The Verse Calculus: A Core Calculus for Deterministic Functional Logic Programming

LENNART AUGUSTSSON, Epic Games, Sweden
JOACHIM BREITNER, Unaffiliated, Germany
KOEN CLAESSEN, Epic Games, Sweden
RANJIT JHALA, Epic Games, USA
SIMON PEYTON JONES, Epic Games, United Kingdom
OLIN SHIVERS, Epic Games, USA
GUY L. STEELE JR., Oracle Labs, 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 deterministic functional logic 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. We also show that the rewrite system is confluent for well-behaved terms.

CCS Concepts: • **Theory of computation** → **Equational logic and rewriting**; *Proof theory*; **Rewrite systems**; *Grammars and context-free languages*; • **Software and its engineering** → **Syntax**; **Semantics**; **Functional languages**; **Constraint and logic languages**; **Multiparadigm languages**.

Additional Key Words and Phrases: choice operator, confluence, declarative programming, evaluation strategy, even/odd problem, functional programming, lambda calculus, lenient evaluation, logic programming, logical variables, normal forms, rewrite rules, skew confluence, substitution, unification, Verse calculus, Verse language

ACM Reference Format:
Lennart Augustsson, Joachim Breitner, Koen Claessen, Ranjit Jhala, Simon Peyton Jones, Olin Shivers, Guy L. Steele Jr., and Tim Sweeney. 2023. The Verse Calculus: A Core Calculus for Deterministic Functional Logic Programming. *Proc. ACM Program. Lang.* 7, ICFP, Article 203 (August 2023), 31 pages. https://doi.org/10.1145/3607845

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:

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

Authors' addresses: Lennart Augustsson, Epic Games, Sweden, lennart.augustsson@epicgames.com; Joachim Breitner, Unaffiliated, Germany, 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; Guy L. Steele Jr., Oracle Labs, USA, guy.steele@oracle.com; Tim Sweeney, Epic Games, USA, tim.sweeney@epicgames.com.

2475-1421/2023/8-ART203
https://doi.org/10.1145/3607845

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 [Antoy and Hanus 2010]. But it is somewhat tricky for programmers to *reason* about functional logic programs: they must think about logical variables, needed narrowing, unification, and the like. This contrasts with functional programming, where one can say "just apply rewrite rules, such as β-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 two main contributions. **Our first contribution** is a new core calculus for functional logic programming, the Verse calculus or $\mathcal{VC}$ for short (Section 2). Like any functional logic language, $\mathcal{VC}$ supports logical variables, equalities, and choice, but it is distinctive in several ways:

- *Natively higher order.* $\mathcal{VC}$ directly supports higher-order functions, 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 defunctionalization [Reynolds 1972; Hanus 2013, 3.3].
- *Deterministic, with native encapsulation.* $\mathcal{VC}$ is *deterministic*, in the sense that when an expression yields more than one value (as is often the case in functional logic programs), those values are returned in a well-specified order. This makes it easy to solve the notoriously tricky issue [Braßel et al. 2004a,b] of how to *encapsulate* the result of a search as a data structure, using the **all** operator (see Section 2.6). It opens up a new approach to dealing with so-called "flexible" *vs.* "rigid" variables (see Section 2.5). It supports an elegant economy of concepts: for example, there is just one equality (other languages may have a suspending equality and a narrowing equality), and conditional expressions are driven by failure rather than booleans (see Section 2.5). On the other hand, it pretty much rules out *laziness* (see Section 3.6) and parallel first-come first-returned search strategies. Most other functional-logic languages (Curry [Hanus et al. 2016] is the brand leader in this design space) are non-deterministic by design; $\mathcal{VC}$ explores a different (and less well-examined) part of the design space.

Our second contribution is to equip $\mathcal{VC}$ with a *small-step term-rewriting semantics* (see 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 as for the lambda calculus. That seems challenging, however, because logical variables and unification involve sharing and non-local communication. How can that be expressed in a rewrite system?

Happily, we can build on prior work: exactly the same difficulty arises with call-by-need in the lambda calculus. 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, unification, and choice into the term itself, and replaces non-deterministic search with a (deterministic) tree of successful results. In $\mathcal{VC}$ the choices are "laid out in space" (in the syntax of the term) rather than, as is more typical, "laid out in time" (via non-deterministic rewrites and backtracking).

Integers	k
Variables	x, y, z, f, g
Programs	$p ::= \mathbf{one}\{e\}$ where $\mathrm{fvs}(e) = \emptyset$
Expressions	$e ::= v \mid eq;\ e \mid \exists x.\ e \mid \mathbf{fail} \mid e_1 \mathbin{\|} e_2 \mid v_1\ v_2 \mid \mathbf{one}\{e\} \mid \mathbf{all}\{e\}$
	$eq ::= e \mid v = e$ Note: "eq" is pronounced "expression or equation"
Values	$v ::= x \mid hnf$
Head values	$hnf ::= k \mid op \mid \langle v_1, \cdots, v_n\rangle \mid \lambda x.\ e$
Primops	$op ::= \mathbf{gt} \mid \mathbf{add}$

Concrete syntax: "$\|$" and ";" are right-associative.
"=" binds more tightly than ";".
"λ" and "$\exists$" each scope as far to the right as possible.
For example, $(\lambda y.\ \exists x.\ x = 1;\ x + y)$ means $(\lambda y.\ (\exists x.\ ((x = 1);\ (x + y))))$.

Parentheses may be used freely to aid readability and override default precedence.
$\mathrm{fvs}(e)$ means the free variables of e; in $\mathcal{VC}$, λ and $\exists$ are the only binders.

Desugaring of extended expressions

$e_1 + e_2$	means	$\mathbf{add}\langle e_1, e_2\rangle$	
$e_1 > e_2$	means	$\mathbf{gt}\langle e_1, e_2\rangle$	
$\exists x_1\, x_2 \cdots x_n.\ e$	means	$\exists x_1.\ \exists x_2. \cdots \exists x_n.\ e$	
$x := e_1;\ e_2$	means	$\exists x.\ x = e_1;\ e_2$	
$e_1\ e_2$	means[†]	$f := e_1;\ x := e_2;\ f\ x$	f, x fresh
$\langle e_1, \cdots, e_n\rangle$	means[†]	$x_1 := e_1;\ \cdots;\ x_n := e_n;\ \langle x_1, \cdots, x_n\rangle$	x_i fresh
$e_1 = e_2$	means[‡]	$x := e_1;\ x = e_2;\ x$	x fresh
$\lambda\langle x_1, \cdots, x_n\rangle.\ e$	means	$\lambda p.\ \exists x_1 \cdots x_n.\ p = \langle x_1, \cdots, x_n\rangle;\ e$	p fresh, $n \geqslant 0$
$\mathbf{if}\ (\exists x_1 \cdots x_n.\ e_1)\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3$	means	$(\mathbf{one}\{(\exists x_1 \cdots x_n.\ e_1;\ \lambda\langle\rangle.\ e_2) \mathbin{\|} (\lambda\langle\rangle.\ e_3)\})\langle\rangle$	

[†]Apply this rule only if at least one of the e_i is not a value v.
[‡]Apply this rule only if either (i) e_1 is not a value v, or (ii) $e_1 = e_2$ is not to the left of a ";".

Fig. 1. $\mathcal{VC}$: Syntax

As an example of rewriting in action, the expression above can be rewritten as follows[1]:

$$
\begin{array}{llll}
 & \exists x\, y\, z.\ x = \langle y, 3\rangle;\ x = \langle 2, z\rangle;\ y & \longrightarrow\{\textsc{subst}\} & \exists x\, y\, z.\ \langle 2, z\rangle = \langle y, 3\rangle;\ x = \langle 2, z\rangle;\ y \\
\longrightarrow\{\textsc{eqn-elim}\} & \exists y\, z.\ \langle 2, z\rangle = \langle y, 3\rangle;\ y & \longrightarrow\{\textsc{u-tup}\} & \exists y\, z.\ 2 = y;\ z = 3;\ y \\
\longrightarrow\{\textsc{eqn-elim}\} & \exists y.\ 2 = y;\ y & \longrightarrow\{\textsc{hnf-swap}\} & \exists y.\ y = 2;\ y \\
\longrightarrow\{\textsc{subst}\} & \exists y.\ y = 2;\ 2 & \longrightarrow\{\textsc{eqn-elim}\} & 2
\end{array}
$$

Rules may be applied anywhere they match, including under binders, again just like the lambda calculus. This freedom only makes sense, however, if each term ultimately reduces to a unique value, regardless of its reduction path, so we show that $\mathcal{VC}$ is *confluent*, in Section 4.

As always with a calculus, the idea is that $\mathcal{VC}$ distills the essence of (deterministic) functional logic programming. Each construct does just one thing, and $\mathcal{VC}$ cannot be made smaller without losing key features. We are 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 design alternatives in Section 5 and related work in Section 6.

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

2 THE VERSE CALCULUS, INFORMALLY

We begin by presenting the Verse calculus, $\mathcal{VC}$, informally. We describe its rewrite rules precisely in Section 3. The (abstract) 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 variable x or a head-normal form *hnf*. 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. A head-normal form is a conventional value: a built-in constant k, an operator *op*, a tuple, or a lambda. Our tiny calculus offers only integer constants k and two illustrative integer operators *op*, namely **gt** and **add**.
- *Expressions e* include values v, and applications $v_1\ v_2$; we will introduce the other constructs as we go. For clarity, we often write $v_1(v_2)$ rather than $v_1\ v_2$ when v_2 is not a tuple.
- A term *eq* is either an ordinary expression e, or an *equation* $v = e$; this syntax ensures that equations can only occur to the left of a "; " (Section 2.1).
- *A program*, p, contains a closed expression from which we extract one result using **one** (see Section 2.5)—unless the expression fails, in which case the program fails (Section 2.2).

The formal syntax for e allows only applications of *values* to *values*, $(v_1\ v_2)$, but the desugaring rules in Fig. 1 show how to desugar more applications $(e_1\ e_2)$. This restriction is not fundamental; it simply reduces the number of rewrite rules we need[2].

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 with the use of equations to perform unification (Section 2.1), and choice (Section 2.2). The details of choice and unification, and especially their interaction, are subtle, so this section does a lot of arm-waving. But fear not: Section 3 spells out the precise details. We only have space to describe one incarnation of $\mathcal{VC}$; Section 5 explores some possible alternative design choices.

2.1 Logical Variables and Equations

The Verse calculus includes first-class *logical variables* and *equations* that constrain their values. You can bring a fresh logical variable into scope with $\exists$, constrain a value to be equal to an expression with an equation $v = e$, and compose expressions in sequence with $eq;\ e$ (see Fig. 1). As an example, what might be written $\textbf{let}\ x\ =\ e_1\ \textbf{in}\, e_2$ in a conventional functional language can be written $\exists x.\ x = e_1;\ e_2$ in $\mathcal{VC}$. The syntax carefully constrains both the form of equations and where they can appear: an equation $(v = e)$ always equates a *value* v to an expression e; and an equation can appear only to the left of a "; " (see *eq* in Fig. 1). The desugaring rules in Fig. 1 rewrite a general equation $e_1 = e_2$ (where e_1 is not a value) into equations of this simpler form.

A program executes by solving its equations, using the process of unification. 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, as in 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 …".

High-level functional languages usually provide some kind of pattern matching; in such a language, we might define *first* by $\mathit{first}\langle a, b\rangle = a$. Such pattern matching is typically desugared to

[2]This is a common pattern, often called "administrative normal form", or ANF [Sabry and Felleisen 1992]

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:

$$\mathit{first} := \lambda p.\ \exists a\, b.\ p = \langle a, b\rangle;\ a$$

For convenience, we allow ourselves to write a term like $\mathit{first}\langle 2, 5\rangle$, where we define the library function *first* separately with ":=". Formally, you can imagine each example e being wrapped with a binding for *first*, thus $\exists \mathit{first}.\ \mathit{first} = \ldots;\ e$, and similarly for 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:

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

Here $\mathit{first}(x)$ 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 $2 = \mathit{first}(x)$), and thereby affect its *input* (here $\langle y, 5\rangle$).

