. For this reason we call this

[7] You may wonder if this local propagation is useful, a point we return to in Section 3.3.

[8] Remember, **if** is syntactic sugar for a use of **one**, see Section 2.5, but using **if** makes the example easier to understand.

approach *spatial choice*, in contrast to approaches that eliminate choice by non-deterministically picking one branch or the other, which immediately gives up confluence.

The rules for **one** and **all** expect multiple choices to be normalized into a right-associative list of non-failing values, and the administrative rules ASSOC-CHOICE, FAIL-L and FAIL-R bring nested choices into that form. But why do these rules need a *SX* context? Again, they are needed to guarantee confluence. Suppose we had a rule FAIL-L-NO-SX that unconditionally rewrites (**fail** | $e$) to $e$. Now consider these two reduction sequences, starting from the same expression:

$$\begin{array}{lll} f\langle\rangle;\ (\mathbf{fail} \mid (3 = (1 \mid 3))) & \longrightarrow\{\text{FAIL-L-NO-SX}\} & f\langle\rangle;\ 3 = (1 \mid 3) \\ f\langle\rangle;\ (\mathbf{fail} \mid (3 = (1 \mid 3))) & \longrightarrow\{\text{CHOOSE}\} & f\langle\rangle;\ (\mathbf{fail} \mid ((3 = 1) \mid (3 = 3))) \\ & \longrightarrow\{\text{ULIT} \times 2\} & f\langle\rangle;\ (\mathbf{fail} \mid (\mathbf{fail} \mid 3)) \\ & \longrightarrow\{\text{FAIL-L-NO-SX} \times 2\} & f\langle\rangle;\ 3 \end{array}$$

The first sequence gets stuck after one step[9], while the second makes more progress; and the two results are not joinable.

### 3.6 $\mathcal{VC}$ is lenient

$\mathcal{VC}$ is *lenient* [Schauser and Goldstein 1995], not lazy (call-by-need), nor strict (call-by-value). Under lenient evaluation, everything is eventually evaluated, but functions can run before their arguments have a value. Consider a function call $f(e)$. In $\mathcal{VC}$ applications are in administrative normal form (ANF), so we must actually write $\exists x.\ x = e;\ f(x)$. This expression will not return a value until $e$ reduces to a value: that is, everything is eventually evaluated. But even so $f(x)$ can proceed to beta reduce (Section 3.1), assuming we know the definition of $f$.

Lenience supports *abstraction*. For example, we can replace an expression $(x = \langle y, 3\rangle;\ y > 7)$ by

$$\exists f.\ f = (\lambda\langle p, q\rangle.\ p = \langle q, 3\rangle;\ q > 7);\ f\langle x, y\rangle$$

Here, we abstract over the free variables of the expression, and define a named function $f$. Calling the function is just the same as writing the original expression. This transformation would not be valid under call-by-value.

This is not just a way to get *parallelism*, which was the original motivation for introducing lenience in the data-flow language Id [Schauser and Goldstein 1995]; it affects *semantics*. Consider

$$\exists f\, x\, y.\ f = (\lambda p.\ x = 7;\ p);\ y = (\mathbf{if}\ (x > 0)\ \mathbf{then}\ 7\ \mathbf{else}\ 8);\ f(y)$$

Here, $y$ does not get a value until $x$ is known; but $x$ does not get its value (in this case 7) until $f$ is called. Without lenience this program would be stuck. Laziness would be another possible design choice, one that is even more expressive, as we discuss in Appendix B.4.

### 3.7 Evaluation strategy

Any rewrite rule can apply anywhere in the term, at any time. For example in the term $(x = 3 + 4;\ y = 3 * 2;\ x + y)$ the rewrite rules do not say whether to rewrite $3 + 4 \longrightarrow 7$ and then $3 * 2 \longrightarrow 6$, or the other way around. The rules do, however, require us to reduce $3 + 4 \longrightarrow 7$ before substituting for $x$ in $x + y$, because the DEREF rules only fire when the RHS is a value. By choosing rewrite rules carefully, we can for example express call-by-name, call-by-name, and call-by-need [Ariola et al. 1995].

An *evaluation strategy* answers the question: given a closed term, which unique redex, out of the many possible redexes, should I rewrite next to make progress towards the result? Any decent evaluation strategy should (a) guarantee to terminate if there is *any* terminating sequence of reductions; and (b) be amenable to compilation into efficient code. For example, in the pure lambda

[9]The strange $f\langle\rangle$ prevents us using CHOOSE to float the $(1 \mid 3)$ upwards.

calculus, *normal-order reduction*, sometimes called *leftmost outermost reduction*, is an evaluation strategy that guarantees to terminate if any strategy does so.

It would be even better if the strategy could (c) guarantee to find the result in the minimal number of rewrite steps—so called "optimal reduction" [Asperti and Guerrini 1999; Lamping 1990; Lévy 1978]—but optimal reduction is typically very hard, even in theory, and invariably involves reducing under lambdas, so for practical purposes it is well out of reach.

Formalising an evaluation strategy for $\mathcal{VC}$ is beyond the scope of this paper, but we can make some informal comments. First, in service of (b) we envisage compiling lambdas to code, and thus we never rewrite under a lambda [Peyton Jones 1987]. Second, it never makes sense to evaluate in the right-hand argument of a choice[10], because $\mathcal{VC}$'s strong-ordering semantics mean that we must first find out what the left-hand argument is (especially, whether it fails) before the right-hand one can be used. So the basic plan is: rewrite the leftmost-outermost redex, subject to these two constraints.

The trouble is that it is hard to say what the "leftmost" redex is. For example in $(e;\ \langle x, 3\rangle = \langle 2, y\rangle)$, the equality may or may not be the leftmost redex, depending on whether $e$ is stuck (*i.e.*, contains no redexes); and whether or not $e$ is stuck is a not syntactic property, and (worse) may depend not only on $e$ itself, but on its context. Even worse, $e$ may subsequently become un-stuck when we rewrite the equality. Any calculus in which a redex to the "right" may unblock one to the "left"—that is, residuation—must grapple with this problem, so we leave evaluation strategy and compilation for future work.

## 4 METATHEORY

The rules of our rewrite semantics can be applied anywhere, in any order, and they give meaning to programs without committing to a particular evaluation strategy. But then it had better be the case that no matter how the rules are applied, one always obtains the same result!

**Reductions and Confluence** A *reduction* $\mathcal{R}$ is a binary relation on a set of terms $\mathcal{E}$. We write $\mathcal{R}^k$ for the $k$-step closure of $\mathcal{R}$ and $\mathcal{R}^*$ for the reflexive and transitive closure of $\mathcal{R}$, *i.e.*$\mathcal{R}^* \equiv \cup_{0 \leqslant k} \mathcal{R}^k$. We write $e \rightarrow_{\mathcal{R}} e'$ (*a steps to b*) if $(e, e') \in \mathcal{R}$ and $e \twoheadrightarrow_{\mathcal{R}} e'$ (*a reduces to b*) if $(e, e') \in \mathcal{R}^*$. A reduction $\mathcal{R}$ is *confluent* if whenever $e \twoheadrightarrow_{\mathcal{R}} e_1$ and $e \twoheadrightarrow_{\mathcal{R}} e_2$, there exists an $e'$ such that $e_1 \twoheadrightarrow_{\mathcal{R}} e'$ and $e_2 \twoheadrightarrow_{\mathcal{R}} e'$. Confluence gives us the assurance that we will not get different results when choosing different rules, or get stuck with some rules and not with others.

**Normal Forms** A term $e$ is an $\mathcal{R}$-*Normal Form* if there does not exist any $e'$ such that $e \rightarrow_{\mathcal{R}} e'$. Confluence implies that ultimately, rewriting terminates with at most one unique normal form, regardless of the evaluation strategy [Barendregt 1984].

