The Verse Calculus: a Core Calculus for Functional Logic Programming

LENNART AUGUSTSSON, Epic Games, Sweden
JOACHIM BREITNER
KOEN CLAESSEN, Epic Games, Sweden
RANJIT JHALA, Epic Games, USA
SIMON PEYTON JONES, Epic Games, United Kingdom
OLIN SHIVERS, Epic Games, USA
GUY STEELE, 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 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.

Additional Key Words and Phrases: confluence, declarative programming, functional programming, lambda calculus, logic programming, rewrite rules, skew confluence, unification, Verse calculus, Verse language

1 INTRODUCTION

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

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

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

Functional logic programming has a long history and a rich literature [Antoy and Hanus 2010]. But it is somewhat tricky for programmers to *reason* about functional logic programs: they must think about logical variables, narrowing, backtracking, Horn clauses, resolution, and the like. This contrasts with functional programming, where one can say "just apply rewrite rules, such as 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 the following contributions in this paper. First, we describe a new core calculus for functional logic programming, the Verse calculus or $\mathcal{VC}$ for short (Section 2). As in any functional logic language, $\mathcal{VC}$ supports logical variables, equalities, and choice, but it is distinctive in several ways:

- $\mathcal{VC}$ natively 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

---

In submission, March 2023,
2023. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

is rooted in a first-order world, and addresses higher-order features via an encoding called defunctionalization [Hanus 2013, 3.3].

- All functional logic languages have some notion of "flexible" *vs.* "rigid" variables. $\mathcal{VC}$ offers a new way to address these notions, through the operators **one** (Section 2.5) and **all** (Section 2.6). This enables an elegant economy of concepts: for example, there is just one equality (other languages may have a suspending equality and a narrowing equality), and conditional expressions are driven by failure rather than booleans (Section 2.5).
- *Choice and determinism.* Choice is a fundamental feature of all functional logic languages. In $\mathcal{VC}$, choice is expressed in the syntax of the term ("laid out in space") rather than, as is more typical, handled by non-deterministic rewrites and backtracking ("laid out in time"). This makes $\mathcal{VC}$ completely *deterministic*, unlike most functional logic languages which are non-deterministic by design (Section 6.1).

As always with a calculus, the idea is that $\mathcal{VC}$ distills the essence of functional logic programming. Each construct does just one thing, and $\mathcal{VC}$ cannot be made smaller without losing key features. We believe that it is possible to use $\mathcal{VC}$ as the compilation target for a variety of functional logic languages such as Curry [Hanus et al. 2016]. We are ourselves working on Verse, a new general purpose programming language, built directly on $\mathcal{VC}$; indeed, our motivation for developing $\mathcal{VC}$ is practical rather than theoretical. No single aspect of $\mathcal{VC}$ is unique, but we believe that their combination is particularly harmonious and orthogonal. We discuss the related work in Section 6, and design alternatives in Section 5.

Our second contribution is to equip $\mathcal{VC}$ with a *small-step term-rewriting semantics* (Section 3). We said that the lambda calculus is a subset of $\mathcal{VC}$, so it is natural to give its semantics using rewrite rules, just 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 and unification into the term itself, and replaces non-deterministic search with a (deterministic) tree of successful results. For example, the expression above can be rewritten as follows[1]:

$$
\begin{array}{llll}
\multicolumn{4}{l}{\exists x\, y\, z.\; x = \langle y, 3\rangle;\; x = \langle 2, z\rangle;\; y} \\
\longrightarrow\{\textsc{subst}\} & \exists x\, y\, z.\; \langle 2, z\rangle = \langle y, 3\rangle;\; x = \langle 2, z\rangle;\; y & \longrightarrow\{\textsc{eqn-elim}\} & \exists y\, z.\; \langle 2, z\rangle = \langle y, 3\rangle;\; y \\
\longrightarrow\{\textsc{u-tup}\} & \exists y\, z.\; 2 = y;\; z = 3;\; y & \longrightarrow\{\textsc{eqn-elim}\} & \exists y.\; 2 = y;\; y \\
\longrightarrow\{\textsc{hnf-swap}\} & \exists y.\; y = 2;\; y & \longrightarrow\{\textsc{subst}\} & \exists y.\; y = 2;\; 2 \\
\longrightarrow\{\textsc{eqn-elim}\} & 2 & &
\end{array}
$$

Rules may be applied anywhere they match, 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.

Syntax

Integers	k
Variables	x, y, z, f, g
Programs	$p ::= \textbf{one}\{e\}$ where $\text{fvs}(e) = \emptyset$
Expressions	$e ::= v \mid eq;\, e \mid \exists x.\, e \mid \textbf{fail} \mid e_1 \,\|\, e_2 \mid v_1\, v_2 \mid \textbf{one}\{e\} \mid \textbf{all}\{e\}$
	$eq ::= e \mid v = e$ 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	$op ::= \textbf{gt} \mid \textbf{add}$

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

Parentheses may be used freely to aid readability and override default precedence.

Desugaring

$e_1 + e_2$	means	$\textbf{add}\langle e_1, e_2 \rangle$	
$e_1 > e_2$	means	$\textbf{gt}\langle e_1, e_2 \rangle$	
$\exists x_1\, x_2 \cdots x_n.\, e$	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, $n \geqslant 0$
$\textbf{if}\ (\exists x_1 \cdots x_n.\, e_1)\ \textbf{then}\ e_2\ \textbf{else}\ e_3$	means	$(\textbf{one}\{(\exists x_1 \cdots x_n.\, e_1;\; \lambda \langle \rangle.\, e_2) \,\|\, (\lambda \langle \rangle.\, e_3)\})\langle \rangle$	

$\text{fvs}(e)$ means the free variables of e; in $\mathcal{VC}$, λ and $\exists$ are the only binders.

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

2 THE VERSE CALCULUS, INFORMALLY

We begin by presenting the Verse calculus, $\mathcal{VC}$, informally. We will 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 includes 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.

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

- 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*, $(v_1\ v_2)$, but the desugaring rules in Fig. 1 show how to desugar more applications $(e_1\ e_2)$. This ANF-like normalization is not fundamental; it simply reduces the number of rewrite rules we need. The desugaring rules are more suggestive than precise; we aim to be precise about $\mathcal{VC}$ but less so about the source language.

Modulo this desugaring, every lambda calculus term is a $\mathcal{VC}$ term, and has the same semantics. Just like the lambda calculus, $\mathcal{VC}$ is untyped; adding a type system is an excellent goal, but is the subject of another paper.

Expressions also include two other key collections of constructs: logical variables and 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 **let** $x = e_1$ **in** e_2 in a conventional functional language can be written $\exists x.\ x = e_1;\ e_2$ in $\mathcal{VC}$. 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 only appear to the left of a "; " (see eq in Fig. 1). The desugaring rules in Fig. 1 rewrite a general equation $e_1 = e_2$ into 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 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][2], and is at the heart of our mantra "just solve the equations."

2.2 Choice

In conventional functional programming, an expression evaluates to a single value. In contrast, a $\mathcal{VC}$ expression evaluates to 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 instead give a name to that choice, and then we can put it in the pair, thus: $\exists x.\ x = (7 \mid 5);\ \langle 3, x\rangle$. 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:

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

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:

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

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

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:

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

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 value for some logical variables, *and propagate those values to occurrences in that branch (only).* Occurrences of that variable outside the choice are unaffected. We call this *local propagation.* This local-propagation rule would allow us to reason thus:

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

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

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

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 $(4 \,\|\, 8)$.

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:

$$\begin{array}{l} append\ [\,]\qquad ys = ys \\ append\ (x : xs)\ ys = x : append\ xs\ ys \end{array}$$

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):

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

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 $\exists zs.\ append\langle zs, single\rangle = single;\ zs$. The call to *append* expands into a choice:

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

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

This example also illustrates that $\mathcal{VC}$ allows an 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 3.8).

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

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 Icon's lead, see Section 6.6. 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 $\exists x\, y.\ y = (\textbf{if}\ (x = 0)\ \textbf{then}\ 3\ \textbf{else}\ 4);\ x = 7$. 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:

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

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[4], which wraps the **then** and **else** branches in a thunk[5]:

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

The argument of **one** reduces to either $(\lambda\langle\rangle.\ e_2) \mid (\lambda\langle\rangle.\ e_3)$ 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.

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 $t(3)$ reduces to **fail**.

What if we apply a tuple to a choice, thus $\langle 10, 27, 32\rangle(1 \mid 0 \mid 1)$? First we must desugar the application to the form $(v_1\ v_2)$, because that is all $\mathcal{VC}$ permits (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 **all**$\{e\}$ reduces to the tuple $\langle v_1, \cdots, v_n\rangle$;

[4] The translation in the figure also allow variables bound in the condition to scope over the **then** branch.

[5] Using thunks for the branches of a conditional is another very old idea; for example, see [Steele Jr. 1978, p. 54].

$\textbf{all}\{v\}$ produces the singleton tuple $\langle v \rangle$; and $\textbf{all}\{\textbf{fail}\}$ 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 actually rewrites to:

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

which (as we will see in Section 3) simplifies to just $(10 \mid 27 \mid 32)$. So **all** allows a choice to be reified into a tuple; and $(\exists i.\ t(i))$ allows a tuple to be turned back into a choice. 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 $(\textbf{all}\{e\})(0)$ instead of $\textbf{one}\{e\}$? 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, $\textbf{one}\{1 \mid loop\langle\rangle\}$ can reduce to 1, while $(\textbf{all}\{1 \mid loop\langle\rangle\})(0)$ 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.*, $flatMap\langle(\lambda x.\ x \mid x+10), \langle 2,3\rangle\rangle$ reduces to $\langle 2, 12, 3, 13\rangle$, whereas *map* keeps the choices. For example:

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

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: $fcn\langle 88, 1, 99, 2\rangle$.

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

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

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

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:

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

so that the pattern depends on the value of a.

Desugaring

$$f(x) := e \quad \text{means} \quad f := \lambda x.\, e$$
$$f\langle x, y\rangle := e \quad \text{means} \quad f := \lambda\langle x, y\rangle.\, e$$

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

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

2.8 for loops

The expression $\textbf{for}(e_1)\ \textbf{do}\ e_2$ 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[6] in that variables bound in e_1 also scope over e_2. So, for example, $\textbf{for}(x := 2 \mid 3 \mid 5)\ \textbf{do}\ (x + 1)$ 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, $\textbf{for}(x := 2 \mid 3 \mid 5;\ x > 2)\ \textbf{do}\ (x + 1)$ reduces to $\langle 4, 6\rangle$. Nested iteration in a **for** works as expected and requires nothing special. So, $\textbf{for}(\exists x\, y.\, x = 10 \mid 20;\ y = 1 \mid 2 \mid 3)\ \textbf{do}\ (x + y)$ 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 $\textbf{all}\{e_1;\ e_2\}$ 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):

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

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, $\textbf{for}(x := 10 \mid 20)\ \textbf{do}\ (x \mid x + 1)$ 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, $\textbf{for}(\exists i\, x.\, x = t(i))\ \textbf{do}\ (x + i)$ 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.

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

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

have such a denotational semantics under development, which we offer for completeness in Appendix E, 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 typically describe how a (heap, expression, stack) configuration evolves, one small step at a time (*e.g.*, [Peyton Jones 1992]). The difficulty is that the description is now so low level that it is again hard to explain to programmers.
- *A rewrite semantics* steers between these two extremes. For example, Ariola *et al.*'s "A call-by-need lambda calculus" [Ariola et al. 1995] shows how to give the semantics of a call-by-need language as a set of rewrite rules. The great advantage of this approach is that it is 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 3.8).
- The rules can be applied by programmers to reason about what their program does, and by compilers to transform (and hopefully optimise) the program.
- There is no "magical rewriting" (Section 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, rule APP-ADD should be familiar: it simply rewrites an application of **add** to integer constants. For example $\textbf{add}\langle 3, 4\rangle \longrightarrow 7$. Rules APP-GT and APP-GT-FAIL are more interesting: $\textbf{gt}\langle k_1, k_2\rangle$ 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).

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 $y \notin \mathrm{fvs}(v)$.

Application:

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

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	$hnf_1 = hnf_2;\ e$	$\longrightarrow \textbf{fail}$	if U-LIT, U-TUP do not match
U-OCCURS	$x = V[\,x\,];\ e$	$\longrightarrow \textbf{fail}$	if $V \neq \Box$
SUBST	$X[\,x = v;\ e\,]$	$\longrightarrow (X\{v/x\})[\,x = v;\ e\{v/x\}\,]$	if $x \in \text{fvs}(X, e)$, $x \notin \text{fvs}(v)$, and $(v = y \implies x \prec y)$
HNF-SWAP	$hnf = x;\ e$	$\longrightarrow x = hnf;\ e$	
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 $x \notin \text{fvs}(e)$
EQN-ELIM	$\exists x.\, X[\,x = v;\ e\,]$	$\longrightarrow X[\,e\,]$	if $x \notin \text{fvs}(X[\,v;\ e\,])$
FAIL-ELIM	$X[\,\textbf{fail}\,]$	$\longrightarrow \textbf{fail}$	if $X \neq \Box$

Normalization:

EXI-FLOAT$^\alpha$	$X[\,\exists x.\, e\,]$	$\longrightarrow \exists x.\, X[\,e\,]$	if $X \neq \Box$, $x \notin \text{fvs}(X)$
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	$\textbf{one}\{\textbf{fail}\}$	$\longrightarrow \textbf{fail}$	
ONE-VALUE	$\textbf{one}\{v\}$	$\longrightarrow v$	
ONE-CHOICE	$\textbf{one}\{v \mid e\}$	$\longrightarrow v$	
ALL-FAIL	$\textbf{all}\{\textbf{fail}\}$	$\longrightarrow \langle\rangle$	
ALL-VALUE	$\textbf{all}\{v\}$	$\longrightarrow \langle v\rangle$	
ALL-CHOICE	$\textbf{all}\{v_1 \mid \cdots \mid v_n\}$	$\longrightarrow \langle v_1, \cdots, v_n\rangle$	
CHOOSE-R	$\textbf{fail} \mid e$	$\longrightarrow e$	
CHOOSE-L	$e \mid \textbf{fail}$	$\longrightarrow e$	
CHOOSE-ASSOC	$(e_1 \mid e_2) \mid e_3$	$\longrightarrow e_1 \mid (e_2 \mid e_3)$	
CHOOSE	$SX[\,CX[\,e_1 \mid e_2\,]\,]$	$\longrightarrow SX[\,CX[\,e_1\,] \mid CX[\,e_2\,]\,]$	if $CX \neq \Box$

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

Fig. 4. The syntax of contexts

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, 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 on two different head-normal forms (see Fig. 1 for the syntax of *hnf*). Note in particular that unification fails if you attempt to unify a lambda with any other value, including itself (see Section 4.3).

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

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

$$\textsc{subst} \quad X[\,x{=}v;\ e\,] \quad\longrightarrow\quad (X\{v/x\})[\,x{=}v;\ e\{v/x\}\,] \quad \text{if } x \in \text{fvs}(X, e),\ x \notin \text{fvs}(v) \text{ and } v = y \implies x \prec y$$

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 $X\{v/x\}$, but X will have no bindings to be avoided). 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 $(y{=}x+1;\ (x{=}3;\ z{=}x+3))$ to $(y{=}x+1;\ (x{=}3;\ z{=}3+3))$, using $X = (\Box;\ z{=}x+3)$, but that leaves an occurrence of x in $(y{=}x+1)$. When there are no remaining occurrences of x we may eliminate the binding: see Section 3.5.

- The side condition $x \notin \text{fvs}(v)$ in SUBST prevents infinite substitution, while $x \in \text{fvs}(X, e)$ ensures that there is at least one occurrence to substitute. The other side condition will be explained next, when we discuss VAR-SWAP.