Although ";" is called "sequencing," the order of that sequence is immaterial for equations that do not contain choices (see Section 2.2 for the latter caveat). For example, consider $(\exists x\, y.\ x = 3 + y;\ y = 7;\ x)$. The sub-expression $3 + y$ is stuck until y gets a value. In $\mathcal{VC}$, we can unify x only with a *value*—we will see why in Section 2.2—and hence the equation $x = 3 + y$ is also 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][3], 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 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 followed by all the values yielded by e_2. Order is maintained and duplicates are not eliminated; we shall see why in Section 2.8. 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, 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 use the desugaring rules of Fig. 1, which give a name to that choice, thus: $\exists x.\ x = (7 \mid 5);\ \langle 3, x\rangle$. Now the expression 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". The same thing happens when there are multiple choices. For example:

$$\exists x\, y.\ x = (7 \mid 22);\ y = (31 \mid 5);\ \langle x, y\rangle \quad \text{yields the sequence } \langle 7, 31\rangle, \langle 7, 5\rangle, \langle 22, 31\rangle, \langle 22, 5\rangle$$

Notice that the order of the two equations now *is* significant:

$$\exists x\, y.\ y = (31 \mid 5);\ x = (7 \mid 22);\ \langle x, y\rangle \quad \text{yields the sequence } \langle 7, 31\rangle, \langle 22, 31\rangle, \langle 7, 5\rangle, \langle 22, 5\rangle$$

[3]Hanus did not invent the terms "residuation" and"narrowing," but his survey is an excellent introduction and bibliography.

Readers familiar with list comprehensions in Haskell and other languages will recognize this nested-loop pattern, but here it emerges naturally from choice as a deeply built-in primitive, rather than being a special construct for lists.

Just as we never bind a variable to a multi-value, we never bind it to **fail** either; rather we iterate over zero values, and that iteration of course returns zero values. So:

$$\exists x.\, x = \mathbf{fail};\ 33 \quad \longrightarrow \quad \mathbf{fail}$$

2.3 Mixing Choice and Equations

In the last section, we discussed what happens if there is a choice in the right-hand side (RHS) of an equation. What if we have equations under choice? For example:

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

Intuitively, "either unify x with 3 and yield $x + 1$, or unify x with 4 and yield $x + 4$". But there is a problem: so far we have said only "a program executes by solving its equations" (Section 2.1). Here, we can see two equations, $(x = 3)$ 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 values 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:

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

Are we stuck now? No, we can float the choice out as before[4],

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

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 equations. But what about multi-equation pattern matching, such as this definition in Haskell:

$\mathbf{append}\ [\,]\qquad ys = ys$
$\mathbf{append}\ (x : xs)\ ys = x : \mathbf{append}\ xs\ ys$

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, where we use the empty tuple $\langle\rangle$ to represent the empty list and pairs to represent cons cells (see Fig. 1 to desugar the pattern-matching lambda):

$$\mathbf{append} := \lambda\langle xs, ys\rangle.\, ((xs = \langle\rangle;\ ys) \mid (\exists x\, xr.\, xs = \langle x, xr\rangle;\ \langle x, \mathbf{append}\langle xr, ys\rangle\rangle))$$

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 $\langle x, xr\rangle$). If xs is of the form $\langle x, xr\rangle$, 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 $\exists zs.\, \mathbf{append}\langle zs, single\rangle = single;\ zs$. The call to **append** expands into a choice:

$$(zs = \langle\rangle;\ single) \quad \mid \quad (\exists x\, xr.\, zs = \langle x, xr\rangle;\ \langle x, \mathbf{append}\langle xr, single\rangle\rangle)$$

[4]Indeed, we could have done so first, had we wished.

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 equation (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 toward 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 [Antoy and Hanus 2021, §3.4], 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, following the lead of Icon (see Section 6.7). 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 (yields one or more values), the **if** returns e_2; otherwise it returns e_3. There are no data constructors *True* and *False*; instead failure (returning zero values) plays the role of falsity.

But something is terribly wrong here. Consider $\exists x\, y.\; y = (\textbf{if}\ (x=0)\ \textbf{then}\ 3\ \textbf{else}\ 4);\ x = 7;\ y$. Presumably this is meant to set x to 7, test whether 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, equations 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** is stuck, and we move on to unify $x=7$. That unblocks the **if** and all is well.

In fact, $\mathcal{VC}$ desugars 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:

$$\textbf{one}\{(e_1;\ e_2)\ \|\ e_3\}$$

This isn't right yet, but the idea is this: 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 what if e_2 or e_3 *themselves* fail or return multiple results? Here is a better translation, the one given in Fig. 1[5], which wraps the **then** and **else** branches within thunks[6]:

$$(\textbf{one}\{(e_1;\ (\lambda\langle\rangle.\ e_2))\ \|\ (\lambda\langle\rangle.\ e_3)\})\langle\rangle$$

The argument of **one** reduces to either $(\lambda\langle\rangle.\ e_2)\ \|\ (\lambda\langle\rangle.\ e_3)$ or $(\lambda\langle\rangle.\ e_3)$ depending on whether e_1 succeeds or fails, respectively; **one** then picks the first value, that is, $\lambda\langle\rangle.\ e_2$ if e_1 succeeded or $\lambda\langle\rangle.\ e_3$ if e_1 failed, 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 unevaluated until the choice is made, just as we expect from a conditional.

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

[5]The translation in the figure also allows variables bound in the condition to scope over the **then** branch.
[6]Using thunks for the branches of a conditional is another very old idea; for example, see [Steele Jr. 1978, p. 54].

2.6 Tuples and all

The main data structure in $\mathcal{VC}$ is the *tuple*. A tuple is a finite sequence of values, $\langle v_1, \cdots, v_n \rangle$, where $n \geqslant 0$. A tuple 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**, as is indexing with a non-integer value. For example, $t := \langle 10, 27, 32 \rangle;\ t(1)$ reduces to 27 and $t := \langle 10, 27, 32 \rangle;\ t(3)$ reduces to **fail**.

What if we apply a tuple to a choice, *e.g.*, $\langle 10, 27, 32 \rangle (1 \mid 0 \mid 1)$? First we must desugar the application to the form $(v_1\ v_2)$, which is all $\mathcal{VC}$ permits (see Fig. 1), giving $x := (1 \mid 0 \mid 1);\ \langle 10, 27, 32 \rangle (x)$, which readily 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 $(v_1 \mid \cdots \mid v_n)$, where $n \geqslant 2$, then $\mathbf{all}\{e\}$ reduces to the tuple $\langle v_1, \cdots, v_n \rangle$; $\mathbf{all}\{v\}$ produces the singleton tuple $\langle v \rangle$; and $\mathbf{all}\{\mathbf{fail}\}$ produces the empty tuple $\langle \rangle$. 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 in $\mathcal{VC}$, the application of a tuple to a value rewrites to a choice of all the possible values of the index. For example, $t := \langle 10, 27, 32 \rangle;\ \exists i.\ t(i)$ looks stuck because we have no value for i, but this expression actually rewrites (via rule APP-TUP in Section 3.1) 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. The idea of rewriting a call of a function with a finite domain into a finite choice is called "narrowing" in the literature.

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

2.7 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 "$\mid$" serves as both *cons* and **append** for choices; the corresponding operations for tuples are defined in Fig. 2. Partial functions, *e.g.*, *head*, will **fail** when the argument is outside of the domain.

Mapping a multi-valued function over a tuple is somewhat subtle. With *flatMap* the choices are flattened in the resulting tuple, *e.g.*, $\mathit{flatMap}\langle(\lambda x.\ x \mid x + 10), \langle 2, 3 \rangle\rangle$ reduces to $\langle 2, 12, 3, 13 \rangle$, whereas *map* keeps the choices. For example:

$$\begin{aligned} &\mathit{map}\langle(\lambda x.\ x \mid x + 10), \langle 2, 3 \rangle\rangle \longrightarrow \langle(\lambda x.\ x \mid x + 10)(2), (\lambda x.\ x \mid x + 10)(3)\rangle \longrightarrow \\ &\langle 2 \mid 12, 3 \mid 13 \rangle \longrightarrow \langle 2, 3 \rangle \mid \langle 2, 13 \rangle \mid \langle 12, 3 \rangle \mid \langle 12, 13 \rangle \end{aligned}$$

Pattern matching for function definitions is simply done by unification of ordinary expressions; see the desugaring of pattern-matching lambda in Fig. 1. This in turn means that we can use ordinary abstraction mechanisms for patterns. For example, here is a function, *fcn*, that could be called as follows: $\mathit{fcn}\langle 88, 1, 99, 2 \rangle$.

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

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

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

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

Desugaring of function definitions

$$
\begin{array}{rcl}
f(x) := e & \text{means} & f := \lambda x.\, e \\
f\langle x, y\rangle := e & \text{means} & f := \lambda\langle x, y\rangle.\, e
\end{array}
$$

Fig. 2. Functions on tuples, analogous to list or array functions in some other languages

Patterns are truly first-class, going well beyond what can be done with, say, pattern synonyms in Haskell. For example, *pat* could be *computed*, like this:

$$\mathit{pat}\langle a, v, w\rangle := \mathbf{if}\ a = 0\ \mathbf{then}\ \langle v, 1, w, 2\rangle\ \mathbf{else}\ \langle 1, 1, w, v\rangle$$

so that the pattern depends on the value of a.

2.8 For Loops

The expression $\mathbf{for}(e_1)\ \mathbf{do}\ e_2$ 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[7] in that variables bound in e_1 also scope over e_2. So, for example, $\mathbf{for}(x := (2 \mid 3 \mid 5))\ \mathbf{do}\ (x + 1)$ will reduce to the tuple $\langle 3, 4, 6\rangle$.

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, $\mathbf{for}(\exists x\, y.\ x = (10 \mid 20);\ y = (1 \mid 2 \mid 3))\ \mathbf{do}\ (x + y)$ 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 $\mathbf{all}\{e_1;\ e_2\}$ would give the wrong semantics. So we put e_2 within a lambda expression, and apply each element of the tuple to $\langle\rangle$ afterwards, using a *map* function (as defined in Fig. 2):

$$\mathbf{for}(\exists x_1 \cdots x_n.\, e_1)\ \mathbf{do}\ e_2 \quad \text{means} \quad v := \mathbf{all}\{\exists x_1 \cdots x_n.\, e_1;\ \lambda\langle\rangle.\, e_2\};\ \mathit{map}\langle \lambda z.\, z\langle\rangle, v\rangle$$

for a fresh variable v. Note how this achieves that peculiar scoping rule: the initial variables in $\exists x_1 \cdots x_n.\, e_1$ are in scope in e_2. Moreover, any effects (like being multi-valued) in e_2 will not affect the choices defined by e_1 since the effects are contained within that lambda. So, for example, $\mathbf{for}(x := (10 \mid 20))\ \mathbf{do}\ (x \mid x + 1)$ 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 that the desugaring of **for** uses *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, each increased by its index within t. Notice the absence of the fencepost-error-prone iteration of i over $(0\,..\,\mathit{size}\ (t) - 1)$, common in most languages.

[7] But similar to C++, Java, Fortress, and Swift, and explained in $\mathcal{VC}$ by the subsequent desugaring into **all**.

3 REWRITE RULES

How can we give a precise semantics to a 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 E of Augustsson et al. [2023], 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. 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.
- *A small-step operational semantics* addresses this criticism: it describes how an abstract machine state (typically a (heap, expression, stack) triple) 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 a middle path between the mathematical abstraction of denotational semantics and the over-concrete details of an operational semantics, whether big or small step. 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 easily explicable to programmers. In fact, teachers almost always explain the execution of Haskell or ML programs as a succession of rewrites of the program, such as: 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 6 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). These rules apply to the core language of Fig. 1, after all desugaring.
- Any rule can be applied, in either direction, anywhere in the program term (including under lambdas).
- The rules are (mostly) 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 optimize) the program.
- There is no "magical rewriting" (Section 6.3): all the free variables on the right-hand side of a rule are bound on the left.

3.1 Functions and Function Application Rules

Looking at Fig. 3, the rule for a primitive operator like addition, 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).

An application of a primitive operator can rewrite only when its arguments are ground values; an application like $\mathbf{gt}\langle x, 3\rangle$ or $\mathbf{add}\langle 7, x\rangle$ is *stuck* awaiting a value for x, which may arrive later, by substitution (Section 3.2). An ill-typed application, such as $\mathbf{gt}\langle 3, \lambda x.\, x\rangle$, is simply stuck forever.