Lemma 4.1 (Unicity). *If $\mathcal{R}$ is confluent then every term reduces to at most one normal form.*

### 4.1 Confluence

Our main result is that $\mathcal{VC}$'s reduction rules are confluent:

Theorem 4.2 (Confluence). *The reduction relation defined in Fig. 3 and 4 is confluent.*

*Proof sketch.* Our proof strategy is to divide the rules into groups, named $\mathcal{U}$, $\mathcal{A}$, etc in the Figures, prove confluence for each separately, and then prove that their combination is confluent. Given two reduction relations $R$ and $S$, we say that $R$ *commutes* with $S$ if for all terms $e, e_1, e_2$ such that $e \twoheadrightarrow_R e_1$ and $e \twoheadrightarrow_S e_2$ there exists $e'$ such that $e_1 \twoheadrightarrow_S e'$ and $e_2 \twoheadrightarrow_R e'$. We prove each individual sub-relation is confluent; and that they pairwise commute. Then confluence of their union follows, using Huet [1980]:

[10] Except perhaps in parallel, of course.

Lemma 4.3 (Commutativity). *If $R$ and $S$ are confluent and commute, then $R \cup S$ is confluent.*

Proving confluence for $\mathcal{R}$, $\mathcal{A}$, $\mathcal{N}$, $\mathcal{F}$ and $\mathcal{G}$ is easy: they all satisfy the *diamond property*, namely, that two different reduction steps can be joined at a common term *by a single step*. This property can be verified easily by taking critical pairs. Any relation satisfying the diamond property is confluent [Barendregt 1984].

Alas, the unification relation $\mathcal{U}$ does not satisfy the diamond property, because it may need multiple steps to join the results of two different one-step reductions. For example, consider the term $(x = \langle 1, y\rangle;\ x = \langle z, 2\rangle;\ x = \langle 1, 2\rangle;\ 3)$. It can be reduced in one step by substituting $x$ in the final equality by either $\langle 1, y\rangle$ or $\langle z, 2\rangle$. After this it will take multiple steps to join the two terms.

Following a well-trodden path in proofs of confluence for the $\lambda$-calculus (e.g. [Barendregt 1984]), our proof of confluence for $\mathcal{U}$ works as follows: we find a *sub-relation* that satisfies three properties. First, it is *locally* confluent, meaning if $e$ single-steps to $e_1$ and $e_2$ then $e_1$ and $e_2$ can be joined at some $e'$. Second, it is terminating. Newman's Lemma [Barendregt 1984] then implies the relation is confluent; and hence so is its reflexive transitive closure. Third, that the *closure* of the sub-relation is the same as the full reduction relation, which then implies that the full reduction relation is also confluent.

### 4.2 Design for confluence

$\mathcal{VC}$ is carefully designed to ensure confluence. Rule deref-h is particularly important. It prevents substituting heap values $h$ under lambdas and inside tuples; and the $A$ context only permits substitution in a place where the value $h$ can be *used immediately*, by application or unification. These restrictions matter for at least three different reasons.

**Nested tuples.** Our proof strategy for the confluence of $\mathcal{U}$ requires that $\mathcal{U}$ terminates. But if deref-h substituted inside tuples, or inside lambdas, it *doesn't* terminate:

$$\exists x.\ x = \langle 1, x\rangle;\ x \ \longrightarrow\ \exists x.\ x = \langle 1, x\rangle;\ \langle 1, x\rangle \ \longrightarrow\ \exists x.\ x = \langle 1, x\rangle;\ \langle 1, \langle 1, x\rangle\rangle \ \longrightarrow\ \ldots$$

Here, each step makes one substitution for $x$. An exactly analogous example can be made for a lambda value. We avoid this fruitless divergence by preventing deref-h from substituting under tuples or lambdas. Instead an equality like $(x = h)$ is left as a "heap-constraint" which can be used (via deref-h) whenever we actually need to access the contents of the value, via unification or appliction; or it can be eliminated via the garbage collection rules.

**The odd/even problem.** Suppose we combined deref-s and deref-h into a a single rule that freely substituted *any* value $v$ for an occurrence of $x$. Then we would lose confluence in the case of mutual recursion:

$$\exists x, y.\ x = \langle 1, y\rangle;\ y = \lambda z.\ x;\ x \quad \longrightarrow^* \quad \exists y.\ y = \lambda z.\ \langle 1, y\rangle;\ \langle 1, y\rangle \qquad \text{(1: substitute for } x \text{ first)}$$
$$\exists x, y.\ x = \langle 1, y\rangle;\ y = \lambda z.\ x;\ x \quad \longrightarrow^* \quad \exists x.\ x = \langle 1, \lambda z.\ x\rangle;\ x \qquad \text{(2: substitute for } y \text{ first)}$$

The result of (1) and (2) have the same meaning (are indistinguishable by a $\mathcal{VC}$ context) but cannot be joined by rewrite rules. This is a well known problem, and an exactly similar phenomenon arises with inlining mutually recursive $\lambda$-terms. Examples like this show that *syntactic* confluence is too strong: what we really need is that our rewrites rules are *semantics* preserving — but of course that requires an independent notion of semantics (see Appendix C for an initial attempt). We restore confluence by restricting deref-h, but an interesting alternative approach would be to seek a weaker form of confluence, such as *skew confluence* [Ariola and Blom 2002].

**Unifying lambdas.** In $\mathcal{VC}$ an attempt to unify two lambdas fails, even if the lambdas are semantically identical (rule u-fail). Why? Because semantic identity of functions is un-implementable. We

cannot instead say that the attempt to unify gets stuck, because that leads to non-confluence. Here is an expression that rewrites in two different ways, depending on which equality we DEREF-H first:

$$(\lambda p.\, 1) = (\lambda q.\, 2);\ 1 \longleftarrow^* \exists x.\, x = (\lambda p.\, 1);\ x = (\lambda q.\, 2);\ x\ \langle\rangle \longrightarrow^* (\lambda q.\, 2) = (\lambda p.\, 1);\ 2$$

These two outcomes cannot be joined. But making unification on lambdas fail, both outcomes lead to **fail**, and confluence is restored.

There is a very delicate interaction between U-FAIL and the apparently-innocuous rule U-SCALAR (Fig. 3). Consider $(\exists x.\, x = (\lambda y.\, y);\ x = x;\ 0)$. If we apply U-SCALAR, and then DEREF-H we get $(\lambda y.\, y)$. But if we *first* apply DEREF-H, twice, we get $((\lambda y.\, y) = (\lambda y.\, y);\ 0)$, and that fails. Yikes!

But in fact all is well: the $A$ context (Fig. 3) *only allows* DEREF-H *to apply in positions where the value is immediately consumed in some way*, by being applied to an argument, or being unified with a value. So in our example, DEREF-H simply does not apply. Only U-SCALAR does, so we get $(\exists x.\, x = (\lambda y.\, y);\ x;\ 0)$. Confluence is restored. But the ice is thin here, so it is reassuring that we have a proof of confluence.

## 5 $\mathcal{VC}$ IN CONTEXT: REFLECTIONS AND RELATED WORK

Functional logic programming has a rich literature; an excellent starting point is Hanus's survey [Hanus 2013]. Now that we know what $\mathcal{VC}$ is, we can identify its distinctive features, and compare them to other approaches.

### 5.1 Choice and non-determinism

A significant difference between our presentation and earlier works is our treatment of choice. Consider an expression like $(3 + (20 \mid 30))$. This is typically handled by a pair of non-deterministic rewrite rules:

$$e_1 \mid e_2 \longrightarrow e_1 \qquad e_1 \mid e_2 \longrightarrow e_2$$

So our expression rewrites (non-deterministically) to either $(3 + 20)$ or $(3 + 30)$; and that in turn allows the addition to make progress. Of course, including non-deterministic choice means the rules are non-confluent by construction. Instead, one must generalize to say that a reduction does not change the *set* of results; in the context of lambda calculi see for example [Kutzner and Schmidt-Schauß 1998; Schmidt-Schauß and Machkasova 2008].