3.3 Swapping and binding order

Rules HNF-SWAP helps SUBST to fire by putting the variable on the left. Rule VAR-SWAP is trickier. Consider this example where a and b are bound further out, perhaps by lambdas. It can rewrite in two different ways:

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

$\longrightarrow\{\text{SUBST}\}$	$\exists x.\ x = \langle a \rangle;\ \langle a \rangle = \langle b \rangle;\ \langle a \rangle$	$\longrightarrow\{\text{SUBST}\}$	$\exists x.\ \langle b \rangle = \langle a \rangle;\ x = \langle b \rangle;\ \langle b \rangle$
$\longrightarrow\{\text{U-TUP}\}$	$\exists x.\ x = \langle a \rangle;\ a = b;\ \langle a \rangle$	$\longrightarrow\{\text{U-TUP}\}$	$\exists x.\ b = a;\ x = \langle b \rangle;\ \langle b \rangle$
$\longrightarrow\{\text{EQN-ELIM}\}$	$a = b;\ \langle a \rangle$	$\longrightarrow\{\text{EQN-ELIM}\}$	$b = a;\ \langle b \rangle$
$\longrightarrow\{\text{SUBST}\}$	$a = b;\ \langle b \rangle$	$\longrightarrow\{\text{SUBST}\}$	$b = a;\ \langle a \rangle$

Each column is a reduction sequence starting from the same common term at the top; the two sequences differ when it comes to which equation for x is chosen for SUBST in the first step. As you can see, they conclude with two terms that are "obviously" the same, but which are syntactically different. Rule VAR-SWAP allows them to be brought together, so that the unification rules are syntactically confluent. Rule SEQ-SWAP is needed for a similar reason. Consider this example:

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

$\longrightarrow\{\text{SUBST}\}$	$c = a;\ a = b;\ a$	$\longrightarrow\{\text{SUBST}\}$	$b = a;\ c = b;\ b$
$\longrightarrow\{\text{VAR-SWAP}\}$	$c = a;\ b = a;\ a$	$\longrightarrow\{\text{SUBST}\}$	$b = a;\ c = a;\ a$

Again, the concluding terms of the two columns are "obviously" the same, because they differ only in the order of the equations $(b = a)$ and $(c = a)$; SEQ-SWAP allows them to be brought together, and makes explicit our intuition that order of equations $(x = v)$ does not matter.

Next we study the mysterious $x \prec y$ side condition in VAR-SWAP, and similar ones in SUBST and SEQ-SWAP. In the overall proof of confluence, it turns out to be very helpful if the unification rules are *terminating* (see Section 4.3). To achieve this, VAR-SWAP fires on $y = x$ *only if x is bound inside y*, 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; and the side condition $(v = y \implies x \prec y)$ on SUBST prevents the rule from firing until VAR-SWAP has done its work.

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.

3.4 Local substitution

Consider this (extremely) tricky term: $\exists x.\ x = \textbf{if}\ (x = 0;\ x > 1)\ \textbf{then}\ 33\ \textbf{else}\ 55$. What should this do? At first you might think it was stuck—how can we simplify the **if** when its condition mentions x, which is not yet defined? But in fact, rule SUBST allows us to substitute *locally* in any X-context surrounding the equation $(x = 0)$ thusly:[7]

$$\begin{array}{ll}
\exists x.\ x = \textbf{if}\ (x = 0;\ x > 1)\ \textbf{then}\ 33\ \textbf{else}\ 55;\ x & \\
\longrightarrow\{\text{SUBST}\} & \exists x.\ x = \textbf{if}\ (x = 0;\ 0 > 1)\ \textbf{then}\ 33\ \textbf{else}\ 55;\ x \\
\longrightarrow\{\text{APP-GT-FAIL,FAIL-ELIM}\} & \exists x.\ x = \textbf{if fail then}\ 33\ \textbf{else}\ 55;\ x \\
\longrightarrow\{\text{simplify } \textbf{if}\}\ \exists x.\ x = 55;\ x & \longrightarrow\{\text{SUBST}\}\ \exists x.\ x = 55;\ 55 \quad \longrightarrow\{\text{EQN-ELIM}\}\ 55
\end{array}$$

[7]Here and elsewhere we rewrite terms that have not been fully desugared, but that is just an expository aid; formally, the rules apply only to programs in the language of Fig. 1.

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

3.5 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 context surrounding a **fail**. Note that none of these rules, except FAIL-ELIM, discard an unevaluated expression, because that expression might fail and we don't want to "lose" that failure (see Section 3.7). The exception is FAIL-ELIM, which propagates failure.

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 then SEQ-ASSOC 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 3.8).

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 $(y=((x=3;\ x+5)\ \|\ (x=4));\ \langle x+1, y\rangle)$. 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.

3.6 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*:

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

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]:

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

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

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

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 approaches that eliminate choice by non-deterministically picking one branch or the other, which immediately gives up confluence.

3.7 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, everything is eventually evaluated, but functions can run before their arguments have a value. Consider a function call $f(e)$, where e is not a value. In $\mathcal{VC}$, applications are in administrative normal form (ANF), so we must actually write $\exists x.\, x = e;\ f(x)$. This expression will not return a value until e reduces to a value: that is, everything is eventually evaluated. But even so, $f(x)$ can proceed to β-reduce (Section 3.1), assuming we know the definition of f.

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

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

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

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

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

Here, y does not get a value until x is known; but x does not get its value (in this case 7) until f is called. Without lenience this program would be stuck.

However, moving to *laziness* seems problematic. For example, consider: $\exists x.\, x = \mathit{wombat}\langle\rangle;\ 3$. In a lazy language this expression would yield 3, but in $\mathcal{VC}$, everything is evaluated, and the expression will not return a value until $\mathit{wombat}\langle\rangle$ converges. There is a good reason for this choice: $\mathit{wombat}\langle\rangle$

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

might fail, and we should not return 3 until we know there is no failure. With laziness, we could easily lose confluence.

3.8 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 only fires 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 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 *good* if it guarantees to terminate if there is *any* terminating sequence of reductions; *i.e.*, if any path terminates with a value, then a good evaluation strategy will terminate with that same value[9]. For example, in the pure lambda calculus, *normal-order reduction*, sometimes called *leftmost-outermost reduction*, is a *good* evaluation strategy.

We believe that the same leftmost-outermost strategy is close to being *good* for $\mathcal{VC}$[10]: just repeatedly reduce the leftmost-outermost redex, with some tweaks to avoid infinite application of EXI-SWAP. That is easy in theory, but it is tricky in practice. 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 substitution to rewrite the $(x+y)$ to $(2+3)$; and *now* the $(2+3)$ is a redex. So a reduction may "unlock" a redex far to its left. A major challenge of an implementation is to find the next redex efficiently.

We have several prototype implementations of $\mathcal{VC}$, each involving an abstract machine with a stack, a heap, a bunch of blocked computations, and so on. Exploring this design space is, however, beyond the scope of this paper.

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.

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.