Application:

APP-ADD	$\mathbf{add}\langle k_1, k_2\rangle \longrightarrow k_3$	where $k_3 = k_1 + k_2$
APP-GT	$\mathbf{gt}\langle k_1, k_2\rangle \longrightarrow k_1$	if $k_1 > k_2$
APP-GT-FAIL	$\mathbf{gt}\langle k_1, k_2\rangle \longrightarrow \mathbf{fail}$	if $k_1 \leqslant k_2$
APP-BETA$^\alpha$	$(\lambda x.\, e)(v) \longrightarrow \exists x.\, x = v;\ e$	if $x \notin \mathrm{fvs}(v)$
APP-TUP	$\langle v_0, \cdots, v_n\rangle(v) \longrightarrow \exists x.\, x = v;\ (x = 0;\ v_0) \mid \cdots \mid (x = n;\ v_n)$	fresh $x \notin \mathrm{fvs}(v, v_0, \cdots, v_n)$
APP-TUP-0	$\langle\rangle(v) \longrightarrow \mathbf{fail}$	

Unification:

U-LIT	$k_1 = k_2;\ e \longrightarrow e$	if $k_1 = k_2$
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	$hnf_1 = hnf_2;\ e \longrightarrow \mathbf{fail}$	if U-LIT, U-TUP do not match and neither hnf_1 nor hnf_2 is a lambda
U-OCCURS	$x = V[x];\ e \longrightarrow \mathbf{fail}$	if $V \neq \square$
SUBST	$X[x = v;\ e] \longrightarrow (X\{v/x\})[x = v;\ e\{v/x\}]$	if $v \neq V[x]$
HNF-SWAP	$hnf = v;\ e \longrightarrow v = hnf;\ e$	
VAR-SWAP	$y = x;\ e \longrightarrow x = y;\ e$	if $x \prec y$
SEQ-SWAP	$eq;\ x = v;\ e \longrightarrow x = v;\ eq;\ e$	unless (eq is $y = v'$ and $y \preceq x$)

Elimination:

VAL-ELIM	$v;\ e \longrightarrow e$	
EXI-ELIM	$\exists x.\, e \longrightarrow e$	if $x \notin \mathrm{fvs}(e)$
EQN-ELIM	$\exists x.\, X[x = v;\ e] \longrightarrow X[e]$	if $x \notin \mathrm{fvs}(X[e])$ and $v \neq V[x]$
FAIL-ELIM	$X[\mathbf{fail}] \longrightarrow \mathbf{fail}$	

Normalization:

EXI-FLOAT$^\alpha$	$X[\exists x.\, e] \longrightarrow \exists x.\, X[e]$	if $x \notin \mathrm{fvs}(X)$
SEQ-ASSOC	$(eq;\ e_1);\ e_2 \longrightarrow eq;\ (e_1;\ e_2)$	
EQN-FLOAT	$v = (eq;\ e_1);\ e_2 \longrightarrow eq;\ (v = e_1;\ e_2)$	
EXI-SWAP	$\exists x.\, \exists y.\, e \longrightarrow \exists y.\, \exists x.\, e$	

Choice:

ONE-FAIL	$\mathbf{one}\{\mathbf{fail}\} \longrightarrow \mathbf{fail}$
ONE-VALUE	$\mathbf{one}\{v\} \longrightarrow v$
ONE-CHOICE	$\mathbf{one}\{v \mid e\} \longrightarrow v$
ALL-FAIL	$\mathbf{all}\{\mathbf{fail}\} \longrightarrow \langle\rangle$
ALL-VALUE	$\mathbf{all}\{v\} \longrightarrow \langle v\rangle$
ALL-CHOICE	$\mathbf{all}\{v_1 \mid \cdots \mid v_n\} \longrightarrow \langle v_1, \cdots, v_n\rangle$
CHOOSE-R	$\mathbf{fail} \mid e \longrightarrow e$
CHOOSE-L	$e \mid \mathbf{fail} \longrightarrow e$
CHOOSE-ASSOC	$(e_1 \mid e_2) \mid e_3 \longrightarrow e_1 \mid (e_2 \mid e_3)$
CHOOSE	$SX[CX[e_1 \mid e_2]] \longrightarrow SX[CX[e_1] \mid CX[e_2]]$

Note: In the rules marked with a superscript α, use α-conversion to satisfy the side condition.

Fig. 3. The Verse Calculus: Rewrite rules

Execution contexts	$X ::= \square \mid v = X;\ e \mid X;\ e \mid eq;\ X$
Value contexts	$V ::= \square \mid \langle v_1, \cdots, V, \cdots, v_n \rangle$
Scope contexts	$SX ::= \mathbf{one}\{SC\} \mid \mathbf{all}\{SC\}$
	$SC ::= \square \mid SC \,\|\, e \mid e \,\|\, SC$
Choice contexts	$CX ::= \square \mid v = CX;\ e \mid CX;\ e \mid ceq;\ CX \mid \exists x.\, CX$
Choice-free exprs	$ce ::= v \mid ceq;\ ce \mid \mathbf{one}\{e\} \mid \mathbf{all}\{e\} \mid \exists x.\, ce \mid op(v)$
	$ceq ::= ce \mid v = ce$

Note: The $\square$ in X can only be an expression, not an equation.

Fig. 4. The syntax of contexts

Note that equality, written $v = e$, is not a primitive operator; it is a language construct, and enjoys a rich set of built-in rewrite rules (see Section 3.2).

β-reduction is performed quite conventionally by APP-BETA; the only unusual feature is that on the RHS of the rule, we use an $\exists$ 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, and a variable 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, α-conversion may be used on $\lambda x.\, e$ to rename x to $y \notin \mathrm{fvs}(v)$.

In $\mathcal{VC}$, tuples behave like (finite) functions in which application is indexing. Rule APP-TUP describes how tuple application works on non-empty tuples, while APP-TUP-0 deals with empty tuples. 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 (a form of narrowing), returning $(0 \,\|\, 2)$. This is a free design decision: a suspending semantics would be equally easy to express.

3.2 Unification Rules

Next we study unification, again in Fig. 3. Rules U-LIT and U-TUP are the standard rules for unification, going back nearly 60 years [Robinson 1965]. Rule U-FAIL makes unification fail (return zero results) on two different head-normal forms (see Fig. 1 for the syntax of *hnf*), except that it gets stuck if you attempt to unify a lambda with any other value, including itself. Why? Because equality of functions is undecidable, so in $\mathcal{VC}$ we simply refuse to run a program that tries to do so, just as we refuse to run $\mathbf{gt}\langle 3, \lambda x.\, x \rangle$ (see Section 3.1). This choice has consequences for confluence: see Section 4.1.1.

The standard "occurs check" is rule U-OCCURS, which makes use of a *context* V, whose syntax is given in Fig. 4. In general, a context [Felleisen and Friedman 1986; Felleisen et al. 1987] is a syntax tree containing a single hole, written $\square$. The notation $V[\,v\,]$ is the term obtained by filling the hole in V with v. For example, U-OCCURS reduces $x = \langle 1, x, 3 \rangle;\ e$ to **fail** using the context $V = \langle 1, \square, 3 \rangle$.

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

$$\textsc{subst}\quad X[\,x = v;\ e\,] \ \longrightarrow\ (X\{v/x\})[\,x = v;\ e\{v/x\}\,] \quad \text{if } v \neq V[\,x\,]$$

The rule says that if we have an equation $(x = v)$, we can replace the occurrences of x by v within the following expression and also within a surrounding context. This rule uses context X (Fig. 4), and uses the notation $e\{v/x\}$ to mean "capture-avoiding substitution of v for x in e" (and similarly $X\{v/x\}$ to mean "capture-avoiding substitution of v for x in X"). There are several things to notice:

- SUBST fires only when the right-hand side of the equation is a *value* v, 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.
- SUBST replaces all occurrences of x in X and e, but *it leaves the original* $(x = v)$ *undisturbed*, because X might not be big enough to encompass all occurrences of x. For example, we can rewrite $(y = x + 1;\ (x = 3;\ x + 3))$ to $(y = x + 1;\ (x = 3;\ 3 + 3))$, using $X = \square$, where the $\square$ is $x = 3;\ x + 3$. But that leaves an occurrence of x in $(y = x + 1)$. When there are no remaining occurrences of x, we may eliminate the binding by using rule EQN-ELIM (see Section 3.4).
- The side condition $v \neq V[x]$ in SUBST avoids an overlap with U-OCCURS.

3.3 Swapping and Binding Order

Two rules allow the two sides of an equation to be swapped. Rule HNF-SWAP helps SUBST to fire by putting the variable on the left. Rule VAR-SWAP, which swaps two variables, is needed for a more subtle reason. Consider the following example expression, where a and b are bound within some containing context, perhaps by lambdas. It can be rewritten in two different ways, where each column is a reduction sequence starting from the same initial term:

$$\exists x.\ x = \langle a\rangle;\ x = \langle b\rangle;\ x$$

$$\begin{array}{ll|ll}
\longrightarrow\{\text{SUBST}\} & \exists x.\ x = \langle a\rangle;\ \langle a\rangle = \langle b\rangle;\ \langle a\rangle & \longrightarrow\{\text{SUBST}\} & \exists x.\ \langle b\rangle = \langle a\rangle;\ x = \langle b\rangle;\ \langle b\rangle \\
\longrightarrow\{\text{U-TUP}\} & \exists x.\ x = \langle a\rangle;\ a = b;\ \langle a\rangle & \longrightarrow\{\text{U-TUP}\} & \exists x.\ b = a;\ x = \langle b\rangle;\ \langle b\rangle \\
\longrightarrow\{\text{EQN-ELIM}\} & a = b;\ \langle a\rangle & \longrightarrow\{\text{EQN-ELIM}\} & b = a;\ \langle b\rangle \\
\longrightarrow\{\text{SUBST}\} & a = b;\ \langle b\rangle & \longrightarrow\{\text{SUBST}\} & b = a;\ \langle a\rangle
\end{array}$$

The two sequences differ when it comes to which equation for x is chosen for SUBST in the first step of each column. As you can see, they conclude with two terms that are "obviously" the same semantically, but which are syntactically different. Rule VAR-SWAP allows them to be brought together (for example, $a = b;\ \langle b\rangle \ \longrightarrow\{\text{VAR-SWAP}\}\ b = a;\ \langle b\rangle \ \longrightarrow\{\text{SUBST}\}\ b = a;\ \langle a\rangle$), so that the unification rules can be syntactically confluent.

Rule SEQ-SWAP, which swaps adjacent equations within a sequence under certain conditions, is needed for a similar reason. Consider this example:

$$c = a;\ c = b;\ c$$

$$\begin{array}{ll|ll}
\longrightarrow\{\text{SUBST}\} & c = a;\ a = b;\ a & \longrightarrow\{\text{SUBST}\} & b = a;\ c = b;\ b \\
\longrightarrow\{\text{VAR-SWAP}\} & c = a;\ b = a;\ a & \longrightarrow\{\text{SUBST}\} & b = a;\ c = a;\ a
\end{array}$$

Again, the concluding terms of the two columns are "obviously" the same (they differ only in the order of the equations $b = a$ and $c = a$); SEQ-SWAP allows one term to be rewritten to match the other, making explicit our intuition that the order of equations of the form $x = v$ should not matter.

Observe the mysterious side condition $x \prec y$ in rule VAR-SWAP, and a similar one in SEQ-SWAP. In the overall proof of confluence, it turns out to be very helpful if the unification rules are *terminating* (see Section 4.2). To achieve this, VAR-SWAP fires on $y = x$ *only if* x*'s binding occurs in the scope of* y*'s binding*, written $x \prec y$, so that the innermost-bound variable ends up on the left. Similarly, the side condition on SEQ-SWAP prevents it firing infinitely.

Other rules, notably EXI-SWAP, may change this binding order and thereby re-enable VAR-SWAP or SEQ-SWAP, but the unification rules *considered in isolation* are terminating and confluent, and that is what we need for the proof (but see Section 5.1).