In contrast, our rules never pick one side or the other of a choice. And yet $(3 + (20 \mid 30))$ can still make progress, by floating out the choice (rule CHOOSE in Fig. 3), thus $(3 + 20) \mid (3 + 30)$. In effect, *choices are laid out in space* (in the syntax of the term), rather than being explored by non-deterministic selection. Rule CHOOSE is not a new idea: it is common in calculi with choice, see *e.g.*, [de'Liguoro and Piperno 1995, Section 6.1, Dal Lago et al. 2020, Section 3] and, more recently, has been used to describe functional logic languages where it is variously called *bubbling* [Antoy et al. 2006] or *pull-tabbing* [Antoy 2011]. However, our formulation appears simpler, because we avoid the need for attaching an identifier to each choice with its attendant complications.

### 5.2 One and all

Logical variables, choice, and equalities are present in many functional logic languages. However **one** and **all** are distinctive features of $\mathcal{VC}$, with the notable exception of Smolka *et al.*'s language Fresh. Introduced in a technical report nearly 40 years ago [Smolka and Panangaden 1985], Fresh has *confinement* (equivalent to **one**) and *collection* (equivalent to **all**). It is a very interesting design, but one does not appear to have been implemented, and its treatment of equality and thus logical variables is rather different to ours.

Several aspects of **all** and **one** are worth noting. First, **all** *reifies* choice (a control operator) into a tuple (a data structure); for example, $\mathbf{all}\{1 \mid 7 \mid 2\}$ returns the tuple $\langle 1, 7, 2\rangle$. In the other

direction, indexing turns a tuple into choice (*e.g.*, $\exists i.\ \langle 1,7,2\rangle(i)$ yields $(1 \mid 7 \mid 2)$). Other languages can reify choices into a (non-deterministic) list, via an operator called `bagof`, or a mechanism called *set-functions* in an extension of Curry [Antoy and Hanus 2021, Section 4.2.7], implemented in the Kiel Curry System interpreter [Antoy and Hanus 2009; Brassel and Huch 2007, 2009]. But this is regarded as a somewhat sophisticated feature, whereas it is part of the foundational fabric of $\mathcal{VC}$. Curry's set-functions need careful explanation about sharing across non-deterministic choices, or what is "inside" and "outside" the set function, something that appears as a straightforward consequence of $\mathcal{VC}$'s single rule CHOOSE.

Second, even under the reification of **all**, $\mathcal{VC}$ is *deterministic*. Choice is not non-deterministic: $\mathcal{VC}$ takes pains to maintain order, so that when reifying choice into a tuple, the order of elements in that tuple is completely determined. This determinism has a price: as we saw in Section 2.3 and Section 3.5, we have to take care to maintain the left-to-right order of choices. However, maintaining that order has other payoffs. For example, it is relatively easy to add effects other than choice, including mutable variables and input/output, to $\mathcal{VC}$.

Thirdly, **one** allows us to reify failure; to try something and take different actions depending on whether or not it succeeds. Prolog's "cut" operator has a similar flavour, and Curry's set-functions allow one to do the same thing.

Finally, **one** and **all** neatly encapsulate the idea of "flexible" *vs.* "rigid" logical variables. As we saw in Section 2.5, logical variables bound outside **one**/**all** cannot be unified inside it; they are "rigid." This notion is nicely captured by the fact that equalities cannot float outside **one** and **all** (Section 3.4).

### 5.3 The semantics of logical variables

Our logical variables, introduced by $\exists$, are often called *extra variables* in the literature, because they are typically introduced as variables that appear on the right-hand side of a function definition, but are not bound on the left. For example, in Curry we can write

```
first x | x =:= (a,b) = a where a,b free
```

Here `a` and `b` are logical variables, not bound on the left; they get their values through unification (written "`=:=`"). In Curry they are explicitly introduced by the "`where a,b free`" clause, while in many other papers their introduction is implicit in the top-level rules, simply by not being bound on the left. These extra variables (our logical variables) are at the heart of the "logic" part of functional logic programming.

Constructor-based ReWrite Logic (CRWL) [González-Moreno et al. 1999] is the brand leader for high-level semantics for non-strict, non-deterministic functional logic languages. CRLW is a "big-step" rewrite semantics that rewrites a term to a value in a single step. López-Fraguas et al. [2007] make a powerful case for instead giving the semantics of a functional logic language using "small-step" rewrite rules, more like those of the lambda calculus, that successively rewrite the term, one step at a time, until it reaches a normal form. Their paper does exactly this, and proves equivalence to the CRWL framework. Their key insight (like us, inspired by Ariola et al. [1995]'s formalisation of the call-by-need lambda calculus) is to use **let** to make sharing explicit.

However both CRWL and Fraguas *et al.* suffer from a major problem: they require something we call *magical rewriting*. A key rewrite rule is this:

$$f(\theta(e_1),\ldots,\theta(e_n)) \longrightarrow \theta(rhs)$$
if $(e_1,\ldots,e_n) \longrightarrow rhs$ is a top-level function binding, and
$\theta$ is a substitution mapping variables to closed values, s.t $dom(\theta) = fvs(e_1,\ldots,e_n,rhs)$

The substitution for the free variables of the left-hand side can readily be chosen by matching the left-hand side against the call. But the substitution for the extra variables must be chosen "magically" [López-Fraguas et al. 2007, Section 7] or clairvoyantly, so as to make the future execution work out. This is admirably high level, because it hides everything about unification, but it is not much help to a programmer trying to understand a program, nor is it directly executable. In a subsequent journal paper they refine CRWL to avoid magical rewriting using "let-narrowing" [López-Fraguas et al. 2014, Section 6]; this system looks rather different to ours, especially in its treatment of choice, but is rather close in spirit.

To explain actual execution, the state of the art is described by Albert et al. [2005]. They give both a big-step operational semantics (in the style of [Launchbury 1993]), and a small-step operational semantics. These two approaches both thread a *heap* through the execution, which holds the unification variables and their unification state; the small-step semantics also has a *stack*, to specify the focus of execution. The trouble is that heaps and stacks are difficult to explain to a programmer, and do not make it easy to reason about program equivalence. In addition to this machinery, the model is further complicated with concurrency to account for residuation.

In contrast, our rewrite rules give a complete, executable (*i.e.*, no "magic") account of logical variables and choice, directly as small-step rewrites on the original program, rather than as the evolution of a (heap, control, stack) configuration. Moreover, we have no problem with residuation.

### 5.4 Flat *vs.* higher order

When giving the semantics of functional logic languages, a first-order presentation is almost universal. User-defined functions can be defined at top level only; and function symbols (the names of such functions) are syntactically distinguished from ordinary variables. As Hanus describes, it is possible to translate a higher-order program into a first-order form[11] using defunctionalisation [Hanus 2013, Section 3.3], and a built-in **apply** function. Sadly, this encoding is hardly a natural rendition of the lambda calculus, and it obstructs the goal of using rewrite rules to explain to programmers how their program might execute. In contrast, a strength of our $\mathcal{VC}$ presentation is that it deals natively with the full lambda calculus.

### 5.5 Intermediate language

Hanus's *Flat Language* [Albert et al. 2005, Fig 1], FLC, plays the same role as $\mathcal{VC}$: it is a small core language into which a larger surface language can be desugared. There are some common features: variables, literals, constructor applications, and sequencing (written `hnf` in FLC). However, it seems that $\mathcal{VC}$ has a greater economy of concepts. In particular, FLC has two forms of equality (`==`) and (`=:=`), and two forms of case-expression, `case` and `fcase`. In each pair, the former suspends if it encounters a logical variable; the latter unifies or narrows respectively. In contrast, $\mathcal{VC}$ has a single equality (=), and the orthogonal **one** construct, to deal with all four concepts.

FLC has let-expressions (`let x=e in b`), where $\mathcal{VC}$ uses $\exists$ and (again) unification. FLC also uses the same construct for a different purpose, to bring a logical variable into scope, using the strange binding `x=x`, thus (`let x=x in e`). In contrast, $\exists x.\, e$ seems more direct.

## 6 LOOKING BACK, LOOKING FORWARD

The semantics of $\mathcal{VC}$ is designed at a level intended to capture the *computational model* of the language; not all formal semantics do so. Defining a language by giving its low-level semantics is *precise* but not necessarily *illuminating*. For example, giving a reference compiler that compiles the

[11] Hanus does not mention this, but for a language with arbitrarily nested lambdas one would need to do lambda-lifting as well, but that is perhaps a minor point.

program to x86 instructions is precise, but is not helpful to a human who is trying to understand exactly what the original program meant. Likewise, a high-level semantics simply provides the eventual answer produced by the program, without insight into the *computational steps* that got us from program start to program completion.