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

[10] We say "close to" being good because we do not yet have a proof; indeed rule FAIL-ELIM may be a bit too powerful.

```
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

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, and they give meaning to programs without committing to a particular evaluation strategy. But then it had better be the case that no matter how the rules are applied, one always obtains the same result! 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*[11] 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 $e \longrightarrow_{\mathcal{R}} e'$ (*a steps to b*) if $(e, e') \in \mathcal{R}$ and $e \twoheadrightarrow_{\mathcal{R}} e'$ (*a reduces to b*) if $(e, e') \in \mathcal{R}^*$. A reduction 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 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:

$$\exists x\, y.\; x = \langle 1, y\rangle;\; y = (\lambda z.\, x);\; x \;\twoheadrightarrow\; \exists y.\; y = (\lambda z.\, \langle 1, y\rangle);\; \langle 1, y\rangle \qquad \text{(1) substitute for } x \text{ first}$$

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

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 3.5). Why not? Because the terms are equivalent only under some kind of graph isomorphism.

We have tackled this problem in three different ways. First, we can simply prohibit recursion, and prove confluence under that restriction (Section 4.2). This is akin to proving confluence for the lambda calculus with **let** but not **letrec**. In $\mathcal{VC}$, excluding recursion is not so simple because $\mathcal{VC}$ has no **letrec**; rather, recursion emerges during execution. For example, is this recursive:

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

$\exists x\, y.\ x = \langle 1, y\rangle;\ f\langle x, y\rangle$? It might be if $f = (\lambda\langle v, w\rangle.\ v = w)$! For *tuples*, we have a simple solution: rule U-OCCURS makes the entire term fail if we get recursion through a tuple. But we cannot do this for lambdas because it leads to non-confluence. Consider $f = (\lambda x.\ const\langle x, f\rangle)$, where $const = (\lambda\langle p, q\rangle.\ p)$. The equation for f *looks* recursive because the RHS mentions f; but if we β-reduce the application of *const*, the occurrence of f disappears.

Thus motivated, we restrict our attention to terms that have *no recursion*:

- A *recursive equation* is an equation of the form $x = V[\lambda y.\ e]$, where $x \in \mathrm{fvs}(e)$, which equates a variable x with a value that contains a lambda in which x is free.
- A term e is *recursive* if it contains a recursive equation.
- A term e is *transitively recursive* if $e \twoheadrightarrow e'$ where e' is recursive.
- A term e has *no recursion* if it is not transitively recursive.

The no-recursion condition is not as severe as it might first appear: it only prohibits recursion through *equations*. 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**.

This approach 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 second 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 4.4, and provide details of our approach to a proof of skew confluence for $\mathcal{VC}$ in Appendix D, including several new lemmas, but we emphasize that the proof of skew confluence is not yet complete.

A third approach 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 that each of our rules be *semantics*-preserving. But, of course, that requires an independent notion of semantics, a direction we sketch in Appendix E.

4.2 Proof of confluence

Our main result is that $\mathcal{VC}$'s reduction rules are confluent for terms with no recursion. We sketch the proof here, with full details in Appendix C (and relevant preliminaries in Appendix B).

THEOREM 4.1 (CONFLUENCE). *The reduction relation in Fig. 3 is confluent for terms with no recursion.*

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 Huet [1980]:

LEMMA 4.2 (COMMUTATIVITY). *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 no-recursion 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 for the gory details.

4.3 Design for confluence

$\mathcal{VC}$ is carefully designed to ensure confluence.

Ensuring that unification terminates. Our proof strategy for the confluence of the unification rules requires that they terminate. The side condition $x \notin \text{fvs}(v)$ in SUBST avoids infinite substitution. If instead we dropped that condition, the following sequence of SUBST reductions would not terminate:

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

Here, each step makes one substitution for x. An exactly analogous example can be made for a lambda value.

Similarly, as we discussed in Section 3.2, rule VAR-SWAP uses the variable-ordering side condition $x \prec y$ to put the equation in a canonical orientation, and thus ensure that the unification rules terminate.

Unifying lambdas. In $\mathcal{VC}$, an attempt to unify two lambdas fails even if the lambdas are semantically identical (rule U-FAIL). Why? Because semantic identity of functions is unimplementable. We cannot instead say that the attempt to unify gets stuck because that leads to non-confluence. Here is an expression that rewrites in two different ways, depending on which 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. Defining unification to fail for lambdas makes both outcomes lead to **fail**, and confluence is restored.

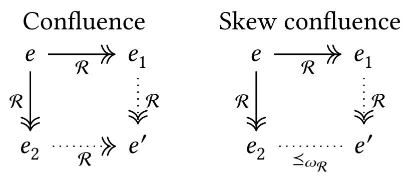
Unifying variables. Note that while U-LIT lets us eliminate equalities on the same literal $k = k$, there is no analogous U-VAR rule to drop equalities on the same variable $x = x$. Perhaps surprisingly, adding that rule would lead to non-confluence. To see why, suppose we had such a U-VAR, and consider 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 fails. Thus, there is 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.

4.4 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.

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

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



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.

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

$\twoheadrightarrow \exists y.\ y = \lambda z.\, \langle 1, y\rangle;\ \langle 1, y\rangle$	$\twoheadrightarrow \exists x.\ x = \langle 1, \lambda z.\, x\rangle;\ x$
$\rightarrow \exists y.\ y = \lambda z.\, \langle 1, y\rangle;\ \langle 1, \lambda z.\, \langle 1, y\rangle\rangle$	$\rightarrow \exists x.\ x = \langle 1, \lambda z.\, x\rangle;\ \langle 1, \lambda z.\, x\rangle)$
$\rightarrow \exists y.\ y = \lambda z.\, \langle 1, y\rangle;\ \langle 1, \lambda z.\, \langle 1, \lambda z.\, \langle 1, y\rangle\rangle\rangle$	$\rightarrow \exists x.\ x = \langle 1, \lambda z.\, x\rangle;\ \langle 1, \lambda z.\, \langle 1, \lambda z.\, x\rangle\rangle$
$\rightarrow \cdots$	$\rightarrow \cdots$

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 a greater amount of 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 Appendix D we show how to adapt the proof strategy of Section 4.2 and Appendix C for skew confluence. To do this we need a new result: if two relations are skew confluent with respect to the same information content function and commute, then their union is also skew confluent. (In fact, it is not required that the two relations fully commute: a slightly weaker precondition suffices.) Using this result, our plan is to (i) define an appropriate information content function for $\mathcal{VC}$ expressions; (ii) prove that all the rewrite rules for $\mathcal{VC}$ are monotonic in this information content function; (iii) prove that the Unification rules (modified to permit recursive substitution) together with the Normalization rules are skew confluent; (iv) prove that this combined set of rules commutes in the necessary way with the rules for Application, Elimination, and Choice (which taken together are already known to be confluent); and (v) then apply our new result to show that the entire set of rewrite rules is skew confluent. At this time steps (iii) and (iv) are incomplete, so we emphasize that we do not yet have a complete proof of skew confluence for $\mathcal{VC}$.

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 Ordering and choices

As we discussed in Section 3.6, 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:

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

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:

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

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:

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

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 directly inspired our denotational semantics (Appendix E.4), where it seems to work rather beautifully.

5.2 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**[12] as $e \;::=\; \cdots \mid \mathbf{split}(e)\{v_1, v_2\}$ and rules:

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

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), 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**:

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

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

[12] The name inspired by Kiselyov et al. [2005].

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

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

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, our formulation appears simpler because we avoid the need for attaching an identifier to each choice with its attendant complications.

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 3.7. 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 interpreter [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, 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 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 3.6, 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 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 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 3.5).

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

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

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

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

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

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

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 function symbols (the names of such functions) are syntactically distinguished from ordinary variables. As Hanus describes, it is possible to translate a higher-order program into a first-order form 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 execute. 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 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 $e_1 \mathbin{|} e_2$ 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. Some operations, especially comparisons, can fail as part of their defined semantics. The expression

if e_1 **then** e_2 **else** e_3 checks to see whether e_1 succeeds; it then produces the values of e_2 (if e_1 succeeded) *or* produces the values of e_3 (if e_1 failed). If e_1 succeeds and then e_2 fails, backtracking does *not* attempt to examine further values from e_1.

- 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;
- 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 between Verse and Icon. 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 an array 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=**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 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 rather syntactic property that is stronger (and hence more delicate and harder to prove) than the contextual equivalence that 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*, *e.g.*, when the condition of an **if** fails, any store effects in the condition must be rolled back. We have preliminary reduction rules for updateable references, see Appendix F.

ACKNOWLEDGMENTS

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

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

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

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

Bernd Braßel and Frank Huch. 2007. On a Tighter Integration of Functional and Logic Programming. In *5th Asian Symposium on Programming Languages and Systems (APLAS 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

Jan Christiansen, Daniel Seidel, and Janis Voigtländer. 2011. An Adequate, Denotational, Functional-Style Semantics for Typed FlatCurry. In *Functional and Constraint Logic Programming: 19th International Workshop, WFLP 2010*, Julio Mariño (Ed.). Springer-Verlag, Berlin, Heidelberg, 119–136. https://doi.org/10.1007/978-3-642-20775-4_7

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