3.4 Elimination and Normalization Rules

Four *elimination* rules allow dead code to be dropped (Fig. 3): VAL-ELIM discards a value to the left of a semicolon; EXI-ELIM discards a dead existential; EQN-ELIM discards an existential $\exists x$ that binds a variable whose only occurrence is a single equation $x = v$; and FAIL-ELIM discards the execution context surrounding a **fail**. Note that none of these rules, except FAIL-ELIM, discards an unevaluated expression, because that expression might fail and we don't want to "lose" that failure (see Section 3.6). The exception is FAIL-ELIM, whose purpose is to propagate a known failure, so losing some other failure that might have been in the discarded execution context does not matter.

Four *normalization* rules help to put the expression in a form that allows other rules to fire (Fig. 3): EXI-FLOAT allows an existential to float outwards; SEQ-ASSOC makes semicolon right-associated; EQN-FLOAT moves work out of the right-hand side of an equation $v = e$. For example, we cannot use SUBST to substitute for x in $(x = (e;\ 3);\ x + 2)$, because the RHS of the x-equation is not a value; but we can instead apply EQN-FLOAT to get $e;\ x = 3;\ x + 2$, and now we *can* apply SUBST.

Rule EXI-SWAP allows you to move an existential inward so that a dead equation can be eliminated by EQN-ELIM. Rule EXI-SWAP is unusual because it can be infinitely applied; avoiding that eventuality is easily achieved by tweaking the evaluation strategy (Section 3.7).

Note that all these swapping and 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, the rules do not float equalities or existentials out of choices: that restriction is the key to localizing unification (Section 2.3) and to the flexible/rigid distinction of Section 2.5. For example, consider the expression $(y = ((x = 3;\ x + 5) \mid (x = 4;\ x + 2));\ \langle x + 1, y \rangle)$. We must not float the binding $(x = 3)$ up to a point where it might interact with the expression $(x + 1)$, because the latter is outside the choice, and a different branch of the choice binds x to 4.

3.5 Rules for Choice

The rules for choice are given in Fig. 3:

- Rules ONE-FAIL, ONE-VALUE, and ONE-CHOICE describe the semantics of **one**, as in Section 2.5.
- Similarly, ALL-FAIL, ALL-VALUE, and ALL-CHOICE describe the semantics of **all** (Section 2.6).
- Rules CHOOSE-R and CHOOSE-L eliminate **fail**, which behaves as an identity for choice.
- Rule CHOOSE-ASSOC associates choice to the right, so that ONE-CHOICE or ALL-CHOICE can fire. (The dots on the left of ALL-CHOICE should be read as a string of right-associated choices.)

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. 4. A *choice context* CX is like an execution context X, *but with no possible choices to the left of the hole*:

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

Here, ce is a guaranteed-choice-free expression (syntax in Fig. 4). 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 the order of choices—see Section 2.3.

The context SX (Fig. 4) in CHOOSE ensures that CX is as large as possible. This is a very subtle point: without this restriction we lose confluence. To see this, consider[8]:

$$\begin{array}{ll} & \exists x.\ (\textbf{if}\ (x > 0)\ \textbf{then}\ 55\ \textbf{else}\ (44 \mid 2));\ x = 1;\ (77 \mid 99) \\ \longrightarrow\{\textsc{subst}\} & \exists x.\ (\textbf{if}\ (1 > 0)\ \textbf{then}\ 55\ \textbf{else}\ (44 \mid 2));\ x = 1;\ (77 \mid 99) \\ \longrightarrow\{\text{simplify}\ \textbf{if}\} & \exists x.\ 55;\ x = 1;\ (77 \mid 99) \quad \longrightarrow\{\textsc{val-elim}, \textsc{eqn-elim}\}\ \ 77 \mid 99 \end{array}$$

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

But suppose instead we floated the choice out *partway*, like this[9]:

$$\begin{array}{l}\exists x.\,(\textbf{if}\ (x>0)\ \textbf{then}\ 55\ \textbf{else}\ (44\,\|\,2));\ x{=}1;\ (77\,\|\,99)\\ \quad\longrightarrow\{\text{Bogus }\textsc{choose}\}\quad \exists x.\,(\textbf{if}\ (x>0)\ \textbf{then}\ 55\ \textbf{else}\ (44\,\|\,2));\ ((x{=}1;\ 77)\,\|\,(x{=}1;\ 99))\end{array}$$

Now the $(x=1)$ is inside the choice branches, so we cannot use SUBST to substitute for x in the condition of the **if**. Nor can we use CHOOSE again to float the choice further out because the **if** is not guaranteed choice-free (in this example, the **else** branch has a choice). So, alas, we are stuck! Our not-entirely-satisfying solution is to force CHOOSE to float the choice all the way to the top. The *SX* context (Fig. 4) formalizes what we mean by "the top": rule CHOOSE can float a choice outward only when it becomes part of the choice tree (context *SC*) immediately under a **one** or **all** construct (context *SX*).

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. This contrasts with other semantic approaches that eliminate choice by non-deterministically picking one branch or the other, which immediately gives up confluence. However, in implementation terms, treating choice a bit like a data constructor is the basis for *pull-tabbing* in logically-complete implementations of non-deterministic functional logic languages like Curry [Antoy 2011].

3.6 The Verse Calculus Is Lenient

$\mathcal{VC}$ is *lenient* [Schauser and Goldstein 1995], not lazy (call-by-need), nor strict (call-by-value). Under lenient evaluation, functions can run before their arguments have a ground value (as in a lazy language), but (almost) everything is eventually evaluated (as in a strict language).

$\mathcal{VC}$ cannot be lazy, because of $\mathcal{VC}$'s commitment to determinism and, particularly, its first-class **all** operator. Consider:

$$\textbf{all}\{\exists x.\ x = e;\ 3\}$$

In a lazy language like Curry, the expression $\exists x.\ x = e;\ 3$ would return one result, 3, regardless of e, because x is unused, and so $x = e$ is simply dead code. But in $\mathcal{VC}$ we must evaluate e, and the **all** expression will yield a tuple with one element for each result yielded by e. If e fails (returns zero results), then the result tuple is empty. Since **all** reifies these results into a tuple, the multiplicity of results is visible to the context of the **all** expression, via the length of the returned tuple. Moreover, consider

$$\textbf{all}\{\exists x, y.\ x = ((y = 3;\ 1)\,\|\,(y = 4;\ 2));\ y\}$$

In $\mathcal{VC}$, the expression to which x is equated must be evaluated, yielding two values, 1 and 2, which are then discarded since x is unused. However, each of these values is accompanied by a distinct binding for y, so the result of the whole expression is the tuple $\langle 3, 4\rangle$. In short, we must eventually evaluate the expression to which x is equated not only to get the *size* of the tuple, but also the *values of its elements*, via the bindings of y.

So it seems hard to reconcile laziness with a deterministic **all**. But $\mathcal{VC}$ is not strict, either; in $\mathcal{VC}$ a function can be called without its argument having a ground value. For example:

$$\exists f.\ f = (\lambda x.\ x = 3;\ x);\ \exists y.\ f(y)$$

Here we can call f on an uninstantiated existential variable y; we do not have to wait until y gets a ground value from the calling context. Rather, in this example it is the body of f that then constrains y to be 3. This is part of the essence of functional logic programming, and indeed in $\mathcal{VC}$ a variable *is* a value (Fig. 1). When lenience was introduced in the data-flow language Id [Schauser and Goldstein

[9]As in the previous example, here and elsewhere we freely rewrite terms that have not been fully desugared, but that is just an expository aid; formally, the rewrite rules of Fig. 3 apply only to programs in the basic "Syntax" language of Fig. 1.

1995], it was a way to get *parallelism*, and it certainly justifies parallelism in an implementation of $\mathcal{VC}$. But for $\mathcal{VC}$ lenience has *semantic* significance, too, as we see in this example.

So in $\mathcal{VC}$ functions can be called on as-yet-unevaluated arguments, but almost all redexes that are not under a lambda get evaluated eventually. You can see this enforced in the rule for **one** in Figure 3, which fires only when the first choice in the body is a value v, and similarly for **all**. Why "almost" all redexes? Because the **one** operator returns the value of its first choice and abandons all other choices. For example **one**$\{1 \parallel \textbf{loop}\langle\rangle\}$ returns 1, discarding the infinite **loop**$\langle\rangle$.

Note that lenience, like laziness, supports *abstraction* in the presence of unification. 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.

3.7 Evaluation Strategy

Any rewrite rule can apply anywhere in the term, at any time. For example, in the term $(x = 3 + 4;\ y = 4 + 2;\ x + y)$ the rewrite rules do not say whether to rewrite $3 + 4 \longrightarrow 7$ and then $4 + 2 \longrightarrow 6$, 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 rule SUBST fires only when the RHS is a value. The rewrite rules thereby express *semantics*.

For example, in the lambda calculus, by changing the rewrite rule β to βV, we change the language from call-by-name to call-by-value; by adding **let**, plus suitable rewrite rules, we can express call-by-need [Ariola et al. 1995]. In $\mathcal{VC}$, the rewrite rules are carefully crafted in a similar way; for example, SUBST will substitute $x = v$ only when the equation binds a variable to a *value*, rather like βV in the lambda calculus. Similarly, the elimination rules never discard a term that could fail.

In any term there may of course be many redexes—that is good. An *evaluation strategy* answers the question: given a closed term, which unique redex, out of the many possible redexes, should be rewritten next to make progress toward the result? Let us call an evaluation strategy *normalizing* if it guarantees to terminate if there is *any* terminating sequence of reductions—that is, if any path terminates with a value, then a normalizing evaluation strategy will terminate with that same value[10]. For example, in the pure lambda calculus, *normal-order reduction*, sometimes called *leftmost-outermost reduction*, is a normalizing strategy.

For $\mathcal{VC}$, a first step towards a normalizing strategy is to avoid no-ops. For example, there is no point in applying rule SUBST if $x \notin \text{fvs}(e)$; nor in applying FAIL-ELIM if $X = \square$. Next, a normalizing strategy must be careful with EXI-SWAP, because it can easily apply infinitely; its role is to make it possible to use EXI-FLOAT. Dealing with these no-ops and flip-flops is not hard.

With that done, we believe that a fair outermost strategy is normalizing for $\mathcal{VC}$: at each step, just select any redex that is not within a lambda or another redex. That sounds easy, but is tricky in practice for two reasons. First, a reduction may "unlock" a redex far to its left. For example, consider $(x + y;\ \langle x, 3\rangle = \langle 2, y\rangle;\ x)$. The $(x + y)$ is not a redex, but the equation is; we can apply unification to get $(x = 2;\ y = 3)$, and then use substitution to rewrite the $(x + y)$ to $(2 + 3)$; and *now* the $(2 + 3)$ is a redex. Hence, a challenge for an implementation is to find the next redex efficiently. Secondly, as with other functional-logic languages such as Curry, a normalizing strategy must be able to pursue redexes within multiple subterms in parallel (or alternately), without starving any of

[10]It would be even better if the strategy could 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.

them, hence "fair". For example ($\textbf{loop}\langle\rangle$; **fail**) and (**fail**; $\textbf{loop}\langle\rangle$) should both fail, so a normalizing strategy must, in both cases, avoid getting stuck in the **loop**.

We have not yet, however, formalized such a strategy for $\mathcal{VC}$ or proved it normalizing. We have several prototype implementations of $\mathcal{VC}$, each involving an abstract machine with a stack, a heap, a bunch of previously blocked computations, and so on. Exploring this design space is, however, beyond the scope of this paper.