The moral here is that *formal specifications can be obfuscatory—or illuminating*. The latter kind shed light because they are defined *in terms of the intended mechanisms of the language*. $\mathcal{VC}$ does this; it respects the conceptual structures of the Verse language.

Note that when we say "illuminating" we mean that in multiple ways. A semantics can be illuminating for humans who are trying to understand what a particular program does, or how a proposed change to the language will affect the language. It can drive analyses that help the compiler optimize programs. It underlies the use of formal methods and verification to provide machine-derived and -checkable proofs of correctness. All of these applications depend on the semantics being defined at the appropriate level: the level of the computational model that underlies the language. This has been our goal in this work.

We have much left to do. The full Verse language has statically checked types. In the dynamic semantics, the types can be represented by partial identity functions—identity of the values of the type and **fail** otherwise. This gives a distinctive new perspective on type systems, one that we intend to develop in future work. The full Verse language also has a statically-checked effect system, including both mutable references and input/output. All these effects must be *transactional*, *e.g.*, when the condition of an **if** fails, any store effects in the condition must be rolled back. We have preliminary reduction rules for updateable references, but they are not included here.

## ACKNOWLEDGMENTS

We thank our colleagues for their helpful and specific feedback on earlier drafts of this paper, including Jessica Augustsson, Francisco López-Fraguas, Andy Gordon, Michael Hanus, Juan Rodríguez Hortalá, John Launchbury, Dale Miller, Andy Pitts, Niklas Röjemo, Jaime Sánches-Hernández. and Andrew Scheidecker.

## REFERENCES

Elvira Albert, Michael Hanus, Frank Huch, Javier Oliver, and German Vidal. 2005. Operational semantics for declarative multi-paradigm languages. *Journal of Symbolic Computation* 40, 1 (2005), 795–829. https://doi.org/10.1016/j.jsc.2004.01.001 Reduction Strategies in Rewriting and Programming special issue.

S. Antoy. 2011. On the Correctness of Pull-Tabbing. *Theory and Practice of Logic Programming* 11, 4-5 (2011), 713–730. https://doi.org/10.1017/S1471068411000263

S Antoy, D Brown, and S Chiang. 2006. Lazy context cloning for non-deterministic graph rewriting. In *Proc. of the 3rd International Workshop on Term Graph Rewriting*. Vienna, Austria, 61–70.

S. Antoy and M. Hanus. 2009. Set Functions for Functional Logic Programming. In *Proceedings of the 11th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'09)*. ACM Press, 73–82. https://doi.org/10.1145/1599410.1599420

S Antoy and M Hanus. 2021. *Curry: a tutorial introduction.* Technical Report. University of Kiel.

Zena M. Ariola and Stefan Blom. 2002. Skew confluence and the lambda calculus with letrec. *Annals of Pure and Applied Logic* 117, 1 (2002), 95–168. https://doi.org/10.1016/S0168-0072(01)00104-X

Zena M. Ariola, John Maraist, Martin Odersky, Matthias Felleisen, and Philip Wadler. 1995. A Call-by-Need Lambda Calculus. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '95)*. Association for Computing Machinery, New York, NY, USA, 233246. https://doi.org/10.1145/199448.199507

Andrea Asperti and Stefano Guerrini. 1999. *The Optimal Implementation of Functional Programming Languages.* Cambridge University Press.

H. P. (Hendrik Pieter) Barendregt. 1984. *The lambda calculus : its syntax and semantics* (rev. ed. ed.). North-Holland, Amsterdam ;.

Bernd Brassel and Frank Huch. 2007. On a tighter integration of functional and logic programming. In *5th Asian Symposium on programming languages and systems (APLAS'07) (LNCS)*, Vol. 4807. Springer, 122–138.

B Brassel and F Huch. 2009. The Kiel Curry System KiCS. In *Applications of Declarative Programing and Knowledge Management*, Vol. 5437. Springer, 195–205.

Jan Christiansen, Daniel Seidel, and Janis Voigtländer. 2010. An Adequate, Denotational, Functional-Style Semantics for Typed FlatCurry, In International Workshop on Functional and Constraint Logic Programming. *International Workshop on Functional and Constraint Logic Programming*, 119–136. https://doi.org/10.1007/978-3-642-20775-4_7

U. Dal Lago, G. Guerrieri, and W. Heijltjes. 2020. Decomposing Probabilistic Lambda-Calculi. In *Foundations of Software Science and Computation Structures (FoSSaCS'20) (Lecture Notes in Computer Science)*, Vol. 12077. Springer.

Ugor de'Liguoro and Adolfo Piperno. 1995. Nondeterministic extensions of untyped lambda calculus. *Information and Computation* 122 (1995), 149–177.

Matthias Felleisen and Daniel P. Friedman. 1986. Control operators, the SECD machine, and the lambda calculus. In *Formal Description of Programming Concepts III*. Elsevier, 193–217.

Matthias Felleisen, Daniel P. Friedman, Eugene Kohlbecker, and Bruce Duba. 1987. A syntactic theory of sequential control. *Theoretical Computer Science* 3 (1987), 205–237. Issue 52.

J.C. González-Moreno, M.T. Hortalá-González, F.J. López-Fraguas, and M. Rodríguez-Artalejo. 1999. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming* 40 (1999), 47–87. https://doi.org/10.1016/S0743-1066(98)10029-8

Michael Hanus. 2013. Functional Logic Programming: From Theory to Curry. In *Programming Logics*, A Voronkov and C Weidenbach (Eds.). Lecture Notes in Computer Science, Vol. 7797. Springer Verlag. https://doi.org/10.1007/978-3-642-37651-1_6

Michael Hanus. 2016. *Curry: an integrated functional logic language.* Technical Report. University of Kiel.

Gérard Huet. 1980. Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems: Abstract Properties and Applications to Term Rewriting Systems. *J. ACM* 27, 4 (oct 1980), 797–821. https://doi.org/10.1145/322217.322230

A. Kutzner and M. Schmidt-Schauß. 1998. A non-deterministic call-by-need lambda calculus. In *3th ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*. ACM, 324–335.

John Lamping. 1990. An algorithm for optimal lambda-calculus reduction. In *Proceedings of the Seventeenth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 16–30.

John Launchbury. 1993. A Natural Semantics for Lazy Evaluation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Charleston, South Carolina, USA) *(POPL '93)*. Association for Computing Machinery, New York, NY, USA, 144154. https://doi.org/10.1145/158511.158618

Jean-Jacques Lévy. 1978. *Réductions Correctes et Optimales dans le Lambda-calcul.* Ph.D. Dissertation. Université Paris VII.

Francisco J. López-Fraguas, Juan Rodríguez-Hortalá, and Jaime Sánchez-Hernández. 2007. A Simple Rewrite Notion for Call-Time Choice Semantics. In *Proceedings of the 9th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming* (Wroclaw, Poland) *(PPDP '07)*. Association for Computing Machinery, New York, NY, USA, 197208. https://doi.org/10.1145/1273920.1273947

Francisco Javier López-Fraguas, Enrique Martin-Martin, Juan Rodríguez-Hortalá, and Jaime Sánchez-Hernández. 2014. Rewriting and narrowing for constructor systems with call-time choice semantics. *Theory and Practice of Logic programming* 14 (2014), 165–213. Issue 2.

Simon Peyton Jones. 1987. *The implementation of functional programming languages.* Prentice Hall.

Simon Peyton Jones. 1992. Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-machine. *Journal of Functional Programming* 2 (July 1992), 127–202. https://www.microsoft.com/en-us/research/publication/implementing-lazy-functional-languages-on-stock-hardware-the-spineless-tagless-g-machine/

JA Robinson. 1965. A machine-oriented logic based on the resolution principle. *J. ACM* 12 (1965), 23–41. Issue 1.