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*. 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. 1976. An Algebraic Interpretation of the $\lambda\beta$K-Calculus; and an Application of a Labelled λ-Calculus. *Theoretical Computer Science* 2, 1 (June 1976), 97–114. https://doi.org/10.1016/0304-3975(76)90009-8

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

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

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/.

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

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.

A EXAMPLE

A complete reduction sequence for a small example can be found in Fig. 6. This example shows how constraining the output of a function call can constrain the argument. While most of the reductions are administrative in nature, these are the highlights: At ① the *swap* function is inlined so that at ② a β-reduction can happen. Step ③ inlines the argument, and ④ does the matching of the tuple. At ⑤ and ⑥ the actual numbers are inlined.

		$swap\langle x, y\rangle := \langle y, x\rangle;\ \exists p.\ swap(p) = \langle 2, 3\rangle;\ p$
	$\longrightarrow$ {DESUGAR}	$\exists swap.\ swap = (\lambda xy.\ \exists x\,y.\ \langle x, y\rangle = xy;\ \langle y, x\rangle);\ \exists p\,t.\ t = swap(p);\ t = \langle 2, 3\rangle;\ p$
①	$\longrightarrow$ {SUBST,EQN-ELIM}	$\exists p\,t.\ t = (\lambda xy.\ \exists x\,y.\ \langle x, y\rangle = xy;\ \langle y, x\rangle)(p);\ t = \langle 2, 3\rangle;\ p$
	$\longrightarrow$ {SUBST,EQN-ELIM}	$\exists p.\ (2, 3) = (\lambda xy.\ \exists x\,y.\ \langle x, y\rangle = xy;\ \langle y, x\rangle)(p);\ p$
②	$\longrightarrow$ {APP-BETA}	$\exists p.\ (2, 3) = (\exists xy.\ xy = p;\ \exists x\,y.\ \langle x, y\rangle = xy;\ \langle y, x\rangle);\ p$
	$\longrightarrow$ {EXI-FLOAT}	$\exists p\,xy.\ (2, 3) = ((xy = p;\ \exists x\,y.\ \langle x, y\rangle = xy;\ \langle y, x\rangle));\ p$
③	$\longrightarrow$ {SUBST,EQN-ELIM}	$\exists p.\ (2, 3) = (\exists x\,y.\ \langle x, y\rangle = p;\ \langle y, x\rangle);\ p$
	$\longrightarrow$ {EXI-FLOAT,EXI-FLOAT}	$\exists p\,x\,y.\ (2, 3) = (\langle x, y\rangle = p;\ \langle y, x\rangle);\ p$
	$\longrightarrow$ {EQN-FLOAT,SEQ-ASSOC}	$\exists p\,x\,y.\ \langle x, y\rangle = p;\ (2, 3) = \langle y, x\rangle;\ p$
	$\longrightarrow$ {HNF-SWAP}	$\exists p\,x\,y.\ p = \langle x, y\rangle;\ (2, 3) = \langle y, x\rangle;\ p$
	$\longrightarrow$ {SUBST,EQN-ELIM}	$\exists x\,y.\ (2, 3) = \langle y, x\rangle;\ \langle x, y\rangle$
④	$\longrightarrow$ {U-TUP,SEQ-ASSOC}	$\exists x\,y.\ 2 = y;\ 3 = x;\ \langle x, y\rangle$
	$\longrightarrow$ {HNF-SWAP}	$\exists x\,y.\ y = 2;\ 3 = x;\ \langle x, y\rangle$
⑤	$\longrightarrow$ {SUBST,EQN-ELIM}	$\exists x.\ 3 = x;\ \langle x, 2\rangle$
	$\longrightarrow$ {HNF-SWAP}	$\exists x.\ x = 3;\ \langle x, 2\rangle$
⑥	$\longrightarrow$ {SUBST,EQN-ELIM}	$\langle 3, 2\rangle$

Fig. 6. A sample reduction sequence

B CONFLUENCE: PRELIMINARIES

B.1 Reduction relations

Definition B.1 (Binary relations). A *binary relation* is a set of pairs of related items; if R is a relation, then we may write $a\,R\,b$ to mean $(a, b) \in R$.

Definition B.2 (Prototype reduction relations and rewrite rules). Let $\widehat{\mathcal{R}}$ be any binary relation on a set of tree-structured *terms*, such as the terms generated by some BNF grammar; we sometimes refer to $\widehat{\mathcal{R}}$ as a *prototype reduction relation.*

Often a prototype reduction relation is specified by a *rewrite rule* of the form $\alpha \longrightarrow \beta$, which indicates that for any substitution σ that consistently instantiates all the metavariables (BNF nonterminals) in α and β, $(\sigma(\alpha), \sigma(\beta))$ is a member of the prototype reduction relation. A prototype reduction relation may also be specified by a set of rewrite rules, in which case the prototype reduction relation is the union of the prototype reduction relations specified by the individual rewrite rules.

Definition B.3 (Reduction relations). A *reduction relation* $\mathcal{R}$ is the *compatible closure* of some prototype reduction relation $\widehat{\mathcal{R}}$; compatibility means that, for any context E and any two terms M and N, if $(M, N) \in \mathcal{R}$ then $(E[M], E[N]) \in \mathcal{R}$. Because most of the relations we consider here are compatible, we find it more convenient to use a hat over relation symbol to indicate that it may *not* be compatible, rather than using some special mark to indicate that a relation *is* compatible or to indicate the taking of a compatible closure.

Definition B.4 (Derived relations). For any relation—but typically for a reduction relation, so we will call it $\mathcal{R}$ here—we write $\mathcal{R}^k$ for the composition of k copies of $\mathcal{R}$ and $\mathcal{R}^*$ for the reflexive and transitive closure of $\mathcal{R}$, *i.e.* $\mathcal{R}^* \equiv \cup_{0 \leqslant k} \mathcal{R}^k$. We write

- $a \longrightarrow_{\mathcal{R}} b$ (*a steps to b*) if $(a, b) \in \mathcal{R}$,
- $a \overset{\epsilon}{\longrightarrow}_{\mathcal{R}} b$ (*a skips to b*) if $a \equiv b$ or $(a, b) \in \mathcal{R}$,
- $a \twoheadrightarrow_{\mathcal{R}} b$ (*a reduces to b*) if $(x, y) \in \mathcal{R}^*$.
- $a \overset{k}{\twoheadrightarrow}_{\mathcal{R}} b$ (*a k-steps to b*) if $(a, b) \in \mathcal{R}^k$, and

Sometimes we use this same notation and terminology with a prototype reduction relation $\widehat{\mathcal{R}}$, thus for example $a \longrightarrow_{\widehat{\mathcal{R}}} b$. In such a case, the arrow indicates rewriting of the entire term a (at the root), and not of some subterm of a.

Definition B.5 (Size). The *size* of a reduction $a \twoheadrightarrow b$ is the smallest i such that $a \overset{i}{\twoheadrightarrow} b$.

Definition B.6 (Normal Forms). A term a is an $\mathcal{R}$*-Normal Form* if there does not exist any b such that $a \longrightarrow_{\mathcal{R}} b$.

For clarity, we will omit the subscript $\mathcal{R}$ when it is clear from the context.

B.2 Confluence

Definition B.7 (Diamond Property). A reduction relation satisfies the *diamond* property if whenever $a \longrightarrow b$ and $a \longrightarrow c$, there is a d such that $b \longrightarrow d$ and $c \longrightarrow d$.

Definition B.8 (Confluence). Two terms b, c can be $\mathcal{R}$*-joined* written $b \downarrow_{\mathcal{R}} c$, if there is a d such that $b \twoheadrightarrow_{\mathcal{R}} d$ and $c \twoheadrightarrow_{\mathcal{R}} d$. A reduction relation $\mathcal{R}$ is *confluent* if whenever $a \twoheadrightarrow_{\mathcal{R}} b$ and $a \twoheadrightarrow_{\mathcal{R}} c$, we have $b \downarrow_{\mathcal{R}} c$.

Definition B.9 (Local Confluence). A reduction relation $\mathcal{R}$ is *locally confluent* if whenever $a \longrightarrow_{\mathcal{R}} b$ and $a \longrightarrow_{\mathcal{R}} c$, we have $b \downarrow_{\mathcal{R}} c$.

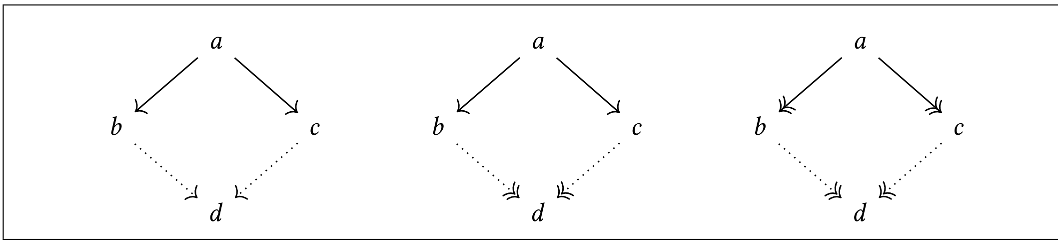


Fig. 7. **Diamond Property (L), Local Confluence (M), and Confluence (R)**

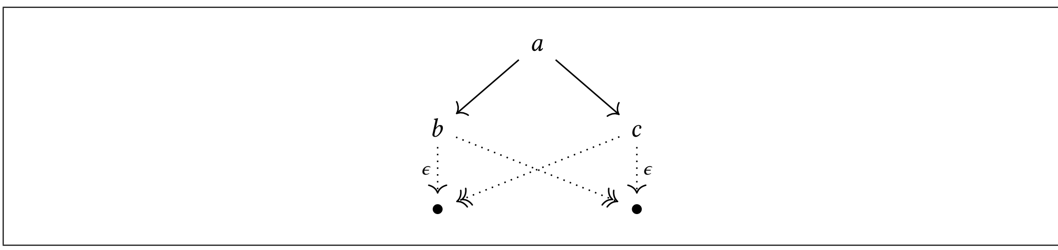


Fig. 8. **Strong Confluence**

Lemma B.10 (Diamond [Barendregt 1984]). *If $\mathcal{R}$ satisfies the diamond property then $\mathcal{R}$ is confluent.*

Lemma B.11 (Unicity [Barendregt 1984]). *If $\mathcal{R}$ is confluent then every term reduces to at most one normal form.*

Lemma B.12 (Closure [Barendregt 1984]). *If $\mathcal{R}$ is confluent then $\mathcal{R}^*$ is confluent.*

Definition B.13 (Noetherian Reduction). A reduction relation $\mathcal{R}$ is *Noetherian* if there is no infinite sequence $a_0 \longrightarrow_{\mathcal{R}} a_1 \longrightarrow_{\mathcal{R}} \ldots \longrightarrow_{\mathcal{R}} a_n \longrightarrow_{\mathcal{R}} \ldots$.

The following result is known as Newman's Lemma [Barendregt 1984; Huet 1980].

Lemma B.14 (Newman's Lemma). *If $\mathcal{R}$ is locally confluent and Noetherian then $\mathcal{R}$ is confluent.*

Definition B.15 (Strong Confluence). A reduction relation is *strongly confluent* if whenever $a \longrightarrow b$ and $a \longrightarrow c$, either $b \twoheadrightarrow c$ or there is a d such that $b \twoheadrightarrow d$ and $c \longrightarrow d$, as shown in Fig. 8, where the ϵ label indicates 0 or 1 step.

Lemma B.16 ([Huet 1980, Lemma 2.5]). *If $\mathcal{R}$ is strongly confluent then $\mathcal{R}$ is confluent.*

B.3 Commutativity

Definition B.17 (Commutativity). A reduction relation R *commutes* with S if for all terms a, b, c such that $a \twoheadrightarrow_R b$ and $a \twoheadrightarrow_S c$ there exists d such that $b \twoheadrightarrow_S d$ and $c \twoheadrightarrow_R d$, as illustrated on the left in Fig. 9.

Definition B.18 (Strong commutativity). A reduction relation R *strongly commutes* with S if for all terms a, b, c such that $a \longrightarrow_R b$ and $a \longrightarrow_R c$ there exists d such that $b \longrightarrow_S d$ and $c \longrightarrow_R d$, as illustrated in the middle in Fig. 9.

Note that if R strongly commutes with itself then, by Definition B.7, R has the diamond property.

Lemma B.19 (Strong-Commutativity). *If R strongly commutes with S then R commutes with S.*

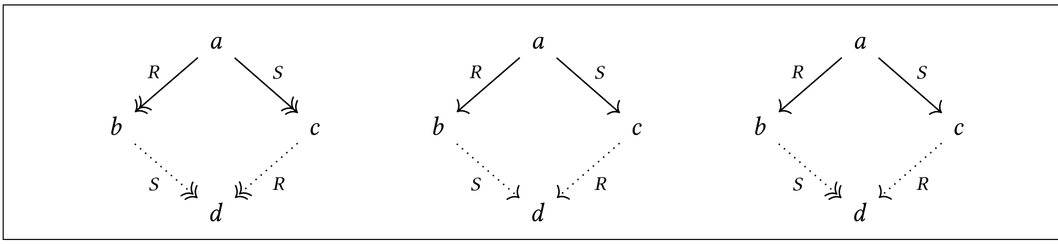
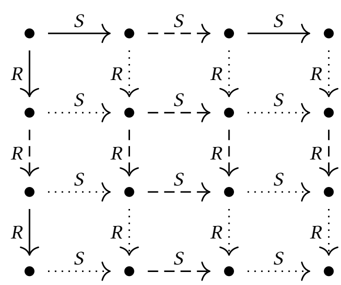


Fig. 9. **Commutativity (L), Strong Commutativity (C), ∗-Commutativity (R)**

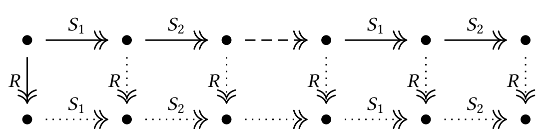
Proof. Via the following "chase" diagram (probably well known?)



□

Lemma B.20 (Union). *If R and S_1 commute and R and S_2 commute then R and $S_1 \cup S_2$ commute.*

Proof. Via the following chase diagram (probably well known?)

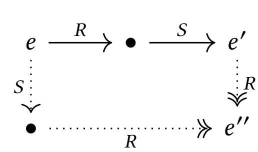


□

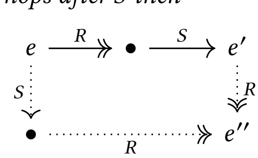
Definition B.21 (Postpones). A reduction relation R *strongly postpones* after S if $e \rightarrow_R \cdot \rightarrow_S e'$ implies $e \twoheadrightarrow_S \cdot \rightarrow_R e'$.

Lemma B.22 ([Hindley 1964]). *If R strongly postpones after S then if $e \twoheadrightarrow_{R \cup S} e'$ then $e \twoheadrightarrow_S \cdot \twoheadrightarrow_R e'$.*

Definition B.23 (Hops). A reduction relation R *hops after* S if $e \rightarrow_R \cdot \rightarrow_S e'$ implies there is an e'' such that $e' \twoheadrightarrow_R e''$ and $e \rightarrow_S \cdot \twoheadrightarrow_R e''$.



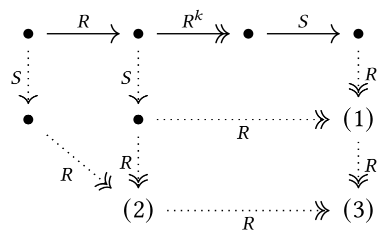
Lemma B.24. *If R is confluent and hops after S then*



Proof. By induction on size of $\twoheadrightarrow_R$.

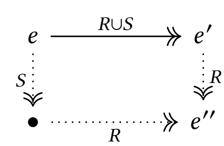
Base case By definition of *hops after.*

Inductive case Assume the lemma for reductions of size upto k and complete the proof via the following diagram where (1) is from the induction hypothesis, (2) is from the definition of hops over, and (3) is from the assumption that R is confluent.

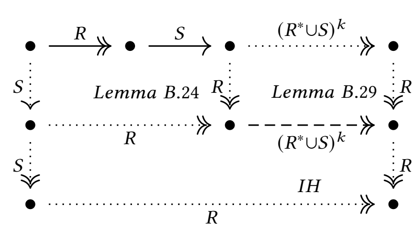


∎

Lemma B.25. *If R is confluent and hops after S then*

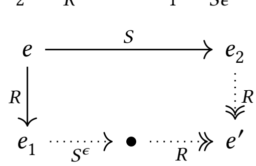


Proof. By induction on the number of S steps in $\twoheadrightarrow_{R\cup S}$, via the following diagram.



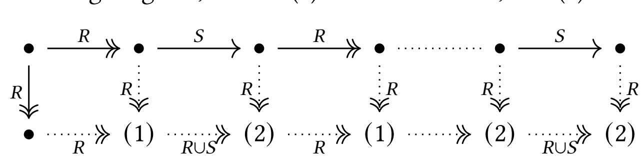
∎

Definition B.26 (half-commutes). A reduction relation R *half-commutes* with S if whenever $e \rightarrow_R e_1$ and $e \rightarrow_S e_2$ there exists e' such that $e_2 \twoheadrightarrow_R e'$ and $e_1 \rightarrow_{S\epsilon} \cdot \twoheadrightarrow_R e'$.



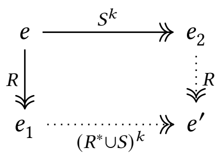
Lemma B.27. *If R is confluent and R half-commutes with S then R commutes with $R \cup S$.*

Proof. Via the following diagram, where: (1) is R is confluent, and (2) is Lemma B.28.

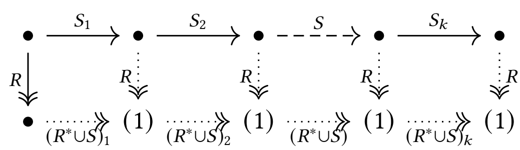


∎

Lemma B.28. *If R is confluent and R half-commutes with S then if $e \twoheadrightarrow_R e_1$ and $e \twoheadrightarrow_S e_2$ then exists e' such that $e_1 \twoheadrightarrow_{R \cup S} e'$ and $e_2 \twoheadrightarrow_R e'$.*

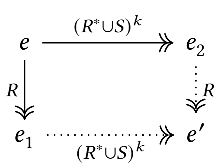


Proof. By repeatedly tiling (1) Lemma B.30 as follows

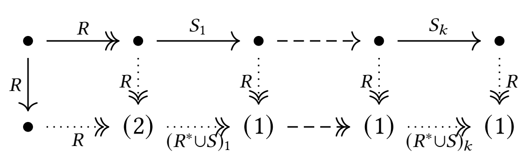


□

Lemma B.29. *If R is confluent and R half-commutes with S then if $e \twoheadrightarrow_R e_1$ and $e \rightarrow_{(R^* \cup S)^k} e_2$ then exists e' such that $e_1 \twoheadrightarrow_{(R^* \cup S)^k} e'$ and $e_2 \twoheadrightarrow_R e'$.*

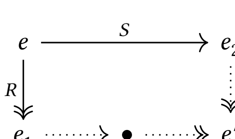


Proof. Similar to Lemma B.28, by repeatedly "tiling" (1) Lemma B.30 and using (2) R is confluent to match the R^* reductions.



□

Lemma B.30. *If R is confluent and R half-commutes with S then if $e \twoheadrightarrow_R e_1$ and $e \rightarrow_S e_2$ then exists e' such that $e_1 \xrightarrow{\epsilon}_S \cdot \twoheadrightarrow_R e'$ and $e_2 \twoheadrightarrow_R e'$.*

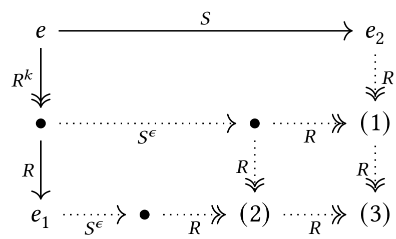


Proof. By induction on the size of $e \twoheadrightarrow_R e_1$.

Base Case Immediate from the definition of half-commutes.

Inductive Case Assume the lemma holds for reductions of size k, complete the proof via the following diagram where (1) is due to the induction hypothesis, (2) is from the definition of

R half-commutes with S and, (3) follows from the fact that R is confluent.



□

B.4 $*$-Commutativity

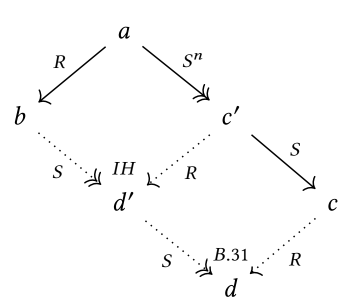