3.8 Logical Completeness

Our rewrite rules seek to implement the idea articulated in Section 2.1 that a program executes by solving its equations, specifically by finding values for the logical variables that satisfy the equations. Ideally, we would like to promise to find *all* substitutions for the logical variables that satisfy the equations: this is *logical completeness.* Alas, in practice logical completeness is well out of reach, at least in systems that support arithmetic. For example, consider $\exists x.\ (x * x * x + 3 * x - 8) = 0;\ x$ (assuming a version of $\mathcal{VC}$ extended to have a multiplication operator $*$ and subtraction operator $-$). To solve this would require solving a cubic equation, and it is equally easy to express very difficult problems such as Fermat's last theorem. Moreover, remembering that $\mathcal{VC}$ is deterministic, in what order would the (infinite sequence of) results of, say, $\exists x.\ x + 1$ be returned? So instead we satisfy ourselves with a computable and deterministic approximation to this Platonic ideal, an approximation that is described precisely by our rewrite rules, using unification as the mechanism. Expressions that unification cannot solve, like the one above, are stuck; no rewrite rule applies. All functional logic languages share this challenge to logical completeness.[11]

The boundary between "stuck" and "soluble" can be quite subtle. Consider this tricky term: $\exists x.\ x = \textbf{if}\ (x{=}0;\ x > 1)\ \textbf{then}\ 33\ \textbf{else}\ 55;\ x$. At first we 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 SUBST allows us to substitute *locally* in any X-context surrounding the equation $(x{=}0)$ thus:

$$\begin{array}{ll}
\exists x.\ x = \textbf{if}\ (x{=}0;\ x > 1)\ \textbf{then}\ 33\ \textbf{else}\ 55;\ x & \\
\longrightarrow\{\textsc{subst}\} & \exists x.\ x = \textbf{if}\ (x{=}0;\ 0 > 1)\ \textbf{then}\ 33\ \textbf{else}\ 55;\ x \\
\longrightarrow\{\textsc{app-gt-fail}\} & \exists x.\ x = \textbf{if}\ (x{=}0;\ \textbf{fail})\ \textbf{then}\ 33\ \textbf{else}\ 55;\ x \\
\longrightarrow\{\textsc{fail-elim}\} & \exists x.\ x = \textbf{if fail then}\ 33\ \textbf{else}\ 55;\ x \\
\longrightarrow\{\text{simplify } \textbf{if}\} & \exists x.\ x = 55;\ x \quad \longrightarrow\{\textsc{subst}\}\ \exists x.\ x = 55;\ 55 \quad \longrightarrow\{\textsc{eqn-elim}\}\ 55
\end{array}$$

Minor variants of the same example get stuck instead of reducing. For example, if we replace the $x{=}0$ with $x{=}100$ then rewriting gets stuck, as the reader may verify (we cannot eliminate the equation $x{=}100$); and yet there is a substitution that will satisfy the equations, namely $\{55/x\}$. And if we replace $x{=}0$ with $x{=}55$, then rewriting again gets stuck; this time there is *no* substitution that will satisfy the equations, and yet the expression gets stuck rather than failing.

3.9 Developing and Debugging Rules

The rules we describe here should both be able to transform a program to its value, and also be confluent. To aid in the development of the rules, we have used several mechanized tools to automate reduction, random test-case generation, and confluence checking. Initially, we used PLT Redex [Felleisen et al. 2009], which is very easy to use but not very efficient. For better efficiency we switched to a Haskell library for term rewriting. The library provides a DSL for writing rules, and provides the infrastructure to apply the rules everywhere, detect cycles, provide traces, *etc.* Some sample rewrite rules can be found in Fig. 5.

[11]For countable domains, like the integers, or even the algebraic numbers—which would be adequate to solve our cubic equation example—we could in theory simply try all possible values of x one by one; but that is hardly practical.

```
rules lhs  =  "APP-ADD"  `name` (do Op Add :@: Tup [Int k1, Int k2] ← [lhs]
                                    pure (Int (k1 + k2)))
           <> "EXI-SWAP" `name` (do EXI x (EXI y e) ← [lhs]
                                    pure (EXI y (EXI x e)))
           <> "EQN-ELIM" `name` (do EXI x a ← [lhs]
                                    (ctx, (Var x′ :=: Val v) :>: e) ← execX a
                                    guard (x = x′ ∧ x ∉ free (ctx (v :>: e)))
                                    pure (ctx e))
```

Fig. 5. Sample Haskell reduction rules

We used this infrastructure in two ways. First, we have a set of examples with known results, against which we can test a potential rule set. Second, before beginning a proof of confluence, we used QuickCheck [Claessen and Hughes 2000] to generate test cases and check them for confluence. QuickCheck turned out to be invaluable at finding counterexamples to otherwise reasonable-looking rules; it has run on the order of 100 million tests on the current rule set.

4 METATHEORY

The rules of our rewrite semantics can be applied anywhere, in any order; they give meaning to programs without committing to a specific evaluation strategy. But then it had better be true that no matter how the rules are applied, one always obtains the same result! That is, our rules should be *confluent*. In this section, we describe our proof of confluence. Because the rule set is quite big (compared, say, to the pure lambda calculus), this proof turns out to be a substantial undertaking.

Reductions and confluence. A *binary relation* is a set of pairs of related items. A *reduction relation* $\mathcal{R}$ is the *compatible closure*[12] of any binary relation on a set of tree-structured *terms*, such as the terms generated by some BNF grammar. We write $\mathcal{R}^*$ for the reflexive transitive closure of $\mathcal{R}$. We write $e \longrightarrow_{\mathcal{R}} e'$ (*e steps to e′*) if $(e, e') \in \mathcal{R}$ and $e \twoheadrightarrow_{\mathcal{R}} e'$ (*e reduces to e′*) if $(e, e') \in \mathcal{R}^*$. A reduction relation $\mathcal{R}$ 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 and unicity. A term e is an *$\mathcal{R}$-normal form* if there does not exist any e' such that $e \longrightarrow_{\mathcal{R}} e'$. Confluence implies uniqueness of normal forms (unicity): if $e \twoheadrightarrow_{\mathcal{R}} e_1$ and $e \twoheadrightarrow_{\mathcal{R}} e_2$, and e_1 and e_2 are normal forms, then $e_1 = e_2$ [Barendregt 1984, Corollary 3.1.13(ii)].

4.1 Two Challenges to Confluence

Confluence is a purely syntactic property, which leads to two tiresome challenges.

4.1.1 Tiresome Challenge 1: Unifying Lambdas. In $\mathcal{VC}$, an attempt to unify a lambda with another value is stuck (Section 3.2). That choice has two consequences. One is easy to handle, one is tiresome.

First, while U-LIT lets us eliminate equalities on the same literal $k = k$, $\mathcal{VC}$ has no analogous U-VAR rule to drop equalities on the same variable $x = x$. To see why not, suppose we had such a U-VAR rule, and consider the term $(\exists x.\ x = (\lambda y.\ y);\ x = x;\ 0)$. If we first apply U-VAR to eliminate the equality $x = x$, then the remainder reduces to 0. However, if we first SUBST the equality $x = (\lambda y.\ y)$, we get $((\lambda y.\ y) = (\lambda y.\ y);\ 0)$, which is stuck. Therefore, to preserve confluence, $\mathcal{VC}$ has no rule U-VAR: such equalities can be eliminated only after the value of x is substituted in and checked to not be a lambda.

[12]"Compatible closure" means that, for any context E and any two terms M and N, if $(M, N) \in \mathcal{R}$ then $(E[M], E[N]) \in \mathcal{R}$.

Second, the inability to fail when unifying different lambdas leads to non-confluence. Here is an expression that rewrites in two different ways, depending on which equation we SUBST first:

$$(\lambda p.\, 1) = (\lambda q.\, 2);\ 1 \quad \twoheadleftarrow \quad \exists x.\, x = (\lambda p.\, 1);\ x = (\lambda q.\, 2);\ x\ \langle\rangle \quad \twoheadrightarrow \quad (\lambda q.\, 2) = (\lambda p.\, 1);\ 2$$

These two outcomes cannot be joined. This non-confluence is tiresome because the program is wrong anyway: we should not be attempting to unify lambdas.

4.1.2 Tiresome Challenge 2: Recursion and the Notorious Even/Odd Problem. It is well known that adding **letrec** to the lambda calculus makes it non-confluent, in a very tiresome, but hard-to-avoid, way [Ariola and Blom 2002]. In our context, consider the term:

$$\begin{array}{lll} \exists x\, y.\, x = \langle 1, y\rangle;\ y = (\lambda z.\, x);\ x \ \twoheadrightarrow\ \exists y.\, y = (\lambda z.\, \langle 1, y\rangle);\ \langle 1, y\rangle & & \text{(1) substitute for } x \text{ first} \\ \exists x\, y.\, x = \langle 1, y\rangle;\ y = (\lambda z.\, x);\ x \ \twoheadrightarrow\ \exists x.\, x = \langle 1, \lambda z.\, x\rangle;\ x & & \text{(2) substitute for } y \text{ first} \end{array}$$

The results of (1) and (2) have the same meaning (are indistinguishable by a $\mathcal{VC}$ context) but cannot be joined by our rewrite rules. Nor is this easily fixed by adding new rules, as we did when we added VAR-SWAP (Section 3.2) and SEQ-SWAP (Section 3.4). Why not? Because the terms are equivalent only under some kind of graph isomorphism.

4.1.3 Resolving the Two Challenges. We explored three different ways to address these challenges. The first is simply to abandon confluence as a goal altogether. Confluence is, after all, purely *syntactic*, and hence much stronger than what we really need, which is only that each of our rules be *semantics*-preserving. But that, of course, requires an independent notion of semantics, a direction we sketch in Appendix E of Augustsson et al. [2023].

Second, we can simply exclude programs that unify lambdas, or that use recursion. To be precise:

- An equation is *obviously problematic* if either
  - it is an equation of form $hnf = \lambda x.\, e$ or $\lambda x.\, e = hnf$, or
  - it is an equation of the form $x = V[\lambda y.\, e]$, where $x \in \text{fvs}(e)$.
- A term e is *obviously problematic* if it contains an obviously problematic equation.
- A term e is *problematic* if there exists an obviously problematic term e' such that $e \twoheadrightarrow e'$.
- A term e is *well-behaved* if it is not problematic.

Then we prove confluence only for well-behaved terms. We take this approach for our main proof in this paper (Section 4.2). Excluding lambda unification is fine; programmers simply cannot expect that to work. Excluding recursion may seem drastic, but no expressiveness is lost thereby: in our untyped setting, one can still write recursive (and non-terminating) programs using one's favorite fixpoint combinator, such as **Y** or **Z**.

But excluding recursion is not entirely satisfying: it is hard to prove that a term has no recursion, and it is clumsy to write recursive programs using only **Y**-combinators. Our third approach is to adopt the idea of *skew confluence* [Ariola and Blom 2002], a clever technique developed specifically to handle the even/odd problem; we give an overview of skew confluence in Section 4.3, and provide details of our approach to a proof of skew confluence in Appendix D of Augustsson et al. [2023], with several new lemmas, but we emphasize that the proof of skew confluence is not yet complete.

4.2 Proof of Confluence

Our main result is that $\mathcal{VC}$'s reduction rules are confluent for well-behaved terms. We sketch the proof here, with full details in Appendix C of Augustsson et al. [2023].

THEOREM 4.1 (CONFLUENCE). *The reduction relation in Fig. 3 is confluent for well-behaved terms.*

Proof sketch. Our proof strategy is to (1) divide the rules into groups for application, unification, *etc.*, approximately as in Fig. 3, (2) prove confluence for each separately, and then (3) prove that their combination is confluent via commutativity. 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 a theorem of Huet [1980]: If R and S are confluent and commute, then $R \cup S$ is confluent. Proving confluence for application, elimination and choice 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*—which suffices to show the relations are confluent [Barendregt 1984]. The diamond property itself can be verified easily by considering critical pairs of transitions. The rules for unification and normalization, however, pose two problems.

The unification problem. The first problem is that the unification relation does not satisfy the diamond property—it may need multiple steps to join the results of two different one-step reductions. For example, consider the term $(x = \langle 1, y\rangle;\ x = \langle z, 2\rangle;\ x = \langle 1, 2\rangle;\ 3)$. The term can be reduced in one step by substituting x in the third equation 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 λ-calculus (*e.g.*, [Barendregt 1984]), our proof of confluence for the unification rules works in two stages. First, we prove that the reductions are *locally confluent*, meaning if e single-steps to each of e_1 and e_2, then e_1 and e_2 can be joined at some e' by taking *multiple* unification rule steps. Second, we prove that the unification reductions are *terminating*, which relies upon eliminating recursion in tuples via U-OCCURS and in lambdas via the well-behaved-ness condition. Newman's Lemma [Huet 1980, Lemma 2.4] then implies that the locally confluent, terminating unification relation is also confluent.