Klaus E. Schauser and Seth C. Goldstein. 1995. How Much Non-Strictness Do Lenient Programs Require?. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture* (La Jolla, California, USA) *(FPCA '95)*. Association for Computing Machinery, New York, NY, USA, 216225. https://doi.org/10.1145/224164.224208

M. Schmidt-Schauß and E. Machkasova. 2008. A finite simulation method in a nondeterministic call-by-need lambda-calculus with letrec, constructors, and case. In *19th International Conference on Rewriting Techniques and Applications (RTA'08) (LNCS)*, Vol. 5117. Springer, 321–335.

Gert Smolka and Prakash Panangaden. 1985. *FRESH: A Higher-Order Language with Unification and Multiple Results.* Technical Report TR85-685. Cornell University. https://hdl.handle.net/1813/6525

| | |
|---|---|
| | $swap\langle x, y\rangle := \langle y, x\rangle;\ \exists p.\ swap(p) = \langle 2, 3\rangle;\ p$ |
| $\longrightarrow$ {DESUGAR} | $\exists swap.\ swap = (\lambda xy.\ \exists x\, y.\ \langle x, y\rangle = xy;\ \langle y, x\rangle);\ \exists p\, t.\ t = swap(p);\ t = \langle 2, 3\rangle;\ p$ |
| ① $\longrightarrow$ {DEREF-H,ELIM-DEF} | $\exists p\, t.\ t = (\lambda xy.\ \exists x\, y.\ \langle x, y\rangle = xy;\ \langle y, x\rangle)(p);\ t = \langle 2, 3\rangle;\ p$ |
| $\longrightarrow$ {NORM-SEQ-SWAP} | $\exists p\, t.\ t = \langle 2, 3\rangle;\ t = (\lambda xy.\ \exists x\, y.\ \langle x, y\rangle = xy;\ \langle y, x\rangle)(p);\ p$ |
| ② $\longrightarrow$ {APP-BETA} | $\exists p\, t.\ t = \langle 2, 3\rangle;\ t = \exists xy.\ (xy = p;\ \exists x\, y.\ \langle x, y\rangle = xy;\ \langle y, x\rangle);\ p$ |
| $\longrightarrow$ {NORM-DEFR,NORM-SEQ-DEFR} | $\exists p\, t.\ t = \langle 2, 3\rangle;\ \exists xy.\ t = (xy = p;\ \exists x\, y.\ \langle x, y\rangle = xy;\ \langle y, x\rangle);\ p$ |
| $\longrightarrow$ {NORM-SEQ-DEFL} | $\exists p\, t\, xy.\ t = \langle 2, 3\rangle;\ t = (xy = p;\ \exists x\, y.\ \langle x, y\rangle = xy;\ \langle y, x\rangle);\ p$ |
| $\longrightarrow$ {NORM-SEQR,NORM-SEQ-ASSOC} | $\exists p\, t\, xy.\ t = \langle 2, 3\rangle;\ xy = p;\ t = \exists x\, y.\ (\langle x, y\rangle = xy;\ \langle y, x\rangle);\ p$ |
| $\longrightarrow$ {NORM-SEQ-SWAP} | $\exists p\, t\, xy.\ xy = p;\ t = \langle 2, 3\rangle;\ t = \exists x\, y.\ (\langle x, y\rangle = xy;\ \langle y, x\rangle);\ p$ |
| ③ $\longrightarrow$ {DEREF-S,ELIM-DEF} | $\exists p\, t.\ t = \langle 2, 3\rangle;\ t = \exists x\, y.\ (\langle x, y\rangle = p;\ \langle y, x\rangle);\ p$ |
| $\longrightarrow$ {NORM-DEFR,NORM-SEQ-DEFR} | $\exists p\, t.\ t = \langle 2, 3\rangle;\ \exists x.\ t = \exists y.\ (\langle x, y\rangle = p;\ \langle y, x\rangle);\ p$ |
| $\longrightarrow$ {NORM-SEQ-DEFL} | $\exists p\, t\, x.\ t = \langle 2, 3\rangle;\ t = \exists y.\ (\langle x, y\rangle = p;\ \langle y, x\rangle);\ p$ |
| $\longrightarrow$ {NORM-DEFR,NORM-SEQ-DEFR} | $\exists p\, t\, x.\ t = \langle 2, 3\rangle;\ \exists y.\ t = (\langle x, y\rangle = p;\ \langle y, x\rangle);\ p$ |
| $\longrightarrow$ {NORM-SEQ-DEFL} | $\exists p\, t\, x\, y.\ t = \langle 2, 3\rangle;\ t = (\langle x, y\rangle = p;\ \langle y, x\rangle);\ p$ |
| $\longrightarrow$ {NORM-SEQR} | $\exists p\, t\, x\, y.\ t = \langle 2, 3\rangle;\ (\langle x, y\rangle = p;\ t = \langle y, x\rangle);\ p$ |
| $\longrightarrow$ {DEREF-H,ELIM-DEF} | $\exists p\, x\, y.\ (\langle x, y\rangle = p;\ \langle 2, 3\rangle = \langle y, x\rangle);\ p$ |
| $\longrightarrow$ {NORM-SEQ-ASSOC,NORM-SWAP-EQ} | $\exists p\, x\, y.\ p = \langle x, y\rangle;\ \langle 2, 3\rangle = \langle y, x\rangle;\ p$ |
| ④ $\longrightarrow$ {U-TUP,NORM-SEQ-ASSOC} | $\exists p\, x\, y.\ p = \langle x, y\rangle;\ 2 = y;\ (3 = x;\ \langle 2, 3\rangle);\ p$ |
| $\longrightarrow$ {NORM-SWAP-EQ} | $\exists p\, x\, y.\ p = \langle x, y\rangle;\ y = 2;\ (3 = x;\ \langle 2, 3\rangle);\ p$ |
| $\longrightarrow$ {NORM-SEQ-SWAP} | $\exists p\, x\, y.\ y = 2;\ p = \langle x, y\rangle;\ (3 = x;\ \langle 2, 3\rangle);\ p$ |
| ⑤ $\longrightarrow$ {DEREF-S,ELIM-DEF} | $\exists p\, x.\ p = \langle x, 2\rangle;\ (3 = x;\ \langle 2, 3\rangle);\ p$ |
| $\longrightarrow$ {NORM-SEQ-ASSOC} | $\exists p\, x.\ p = \langle x, 2\rangle;\ 3 = x;\ \langle 2, 3\rangle;\ p$ |
| $\longrightarrow$ {NORM-SWAP-EQ} | $\exists p\, x.\ p = \langle x, 2\rangle;\ x = 3;\ \langle 2, 3\rangle;\ p$ |
| $\longrightarrow$ {NORM-SEQ-SWAP} | $\exists p\, x.\ x = 3;\ p = \langle x, 2\rangle;\ \langle 2, 3\rangle;\ p$ |
| ⑥ $\longrightarrow$ {DEREF-S,ELIM-DEF} | $\exists p.\ p = \langle 3, 2\rangle;\ \langle 2, 3\rangle;\ p$ |
| ⑦ $\longrightarrow$ {NORM-VAL} | $\exists p.\ p = \langle 3, 2\rangle;\ p$ |
| $\longrightarrow$ {POST-REDUCTION-INLINE} | $\langle 3, 2\rangle$ |

Fig. 6. Sample reduction sequence

## A EXAMPLE

A complete reduction sequence for a small example can be found in figure 6. This example shows how constraining the output of a function call can constrain the argument. While most of the reductions are administrative in nature, these are the highlights: At ① the *swap* function is inlined so that at ② a $\beta$-reduction can happen.

Step ③ inlines the argument, and ④ does the matching of the tuple. At ⑤ and ⑥ the actual numbers are inline. After removing some garbage we reach the result at ⑦.

## B VARIATIONS AND CHOICES

In a calculus like $\mathcal{VC}$ there is room for many design variations. We discuss some of them here.

### B.1 Dead existentials

Consider the term $(\exists x.\ 99)$. This rewrites to 99 by DEF-ELIM, but you could argue that it should instead be stuck. For example, the term $(\exists x.\ x = (1 \mid 2);\ 99)$ rewrites to $(99 \mid 99)$, producing two results, *one for each solution for* $x$. So, if $x$ is entirely unconstrained, maybe we should return an infinite number of results? It would be easy to change this decision, by adjusting the rules in Fig. 5 for well-formed results.