Definition B.31 ($*$-Commutativity). A reduction relation R $*$-commutes with S if for all terms a, b, c such that $a \longrightarrow_R b$ and $a \longrightarrow_S c$ there exists d such that $b \twoheadrightarrow_S d$ and $c \xrightarrow{\epsilon}_R d$ (right in Fig. 9.)

Lemma B.32. *If R $*$-commutes with S then for all a, b, c if $a \longrightarrow_R b$ and $a \twoheadrightarrow_S c$ then there exists d such that $b \twoheadrightarrow_S d$ and $c \xrightarrow{\epsilon}_R d$.*

Proof. By induction on the size of the reduction $a \twoheadrightarrow_S c$.

(Base case) Here c is the same as a, so just pick $d = b$.

(Ind. case) Assume the lemma for reductions of size less than or equal to n. Suppose that $a \xrightarrow{n+1}\!\!\twoheadrightarrow_S c$. Then there exists c' such that $a \xrightarrow{n}\!\!\twoheadrightarrow_S c'$ and $c' \longrightarrow_S c$. The proof is completed by the diagram:



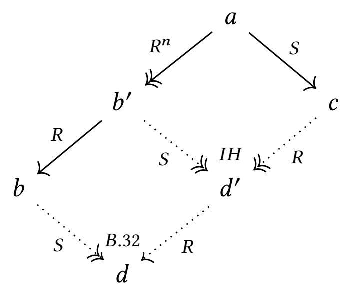
□

Lemma B.33. *If R $*$-commutes with S then for all a, b, c if $a \twoheadrightarrow_R b$ and $a \longrightarrow_S c$ then there exists d such that $b \twoheadrightarrow_S d$ and $c \twoheadrightarrow_R d$.*

Proof. By induction on the size of the reduction $a \twoheadrightarrow_R b$.

(Base case) Here b is the same as a, so just pick $d = c$.

(Ind. case) Assume the lemma for reductions of size less than or equal to n. Suppose that $a \xrightarrow{n+1}\!\!\twoheadrightarrow_R b$. Then there exists some b' such that $a \xrightarrow{n}\!\!\twoheadrightarrow_R b'$ and $b' \longrightarrow_R b$. The proof is completed by the diagram below.



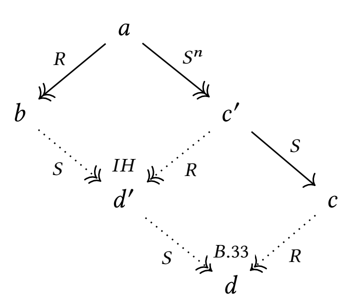
□

Lemma B.34 ($*$-Commutativity). *If R $*$-commutes with S then R commutes with S.*

Proof. By induction on the size of the reduction $a \twoheadrightarrow_S c$.

(Base case) Here c is the same as a, so just pick $d = b$.

(Ind. case) Assume the lemma for reductions of size less than or equal to n. Suppose that $a \xrightarrow{n+1}\!\!\!\!\twoheadrightarrow_S c$. Then there exists some c' such that $a \overset{n}{\twoheadrightarrow}_S c'$ and $c' \rightarrow_S c$. The proof is completed by the diagram below.



□

B.5 Commutativity and Confluence

Lemma B.35 (Commutativity). *If R and S are confluent and commute, then $R \cup S$ is confluent.*

Lemma B.36 (N-Commutativity). *If (i) $\forall 0 \leqslant i \leqslant n$, the reduction relation R_i is confluent, and (ii) $\forall 0 \leqslant i < j \leqslant n$, the reduction relations R_i and R_j commute then $\cup_{i=0}^{n} R_i$ is confluent.*

Proof. By induction on n using Lemma B.35 and Lemma B.20. □

B.6 Confluent Kernels

Definition B.37 (Kernel). A reduction relation S is a *kernel* of R, written $S \preceq R$ if (1) $S \subseteq R$ and (2) If $a \rightarrow_R b$ there exists c such that $a, b \twoheadrightarrow_S c$.

Lemma B.38 (Kernel-Steps). *If $S \preceq R$ and S is confluent and $a \twoheadrightarrow_R b$ then $\exists c.\ a, b \twoheadrightarrow_S c$.*

Proof. By induction on $a \twoheadrightarrow_R b$.

Base Case: $a \equiv b$ so trivially $a, b \twoheadrightarrow_S a$.

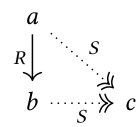
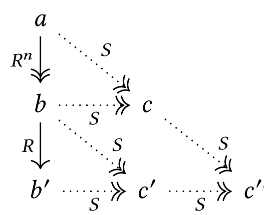


Fig. 10. **S is a kernel of R written $S \preceq R$**

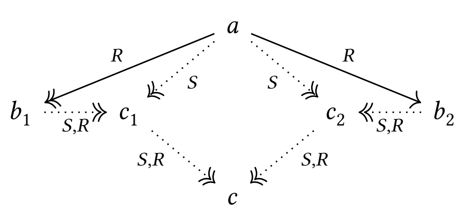
Inductive Case: Assume theorem for $a \xrightarrow{n}\!\!\!\!\rightarrow_R b$. Suppose that $a \xrightarrow{n}\!\!\!\!\rightarrow_R b'$ via $a \xrightarrow{n}\!\!\!\!\rightarrow_R b$ and $b \longrightarrow_R b'$. The proof follows from the diagram below: c is from the IH, c' from $S \preceq R$ and c'' from the confluence of S.



□

Theorem B.39. *Kernel Confluence If $S \preceq R$ and S is confluent, then R is confluent.*

Proof. Suppose that $a \twoheadrightarrow_R b_1$ and $a \twoheadrightarrow_R b_2$. The following diagram shows how to construct c such that $b_1 \twoheadrightarrow_R c$ and $b_2 \twoheadrightarrow_R c$. c_1 (resp. c_2) follows from Lemma B.38 using a and b_1 (resp. b_2). Recall that $S \preceq R$ implies every S reduction is also an R reduction.



□

C CONFLUENCE OF $\mathcal{VC}$: PROOF

Definition C.1 (Reductions). Let $\mathcal{R}$ be the reduction relation defined as the union $\mathcal{U} \cup \mathcal{N} \cup \mathcal{A} \cup \mathcal{G} \cup \mathcal{C}$ of five distinct reduction relations, each of which is defined as the compatible closure of a prototype reduction relation that is in turn defined by rewrite rules in Fig. 3, as follows:

- $\mathcal{U}$ (Unification) is the compatible closure of $\widehat{\mathcal{U}}$, which is the union of the prototype reduction relations specified by rules U-LIT, U-TUP, U-FAIL, U-OCCURS, SUBST, HNF-SWAP, VAR-SWAP, CHOOSE, SEQ-ASSOC, EQN-FLOAT, and SEQ-SWAP.
- $\mathcal{N}$ (Normalization) is the compatible closure of $\widehat{\mathcal{N}}$, which is the union of the prototype reduction relations specified by rules EXI-SWAP, EXI-FLOAT, SUBST (restricted to $x = y$), and VAR-SWAP.
- $\mathcal{A}$ (Application) is the compatible closure of $\widehat{\mathcal{A}}$, which is the union of the prototype reduction relations specified by rules APP-ADD, APP-GT, APP-GT-FAIL, APP-BETA, APP-TUP, and APP-TUP-0.
- $\mathcal{G}$ (Garbage Collection) is the compatible closure of $\widehat{\mathcal{G}}$, which is the union of the prototype reduction relations specified by rules FAIL-ELIM, VAL-ELIM, EXI-ELIM, and EQN-ELIM.

$\mathcal{U}$ and $\widehat{\mathcal{U}}$	$\mathcal{N}$ and $\widehat{\mathcal{N}}$	$\mathcal{A}$ and $\widehat{\mathcal{A}}$	$\mathcal{G}$ and $\widehat{\mathcal{G}}$	$\mathcal{C}$ and $\widehat{\mathcal{C}}$
U-LIT	EXI-SWAP	APP-ADD	FAIL-ELIM	ONE-FAIL
U-TUP	EXI-FLOAT	APP-GT	VAL-ELIM	ONE-VALUE
U-FAIL	SUBST (restricted to $x = y$)	APP-GT-FAIL	EXI-ELIM	ONE-CHOICE
U-OCCURS	VAR-SWAP	APP-BETA	EQN-ELIM	ALL-FAIL
SUBST		APP-TUP		ALL-VALUE
HNF-SWAP		APP-TUP-0		ALL-CHOICE
VAR-SWAP				CHOOSE-L
CHOOSE				CHOOSE-R
SEQ-ASSOC				
EQN-FLOAT				
SEQ-SWAP				

Fig. 11. Division of the rewrite rules shown in Fig. 3 into groups

		$\mathcal{U}$	$\mathcal{N}$	$\mathcal{A}$	$\mathcal{G}$	$\mathcal{C}$
Unification	$\mathcal{U}$	C.19	C.42	C.49	C.50	C.51
Normalization	$\mathcal{N}$		C.31	C.52	C.53	C.54
Application	$\mathcal{A}$			C.55	C.56	C.57
Garbage Collection	$\mathcal{G}$				C.58	C.59
Choice	$\mathcal{C}$					C.60

Fig. 12. Summary of the confluence and commutativity of the reductions in Definition C.1. The lemmas on the diagonal (resp. non-diagonal) entries establish confluence (resp. commutativity) for the respective relation (resp. pairs of relations).

- $\mathcal{C}$ (Choice) is the compatible closure of $\widehat{\mathcal{C}}$, which is the union of the prototype reduction relations specified by rules ONE-FAIL, ONE-VALUE, ONE-CHOICE, ALL-FAIL, ALL-VALUE, ALL-CHOICE, CHOOSE-L, and CHOOSE-R.

Let $\widehat{\mathcal{R}} = \widehat{\mathcal{U}} \cup \widehat{\mathcal{N}} \cup \widehat{\mathcal{A}} \cup \widehat{\mathcal{G}} \cup \widehat{\mathcal{C}}$; then $\mathcal{R}$ may also be described as the compatible closure of $\widehat{\mathcal{R}}$ (because the operation of taking a compatible closure distributes over $\cup$).

These groups correspond approximately to the sub-headings in Fig. 3, *but not precisely*. In particular, some rewrite rules appear in more than one group: VAR-SWAP is used in both $\mathcal{U}$ and $\mathcal{N}$, and SUBST is used in both $\mathcal{U}$ and (in restricted form) $\mathcal{N}$. Moreover, CHOOSE is used in $\mathcal{U}$ but not in $\mathcal{C}$, although it is listed under "Choice" in Fig. 3.

For convenient reference, the five lists of rules are also displayed in tabular form in Fig. 11.

Definition C.2 (Recursive Equations). A *recursive equation* is a term of the form

$$x = V[\lambda y.\, e] \quad \text{where } x \in \mathrm{fvs}(e)$$

where the LHS is a variable and the RHS is a *value* that is or contains a λ in which x occurs free. A term e is *recursive* if it contains a recursive equation. A term e is *transitively recursive* if $e \twoheadrightarrow_{\mathcal{R}} e'$ where e' is recursive. A term e *has no recursion* if it is not transitively recursive.

Our main confluence theorem is as follows:

THEOREM C.3 (CONFLUENCE). *If e has no recursion and $e \twoheadrightarrow_{\mathcal{R}} e_1$ and $e \twoheadrightarrow_{\mathcal{R}} e_2$ then $e_1 \downarrow_{\mathcal{R}} e_2$.*

Notation

a, b, c, d, e	Expressions (syntax in Fig. 1)
Δ	An expression e that has a redex at the root
$e_1 \subset e_2$	The expression e_1 is a strict sub-term of e_2
$e_1 \subseteq e_2$	The expression e_1 is a sub-term of e_2, including e_2 itself
$a \longrightarrow_{\widehat{\mathcal{R}}} b$	a reduces to b via one root-level step of $\mathcal{R}$
$a \longrightarrow_{\mathcal{R}} b$	a reduces to b in one step of $\mathcal{R}$
$a \xrightarrow{\epsilon}_{\mathcal{R}} b$	a reduces to b in zero or one step of $\mathcal{R}$
$a \twoheadrightarrow_{\mathcal{R}} b$	a reduces to b in zero or more steps of $\mathcal{R}$
$a \overset{k}{\twoheadrightarrow}_{\mathcal{R}} b$	a reduces to b in k steps of $\mathcal{R}$

Expression contexts

$$E \quad ::= \quad \square \mid E;\, e \mid v\!=\!E;\, e \mid \exists x.\, E \mid E \,\|\, e \mid e \,\|\, E \mid \mathbf{one}\{E\} \mid \mathbf{all}\{E\}$$
$$\mid \quad E(v) \mid v(E) \mid \langle v_1, \cdots, E, \cdots, v_n \rangle \mid \lambda x.\, E$$

Note: $e_1 \subseteq e_2$ is equivalent to $\exists E.\; E[e_1] \equiv e_2$.

Fig. 13. Summary of notation

Proof. First, we partition $\mathcal{R}$ into the relations $\mathcal{U} \cup \mathcal{N}$, $\mathcal{A}$, $\mathcal{G}$ and $\mathcal{C}$. Next, we show that each of these relations is confluent and pairwise commutative (Fig. 12). Finally, we use Lemma B.36 to prove their union $\mathcal{R}$ is confluent. □

The no-recursion condition is only needed to prove $\mathcal{U}$ is confluent, but we assume it globally for clarity.

C.1 Disjointness, Reduction under, and the Diamond property

In talking about confluence we often speak of two different reduction steps with a common starting point, thus $e \longrightarrow_{\mathcal{R}} e_1$ and $e \longrightarrow_{\mathcal{R}} e_2$. In the first of these there is a sub-term of e, say Δ_1, that is the actual redex; the root of Δ_1 matches some rule in $\mathcal{R}$. Δ_1 is just an ordinary expression, but we use the notation "Δ" to stress that it is the root of a redex (see Fig. 13). Δ_1 is a sub-term of e (or possibly $\Delta = e$), which we write $\Delta_1 \subseteq e$ (again in Fig. 13). Note that $e_1 \subseteq e_2$ is equivalent to saying that there exists some expression context E such that $E[e_1] \equiv e_2$, *i.e.* that e_2 can be decomposed into a context E whose hole is filled by e_1.

Similarly we may identify Δ_2, the redex that is reduced by $e \longrightarrow_{\mathcal{R}} e_2$. Now there are two cases to consider:

(1) Δ_1 is disjoint from Δ_2 in e; or
(2) $\Delta_1 \subseteq \Delta_2$, or $\Delta_2 \subseteq \Delta_1$.

One might wonder if Δ_1 can *overlap* Δ_2, but that is not possible: we are discussing syntax trees, not graphs, and so for distinct Δ_1 and Δ_2, either the root of Δ_1 is a child of the root of Δ_2, or vice versa, or neither.

In the first case (a) we have the diamond property immediately:

Lemma C.4 (Disjoint). *Let $e \equiv \ldots \Delta_1 \ldots \Delta_2 \ldots$ be an expression with two* disjoint *redexes Δ_1 and Δ_2. If $e \longrightarrow \ \ldots \Delta_1' \ldots \Delta_2 \ldots \equiv e_1$ and $e \longrightarrow \ \ldots \Delta_1 \ldots \Delta_2' \ldots \equiv e_2$ then there exists e' such that $e_1 \longrightarrow e'$ and $e_2 \longrightarrow e'$.*

Proof. Trivial: $e' = \ldots \Delta_1' \ldots \Delta_2' \ldots$. □

C.2 Lemmas for Reductions-Under

So to prove the diamond property for a relation $\mathcal{R}$, we should focus attention only on case (b) where the redexes are not disjoint, *i.e.* one occurs under the other. To this end, it suffices to consider the case where one of the reductions is *at the root*, written $e \rightarrow_{\widehat{\mathcal{R}}} e_1$ (see Fig. 13 and Appendix B.1), and the *other* occurs under e *i.e.* is of the form $E[\Delta] \rightarrow_{\mathcal{R}} e_2$ where $e_2 \equiv E[\Delta']$, and $\Delta \rightarrow_{\widehat{\mathcal{R}}} \Delta'$.

Next, we prove a set of "reductions-under" R lemmas that say that if a term e can be (1) reduced using two different rules R and S as $e \rightarrow_R e_R$ and $e_S \rightarrow_S$, such that (2) the redex for the S reduction *occurs under* the redex for the R reduction, then there exists some e' such that e_R (resp. e_S) can be reduced to e' using some number of S (resp. R) reductions.

The lemmas will be used in two ways. First, to show that two *different* relations commute. Second, that a relation (strongly) commutes with *itself*, *i.e.* has the diamond property, and hence is confluent. In each case, we will split cases on which relation is the "outer" reduction and which is the "inner" and then applying the appropriate "reduction-under" lemma for the outer relation, and using Lemma C.4 for the case where the redexes are disjoint.