The normalization problem. The second problem is that the normalization rules do not commute with the unification rules. Recall from Section 3.3 that the unification rules rely upon variable ordering to orient equations between variables in a canonical fashion. The normalization rule EXI-SWAP can *change* the variable order and hence, its behavior is deeply intertwined with unification and cannot be factored out via a commutativity argument. Instead, we prove that the union of unification and normalization is confluent by showing that unification *postpones after* normalization [Hindley 1964]; see Appendix C of Augustsson et al. [2023] for the gory details.

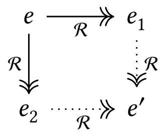
4.3 Overview of Skew Confluence

We travel a path very similar to the one blazed by Ariola and her co-authors. Ariola and Klop studied a form of the lambda calculus with an added letrec construct and determined (like us) that their calculus was not confluent; then they added a specific constraint on recursive substitution and proved that the modified calculus is confluent [Ariola and Klop 1994, 1997]. In a later paper, Ariola and Blom proved that their calculus without the constraint, while not confluent, does obey a weaker related property, which they invented, called *skew confluence* [Ariola and Blom 2002]. We believe, and currently are trying to prove, that $\mathcal{VC}$ without the pesky no-recursion side condition of Theorem 4.1 is skew confluent.

$$\text{Confluence:}\quad \forall e, e_1, e_2.\ e \twoheadrightarrow_{\mathcal{R}} e_1 \wedge e \twoheadrightarrow_{\mathcal{R}} e_2 \implies \exists e'.\ e_1 \twoheadrightarrow_{\mathcal{R}} e' \wedge e_2 \twoheadrightarrow_{\mathcal{R}} e'.$$
$$\text{Skew confluence:}\quad \forall e, e_1, e_2.\ e \twoheadrightarrow_{\mathcal{R}} e_1 \wedge e \twoheadrightarrow_{\mathcal{R}} e_2 \implies \exists e'.\ e_1 \twoheadrightarrow_{\mathcal{R}} e' \wedge e_2 \preceq_{\omega\mathcal{R}} e'.$$

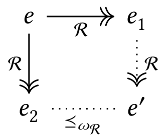
These are depicted here as two commutative diagrams, which differ only on the bottom edge:

Confluence



$$\begin{tikzcd} e \arrow[r, ->>, "\mathcal{R}"'] \arrow[d, ->>, "\mathcal{R}"'] & e_1 \arrow[d, ->>, dotted, "\mathcal{R}"] \\ e_2 \arrow[r, ->>, dotted, "\mathcal{R}"'] & e' \end{tikzcd}$$

Skew confluence



$$\begin{tikzcd} e \arrow[r, ->>, "\mathcal{R}"'] \arrow[d, ->>, "\mathcal{R}"'] & e_1 \arrow[d, ->>, dotted, "\mathcal{R}"] \\ e_2 \arrow[r, dash, dotted, "\preceq_{\omega\mathcal{R}}"'] & e' \end{tikzcd}$$

For each diagram, given e, e_1, e_2 that obey the relationships indicated by all the solid lines, there exists e' such that all relationships indicated by dotted lines are also satisfied.

You can understand skew confluence as follows: if two different reduction paths from e produce terms e_1, e_2, then e_1 can be further reduced to some e' such that all of e_2's permanent structure is present in e', written $e_2 \preceq_{\omega_{\mathcal{R}}} e'$. By "permanent structure" we mean an outer shell of tuples, lambdas, and constants, that will never change no matter how much further reduction takes place. For example, however far we reduce the term $\langle 1, \lambda z.\, e\rangle$, the result will always look like $\langle 1, \lambda z.\, e'\rangle$, where $e \twoheadrightarrow_{\mathcal{R}} e'$. We can formalize the notion of permanent structure by defining an *information content function* $\omega_{\mathcal{R}}(e)$ that replaces all the impermanent bits of e with a new dummy term Ω. Thus $\omega_{\mathcal{R}}(\langle 1, \lambda z.\, x\rangle) = \langle 1, \lambda z.\, \Omega\rangle$. Then $e_2 \preceq_{\omega_{\mathcal{R}}} e'$ if $\omega_{\mathcal{R}}(e_2)$ can be made equal to e' by replacing each occurrence of Ω in $\omega_{\mathcal{R}}(e_2)$ with an (individually-chosen) term.

Consider the even-odd problem discussed in Section 4.1.2.

$$\exists x\, y.\; x = \langle 1, y\rangle;\; y = \lambda z.\, x;\; x$$

$$\begin{array}{l|l}
\twoheadrightarrow \exists y.\; y = \lambda z.\, \langle 1, y\rangle;\; \langle 1, y\rangle & \twoheadrightarrow \exists x.\; x = \langle 1, \lambda z.\, x\rangle;\; x \\
\longrightarrow\; \exists y.\; y = \lambda z.\, \langle 1, y\rangle;\; \langle 1, \lambda z.\, \langle 1, y\rangle\rangle & \longrightarrow\; \exists x.\; x = \langle 1, \lambda z.\, x\rangle;\; \langle 1, \lambda z.\, x\rangle \\
\longrightarrow\; \exists y.\; y = \lambda z.\, \langle 1, y\rangle;\; \langle 1, \lambda z.\, \langle 1, \lambda z.\, \langle 1, y\rangle\rangle\rangle & \longrightarrow\; \exists x.\; x = \langle 1, \lambda z.\, x\rangle;\; \langle 1, \lambda z.\, \langle 1, \lambda z.\, x\rangle\rangle \\
\longrightarrow\; \cdots & \longrightarrow\; \cdots
\end{array}$$

The two columns can never join up, but if you pick any term in either column, there is a term in the other column that has at least as much permanent structure. That in turn means that the terms in the left-hand column are contextually equivalent to those in the right-hand column, because the context can inspect only the permanent structure. This contextual equivalence is the real reason for seeking confluence in the first place.

In $\mathcal{VC}$, the notion of permanent structure must be generalized from an outer shell of tuples, lambdas, and constants to a *sequence* (right-associative tree) of *choices* of such outer shells. For example, however far we reduce the term $\langle 68, \lambda x.\, e_1\rangle \,\|\, \langle \lambda y.\, e_2, 57, \lambda z.\, e_3\rangle \,\|\, e_4$, the result will always look like $\langle 68, \lambda x.\, e'_1\rangle \,\|\, \langle \lambda y.\, e'_2, 57, \lambda z.\, e'_3\rangle \,\|\, e'_4$, where $e_k \twoheadrightarrow_{\mathcal{R}} e'_k$ for $1 \leqslant k \leqslant 4$; or $\langle 68, \lambda x.\, e'_1\rangle \,\|\, \langle \lambda y.\, e'_2, 57, \lambda z.\, e'_3\rangle$ if $e_4 \twoheadrightarrow_{\mathcal{R}}$ **fail**. In Appendix D of Augustsson et al. [2023] we sketch our plan to adapt the proof strategy of Section 4.2 and Appendix C of Augustsson et al. [2023] for skew confluence.

5 VARIATIONS AND CHOICES

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

5.1 Simplifying the Rules

The rules of Figure 3 are more complicated than we would like, because our proof of confluence requires that the unification rules form a *terminating* rewrite system (Section 4.2). Tantalizingly, we have found simpler versions of four of the unification and elimination rules that we believe are equally expressive and still support (modified versions of) our proofs of termination and confluence:

SUBST′	$x = v;\; e$	$\longrightarrow$	$x = v;\; e\{v/x\}$	if $v \neq V[x]$
VAR-SWAP′	$y = x;\; e$	$\longrightarrow$	$x = y;\; e$	(unconditionally)
SEQ-SWAP′	$eq;\; x = v;\; e$	$\longrightarrow$	$x = v;\; eq;\; e$	(unconditionally)
EQN-ELIM′	$\exists x.\; x = v;\; e$	$\longrightarrow$	e	if $x \notin \mathrm{fvs}(v, e)$

Each of these rules is simpler than its counterpart in Figure 3. In particular, rules SUBST′ and EQN-ELIM′ need no context X: instead, they rely on other rules (EQN-FLOAT and SEQ-SWAP′) to float the equation of interest upward and to the left. Our testing framework (Section 3.9) has given us strong confidence that the entire set of rules, with these four simplifications, remains confluent; actually *proving* it confluent is work in progress. We speculate that these same simplifications will also support a proof of skew confluence.

5.2 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:

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

The trouble with this 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:

$$\text{LABELED-CHOOSE} \quad CX[\,e_1 \mathbin{|} e_2\,] \longrightarrow CX[\,L \triangleright e_1\,] \mathbin{|} CX[\,R \triangleright e_2\,]$$

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

$$\text{SORT} \quad (R \triangleright e_1) \mathbin{|} (L \triangleright e_2) \longrightarrow (L \triangleright e_2) \mathbin{|} (R \triangleright e_1)$$

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. However, the idea inspired our denotational semantics (Appendix E.4 of Augustsson et al. [2023]), where it seems to work rather beautifully.

5.3 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**[13] as $e ::= \cdots \mid \mathbf{split}\{e\}\langle v_1, v_2\rangle$ and rules:

$$\begin{array}{lrcl}
\text{SPLIT-FAIL} & \mathbf{split}\{\mathbf{fail}\}\langle f, g\rangle & \longrightarrow & f\langle\rangle \\
\text{SPLIT-VALUE} & \mathbf{split}\{v\}\langle f, g\rangle & \longrightarrow & g\langle v, \lambda\langle\rangle.\,\mathbf{fail}\rangle \\
\text{SPLIT-CHOICE} & \mathbf{split}\{v \mathbin{|} e\}\langle f, g\rangle & \longrightarrow & g\langle v, \lambda\langle\rangle.\,e\rangle
\end{array}$$

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; but if e returns at least one value, it passes that to g together with the remaining computation, safely tucked away within a lambda. When adding more effects to $\mathcal{VC}$ (see Appendix F of Augustsson et al. [2023]), it is in fact crucial to use **split** to exactly control the order of effects.

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

$$\begin{array}{rll}
\mathbf{one}\{e\} \equiv f(x) := \mathbf{fail}; & g\langle x, y\rangle := x; & \mathbf{split}\{e\}\langle f, g\rangle \\
\mathbf{all}\{e\} \equiv f(x) := \langle\rangle; & g\langle x, y\rangle := cons\langle x, \mathbf{split}\{y\langle\rangle\}\langle f, g\rangle\rangle; & \mathbf{split}\{e\}\langle f, g\rangle
\end{array}$$

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.

6 $\mathcal{VC}$ IN CONTEXT: REFLECTIONS AND RELATED WORK

Functional logic programming has a rich literature; excellent starting points are the CACM review article by Antoy and Hanus [2010] and the longer survey by Hanus [2013]. Now that we know what $\mathcal{VC}$ is, we can identify its distinctive features, and compare them to other approaches.

[13]The name **split** was inspired by Kiselyov et al. [2005].

6.1 Choice and Non-determinism

A significant difference between our presentation and earlier works is our treatment of choice. Consider an expression like $(3+(20 \mid 30))$. Choice 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] and 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. 2007] or *pull-tabbing* [Antoy 2011]. However, laziness imposes significant additional complications, including the need to attach an identifier to each choice.

Recent work by Barenbaum et al. [2020, 2021] describes a functional logic calculus that, like $\mathcal{VC}$, is just an extension of lambda calculus. They deal with lambda, existentials, choice, and equations in a similar way to $\mathcal{VC}$, but lack the **all** and **one** operators which are central to $\mathcal{VC}$. As a result they can always float choice up to the top level of the whole program, and indeed they do so in the very structure of their terms.

6.2 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 Fresh, a very interesting design introduced in a technical report nearly 40 years ago [Smolka and Panangaden 1985] that also aims to unify functional and logical constructs. Fresh reifies choice into data via *confinement* (corresponding to **one**) and *collection* (corresponding to **all**). However, Fresh differs from $\mathcal{VC}$ in crucial ways. First, it solves equations in a strictly left-to-right fashion, which means that it is not lenient in the sense discussed in Section 3.6. Second, its semantics are presented in an operational fashion with explicit stacks and heaps, in contrast to our focus on developing an equational account of functional logic programming. Finally, Fresh appears not to have been implemented.