### B.2 Ordering and choices

As we discussed in Section 3.5, rule CHOOSE is less than satisfying, for two reasons. First, the $CX$ context uses a conservative, syntactic analysis for choice-free expressions; and second, the $SX$ context is needed to force $CX$ to be maximal. A rule like this would be more satisfying:

$$\text{SIMPLER-CHOOSE} \quad CX[\,e_1 \mid e_2\,] \longrightarrow CX[\,e_1\,] \mid CX[\,e_2\,]$$

The trouble with that is that it may change the order of the results (Section 2.3). Another possibility would be to accept that results may come out in the "wrong" order, but have some kind of sorting mechanism to put them back into the "right" order. Something like this:

$$\textsc{labeled-choose} \quad CX[\,e_1 \mid e_2\,] \longrightarrow CX[\,L;\ e_1\,] \mid CX[\,R;\ e_2\,]$$

Here the two branches are labeled with $L$ and $R$. We can add new rules to reorder such labelled expressions, something in the spirit of

$$\textsc{sort} \quad (R;\ e_1) \mid (L;\ e_2) \longrightarrow (L;\ e_2) \mid (R;\ e_1)$$

We believe this can be made to work, and it would allow more programs to evaluate, but it adds unwelcome clutter to program terms, and the cure may be worse than the disease.

### B.3 Generalizing one and all

In $\mathcal{VC}$, we introduced **one** and **all** as the primitive choice-consuming operators, and neither is more general than the other, as discussed in Section 2.6. We could have introduced a more general operator **split** as $e ::= \cdots \mid \mathbf{split}\{e, v_1, v_2\}$ and rules

$$\begin{array}{lrcll}
\textsc{split-fail} & \mathbf{split}\{\mathbf{fail}, f, g\} & \longrightarrow & f\langle\rangle & \\
\textsc{split-choice} & \mathbf{split}\{e_1 \mid e_2, f, g\} & \longrightarrow & g\langle e_1, \lambda x.\, e_2\rangle & \text{if } \emptyset \vdash e_1 \rightsquigarrow (\overline{x} \mid \overline{c} \mid v),\ x \text{ fresh} \\
\textsc{split-value} & \mathbf{split}\{e, f, g\} & \longrightarrow & g\langle e, \lambda x.\, \mathbf{fail}\rangle & \text{if } \emptyset \vdash e \rightsquigarrow (\overline{x} \mid \overline{c} \mid v),\ x \text{ fresh}
\end{array}$$

The intuition behind **split** is that it distinguishes a failing computation from one that returns at least one value. If $e$ fails, it calls $f$, and if $e$ returns at least one value, passes that to $g$ together with the remaining computation, safely tucked away behind a lambda.

Indeed, this is more general, as we can implement **one** and **all** with **split**:

$$\begin{array}{rl}
\mathbf{one}\{e\} \equiv f(x) \coloneqq \mathbf{fail};\ g\langle x, y\rangle \coloneqq x; & \mathbf{split}\{e, f, g\} \\
\mathbf{all}\{e\} \equiv f(x) \coloneqq \langle\rangle; \quad g\langle x, y\rangle \coloneqq \mathit{cons}\langle x, \mathbf{split}\{y\langle\rangle, f, g\}\rangle; & \mathbf{split}\{e, f, g\}
\end{array}$$

For this paper we stuck to the arguably simpler **one** and **all**, to avoid confusing the presentation with these higher-order encodings, but there are no complications using **split** instead.

### B.4 Laziness

As Section 3.6 discussed, $\mathcal{VC}$ is lenient. Unlike Curry however, $\mathcal{VC}$ is not *lazy*. For example, consider: $\exists x.\ x = \mathit{loop}\langle\rangle;\ 3$. In a lazy language this expression would yield 3, but in $\mathcal{VC}$ everything is evaluated, and the infinite computation $\mathit{loop}\langle\rangle$ will prevent the expression from returning a value. There a good reason for this choice: the call to $\mathit{loop}\langle\rangle$ might fail, and we should not return 3 until we know there is no failure. With laziness we could easily lose confluence.

Another place that laziness could play a role is this. Remembering the duality between values and choices, one might also want **all** to return a lazy stream of results, one by one, rather than waiting for them all to terminate. For example, one might hope that this program would converge:

$$\exists y z.\ \langle y, z\rangle = \mathbf{all}\{\exists \mathit{onec}.\ \mathit{onec} = (1 \mid \mathit{onec})\};\ y$$

Here we suppose that **all** returns a lazy stream of values (represented as nested pairs), from which we may pick the first and discard the rest.

In short, there are good reasons for lenience, but a lazy variant of $\mathcal{VC}$ could be worth exploring.

**Domains**

$$
\begin{array}{rcl}
W & = & \mathbb{Z} + \langle W \rangle + (W \to W^*) \\
\langle W \rangle & = & \text{a finite tuple of values } W \\
Env & = & Ident \to W
\end{array}
$$

**Semantics of expressions and values**

$$
\begin{array}{rcl}
\mathcal{E}[\![e]\!] & : & Env \to W^* \\
\mathcal{E}[\![v]\!]\,\rho & = & unit(\mathcal{V}[\![v]\!]\,\rho) \\
\mathcal{E}[\![\mathbf{fail}]\!]\,\rho & = & empty \\
\mathcal{E}[\![e_1 \mid e_2]\!]\,\rho & = & \mathcal{E}[\![e_1]\!]\,\rho \;\Cup\; \mathcal{E}[\![e_2]\!]\,\rho \\
\mathcal{E}[\![e_1 = e_2]\!]\,\rho & = & \mathcal{E}[\![e_1]\!]\,\rho \;\Cap\; \mathcal{E}[\![e_2]\!]\,\rho \\
\mathcal{E}[\![e_1;\, e_2]\!]\,\rho & = & \mathcal{E}[\![e_1]\!]\,\rho \;\fatsemi\; \mathcal{E}[\![e_2]\!]\,\rho \\
\mathcal{E}[\![v_1\ v_2]\!]\,\rho & = & apply(\mathcal{V}[\![v_1]\!]\,\rho,\ \mathcal{V}[\![v_2]\!]\,\rho) \\
\mathcal{E}[\![\exists x.\, e]\!]\,\rho & = & \bigcup_{w \in W} \mathcal{E}[\![e]\!]\,(\rho[x \mapsto w]) \\
\mathcal{E}[\![\mathbf{one}\{e\}]\!]\,\rho & = & one(\mathcal{E}[\![e]\!]\,\rho) \\
\mathcal{E}[\![\mathbf{all}\{e\}]\!]\,\rho & = & unit(all(\mathcal{E}[\![e]\!]\,\rho))
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{V}[\![v]\!] & : & Env \to W \\
\mathcal{V}[\![x]\!]\,\rho & = & \rho(x) \\
\mathcal{V}[\![k]\!]\,\rho & = & k \\
\mathcal{V}[\![op]\!]\,\rho & = & O[\![op]\!] \\
\mathcal{V}[\![\lambda x.\, e]\!]\,\rho & = & \lambda w.\mathcal{E}[\![e]\!]\,(\rho[x \mapsto w]) \\
\mathcal{V}[\![\langle v_1, \cdots, v_n \rangle]\!]\,\rho & = & \langle \mathcal{V}[\![v_1]\!]\,\rho, \cdots, \mathcal{V}[\![v_n]\!]\,\rho \rangle
\end{array}
$$

$$
\begin{array}{rcl}
O[\![op]\!] & : & W \\
O[\![\mathbf{add}]\!] & = & \lambda w.\, \mathbf{if}\,(w = \langle k_1, k_2 \rangle)\ \mathbf{then}\ unit(k_1 + k_2)\ \mathbf{else}\ \text{WRONG} \\
O[\![\mathbf{gt}]\!] & = & \lambda w.\, \mathbf{if}\,(w = \langle k_1, k_2 \rangle \wedge k_1 > k_2)\ \mathbf{then}\ unit(k_1)\ \mathbf{else}\ empty \\
O[\![\mathbf{int}]\!] & = & \lambda w.\, \mathbf{if}\,(w = k)\ \mathbf{then}\ unit(k)\ \mathbf{else}\ empty
\end{array}
$$

$$
\begin{array}{rcll}
apply & : & (W \times W) \to W^* & \\
apply(k, w) & = & \text{WRONG} & k \in \mathbb{Z} \\
apply(\langle v_0, \ldots, v_n \rangle, k) & = & unit(v_k) & 0 \leqslant k \leqslant n \\
 & = & empty & \text{otherwise} \\
apply(f, w) & = & f(w) & f \in W \to W^*
\end{array}
$$

Fig. 7. Expression semantics

## C A DENOTATIONAL SEMANTICS FOR $\mathcal{VC}$

It is highly desirable to have a denotational semantics for $\mathcal{VC}$. A denotational semantics says directly what an expression *means* rather than how it *behaves*, and that meaning can be very perspicuous. Equipped with a denotational semantics we can, for example, prove that the left hand side and right hand side of each rewrite rule have the same denotation; that is, the rewrites are meaning-preserving.