C.2.1 Application. The following lemma says that if a term $\Delta_{\mathcal{A}}$ is the root of an $\mathcal{A}$ reduction $\Delta_{\mathcal{A}} \rightarrow_{\mathcal{A}} \Delta'_{\mathcal{A}}$ and the $\Delta_{\mathcal{A}}$ additionally contains under it a *subterm* Δ that is the root of some $\mathcal{R}$ reduction $\Delta \rightarrow_{\mathcal{R}} \Delta'$ then it is possible to join the result of the $\mathcal{R}$ and $\mathcal{A}$ reduction at a common term $\Delta''_{\mathcal{A}}$ by executing a single step of the *other* reduction, *i.e.* $\mathcal{A}$ and $\mathcal{R}$ respectively. (Recall that $E[e'] \equiv e$ means that $e' \subseteq e$ *i.e.* e' occurs under or is a sub-term of e).

Lemma C.5 (Under-$\mathcal{A}$). *If $\Delta_{\mathcal{A}} \rightarrow_{\widehat{\mathcal{A}}} \Delta'_{\mathcal{A}}$ and $\Delta_{\mathcal{A}} \equiv E[\Delta]$ and $\Delta \rightarrow_{\widehat{\mathcal{R}}} \Delta'$ then there exists $\Delta''_{\mathcal{A}}$ such that $\Delta'_{\mathcal{A}} \rightarrow_{\mathcal{R}} \Delta''_{\mathcal{A}}$ and $E[\Delta'] \rightarrow_{\widehat{\mathcal{A}}} \Delta''_{\mathcal{A}}$.*

$$\begin{array}{ccc} \Delta_{\mathcal{A}} \equiv E[\Delta] & \xrightarrow{\mathcal{R}} & E[\Delta'] \\ \widehat{\mathcal{A}}\downarrow & & \vdots\ \widehat{\mathcal{A}} \\ \Delta'_{\mathcal{A}} & \underset{\mathcal{R}}{\cdots\cdots\rightarrow} & \Delta''_{\mathcal{A}} \end{array}$$

Proof. Split cases on the rule used in $\widehat{\mathcal{A}}$.

Case: app-beta *i.e.* $\Delta_{\mathcal{A}} \rightarrow_{\mathcal{A}} \Delta'_{\mathcal{A}} \equiv (\lambda x.\ e) v \longrightarrow \exists x.\ x = v;\ e$. If $\Delta \subseteq e$, *i.e.* $\mathcal{R} : e \longrightarrow e'$, then join at $\Delta''_{\mathcal{A}} \equiv \exists x.\ x = v;\ e'$. If $\Delta \subseteq v$, *i.e.* $\mathcal{R} : v \longrightarrow v'$, then join at $\Delta''_{\mathcal{A}} \equiv \exists x.\ x = v';\ e$.

Case: app-tup *i.e.* $\Delta_{\mathcal{A}} \rightarrow_{\mathcal{A}} \Delta'_{\mathcal{A}} \equiv \langle v_0 \ \ldots\ v_n \rangle v \longrightarrow \exists x.\ x = v;\ (x = 0;\ v_0 \,\|\, \ldots \,\|\, x = n;\ v_n)$. If $\Delta \subseteq v_i$, *i.e.* $\mathcal{R} : v_i \longrightarrow v'_i$, then join at $\exists x.\ x = v;\ (x = 0;\ v_0 \,\|\, \ldots\ x = i;\ v'_i\ \ldots \,\|\, x = n;\ v_n)$. If $\Delta \subseteq v$, *i.e.* $\mathcal{R} : v \longrightarrow v'$, then join at $\exists x.\ x = v';\ (x = 0;\ v_0 \,\|\, \ldots \,\|\, x = n;\ v_n)$.

Case: app-tup0 *i.e.* $\Delta_{\mathcal{A}} \rightarrow_{\mathcal{A}} \Delta'_{\mathcal{A}} \equiv \langle\rangle v \longrightarrow$ **fail**. Here, $\Delta \subseteq v$, *i.e.* $\mathcal{R} : v \longrightarrow v'$, then join at $\Delta''_{\mathcal{A}} \equiv$ **fail**.

Case: app-add, app-gt-* In any of the primitive application rules, $\Delta \not\subseteq \Delta_{\mathcal{A}}$.

□

C.2.2 Unification.

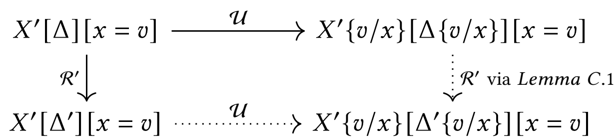
Lemma C.6 (Under-$\mathcal{U}$). *Let $\mathcal{R}' \equiv \mathcal{R} -$ subst $-$ var-swap. If $\Delta_{\mathcal{U}} \rightarrow_{\widehat{\mathcal{U}}} \Delta'_{\mathcal{U}}$ and $\Delta_{\mathcal{U}} \equiv E[\Delta]$ and $\Delta \rightarrow_{\widehat{\mathcal{R}'}} \Delta'$ then there exists $\Delta''_{\mathcal{U}}$ such that $\Delta'_{\mathcal{U}} \twoheadrightarrow_{\mathcal{R}'} \Delta''_{\mathcal{U}}$ and $E[\Delta'] \rightarrow_{\widehat{\mathcal{U}}} \Delta''_{\mathcal{U}}$.*

$$\begin{array}{ccc} \Delta_{\mathcal{U}} \equiv E[\Delta] & \xrightarrow{\mathcal{R}'} & E[\Delta'] \\ \widehat{\mathcal{U}}\downarrow & & \vdots\ \widehat{\mathcal{U}} \\ \Delta'_{\mathcal{U}} & \underset{\mathcal{R}'}{\cdots\cdots\twoheadrightarrow} & \Delta''_{\mathcal{U}} \end{array}$$

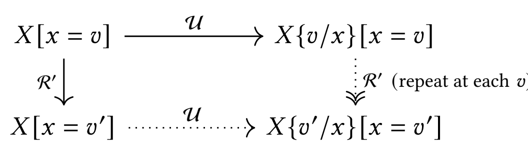
Proof. Split cases on the rule used in $\widehat{\mathcal{U}}$.

Case subst : Here, $\Delta_{\mathcal{U}} \equiv X[x=v]$. Split cases on the occurrence of Δ.

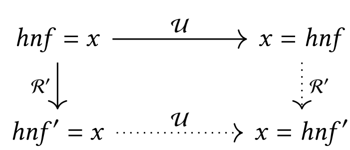
Case $\Delta \subseteq X$, *i.e.* $X \equiv X'[\Delta]$.



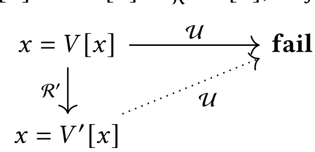
Case $\Delta \subseteq v$, *i.e.* $v \longrightarrow_{\mathcal{R}'} v'$



Case hnf-swap : $\Delta_{\mathcal{U}} \equiv hnf = x$ and $h \longrightarrow_{\mathcal{R}'} h'$, so join at $\Delta''_{\mathcal{U}} \equiv x = hnf'$.



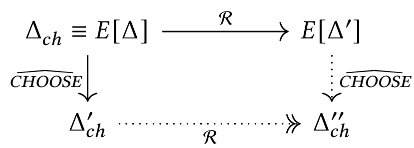
Case u-occurs : $\Delta_{\mathcal{U}} \equiv x = V[x]$ and $V[x] \longrightarrow_{\mathcal{R}'} V'[x]$, so join at $\Delta''_{\mathcal{U}} \equiv$ **fail**.



Case var-swap : Impossible, no $\Delta \subseteq \Delta_{\mathcal{U}}$
Case u-lit : Impossible, no $\Delta \subseteq \Delta_{\mathcal{U}}$
Case u-fail : Join at $\Delta''_{\mathcal{U}} \equiv$ **fail**.
Case u-tup : $\Delta_{\mathcal{U}} \equiv (u_1 \dots u_n) == (v_1 \dots v_n)$.
Case $\Delta \subseteq u_i$ *i.e.* $u_i \longrightarrow_{\mathcal{R}'} u'_i$ Join at $\Delta''_{\mathcal{U}} \equiv u_1 = v_1; \dots u'_i = v_i; \dots u_n = v_n$.
Case $\Delta \subseteq v_j$ *i.e.* $v_j \longrightarrow_{\mathcal{R}'} v'_j$ Join at $\Delta''_{\mathcal{U}} \equiv u_1 = v_1; \dots u_j = v'_j; \dots u_n = v_n$.
Case seq-assoc : $\Delta_{\mathcal{U}} \equiv (eq; e_1); e_2 \longrightarrow eq; (e_1; e_2) \equiv \Delta'_{\mathcal{U}}$. Split cases on where Δ occurs, which as we're precluding subst is either in eq or in e_1 or in e_2.
Case $\Delta \subseteq eq$ *i.e.* $eq \longrightarrow_{\mathcal{R}'} eu'$ Join at $\Delta''_{\mathcal{U}} \equiv eu'; (e_1; e_2)$.
Case $\Delta \subseteq e_1$ *i.e.* $e_1 \longrightarrow_{\mathcal{R}'} e'_1$ Join at $\Delta''_{\mathcal{U}} \equiv eq; (e'_1; e_2)$.
Case $\Delta \subseteq e_2$ *i.e.* $e_2 \longrightarrow_{\mathcal{R}'} e'_2$ Join at $\Delta''_{\mathcal{U}} \equiv eq; (e_1; e'_2)$.
Case Δ spans $(eq; e_1)$ or $(eq; e_1); e_2$ via fail-elim. Join at **fail**.
Case eqn-float : $\Delta_{\mathcal{U}} \equiv v = (eq; e_1); e_2 \longrightarrow eq; (v = e_1; e_2) \equiv \Delta'_{\mathcal{U}}$. Split cases on where Δ occurs, which as we're precluding subst is either in v, eq, e_1 or in e_2.
Case $\Delta \subseteq v$ *i.e.* $v \longrightarrow_{\mathcal{R}'} v'$ Join at $\Delta''_{\mathcal{U}} \equiv eq; (v' = e_1; e_2)$.
Case $\Delta \subseteq eq$ *i.e.* $eq \longrightarrow_{\mathcal{R}'} eu'$ Join at $\Delta''_{\mathcal{U}} \equiv eu'; (v = e_1; e_2)$.
Case $\Delta \subseteq e_1$ *i.e.* $e_1 \longrightarrow_{\mathcal{R}'} e'_1$ Join at $\Delta''_{\mathcal{U}} \equiv eq; (v = e'_1; e_2)$.
Case $\Delta \subseteq e_2$ *i.e.* $e_2 \longrightarrow_{\mathcal{R}'} e'_2$ Join at $\Delta''_{\mathcal{U}} \equiv eq; (v = e_1; e'_2)$.
Case Δ spans $(eq; e_1)$ or $v = (eq; e_1); e_2$ via fail-elim. Join at **fail**.
Case choose : via Lemma C.7.

□

Lemma C.7 (Under-choose). *If $\Delta_{ch} \rightarrow_{\widehat{CHOOSE}} \Delta'_{ch}$ and $\Delta_{ch} \equiv E[\Delta]$ and $\Delta \rightarrow_{\widehat{\mathcal{R}}} \Delta'$ then there exists Δ''_{ch} such that $\Delta'_{ch} \twoheadrightarrow_{\mathcal{R}} \Delta''_{ch}$ and $E[\Delta'] \rightarrow_{\widehat{CHOOSE}} \Delta''_{ch}$.*



Proof. By the definition of choose we have

$$\Delta_{ch} \equiv SX[CX[e_1 \mid e_2]] \longrightarrow SX[CX[e_1] \mid CX[e_2]] \equiv \Delta'_{ch}$$

Split cases on where Δ occurs

Case $\Delta \subseteq e_1$ *i.e.* $e_1 \rightarrow_{\mathcal{R}} e'_1$, so join at $SX[CX[e'_1] \mid CX[e_2]]$.
Case $\Delta \subseteq e_2$ *i.e.* $e_2 \rightarrow_{\mathcal{R}} e'_2$, so join at $SX[CX[e_1 \mid e'_2]]$.
Case $\Delta \subseteq e_1 \mid e_2$ *i.e.* $e_1 \mid e_2 \rightarrow_{\mathcal{R}} e_i$ where e_{3-i} is **fail** so join at $SX[CX[e_i]]$.
Case $\Delta \subseteq CX$ *i.e.* $CX \rightarrow_{\mathcal{R}} CX'$ so join (via *two* $\mathcal{R}$ steps) at $SX[CX'[e_1] \mid CX'[e_2]]$.
Case $\Delta \subseteq CX[e_1 \mid e_2]$ *i.e.* $CX[e_1 \mid e_2] \rightarrow_{\mathcal{R}} CX'[e'_1 \mid e'_2]$, so join at $SX[CX'[e'_1] \mid CX'[e'_2]]$.

□

C.2.3 Normalization.

Lemma C.8 (Under-$\mathcal{N}$). *Let $\mathcal{R}' = \mathcal{R} - \mathcal{N} - \mathcal{U}$. If $\Delta_{\mathcal{N}} \rightarrow_{\widehat{\mathcal{N}}} \Delta'_{\mathcal{N}}$ and $\Delta_{\mathcal{N}} \equiv E[\Delta]$ and $\Delta \rightarrow_{\widehat{\mathcal{R}'}} \Delta'$ then exists $\Delta''_{\mathcal{N}}$ such that $\Delta'_{\mathcal{N}} \rightarrow_{\mathcal{R}'} \Delta''_{\mathcal{N}}$ and $E[\Delta'] \rightarrow_{\widehat{\mathcal{N}}} \Delta''_{\mathcal{N}}$.*

Proof. Split cases on the reduction rule used in $\Delta_{\mathcal{N}} \rightarrow_{\widehat{\mathcal{N}}} \Delta'_{\mathcal{N}}$

Case exi-swap : *i.e.* $\mathcal{N} : \exists x. \exists y. e \longrightarrow \exists y. \exists x. e$. Split cases on the position of Δ.
Case $\Delta \subseteq e$: *i.e.* $e \rightarrow_{\mathcal{R}'} e'$; join at $\exists x. \exists y. e'$.
Case $\Delta \subseteq (\exists y. e)$: *i.e.* y eliminated via an exi-elim or eqn-elim $\exists y. e \rightarrow_{\mathcal{R}'} e'$; join at $\exists x. e'$.
Case $\Delta \subseteq (\exists x. \exists y. e)$: *i.e.* x eliminated via an exi-elim or eqn-elim $\exists x. \exists y. e \rightarrow_{\mathcal{R}'} \exists y. e'$; join at $\exists y. e'$.
Case exi-float : *i.e.* $\mathcal{N} : X[\exists x. e] \longrightarrow \exists x. X[e]$. Split cases on the position of Δ.
Case $\Delta \subseteq e$: *i.e.* $e \rightarrow_{\mathcal{R}'} e'$; join at $\exists x. X[e']$.
Case $\Delta \subseteq (\exists x. e)$: *i.e.* x eliminated via an exi-elim or eqn-elim $\exists x. e \rightarrow_{\mathcal{R}'} e'$; join at $X[e']$.
Case $\Delta \subseteq X$: *i.e.* $X[\exists x. e] \rightarrow_{\mathcal{R}'} X'[\exists x. e']$; join at $\exists x. X'[e']$.
Case subst-var : *i.e.* $\mathcal{N} : X[x = y] \longrightarrow (X\{y/x\})[x = y]$. The only possible position of Δ is $\Delta \subseteq X$ *i.e.* $X^{[}x = y] \rightarrow_{\mathcal{R}'} X'^{[}x = y]$; join at $(X'\{y/x\})[x = y]$.
Case var-swap : *i.e.* $\mathcal{N} : x = y \longrightarrow y = x$. Impossible to have $\Delta \subseteq x = y$.

□

C.2.4 Garbage Collection.

Lemma C.9 (Under-$\mathcal{G}$). *If $\Delta_{\mathcal{G}} \rightarrow_{\widehat{\mathcal{G}}} \Delta'_{\mathcal{G}}$ and $\Delta_{\mathcal{G}} \equiv E[\Delta]$ and $\Delta \rightarrow_{\widehat{\mathcal{R}}} \Delta'$ then there exists $\Delta''_{\mathcal{G}}$ such that $\Delta'_{\mathcal{G}} \xrightarrow{\epsilon}_{\mathcal{R}} \Delta''_{\mathcal{G}}$ and $E[\Delta'] \rightarrow_{\widehat{\mathcal{G}}} \Delta''_{\mathcal{G}}$.*

Proof. Let $\Delta_{\mathcal{G}} \rightarrow_{\widehat{\mathcal{G}}} \Delta'_{\mathcal{G}}$ be the $\mathcal{G}$ redex and split cases on the reduction rule used in the step.

Case val-elim : *i.e.* $\mathcal{G} : v;\ e \longrightarrow e$. Split cases on position of Δ
Case $\Delta \subseteq v$: Join at e.
Case $\Delta \subseteq e$: *i.e.* $e \rightarrow_{\mathcal{R}} e'$; join at e'.
Case $\Delta \subseteq v;\ e$: *i.e.* $v; e \rightarrow_{\text{FAIL-ELIM}}$ **fail** as $e \equiv X[\textbf{fail}]$; join at **fail**.
Case fail-elim : *i.e.* $\mathcal{G} : X[\textbf{fail}] \longrightarrow \textbf{fail}$. Then $X[\textbf{fail}] \rightarrow_{\mathcal{R}} X'[\textbf{fail}]$ hence join at **fail**.

Case exi-elim : *i.e.* $\mathcal{G} : \exists\overline{y}, x, \overline{z}.\ e \longrightarrow \exists\overline{y}, \overline{z}.\ e$; (We can generalize exi-elim to first use a sequence of exi-swap to bring the x binder to the end before applying exi-elim as this does not change the order of the *remaining* binders.) Split cases on position of Δ

Case $\Delta \subseteq e$: *i.e.* $e \rightarrow_{\mathcal{R}} e'$; join at $\exists\overline{y}, \overline{z}.\ e'$.

Case $\Delta \subseteq \exists\overline{y}, x, \overline{z}.\ e$: *i.e.* via exi-swap; join at $\exists\overline{y}, \overline{z}.\ e$.

Case eqn-elim : *i.e.* $\mathcal{G} : \exists x.\ X[\,x = v;\ e\,] \longrightarrow X[\,e\,]$ where $x \notin \text{fvs}(X[\,v;\ e\,])$. (We can generalize exi-elim to first use a sequence of exi-swap to bring the x binder to the end before applying eqn-elim as this does not change the order of the *remaining* binders.) Split cases on position of Δ

Case $\Delta \subseteq v$: *i.e.* $v \rightarrow_{\mathcal{R}} v'$; join at $X[\,e\,]$.

Case $\Delta \subseteq e$: *i.e.* $e \rightarrow_{\mathcal{R}} e'$ (where $\text{fvs}(e') = \text{fvs}(e)$); join at $X[\,e'\,]$.

Case $\Delta \subseteq X$: *i.e.* $X[\,x = v;\ e\,] \rightarrow_{\mathcal{R}} X'[\,x = v;\ e\,]$ (where $\text{fvs}(X') = \text{fvs}(X)$); join at $X'[\,e\,]$.

□

C.2.5 Choice.

Lemma C.10 (Under-C). *If* $\Delta_C \rightarrow_{\widehat{C}} \Delta'_C$ *and* $\Delta_C \equiv E[\Delta]$ *and* $\Delta \rightarrow_{\mathcal{R}} \Delta'$ *then there exists* Δ''_C *such that* $\Delta'_C \xrightarrow{\epsilon}_{\mathcal{R}} \Delta''_C$ *and* $E[\Delta'] \rightarrow_{\widehat{C}} \Delta''_C$.

Proof. Split cases on the rule used in $\Delta_C \rightarrow_{\widehat{C}} \Delta'_C$.