Several aspects of **all** and **one** are worth noting. First, **all** *reifies* choice (a control operator) into a tuple (a data structure); for example, **all**$\{1 \mid 7 \mid 2\}$ returns the tuple $\langle 1, 7, 2\rangle$. In the other direction, indexing turns a tuple into choice (for example, $\exists i.\ \langle 1, 7, 2\rangle(i)$ 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 [Antoy and Hanus 2009; Braßel and Huch 2007, 2009]. But in Curry, 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[14], 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*. $\mathcal{VC}$ takes pains to maintain order, so that when reifying choice into a tuple, the order of elements in that tuple is completely

[14]Again, this complexity in Curry appears to be a consequence of laziness.

determined. This determinism has a price: we have to take care to maintain the left-to-right order of choices (see Section 2.3 and Section 3.5, for example). 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 $\mathcal{VC}$. To substantiate this claim, Appendix F of Augustsson et al. [2023] gives the additional syntax and rewrite rules for mutable variables.

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 [Clocksin and Mellish 2003, Chapter 4] has a similar flavor, 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).

6.3 The semantics of logical variables

Our logical variables, introduced by ∃, 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. CRWL 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 formalization of the call-by-need lambda calculus) is to use **let** to make sharing explicit.

However, both CRWL and López-Fraguas *et al.* are in some ways *too* high level: they require something we call *magical rewriting*. A key rewrite rule is this:

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

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 on the right-hand side 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 by 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.

The Escher language [Lloyd 1999] extends a Haskell-like functional base with logical (existentially quantified) variables and constraints to yield an integration of functional and logical constructs, and informally describes a rewriting based evaluation mechanism. However, the language does not have the equivalent of **one** and **all** which reify choice as data, the rules are not precisely formalized, and no confluence result is established for them.

Goffin [Chakravarty et al. 1998] also presents a way to extend a higher-order functional language with existentially quantified logical variables. However, Goffin is rather different from $\mathcal{VC}$ in that it falls in the tradition of Concurrent Constraint Programming [Saraswat and Rinard 1990], where the constraints over logical variables are used to declare data dependencies as a means to co-ordinate efficient concurrent execution of sub-computations, rather than in the tradition of logic programming (and $\mathcal{VC}$) where we seek to declare the properties of solutions. Consequently, in Goffin, the logical variables are never "unified" but instead are "updated" as values are computed in parallel, yielding a completely different model of computation.

6.4 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 the names of functions are syntactically distinguished from ordinary variables. As Hanus describes, it is possible to translate a higher-order program into a first-order form using defunctionalization [Hanus 2013, Section 3.3] and a built-in **apply** function. (Hanus does not mention this, but for a language with arbitrarily nested lambdas, one would need to do lambda-lifting [Johnsson 1985] as well; this is perhaps a minor point.) 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 behave. In contrast, a strength of our $\mathcal{VC}$ presentation is that it deals natively with the full lambda calculus.

6.5 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.

6.6 Comparison with Various Functional Language Extensions to Datalog

Datalog [Ceri et al. 1989] in its purest form is a subset of Prolog [Clocksin and Mellish 2003; Warren et al. 1977] that is carefully limited so that every Datalog program is guaranteed to terminate.

Datalog variants are often used as database query languages and to concisely express fixpoint computations on lattices (for example, static-analysis phases for compilers). Some of these variants support functional programming while preserving the guarantee that every program terminates.

- Datafun [Arntzenius and Krishnaswami 2016] is perhaps closest in spirit to Verse in that it supports higher-order functions as first-class values; it uses a strong type system to prohibit testing functions for equality and to prohibit unbounded recursion.
- Flix [Madsen et al. 2016] makes it easy to work with arbitrary user-defined lattices; programs are guaranteed to terminate as long as (a) all declared lattices are complete and of finite height, and (b) all functions are strict and monotone. Functions are not first-class and their use is syntactically restricted.
- QL [Avgustinov et al. 2016] has an object-oriented, class-based syntax that is then compiled to pure Datalog; this allows the programmer to use inheritance to organize code on related types of database records. In some cases what looks like a "method call" appears to produce multiple values, but this is just a bit of syntactic sugar for expressing relations.
- Formulog [Bembenek et al. 2020] extends Datalog with a first-order functional language (a subset of ML) and an SMT solver. Functions are not first-class; functions, predicates, and variables have separate namespaces.
- Functional IncA [Pacak and Erdweg 2022] is a language that supports first-class higher-order functions over sets and algebraic data types; this language is compiled to pure Datalog. The compilation process eliminates first-class functions through defunctionalization. The paper does not discuss an equality operator or whether it may be applied to functions.

None of these languages supports unbounded computation, a choice operator, or failure.

6.7 Comparison with Icon

There are many obvious similarities between Verse and the Icon programming language [Griswold 1979; Griswold and Griswold 1983, 2002; Griswold et al. 1979, 1981]:

- An expression can (successively) produce any number of values. An expression that produces zero values is said to *fail* [Griswold et al. 1981, §3.1]; an expression that produces at least one value is said to *succeed.*
- The expression $e_1 \,|\, e_2$ produces all the values of e_1 followed by all the values of e_2.
- There is a way to turn an array (or tuple) a into a sequence of produced values. In Icon, this is written $!\,a$ [Griswold et al. 1979, §3]; in Verse, a?; in $\mathcal{VC}$, $\exists i.\ a(i)$.
- Most "scalar" operations (such as addition and comparisons) run through all possible combinations of values of their operand expressions, using a specific left-to-right evaluation order and automatic chronological backtracking.
- Success and failure are used in place of boolean values for control-structure purposes, just as in Section 2.5.
- The "|" construct is idiomatically used as a *logical or* operation [Griswold et al. 1979, §3].
- There is a control structure that executes a specified expression once for every value produced by another expression. In Icon, this is `every` e_1 `do` e_2 and in Verse, it is written `for(`e_1`) do` e_2. In each language, the "`do` e_2" may be omitted to simply evaluate e_1 repeatedly until it has yielded all its values. In Icon, `every` e may also be written as `repeat` e.
- It is impossible to name a generator (Icon) or choice (Verse); if e produces multiple values, $x\ :=\ e$ will provide one value at a time from e to be named by variable x.

But there are also major differences. Icon was designed primarily to use expressions as generators to automatically explore a combinatorial space of possibilities ("goal-directed evaluation"), and secondarily to use success/failure rather than booleans to drive control structure. But in other respects, *Icon is a fairly conventional imperative language*, relying on side effects (assignments) to process the generated combinations. The designers judged that the interactions of such side effects with completely unrestrained control backtracking would be difficult for programmers to understand [Griswold et al. 1981, §3.1]; therefore, the design of Icon emphasizes limited scopes for control backtracking and tools for controlling the backtracking process [Griswold et al. 1981, §3.3].

In contrast, *Verse is a declarative language* and avoids these difficulties by using a functional logic approach rather than an imperative approach to processing generated combinations:

- While Icon typically processes multiple values from an expression by using *assignment*, Verse typically processes multiple values by using *equations* (which are then *solved*).
- Verse also has a concise way to turn a finite sequence of multiple values into a tuple directly. For example, to make variable `a` refer to an array containing all values generated by expression `e`, code such as the following (using a `repeat` loop containing an assignment) is idiomatic in Icon [Griswold et al. 1979, §8]:
  ```
          a := array 0 string; i := 0; repeat a[i+] := e; close(a)
  ```
  In Verse, `a = for{e}` does the job; in $\mathcal{VC}$, $a = \textbf{all}\{e\}$ is all it takes.
- Backtracking in Icon is "only control backtracking"; side effects, such as assignments, are not undone [Griswold et al. 1981, §3.1].
- Both languages have an implicit "cut" (permanent acceptance of the first produced value) after the predicate part of an **if-then-else**, but Icon furthermore has an implicit cut at each statement end (semicolon or end of line) [Griswold et al. 1981, §3.1], each closing brace "}", and most keywords [Icon PC 1980].

7 LOOKING BACK, LOOKING FORWARD

We believe that this is the first presentation of a functional logic language as a deterministic rewrite system. A rewrite system has the advantage (compared to more denotational, or more operational, methods) that it is sufficiently low-level to capture the *computational model* of the language, and yet sufficiently high-level to be *illuminating* to a programmer or compiler writer. Our focus on rewriting as a way to define the semantics has forced us to focus on confluence, a syntactic property that is stronger (and hence more delicate and harder to prove) than the contextual equivalence that is all we really need. That in turn led us to study the elegant and ingenious notion of skew confluence, which has been barely revisited during the last 20 years, but which we believe deserves a wider audience.

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 for 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*; for example, when the condition of an **if** fails, any store effects in the condition must be rolled back. We have preliminary reduction rules for references see Appendix F of Augustsson et al. [2023].

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, David Holz, Juan Rodríguez Hortalá, John Launchbury, Dale Miller, Andy Pitts, Niklas Röjemo, Jaime Sánches-Hernández, Andrew Scheidecker, Stephanie Weirich, and the anonymous ICFP reviewers. We are particularly grateful to Michael Hanus, who helped us a great deal to position $\mathcal{VC}$ more accurately in the rich space of functional logic languages.

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.

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

Sergio Antoy, Daniel W. Brown, and Su-Hui Chiang. 2007. Lazy Context Cloning for Non-Deterministic Graph Rewriting. *Electronic Notes in Theoretical Computer Science* 176, 1 (May 2007), 3–23. https://doi.org/10.1016/j.entcs.2006.10.026 Proceedings of the Third International Workshop on Term Graph Rewriting (TERMGRAPH 2006).

Sergio Antoy and Michael Hanus. 2009. Set Functions for Functional Logic Programming. In *Proceedings of the 11th ACM SIGPLAN Conference on Principles and Practice of Declarative Programming* (Coimbra, Portugal) *(PPDP '09)*. Association for Computing Machinery, New York, NY, USA, 73–82. https://doi.org/10.1145/1599410.1599420

Sergio Antoy and Michael Hanus. 2010. Functional Logic Programming. *Commun. ACM* 53, 4 (April 2010), 74–85. https://doi.org/10.1145/1721654.1721675

Sergio Antoy and Michael Hanus. 2021. *Curry: A Tutorial Introduction*. Technical Report. Kiel University (Christian-Albrechts-Universität zu Kiel). https://web.archive.org/web/20220121070135/https://www.informatik.uni-kiel.de/~curry/tutorial/tutorial.pdf

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 and Jan Willem Klop. 1994. Cyclic lambda graph rewriting. In *Proceedings of the Ninth Annual IEEE Symposium on Logic in Computer Science (LICS '94)*. IEEE, 416–425. https://doi.org/10.1109/LICS.1994.316066

Zena M. Ariola and Jan Willem Klop. 1997. Lambda Calculus with Explicit Recursion. *Information and Computation* 139, 2 (Dec. 1997), 154–233. https://doi.org/10.1006/inco.1997.2651

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

Michael Arntzenius and Neelakantan R. Krishnaswami. 2016. Datafun: A Functional Datalog. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming* (Nara, Japan) *(ICFP 2016)*. Association for Computing Machinery, New York, NY, USA, 214–227. https://doi.org/10.1145/2951913.2951948

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

Lennart Augustsson, Joachim Breitner, Koen Claessen, Ranjit Jhala, Simon Peyton Jones, Olin Shivers, Guy Steele, and Tim Sweeney. 2023. *The Verse Calculus: A Core Calculus for Deterministic Functional Logic Programming*. Technical Report. Epic Games. https://simon.peytonjones.org/verse-calculus/

Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. 2016. QL: Object-oriented Queries on Relational Data. In *30th European Conference on Object-Oriented Programming (ECOOP 2016) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Schloss Dagstuhl—Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2:1–2:25. https://doi.org/10.4230/LIPIcs.ECOOP.2016.2

Pablo Barenbaum, Federico Lochbaum, and Mariana Milicich. 2020. Semantics of a Relational λ-Calculus. In *Theoretical Aspects of Computing (ICTAC 2020)*, Violet Ka I Pun, Volker Stolz, and Adenilso Simao (Eds.). Springer International Publishing, Cham, 242–261. https://doi.org/10.1007/978-3-030-64276-1_13

Pablo Barenbaum, Federico Lochbaum, and Mariana Milicich. 2021. Semantics of a Relational λ-Calculus (Extended Version), version 4. *CoRR* abs/2009.10929 (28 Feb. 2021), 51 pages. arXiv:2009.10929 https://arxiv.org/abs/2009.10929

H. P. (Hendrik Pieter) Barendregt. 1984. *The Lambda Calculus: Its Syntax and Semantics* (revised ed.). Studies in Logic and the Foundations of Mathematics, Vol. 103. North-Holland (Elsevier Science Publishers), Amsterdam.