**Domains**

$W^* = (\text{WRONG} + \mathcal{P}(W))_\perp$

**Operations**

| | | | | |
|---|---|---|---|---|
| Empty | $empty$ | : | $W^*$ | |
| | $empty$ | = | $\{\}$ | |
| Unit | $unit$ | : | $W \to W^*$ | |
| | $unit(w)$ | = | $\{w\}$ | |
| Union | $\Cup$ | : | $W^* \to W^* \to W^*$ | |
| | $s_1 \Cup s_2$ | = | $s_1 \cup s_2$ | |
| Intersection | $\Cap$ | : | $W^* \to W^* \to W^*$ | |
| | $s_1 \Cap s_2$ | = | $s_1 \cap s_2$ | |
| Sequencing | $\fatsemi$ | : | $W^* \to W^* \to W^*$ | |
| | $s_1 \fatsemi s_2$ | = | $s_2$ | if $s_1$ is non-empty |
| | | = | $\{\}$ | otherwise |
| One | $one$ | : | $W^* \to W^*$ | The result is either empty or a singleton |
| | $one(s)$ | = | ??? | |
| All | $all$ | : | $W^* \to \langle W \rangle$ | |
| | $all(s)$ | = | ??? | |

All operations over $W^*$ implicitly propagate $\perp$ and WRONG. E.g.

| | | | |
|---|---|---|---|
| $s_1 \Cup s_2$ | = | $\perp$ | if $s_1 = \perp$ or $s_2 = \perp$ |
| | = | WRONG | if ($s_1$ = WRONG and $s_2 \neq \perp$) or ($s_2$ = WRONG and $s_1 \neq \perp$) |
| | = | $s_1 \cup s_2$ | otherwise |

Fig. 8. Set semantics for $W^*$

But a denotational semantics for a functional logic language is tricky. Typically one writes a denotation function something like

$$\mathcal{E}[\![e]\!] : Env \to W$$

where $Env = Ident \to W$. So $\mathcal{E}$ takes an expession $e$ and an environment $\rho : Env$ and returns the value, or denotation, of the expresssion. The environment binds each free variable of $e$ to its value. But what is the semantics of $\exists x.\ e$? We need to extend $\rho$ with a binding for $x$, but what is $x$ bound to? In a functional logic language $x$ is given its value by various equalities scattered throughout $e$.

This section sketches our approach to this challenge. It is not finished work, and does not count as a contribution of our paper. We offer it because we have found it an illuminating alternative way to understand $\mathcal{VC}$, one that complements the rewrite rules that are the substance of the paper.

## C.1 A first attempt at a denotational semantics

Our denotational semantics for $\mathcal{VC}$ is given in Fig. 7.

- We have one semantic function (here $\mathcal{E}$ and $\mathcal{V}$) for each syntactic non terminal (here $e$ and $v$ respectively.)
- Each function has one equation for each form of the construct.

- Both functions take an environment $\rho$ that maps in-scope identifiers to a *single* value; see the definition $Env = Ident \to W$.
- The value function $\mathcal{V}$ returns a *single value* $W$, while the expression function $\mathcal{E}$ returns a *collection of values* $W^*$ (Appendix C.1).

The semantics is parameterised over the meaning of a "collection of values $W^*$". To a first approximation, think of $W^*$ a (possibly infinite) set of values $W$, with union, intersection etc having their ordinary meaning.

Our first interpretation, given in Figure 8, is a little more refined: $W^*$ includes $\bot$ and WRONG as well as a set of values. Our second interpretation is given in Figure 9, and discussed in Appendix C.4.

The equations themselves, in Fig. 7 are beautifully simple and compositional, as a denotational semantics should be.

The equations for $\mathcal{V}$ are mostly self-explanatory, but an equation like $\mathcal{V}[\![k]\!]\,\rho = k$ needs some explanation: the $k$ on the left hand side (e.g. "3") is a piece of *syntax*, but the $k$ on the right is the corresponding element of the *semantic world of values* $W$ (e.g. 3). As is conventional, albeit a bit confusing, we use the same $k$ for both. Same for *op*, where the semantic equivalent is the corresponding mathematical function.

The equations for $\mathcal{E}$ are more interesting.

- Values $\mathcal{E}[\![v]\!]\,\rho$: compute the single value for $v$, and return a singleton sequence of results. The auxiliary function *unit* is defined at the bottom of Fig. 7.
- In particular, values include lambdas. The semantics says that a lambda evaluates to a *singleton* collection, whose only element is a function value. But that function value has type $W \to W^*$; that is, it is a function that takes a single value and returns a *collection* of values.
- Function application $\mathcal{E}[\![v_1\ v_2]\!]\,\rho$ is easy, because $\mathcal{V}$ returns a single value: just apply the meaning of the function to the meaning of the argument. The *apply* function is defined in Figure 7.
- Choice $\mathcal{E}[\![e_1 \mid e_2]\!]\,\rho$: take the union (written $\Cup$) of the values returned by $e_1$ and $e_2$ respectively. For bags this union operator is just bag union (Figure 8).
- Unification $\mathcal{E}[\![e_1 \mid e_2]\!]\,\rho$: take the *intersection* of the values returned by $e_1$ and $e_2$ respectively. For bags, this "intersection" operator $\Cap$ is defined in Fig. 8. In this definition, the equality is mathematical equality of functions; which we can't implement for functions; see Appendix C.1.
- Sequencing $\mathcal{E}[\![e_1;\ e_2]\!]\,\rho$. Again we use an auxiliary function $\fatsemi$ to combine the meanings of $e_1$ and $e_2$. For bags, the function $\fatsemi$ (Fig. 8 again) uses a bag comprehension. Again it does a cartesian product, but without the equality constraint of $\Cap$.
- The semantics of (**one**$\{e\}$) simply applies the semantic function $one : W^* \to W^*$ to the collection of values returned by $e$. If $e$ returns no values, so does (**one**$\{e\}$); but if $e$ returns one or more values, (**one**$\{e\}$) returns the first. Of course that begs the question of what "the first" means – for bags it would be non-deterministic. We will fix this problem in Appendix C.4, but for now we simply ignore it.
- The semantics of (**all**$\{e\}$) is similar, but it always returns a singleton collection (hence the *unit* in the semantics of **all**$\{\cdot\}$) whose element is a (possibly-empty) tuple that contains all the values in the collection returned by $e$.

The fact that unification "=" maps onto intersection, and choice "|" onto union, is very satisfying.

The big excitement is the treatment of $\exists$. We must extend $\rho$, but what should we bind $x$ to? (Compare the equation for $\mathcal{V}[\![\lambda x.\ e]\!]$, where we have a value $w$ to hand.) Our answer is simple: *try*

*all possible values, and union the results*:

$$\mathcal{E}[\![\exists x.\, e]\!]\,\rho \;=\; \bigcup_{w \in W} \mathcal{E}[\![e]\!]\,(\rho[x \mapsto w])$$

That $\bigcup_{w \in W}$ means: enumerate all values in $w \in W$, in some arbitrary order, and for each: bind $x$ to $w$, find the semantics of $e$ for that value of $x$, namely $\mathcal{E}[\![e]\!]\,(\rho[x \mapsto w])$, and take the union (in the sense of $\Cup$) of the results.