Case one-fail (symmetric all-fail) Impossible as $\Delta \not\subseteq \Delta_C$.

Case one-value : Here $\Delta_C \rightarrow_C \Delta'_C \equiv \textbf{one}\{v\} \longrightarrow v$. Hence $\Delta \subseteq v$ *i.e.* $\mathcal{R} : v \longrightarrow v'$, so join at v'.

Case all-value : Here $\Delta_C \rightarrow_C \Delta'_C \equiv \textbf{all}\{v\} \longrightarrow \langle v \rangle$. Hence $\Delta \subseteq v$ *i.e.* $\mathcal{R} : v \longrightarrow v'$, so join at $\langle v' \rangle$.

Case one-choice : Here $\Delta_C \rightarrow_C \Delta'_C \equiv \textbf{one}\{v \,|\, e\} \longrightarrow v$. If $\Delta \subseteq v$, *i.e.* $\mathcal{R} : v \longrightarrow v'$ then join at v'. If $\Delta \subseteq e$, *i.e.* $\mathcal{R} : e \longrightarrow e'$ then join at v.

Case all-choice : Here $\Delta_C \rightarrow_C \Delta'_C \equiv \textbf{all}\{v_1 \,|\, \ldots \,|\, v_n\} \longrightarrow \langle v_1, \ldots, v_n \rangle$. If $\Delta \subseteq v_i$ ie $\mathcal{R} : v_i \longrightarrow v'_i$ then join at $\langle v_1, \ldots, v'_i, \ldots, v_n \rangle$.

Case choose-l : (symmetric choose-r) Here $\Delta_C \rightarrow_C \Delta'_C \equiv \textbf{fail} \,|\, e \longrightarrow e$. Here, $\Delta \subseteq e$, *i.e.* $\mathcal{R} : e \longrightarrow e'$ so join at e'.

Case choose-assoc : *i.e.* $\Delta_C \rightarrow_C \Delta'_C \equiv (e_1 | e_2) | e_3 \longrightarrow e_1 | (e_2 | e_3)$. Split cases on where Δ occurs.

Case $\Delta \subseteq e_1$, *i.e.* $\mathcal{R} : e_1 \longrightarrow e'_1$ so join at $e'_1 | (e_2 | e_3)$.

Case $\Delta \subseteq e_2$, *i.e.* $\mathcal{R} : e_2 \longrightarrow e'_2$ so join at $e_1 | (e'_2 | e_3)$.

Case $\Delta \subseteq e_3$, *i.e.* $\mathcal{R} : e_3 \longrightarrow e'_3$ so join at $e_1 | (e_2 | e'_3)$.

Case $\Delta \equiv e_1 \,|\, \textbf{fail}$, *i.e.* $\mathcal{R} : e_1 \,|\, \textbf{fail} \longrightarrow e_1$ so join at $e_1 | e_3$.

Case $\Delta \equiv \textbf{fail} \,|\, e_2$, *i.e.* $\mathcal{R} : \textbf{fail} \,|\, e_2 \longrightarrow e_2$ so join at $e_2 | e_3$.

□

C.3 Lemmas for Substitution and Unification

Lemma C.11 (Substitution). *Let* $\mathcal{R}' \equiv \mathcal{R} - \mathcal{U}$. *If* $\Delta \rightarrow_{\widehat{\mathcal{R}'}} \Delta'$ *then* $\Delta\{v/x\} \rightarrow_{\widehat{\mathcal{R}'}} \Delta'\{v/x\}$.

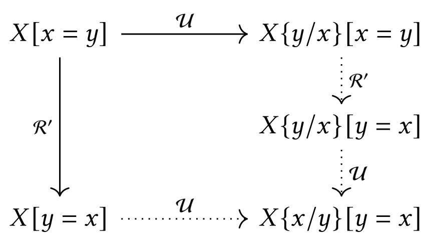
Proof. By induction on the structure of Δ, splitting cases on the reduction rule used and using the fact that e, v, X, CX, SX are all closed under value substitution. □

Lemma C.12 (Subst-Swap). *If* $e \rightarrow_{\textsc{subst}} e_1$ *and* $e \rightarrow_{\textsc{swap}} e_2$ *then exists* e' *such that* $e_1, e_2 \twoheadrightarrow_{\mathcal{U}} e'$.

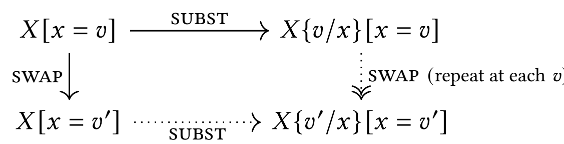
Proof. Let $\Delta_1 \rightarrow_{\textsc{subst}} \Delta'_1$ and $\Delta_2 \rightarrow_{\textsc{swap}} \Delta'_2$ be the respective reducts. Via Lemma C.4 it suffices to consider two cases:

Case swap under subst : *i.e.* $\Delta_2 \subseteq \Delta_1$ Let $\Delta_1 \equiv X[\,x{=}v\,]$; split cases on Δ_2 position.

Case $\Delta_2 \equiv x = v$: via rule VAR-SWAP:

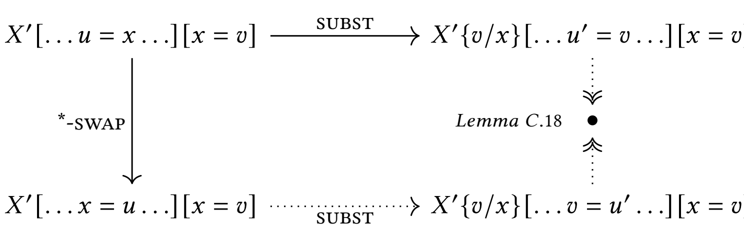


Case $\Delta_2 \subseteq v$: *i.e.* $v \longrightarrow_{\text{SWAP}} v'$.

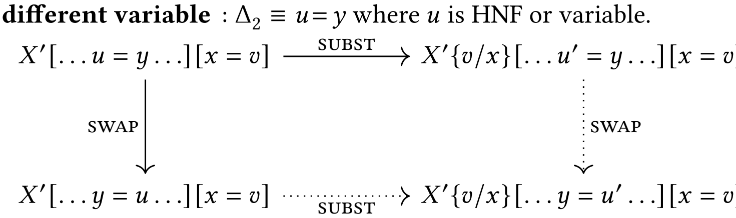


Case $\Delta_2 \subseteq X$: *i.e.* $X \equiv X'[\dots\Delta_2\dots]$. Let $u' \equiv u\{v/x\}$, split cases on SWAP RHS.

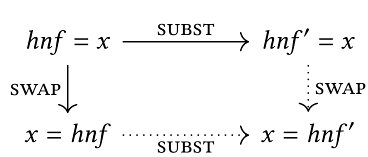
Case same variable : $\Delta_2 \equiv u = x$ where u is HNF or variable.



Case different variable : $\Delta_2 \equiv u = y$ where u is HNF or variable.



Case SUBST under SWAP : *i.e.* $\Delta_1 \subseteq \Delta_2$ Let $\Delta_2 \equiv hnf = x$, so $\Delta_1 \subseteq hnf$, *i.e.* $hnf \longrightarrow_{\text{SUBST}} hnf'$, so join at $x = hnf'$.



□

Definition C.13 (Levels). Let $eq_1 \equiv x_1 = v_1$ and $eq_2 \equiv x_2 = v_2$ be two equations in a term e. We say eq_2 *is under* eq_1 if $eq_2 \subseteq X$ and $X[eq_1] \subseteq e$.

LEMMA C.14 (SUBST-SUBST). *If* $e \longrightarrow_{SUBST} e_1$ *and* $e \longrightarrow_{SUBST} e_2$ *then* $e_1 \downarrow_{\mathcal{U}} e_2$.

PROOF. Suppose that the redex $e \longrightarrow e_i$ is using the equation $eq_i \equiv x_i = v_i$. Split cases on

Case eq_1 is under eq_2 and eq_2 is under eq_1: Lemma C.15 completes the proof.
Case eq_1 is under eq_2 and eq_2 is not under eq_1: Lemma C.16 completes the proof.
Case eq_1 is not under eq_2 and eq_2 is under eq_1: Lemma C.16 completes the proof.

Case eq_1 is not under eq_2 and eq_2 is not under eq_1: The substitutions are disjoint, so Lemma C.4 completes the proof.

□

Lemma C.15 (Subst-Same). *If $e \longrightarrow_{SUBST} e_1$ using eq_1 and $e \longrightarrow_{SUBST} e_2$ using eq_2 such that eq_1 is under eq_2 and eq_2 is under eq_1, then $e_1 \downarrow_{\mathcal{U}} e_2$.*

Proof. As eq_1 is under eq_2 and eq_2 is under eq_1, we have $e \equiv X[x = v_1;\ y = v_2]$, *i.e.* wlog the equations eq_1 and eq_2 are adjacent. Let us split cases on whether $x \equiv y$

Case $x \equiv y$: We join e_1 and e_2 using Lemma C.18 via the context $X' \equiv X\{z/x\}\lceil x = z]$ where z is a fresh variable.

$$\begin{array}{ccc}
X[x = u; x = v] & \xrightarrow{eq_1} & X\{u/x\}[x = u; u = v] \\
\downarrow{\scriptstyle eq_2} & & \vdots \\
X\{v/x\}[v = u; x = v] & \xrightarrow[\cdots]{Lemma\ C.18} & \bullet
\end{array}$$

Case $x \not\equiv y$: Let us split cases on whether x, y appear in $\mathrm{fvs}(u), \mathrm{fvs}(v)$ respectively.

Case $x \notin \mathrm{fvs}(v), y \notin \mathrm{fvs}(u)$:

$$\begin{array}{ccc}
X[x = u; y = v] & \xrightarrow{eq_1} & X\{u/x\}[x = u; y = v] \\
\downarrow{\scriptstyle eq_2} & & \vdots{\scriptstyle eq_2} \\
X\{v/x\}[x = u; y = v] & \xrightarrow[\cdots]{eq_1} & X\{v/x\}[x = u; y = v]
\end{array}$$

Case $x \notin \mathrm{fvs}(v), y \in \mathrm{fvs}(u)$:

$$\begin{array}{ccc}
X[x = u; y = v] & \xrightarrow{eq_1} & X\{u/x\}[x = u; y = v] \\
\downarrow{\scriptstyle eq_2} & & \vdots{\scriptstyle eq_2} \\
X\{v/y\}[x = u\{v/y\}; y = v] & \xrightarrow[\cdots]{eq_1} & X\{u\{v/y\}/x, v/y\}[x = u\{v/y\}; y = v]
\end{array}$$

Case $x \in \mathrm{fvs}(v), y \notin \mathrm{fvs}(u)$: Symmetric to previous case.
Case $x \in \mathrm{fvs}(v), y \in \mathrm{fvs}(u)$: Join at **fail** if u-occurs, else impossible due to no recursion.

□

Lemma C.16 (Subst-Diff). *If $e \longrightarrow_{SUBST} e_1$ using eq_1 and $e \longrightarrow_{SUBST} e_2$ using eq_2 such that eq_1 is not under eq_2 and eq_2 is under eq_1, then $e_1 \downarrow_{\mathcal{U}} e_2$.*

Proof. Here, we have $e \equiv X_1[...X_2[x_2 = v_2]...][x_1 = v_1]$ where the substitution with $x_2 = v_2$ does not affect X_1, x_1, v_1. Split cases on whether $x_1 \equiv x_2$.

Case $x_1 \equiv x_2 \equiv x$: By no recursion we have $x \notin \mathrm{fvs}(v_1), x \notin \mathrm{fvs}(v_2)$. Hence, we can join e_1 and e_2 using Lemma C.17 on the sub-terms $X_2\{v_1/x\}[v_1 = v_2]$ and $X_2\{v_2/x\}[v_1 = v_2]$.

$$\begin{array}{ccccc}
X_1[\ldots X_2[x = v_2]\ldots][x = v_1] & & \xrightarrow{eq_2} & & X_1[\ldots X_2\{v_2/x\}[x = v_2]\ldots][x = v_1] \\
\downarrow{\scriptstyle eq_1} & & & & \vdots{\scriptstyle eq_1} \\
X_1\{v_1/x\}[\ldots X_2\{v_1/x\}[v_1 = v_2]\ldots][x = v_1] & \cdots\twoheadrightarrow & \underset{Lemma\ C.17}{\bullet} & \twoheadleftarrow\cdots & X_1\{v_1/x\}[\ldots X_2\{v_2/x\}[v_1 = v_2]\ldots][x = v_1]
\end{array}$$

Case $x_1 \not\equiv x_2$: Let $v\_1' \equiv v_1\{v_2/x_2\}$ and $v\_2' \equiv v_2\{v_1/x_1\}$. Split cases on whether $x_i \in \mathrm{fvs}(v_{3-i})$.

Case $x_2 \notin \mathsf{fvs}(v_1)$

$$\begin{array}{ccc}
X_1[\ldots X_2[x_2 = v_2]\ldots][x_1 = v_1] & \xrightarrow{eq_2} & X_1[\ldots X_2\{v_2/x_2\}[x_2 = v_2]\ldots][x_1 = v_1] \\
\downarrow{\scriptstyle eq_1} & & \vdots{\scriptstyle eq_1} \\
X_1\{v_1/x_1\}[\ldots X_2\{v_1/x\}[x_2 = v_2']\ldots][x_1 = v_1] & \overset{}{\underset{eq_2}{\dashrightarrow}} & X_1\{v_1/x\}[\ldots X_2\{v_2'/x_2, v_1/x_1\}[x_2 = v_2']\ldots][x_1 = v_1]
\end{array}$$

Case $x_2 \in \mathsf{fvs}(v_1), x_1 \notin \mathsf{fvs}(v_2)$

$$\begin{array}{ccc}
X_1[\ldots X_2[x_2 = v_2]\ldots][x_1 = v_1] & \xrightarrow{eq_2} & X_1[\ldots X_2\{v_2/x_2\}[x_2 = v_2]\ldots][x_1 = v_1] \\
\downarrow{\scriptstyle eq_1} & & \vdots{\scriptstyle eq_1} \\
X_1\{v_1/x_1\}[\ldots X_2\{v_1/x_1\}[x_2 = v_2]\ldots][x_1 = v_1] & & X_1\{v_1/x_1\}[\ldots X_2\{v_1/x_1, v_2/x_2\}[x_2 = v_2]\ldots][x_1 = v_1] \\
 & \underset{eq_2}{\dashrightarrow} & \vdots{\scriptstyle eq_2} \\
 & & X_1\{v_1/x_1\}[\ldots X_2\{v_1'/x_1, v_2/x_2\}[x_2 = v_2]\ldots][x_1 = v_1]
\end{array}$$

Case $x_2 \in \mathsf{fvs}(v_1), x_1 \in \mathsf{fvs}(v_2)$ In this case, we get the below diagram where, since $x_2 \in \mathsf{fvs}(v_2')$, the term $x_2 = v_2'$ either steps to **fail** (and so we can join at **fail**) or the term violates the no recursion assumption.

$$\begin{array}{ccc}
X_1[\ldots X_2[x_2 = v_2]\ldots][x_1 = v_1] & \xrightarrow{eq_2} & X_1[\ldots X_2\{v_2/x_2\}[x_2 = v_2]\ldots][x_1 = v_1] \\
\downarrow{\scriptstyle eq_1} & & \vdots{\scriptstyle eq_1} \\
X_1\{v_1/x_1\}[\ldots X_2\{v_1/x_1\}[x_2 = v_2']\ldots][x_1 = v_1] & & X_1\{v_1/x_1\}[\ldots X_2\{v_1/x_1, v_2'/x_2\}[x_2 = v_2']\ldots][x_1 = v_1]
\end{array}$$

□

Unification Lemmas The next two *unification* lemmas state that our rewrite rules encode classical unification algorithms.

Lemma C.17 (Unify). *If* $\overline{z} \cap (\mathsf{fvs}(\overline{u}) \cup \mathsf{fvs}(\overline{v})) = \emptyset$ *then* $X\{\overline{u}/\overline{z}\}[\overline{u = v}] \downarrow_{\mathcal{U}} X\{\overline{v}/\overline{z}\}[\overline{u = v}]$

Proof. Let $X_{\overline{u}} \equiv X\{\overline{u}/\overline{z}\}$ and $X_{\overline{v}} \equiv X\{\overline{v}/\overline{z}\}$. The proof follows by induction on the triple $(\sharp\text{free}, \sharp\text{size}, \sharp n)$ where

$$\begin{aligned}
\sharp\text{free} &\doteq \sharp\mathsf{fvs}(\overline{u}) + \sharp\mathsf{fvs}(\overline{v}) \\
\sharp\text{size} &\doteq \Sigma_{i=1}^{n} size(u_i) + size(v_i) \\
\sharp n &\doteq \text{the cardinality of } \overline{u}, \overline{v}
\end{aligned}$$

Split cases on the *first* equation $u_1 = v_1$.

Case $hnf_1 = hnf_2$ with *incompatible* values: Here,

$$X_{\overline{u}}[hnf_1 = hnf_2; \overline{u' = v'}] \longrightarrow_{\text{U-FAIL}} X_{\overline{u}}[\mathtt{fail}]$$

and

$$X_{\overline{v}}[hnf_1 = hnf_2; \overline{u' = v'}] \longrightarrow_{\text{U-FAIL}} X_{\overline{v}}[\mathtt{fail}]$$

after which we can join at **fail** via fail-elim.

Case $\langle u_1, \ldots, u\_k\rangle = \langle v_1, \ldots, v_k\rangle$ with tuples of the same arity k: use U-TUP to get equations per component and join using the induction hypothesis, which is well-founded as the ♯size is strictly smaller:

$$\begin{array}{ccc} X_{\overline{u}}[\langle u_1, \ldots, u_k\rangle = \langle v_1, \ldots, v_k\rangle ; \overline{u' = v'}] & \xrightarrow{\text{U-TUP}} & X_{\overline{u}}[u_1 = v_1, \ldots, u_k = v_k ; \overline{u' = v'}] \\ & & \downarrow\!\!\downarrow \\ & IH & \bullet \\ & & \uparrow\!\!\uparrow \\ X_{\overline{v}}[\langle u_1, \ldots, u_k\rangle = \langle v_1, \ldots, v_k\rangle ; \overline{u' = v'}] & \xrightarrow[\text{U-TUP}]{} & X_{\overline{v}}[u_1 = v_1, \ldots, u_k = v_k ; \overline{u' = v'}] \end{array}$$

Case $x = y$ use SUBST to replace all occurrences of x with y, and then apply the IH on the *remaining* $n - 1$ equations $\overline{u' = v'}$. Note that the induction is well-founded as in this case ♯free and ♯size are unchanged but the number of equations decreases by one.

$$\begin{array}{ccc} X_{\overline{u}}[x = y; \overline{u' = v'}] & \xrightarrow{\text{SUBST}} & X_{\overline{u}}\{y/x\}[x = y; \overline{u'\{y/x\} = v'\{y/x\}}] \\ & & \downarrow\!\!\downarrow \\ & IH & \bullet \\ & & \uparrow\!\!\uparrow \\ X_{\overline{v}}[x = y; \overline{u' = v'}] & \xrightarrow[\text{SUBST}]{} & X_{\overline{v}}\{y/x\}[x = y; \overline{u'\{y/x\} = v'\{y/x\}}] \end{array}$$

Case $x = h$ where h is an HNF value and $x \notin \mathsf{fvs}(h)$: use SUBST to replace all occurrences of x with h, and then apply the IH on the remaining $n-1$ equations $\overline{u' = v'}$. Note that the induction is well-founded in this case as ♯free decreases since x is removed from the free variables of $\overline{u'}$ and $\overline{v'}$ and $X_{\overline{u}}$ and $X_{\overline{v}}$ even though the ♯size may increase due to the substitution.

$$\begin{array}{ccc} X_{\overline{u}}[x = h; \overline{u' = v'}] & \xrightarrow{\text{SUBST}} & X_{\overline{u}}\{h/x\}[x = h; \overline{u'\{h/x\} = v'\{h/x\}}] \\ & & \downarrow\!\!\downarrow \\ & IH & \bullet \\ & & \uparrow\!\!\uparrow \\ X_{\overline{v}}[x = h; \overline{u' = v'}] & \xrightarrow[\text{SUBST}]{} & X_{\overline{v}}\{h/x\}[x = h; \overline{u'\{h/x\} = v'\{h/x\}}] \end{array}$$

Case $x = v$ where $x \in \mathsf{fvs}(v)$: either join at **fail** via U-OCCURS or violates the no recursion assumption.

□