Aaron Bembenek, Michael Greenberg, and Stephen Chong. 2020. Formulog: Datalog for SMT-Based Static Analysis. *Proc. ACM Programming Languages* 4, OOPSLA, Article 141 (Nov. 2020), 31 pages. https://doi.org/10.1145/3428209

Bernd Braßel, Michael Hanus, and Frank Huch. 2004a. Encapsulating Non-Determinism in Functional Logic Computations [extended journal version]. *Journal of Functional and Logic Programming* 2004, 6 (Dec. 2004), 28 pages. http://danae.uni-muenster.de/lehre/kuchen/JFLP/articles/2004/S04-01/A2004-06/JFLP-A2004-06.pdf Special Issue 1. Archived at https://web.archive.org/web/20060505093657/http://danae.uni-muenster.de/lehre/kuchen/JFLP/articles/2004/S04-01/A2004-06/JFLP-A2004-06.pdf..

Bernd Braßel, Michael Hanus, and Frank Huch. 2004b. Encapsulating Non-Determinism in Functional Logic Computations [workshop version]. In *13th International Workshop on Functional and (Constraint) Logic Programming (WFLP'04)*. 74–90. Full proceedings available at http://www-i2.informatik.rwth-aachen.de/WFLP04/WFLP-proceedings.pdf. RWTH Aachen, Department of Computer Science, Technical Report AIB-2004-05. Archived at https://web.archive.org/web/

20040713053113/http://www-i2.informatik.rwth-aachen.de/WFLP04/WFLP-proceedings.pdf..

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 2007) (LNCS 4807)*, Zhong Shao (Ed.). Springer-Verlag, Berlin, Heidelberg, 122–138. https://doi.org/10.1007/978-3-540-76637-7_9

Bernd Braßel and Frank Huch. 2009. The Kiel Curry System KiCS. In *Applications of Declarative Programming and Knowledge Management (LNAI 5437)*, Dietmar Seipel, Michael Hanus, and Armin Wolf (Eds.). Springer-Verlag, Berlin, Heidelberg, 195–205. https://doi.org/10.1007/978-3-642-00675-3_13

Stefano Ceri, Georg Gottlob, and Letizia Tanca. 1989. What You Always Wanted to Know about Datalog (and Never Dared to Ask). *IEEE Transactions on Knowledge and Data Engineering* 1, 1 (March 1989), 146–166. https://doi.org/10.1109/69.43410 A comprehensive survey with an extensive bibliography.

Manuel M.T. Chakravarty, Yike Guo, Martin Köhler, and Hendrik C.R. Lock. 1998. GOFFIN: Higher-order Functions Meet Concurrent Constraints. *Science of Computer Programming* 30, 1 (June 1998), 157–199. https://doi.org/10.1016/S0167-6423(97)00010-5

Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming* (Montreal, Canada) *(ICFP '00)*. Association for Computing Machinery, New York, NY, USA, 268–279. https://doi.org/10.1145/351240.351266

William F. Clocksin and Christopher S. Mellish. 2003. *Programming in Prolog (Using the ISO Standard)* (fifth ed.). Springer-Verlag New York Berlin Heidelberg.

Ugo Dal Lago, Giulio Guerrieri, and Willem Heijltjes. 2020. Decomposing Probabilistic Lambda-Calculi. In *Foundations of Software Science and Computation Structures: 23rd International Conference (FoSSaCS'20) (LNCS 12077)*, Jean Goubault-Larrecq and Barbara König (Eds.). Springer International, 136–156. https://doi.org/10.1007/978-3-030-45231-5_8

Ugo de'Liguoro and Adolfo Piperno. 1995. Nondeterministic Extensions of Untyped λ-Calculus. *Information and Computation* 122, 2 (1995), 149–177. https://doi.org/10.1006/inco.1995.1145

Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. 2009. *Semantics Engineering with PLT Redex*. MIT Press, Cambridge, Massachusetts, USA. https://mitpress.mit.edu/9780262062756/semantics-engineering-with-plt-redex/

Matthias Felleisen and Daniel P. Friedman. 1986. Control Operators, the SECD Machine, and the λ-Calculus. In *Formal Description of Programming Concepts III: Proceedings of the IFIP TC 2/WG 2.2 Working Conference* (Ebberup, Denmark). Elsevier Science Publishers (North-Holland), 193–217. https://web.archive.org/web/20220709064643/https://www.cs.tufts.edu/~nr/cs257/archive/matthias-felleisen/cesk.pdf

Matthias Felleisen, Daniel P. Friedman, Eugene Kohlbecker, and Bruce Duba. 1987. A Syntactic Theory of Sequential Control. *Theoretical Computer Science* 52, 3 (1987), 205–237. https://doi.org/10.1016/0304-3975(87)90109-5

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. *The Journal of Logic Programming* 40, 1 (July 1999), 47–87. https://doi.org/10.1016/S0743-1066(98)10029-8

Ralph E. Griswold. 1979. *User's Manual for the Icon Programming Language.* Technical Report TR 78-14. Department of Computer Science, University of Arizona. https://www2.cs.arizona.edu/icon/ftp/doc/tr78_14.pdf

Ralph E. Griswold and Madge T. Griswold. 1983. *The Icon Programming Language.* Prentice-Hall, Englewood Cliffs, New Jersey.

Ralph E. Griswold and Madge T. Griswold. 2002. *The Icon Programming Language* (third ed.). Peer-to-Peer Communications. https://web.archive.org/web/20040723085807/https://www2.cs.arizona.edu/icon/ftp/doc/lb1up.pdf

Ralph E. Griswold, David R. Hanson, and John T. Korb. 1979. The Icon Programming Language: An Overview. *SIGPLAN Notices* 14, 4 (April 1979), 18–31. https://doi.org/10.1145/988078.988082

Ralph E. Griswold, David R. Hanson, and John T. Korb. 1981. Generators in Icon. *ACM Trans. Programming Languages and Systems* 3, 2 (April 1981), 144–161. https://doi.org/10.1145/357133.357136

Michael Hanus. 2013. Functional Logic Programming: From Theory to Curry. In *Programming Logics: Essays in Memory of Harald Ganzinger*, Andrei Voronkov and Christoph Weidenbach (Eds.). LNCS, Vol. 7797. Springer, Berlin, Heidelberg, 123–168. https://doi.org/10.1007/978-3-642-37651-1_6

Michael Hanus, Sergio Antoy, Bernd Braßel, Herbert Kuchen, Francisco J. López-Fraguas, Wolfgang Lux, Juan José Moreno Navarro, Björn Peemöller, and Frank Steiner. 2016. *Curry: An Integrated Functional Logic Language (Version 0.9.0).* Technical Report. University of Kiel. https://web.archive.org/web/20161020144634/https://www-ps.informatik.uni-kiel.de/currywiki/_media/documentation/report.pdf

J. Roger Hindley. 1964. *The Church-Rosser Property and a Result in Combinatory Logic.* Ph. D. Dissertation. University of Newcastle-upon-Tyne, United Kingdom.

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

Icon PC 1980. Programming Corner from Icon Newsletter 4. https://www2.cs.arizona.edu/icon/progcorn/pc_inl04.htm

Thomas Johnsson. 1985. Lambda lifting: Transforming programs to recursive equations. In *Functional Programming Languages and Computer Architecture*, Jean-Pierre Jouannaud (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 190–203.

Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman, and Amr Sabry. 2005. Backtracking, interleaving, and terminating monad transformers: (functional pearl). In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005, Tallinn, Estonia, September 26-28, 2005*, Olivier Danvy and Benjamin C. Pierce (Eds.). ACM, 192–203. https://doi.org/10.1145/1086365.1086390

Arne Kutzner and Manfred Schmidt-Schauß. 1998. A Non-Deterministic Call-by-Need Lambda Calculus. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming* (Baltimore, Maryland, USA) *(ICFP '98)*. Association for Computing Machinery, New York, NY, USA, 324–335. https://doi.org/10.1145/289423.289462

John Lamping. 1990. An Algorithm for Optimal Lambda Calculus Reduction. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '90)*. Association for Computing Machinery, New York, NY, USA, 16–30. https://doi.org/10.1145/96709.96711

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, 144–154. 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. https://web.archive.org/web/20051016053439/http://pauillac.inria.fr/~levy/pubs/78phd.pdf

John W. Lloyd. 1999. Programming in an Integrated Functional and Logic Language. *Journal of Functional and Logic Programming* 1999, 3 (March 1999). http://danae.uni-muenster.de/lehre/kuchen/JFLP//articles/1999/A99-03/JFLP-A99-03.pdf Archived at https://web.archive.org/web/20040627000956/http://danae.uni-muenster.de/lehre/kuchen/JFLP/articles/1999/A99-03/JFLP-A99-03.pdf. Also available at https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=31d9a554f61ced1304ecd0ded9fca6fee2173b99.

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, 2 (March 2014), 165–213. https://doi.org/doi:10.1017/S1471068412000373 Published online on 30 October 2012.

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, 197–208. https://doi.org/10.1145/1273920.1273947

Magnus Madsen, Ming-Ho Yee, and Ondřej Lhoták. 2016. From Datalog to Flix: A Declarative Language for Fixed Points on Lattices. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, California, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 194–208. https://doi.org/10.1145/2908080.2908096

André Pacak and Sebastian Erdweg. 2022. Functional Programming with Datalog. In *36th European Conference on Object-Oriented Programming (ECOOP 2022) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 222)*, Karim Ali and Jan Vitek (Eds.). Schloss Dagstuhl—Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 7:1–7:28. https://doi.org/10.4230/LIPIcs.ECOOP.2022.7

Simon L. Peyton Jones. 1992. Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-machine. *Journal of Functional Programming* 2, 2 (April 1992), 127–202. https://doi.org/10.1017/S0956796800000319 Also available at https://www.microsoft.com/en-us/research/publication/implementing-lazy-functional-languages-on-stock-hardware-the-spineless-tagless-g-machine/.

John C. Reynolds. 1972. Definitional Interpreters for Higher-Order Programming Languages. In *Proceedings of the ACM Annual Conference—Volume 2* (Boston, Massachusetts, USA) *(ACM '72)*. Association for Computing Machinery, New York, NY, USA, 717–740. https://doi.org/10.1145/800194.805852

J. A. Robinson. 1965. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM* 12, 1 (Jan. 1965), 23–41. https://doi.org/10.1145/321250.321253

Amr Sabry and Matthias Felleisen. 1992. Reasoning about Programs in Continuation-Passing Style. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming, LFP '92.* ACM.

Vijay A. Saraswat and Martin C. Rinard. 1990. Concurrent Constraint Programming. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '90)*, Frances E. Allen (Ed.). ACM Press, 232–245. https://doi.org/10.1145/96709.96733

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, 216–225. https://doi.org/10.1145/224164.224208

Manfred Schmidt-Schauß and Elena Machkasova. 2008. A Finite Simulation Method in a Non-deterministic Call-by-Need Lambda-Calculus with Letrec, Constructors, and Case. In *19th International Conference on Rewriting Techniques and Applications (RTA '08) (LNCS 5117)*. Springer, Berlin, Heidelberg, 321–335. https://doi.org/10.1007/978-3-540-70590-1_22

Gert Smolka and Prakash Panangaden. 1985. *FRESH: A Higher-Order Language with Unification and Multiple Results*. Technical Report TR 85-685. Cornell University, Ithaca, New York, USA. https://hdl.handle.net/1813/6525

Guy Lewis Steele Jr. 1978. *Rabbit: A Compiler for Scheme*. Technical Report 474. Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, USA. https://web.archive.org/web/20211108071621/https://dspace.mit.edu/bitstream/handle/1721.1/6913/AITR-474.pdf Master's Dissertation.

David H D Warren, Luis M. Pereira, and Fernando Pereira. 1977. Prolog - The Language and Its Implementation Compared with Lisp. In *Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages* (Rochester, New York, USA). Association for Computing Machinery, New York, NY, USA, 109–115. https://doi.org/10.1145/800228.806939

Received 2023-03-01; accepted 2023-06-27