Of course we can't possibly implement it like this, but it makes a great specification. For example $\exists x.\, x = 3$ tries all possible values for $x$, but only one of them succeeds, namely 3, so the semantics is a singleton sequence $[3]$.

### C.2 The denotational semantics is un-implementable

This semantics is nice and simple, but we definitely can't implement it! Consider

$$\exists x.\, (x^2 - x - 6) = 0;\; x$$

The semantics will iterate over all possible values for $x$, returning all those that satisfy the equality; including 3, for example. But unless our implementation can guarantee to solve quadratic equations, we can't expect it to return 3. Instead it'll get stuck.

Another way in which the implementation might get stuck is through unifying functions:

$$(\lambda x.\, x + x) = (\lambda y.\, y * 2) \quad \text{or even} \quad (\lambda x.\, x + 1) = (\lambda y.\, y + 1)$$

But not all unification-over-functions is ruled out. We do expect the implementation to succeed with

$$\exists f.\, ((\lambda x.\, x + 1) = f);\; f\,3$$

Here the $\exists$ will iterate over all values of $f$, and the equality will pick out the (unique) iteration in which $f$ is bound to the incrementing function.

So our touchstone must be:

- If the implementation returns a value at all, it must be the value given by the semantics.
- Ideally, the verifier will guarantee that the implementation does not get stuck, or go WRONG.

### C.3 Getting WRONG right

Getting WRONG right is a bit tricky.

- What is the value of $(3 = \langle\rangle)$? The intersection semantics would say *empty*, the empty collection of results, but we might want to say WRONG.
- Should WRONG be an element of $W$ or of $W^*$? We probably want (**one**{3 | **wrong**} to return a $\mathit{unit}(3)$ rather then WRONG?
- What about $\mathit{fst}(\langle 3, \textbf{wrong}\rangle)$? Is that wrong or 3?

There is probably more than one possible choice here.

### C.4 An order-sensitive denotational semantics

There is a Big Problem with this approach. Consider $\exists x.\, x = (4 \mid 3)$. The existential enumerates all possible values of x *in some arbitrary order*, and takes the union (i.e. concatention) of the results from each of these bindings. Suppose that $\exists$ enumerates 3 before 4; then the semantics of this expression is the sequence $[3, 4]$, and not $[4, 3]$ as it should be. And yet returning a sequence (not a set nor a bag) is a key design choice in Verse. What can we do?

Figure 9 give a new denotational semantics that *does* account for order. The key idea (due to Joachim Breitner) is this: return a sequence of *labelled* values; and then sort that sequence (in *one* and *all*) into canonical order before exposing it to the programmer.

**Domains**

$$
\begin{array}{rcll}
W^* & = & (\text{WRONG} + \mathcal{P}(LW))_\perp & \\
W^? & = & \{W\} & \text{Set with 0 or 1 elements} \\
LW & = & [L] \times W & \text{Sequence of } L \text{ and a value} \\
L & = & \mathbf{L} + \mathbf{R} &
\end{array}
$$

**Operations**

$$
\begin{array}{rrcl}
\text{Empty} & \mathit{empty} & : & W^* \\
 & \mathit{empty} & = & \varnothing \\
\text{Singleton} & \mathit{unit}(.) & : & W \to W^* \\
 & \mathit{unit}(w) & = & \{([], w)\} \\
\text{Union} & \Cup & : & W^* \to W^* \to W^* \\
 & s_1 \Cup s_2 & = & \{(\mathbf{L} : l, w) \mid (l, w) \in s_1\} \cup \{(\mathbf{R} : l, w) \mid (l, w) \in s_2\} \\
\text{Intersection} & \Cap & : & W^* \to W^* \to W^* \\
 & s_1 \Cap s_2 & = & \{(l_1 \bowtie l_2, w_1) \mid (l_1, w_1) \in s_1, (l_2, w_2) \in s_2, w_1 = w_2\} \\
\text{Sequencing} & \fatsemi & : & W^* \to W^* \to W^* \\
 & s_1 \fatsemi s_2 & = & \{(l_1 \bowtie l_2, w_2) \mid (l_1, w_1) \in s_1, (l_2, w_2) \in s_2\} \\
\text{One} & \mathit{one} & : & W^* \to W^* \\
 & \mathit{one}(s) & = & \mathit{head}(\mathit{sort}(s)) \\
\text{All} & \mathit{all} & : & W^* \to W^* \\
 & \mathit{all}(s) & = & \mathit{tuple}(\mathit{sort}(s)) \\
\text{Head} & \mathit{head} & : & [W] \to W^? \\
 & \mathit{head}[] & = & \mathit{empty} \\
 & \mathit{head}(w : s) & = & \mathit{unit}(w) \\
\text{To tuple} & \mathit{tuple} & : & [W] \to \langle W \rangle \\
 & \mathit{tuple}[w_1, \cdots, w_n] & = & \langle w_1, \cdots, w_n \rangle
\end{array}
$$

$$
\begin{array}{rrcll}
\text{Sort} & \mathit{sort} & : & LW^* \to ([W] + \text{WRONG})_\perp & \\
 & \mathit{sort}(s) & = & [] & \text{if } s \text{ is empty} \\
 & & = & \text{WRONG} & \text{if } ws \text{ has more than one element} \\
 & & = & ws & \text{otherwise} \\
 & & & \bowtie \mathit{sort}\{(l, w) \mid (\mathbf{L} : l, w) \in s\} & \\
 & & & \bowtie \mathit{sort}\{(l, w) \mid (\mathbf{R} : l, w) \in s\} & \\
 & & & \text{where } ws = [w \mid ([], w) \in s] &
\end{array}
$$

Fig. 9. Labelled set semantics for $W^*$

We do not change the equations for $\mathcal{E}$, $\mathcal{V}$, and $\mathcal{O}$ at all; they remain precisely as they are in Figure 7. However the semantics of a collection of values, $W^*$, does change, and is given in Figure 9:

- A collection of values $W^*$ is now $\perp$ or WRONG (as before), or a *set of labelled values*, each of type $LW$.

- A labelled value (of type $LW$) is just a pair $([L] \times W)$ of a *label* and a value.
- A label is a sequence of tags $L$, where a tag is just **L** or **R**.
- The union (or concatention) operation $\Cup$, defined in Fig. 9, adds a **L** tag to the labels of the values in the left branch of the choice, and a **R** tag to those coming from the right. So the labels specify where in the tree the value comes from.
- Sequencing $\fatsemi$ and $\Cap$ both concatenate the labels from the values they combine.
- Finally *sort* puts everything in the "right" order: first the values with an empty label, then the values whose label starts with **L** (notice the recursive sort of the trimmed-down sequence), and then those that start with **R**. Notice that *sort* removes all the labels, leaving just a bare sequence of values $W^*$.
- Note that if *sort* encounters a set with more than one unlabelled element then this considered WRONG. This makes ambiguous expressions, like **one**$\{\exists x.\, x\}$, WRONG.

Let us look at our troublesome example $\exists x.\, x = (4 \mid 3)$, and assume that $\exists$ binds $x$ to 3 and then 4. The meaning of this expression will be

$$\mathcal{E}[\![\exists x.\, x = (4 \mid 3)]\!]\, \epsilon \quad = \quad [(\mathbf{R}, 3), (\mathbf{L}, 4)]$$

Now if we take **all** of that expression we will get a singleton sequence containing $\langle 4, 3\rangle$, because **all** does a sort, stripping off all the tags.

$$\mathcal{E}[\![\mathbf{all}\{\exists x.\, x = (4 \mid 3)\}]\!]\, \epsilon \quad = \quad [([], \langle 4, 3\rangle)]$$

### C.5 Related work

[Christiansen et al. 2010] gives another approach to a denotational semantics for a functional logic language. We are keen to learn of others.