LEMMA C.18 (UNIFY-FLIP). *If* $\overline{z} \cap (\mathsf{fvs}(\overline{u}) \cup \mathsf{fvs}(\overline{v})) = \emptyset$ *then* $X\{\overline{u}/\overline{z}\}[\overline{u = v}] \downarrow_{\mathcal{U}} X\{\overline{v}/\overline{z}\}[\overline{v = u}]$

PROOF. Same as Lemma C.17 except using HNF-SWAP and VAR-SWAP to make the equations the same on both sides. □

C.4 Unification is Confluent

LEMMA C.19 ($\mathcal{U}$-CONFLUENT). *$\mathcal{U}$ is confluent.*

PROOF. We prove that $\mathcal{U}$ is confluent via the following strategy inspired by *labeled reductions* [Lévy 1976]. Let $\mathcal{U}_k$ which is a subset of $\mathcal{U}$ that only applies reductions to terms that are *under less than k* λs.

$$\begin{array}{lll} \text{Block} & b ::= \{v = r;\ b\}_\ell \mid \{b; b\}_\ell \mid t \\ \text{RHS} & r ::= b \mid t \\ \text{Tail} & t ::= v \mid v_1\ v_2 \mid \exists x.\, e \mid e_1 \,\|\, e_2 \mid \mathbf{one}\{e\} \mid \mathbf{all}\{e\} \mid \mathbf{fail} \end{array}$$

Fig. 14. Labeled Blocks

(1) First, we show that $\mathcal{U}_k$ is *locally confluent* for all k (Lemma C.21).
(2) Second, we show that $\mathcal{U}_k$ is *terminating* for all k (Lemma C.23).
(3) Third, consequently, by Lemma B.14 we obtain that $\mathcal{U}_k$ is confluent for all k.
(4) Finally, we show that $\mathcal{U}$ is confluent by using the largest k in two traces, to join two arbitrary sequences of $\mathcal{U}$ reductions Lemma C.22.

□

Definition C.20 (k-Unification). A k-labeled term is a term where each subterm occurring under at most k λ's is marked by a special label ℓ. Let $\mathcal{U}_k$ be defined as the set of all $\mathcal{U}$ reductions where: (1) the $\mathcal{U}$-redex is a ℓ-labeled or occurs under $\leqslant k$ λs, and (2) the SUBST preserves labels.

LEMMA C.21. *$\mathcal{U}_k$ is locally confluent.*

PROOF. For simplicity, we directly prove that $\mathcal{U}$ is locally confluent (Lemma C.28). The proof carries over to $\mathcal{U}_k$ as the only required $\mathcal{U}$-reductions under $> k$ λs are on *labeled* subterms. □

We can now prove that any two $\mathcal{U}_k$ reductions (and hence $\mathcal{U}$ reductions) can be joined.

LEMMA C.22 ($\mathcal{U}_k$-JOIN). *If $e \twoheadrightarrow_{\mathcal{U}_i} e_i$ and $e \twoheadrightarrow_{\mathcal{U}_j} e_j$ then there exists e' such that $e_i, e_j \twoheadrightarrow_{\mathcal{U}} e'$.*

PROOF. Let $k = \max(i, j)$. As $\mathcal{U}_i, \mathcal{U}_j \subseteq \mathcal{U}_k$ we have $e \twoheadrightarrow_{\mathcal{U}_k} e_i$ and $e \twoheadrightarrow_{\mathcal{U}_k} e_j$. By Lemma C.23 and Lemma C.21 and Lemma B.14, $\mathcal{U}_k$ is confluent, hence there exists e' such that $e_i, e_j \twoheadrightarrow_{\mathcal{U}_k} e'$, after which $\mathcal{U}_k \subseteq \mathcal{U}$ completes the proof. □

LEMMA C.23. *$\mathcal{U}_k$ is Noetherian.*

PROOF. By induction on k.

Base case ($k \equiv 0$) via Lemma C.27.

Inductive case Assume the induction hypothesis that $\mathcal{U}_k$ is Noetherian and prove $\mathcal{U}_{k+1}$ is Noetherian. Let σ be a $\mathcal{U}_{k+1}$ reduction sequence $e \longrightarrow \ldots$. We will prove that σ is finite. By the IH there is some *finite prefix* of the trace $e \twoheadrightarrow_{\mathcal{U}_{k+1}} e'$ after which there are no more $\mathcal{U}$ steps at level $\leqslant k$. Note that e' is finite and of the form $\ldots(\lambda x_1.\, e_1)\ldots(\lambda x_n.\, e_n)\ldots$ comprising n disjoint λ terms. Every $\mathcal{U}_{k+1}$ reduction from e' is a $\mathcal{U}_k$ reduction from some e_i, that occur "in parallel" *i.e.* without influencing each other, and which can be sequenced to get a $\mathcal{U}_{k+1}$ reduction sequence. Again, by the induction hypothesis, each of these reduction sequences (for each e_i) is finite, and hence their sequencing is finite, hence σ must be finite.

□

Labeled Blocks We prove the base case of Lemma C.23 by stratifying expressions into *labeled blocks*, *tails*, *rhs* and *expressions* as shown in Fig. 14. A *tail* is a term that is "inert" for the purposes of $\mathcal{U}_0$ reduction: namely a value, application, existential, one, all or choice. An *rhs* is either a block or a tail (which includes a value). A *labeled block* is a sequence of equations $v = r$ of a value and an RHS r followed by a tail t. We assume each block carries a unique "ghost" label ℓ (that will be used to prove termination). In any block b, for any two labels ℓ_1 and ℓ_2 we write $\ell_1 \prec_b \ell_2$ if the block labelled by ℓ_2 occurs *inside* (under) the block labelled by ℓ_1 in b. We will use b_ℓ to denote the (unique) sub-block of b labeled by ℓ. For rewrites like CHOOSE, SEQ-ASSOC, SEQ-SWAP and EQN-FLOAT, we assume that the rewritten term is given a fresh set of distinct block labels. For rewrites with U-TUP, we assume fresh labels are given to the new (inner) blocks created by tuple matching equations. All other $\mathcal{U}$ rewrites preserve blocks or delete them, so we assume that the same labels carry over to the rewritten terms.

LEMMA C.24. *SEQ-SWAP strongly postpones after* $\mathcal{U}$.

PROOF. Split cases on each reduction of $\mathcal{U}$; the diamond is completed as the rules are non-overlapping. ◻

Definition C.25 (Elimination). We say a reduction *eliminates* a variable x from a block b if the reduction is (1) a SUBST reduction spanning b or an enclosing block (2) using an equation $x = v$ where (3) v is either an HNF or a variable y such that $x \prec y$. A reduction sequence *eliminates* a variable x from a block b if there is some reduction in the sequence that eliminates x from b, and the sequence contains no subsequent SUBST reductions spanning any block strictly enclosing b.

LEMMA C.26. $\mathcal{U}_0$ *is Noetherian for all* blocks b.

PROOF. We prove that for any term b that it is only possible to take finitely many $\mathcal{U}_0$ steps from b. Let $\sigma \doteq b \longrightarrow b_1 \longrightarrow b_2 \longrightarrow \ldots$ be a $\mathcal{U}_0$ reduction sequence starting at b. Write σ_i for the prefix $b \longrightarrow \ldots \longrightarrow b_i$. We will show that σ must be finite. Let $\mathcal{U}_0' \doteq \mathcal{U}_0 -$ SEQ-SWAP. As SEQ-SWAP strongly postpones after $\mathcal{U}$ Lemma C.24, any infinite σ can be translated to a either: (a) A sequence with a *finite* prefix of $\mathcal{U}_0'$ reductions followed by infinitely many SEQ-SWAP, or (b) An infinite sequence of $\mathcal{U}_0'$ reductions. Next, we show neither case is possible.

Case (a) This case is ruled out by the ordering restriction on SEQ-SWAP which ensures that after the finite prefix of $\mathcal{U}_0'$ reductions, we can only keep swapping equations till they reach a canonical linear order after which no further swaps are possible.

Case (b) Next, we (ignore SEQ-SWAP to) show there is no infinite sequence of $\mathcal{U}_0'$ reductions. To do so, suppose that σ is such a reduction sequence. For each prefix (σ_i, e_i) we define the following lexicographic termination metric

$$\sharp(\sigma_i, b_i) \doteq (\sharp\mathsf{choose}(b_i), \sharp\mathsf{semi}(b_i), \mathsf{cands}(\sigma_i, b_i), \mathsf{size}(b_i), \sharp\mathsf{swaps}(b_i))$$

where

$$\begin{aligned}
\mathsf{choose}(b_i) &\doteq \text{CHOOSE redexes in } b_i \\
\mathsf{semi}(b_i) &\doteq \text{SEQ-ASSOC or EQN-FLOAT redexes in } b_i \\
\mathsf{cands}(\sigma_i, e_i) &\doteq [\ldots \ell \mapsto (\sharp\mathsf{cand}(\sigma_i, b_i, \ell) \ldots \mid \ell \in b_i] \\
&\qquad \text{where labels are ordered by } \prec_{b_i} \\
\mathsf{size}(b_i) &\doteq \text{size of the block } b_i \\
\mathsf{swaps}(b_i) &\doteq \text{VAR-SWAP redexes in } b_i
\end{aligned}$$

and where, for a finite reduction (prefix) σ' block b and label ℓ

$$\begin{aligned}
\mathsf{cand}(\sigma', b, \ell) &\doteq \mathsf{fvs}(b_\ell) - \mathit{elim}(\sigma', b, \ell) \\
\mathsf{elim}(\sigma', b, \ell) &\doteq \{x \mid \sigma' \text{ eliminates } x \text{ from } b_\ell\}
\end{aligned}$$

The unification reductions preserve the following invariant: once a variable x has been eliminated from a block, it appears at most once in the block as an LHS of an equation $x = v$, and that equation can *never* again be used to perform a substitution in that block *unless* new occurrences of x are injected into the block by a substitution performed in an *enclosing* block, in which case, the block metric for the outer block will be strictly smaller. Specifically, each application of

- CHOOSE strictly reduces ♯choose;
- SEQ-ASSOC or EQN-FLOAT strictly reduces ♯semi (leaving ♯choose unchanged);
- SUBST strictly reduces ♯cands (leaving ♯semi, ♯choose unchanged), as it *eliminates* a variable from the block ℓ that the substitution spans;
- U-TUP strictly reduces size (leaving cands, ♯semi, ♯choose unchanged), as it preserves elim and hence cand, but reduces the size of ℓ;
- U-LIT, U-FAIL, U-OCCURS strictly reduces size (leaving cands, ♯semi, ♯choose unchanged);
- VAR-SWAP strictly reduces swaps leaving the other components unchanged.

Thus, as $\sharp(\sigma_i, b_i)$ is a strictly decreasing well-founded metric, the sequence $(\sigma_1, b_1), \ldots,$ is finite, and so any sequence of $\mathcal{U}_0$' steps is guaranteed to terminate. □

Lemma C.27. *$\mathcal{U}_0$ is Noetherian for all* tails t, rhs r *and* expressions e.

Proof. By induction on the structure of t, r and e, using Lemma C.26 for the base case. □

Lemma C.28. *$\mathcal{U}$ is locally confluent.*

Proof. Let $\Delta_1 \rightarrow_1 \Delta_1'$ and $\Delta_2 \rightarrow_2 \Delta_2'$ denote the two $\mathcal{U}$ reducts. If the reducts are disjoint, then the terms can be joined trivially in a single step via Lemma C.4. By symmetry it suffices to consider the case where Δ_1 occurs under Δ_2. Let us split cases on the rule used for Δ_1.

Case Δ_1 via $\mathcal{U}$ − SUBST − VAR-SWAP join using Lemma C.6.
Case Δ_1 via VAR-SWAP join using Lemma C.29.
Case Δ_1 via SUBST join using Lemma C.30.

□

Lemma C.29 (VAR-SWAP UNDER). *If $\Delta_{\mathcal{U}} \rightarrow_{\mathcal{U}} \Delta_{\mathcal{U}}'$ and $\Delta_{\mathcal{U}} \equiv E[\Delta]$ and $\Delta \rightarrow_{SWAP} \Delta'$ then there exists $\Delta_{\mathcal{U}}''$ such that $\Delta_{\mathcal{U}}' \twoheadrightarrow_{SWAP} \Delta_{\mathcal{U}}''$ and $E[\Delta'] \rightarrow_{\mathcal{U}} \Delta_{\mathcal{R}}''$.*

$$\begin{array}{ccc} \Delta_{\mathcal{U}} \equiv E[\Delta] & \xrightarrow{VAR\text{-}SWAP} & E[\Delta'] \\ \mathcal{U}\downarrow & & \downarrow \mathcal{U} \\ \Delta_{\mathcal{U}}' & \underset{VAR\text{-}SWAP}{\twoheadrightarrow} & \Delta_{\mathcal{U}}'' \end{array}$$

Proof. Split cases on the rule used in $\mathcal{U}$.

Case U-LIT or VAR-SWAP : impossible as no VAR-SWAP redex under $k_1 = k_2$ or $x = y$.
Case U-TUP : Here, $\Delta_{\mathcal{U}} \equiv \langle u_1, \ldots, u_n \rangle = \langle v_1, \ldots, v_n \rangle$ and wlog the VAR-SWAP redex is $u_1' \rightarrow_{u_1}$ so join at $u\_1' = v_1; \ldots; u_n = v_n$.
Case U-FAIL : Here, $\Delta_{\mathcal{U}} \equiv hnf_i \rightarrow hnf_i'$ so join at **fail**
Case U-OCCURS : Here, $\Delta_{\mathcal{U}} \equiv x = V[x]$ and the VAR-SWAP redex is under $V[x]$, *i.e.* $V^{[}x] \rightarrow_{\text{SUBST}} V^{[}x]'$ as the free variables are preserved by VAR-SWAP hence we can join at **fail**.
Case HNF-SWAP : Here, $\Delta_{\mathcal{U}} \equiv hnf = x$ and the VAR-SWAP redex is under hnf *i.e.* $hnf \rightarrow_{\text{SUBST}} hnf'$, hence join at $x = hnf'$.
Case SUBST : via Lemma C.12.
Case CHOOSE : via Lemma C.7.
Case SEQ-ASSOC : Here, $\Delta_{\mathcal{U}} \equiv (eq; e_1); e_2 \rightarrow eq; (e_1; e_2) \equiv \Delta_{\mathcal{U}}'$. Split cases on where Δ occurs.
  Case $\Delta \subseteq eq$ *i.e.* $eq \rightarrow_{\text{VAR-SWAP}} eu'$ Join at $\Delta_{\mathcal{U}}'' \equiv eu'; (e_1; e_2)$.
  Case $\Delta \subseteq e_1$ *i.e.* $e_1 \rightarrow_{\text{VAR-SWAP}} e_1'$ Join at $\Delta_{\mathcal{U}}'' \equiv eq; (e_1'; e_2)$.

Case $\Delta \subseteq e_2$ *i.e.* $e_2 \rightarrow_{\text{VAR-SWAP}} e_2'$ Join at $\Delta''_{\mathcal{U}} \equiv eq; (e_1; e_2')$.

Case EQN-FLOAT : $\Delta_{\mathcal{U}} \equiv v = (eq; e_1); e_2 \longrightarrow eq; (v = e_1; e_2) \equiv \Delta'_{\mathcal{U}}$. Split cases on where Δ occurs.

Case $\Delta \subseteq v$ *i.e.* $v \rightarrow_{\text{VAR-SWAP}} v'$ Join at $\Delta''_{\mathcal{U}} \equiv eq; (v' = e_1; e_2)$.

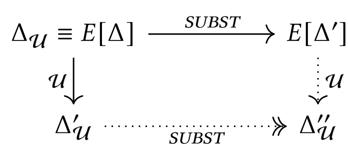
Case $\Delta \subseteq eq$ *i.e.* $eq \rightarrow_{\text{VAR-SWAP}} eu'$ Join at $\Delta''_{\mathcal{U}} \equiv eu'; (v = e_1; e_2)$.

Case $\Delta \subseteq e_1$ *i.e.* $e_1 \rightarrow_{\text{VAR-SWAP}} e_1'$ Join at $\Delta''_{\mathcal{U}} \equiv eq; (v = e_1'; e_2)$.

Case $\Delta \subseteq e_2$ *i.e.* $e_2 \rightarrow_{\text{VAR-SWAP}} e_2'$ Join at $\Delta''_{\mathcal{U}} \equiv eq; (v = e_1; e_2')$.

□

Lemma C.30 (subst-under). *If $\Delta_{\mathcal{U}} \rightarrow_{\mathcal{U}} \Delta'_{\mathcal{U}}$ and $\Delta_{\mathcal{U}} \equiv E[\Delta]$ and $\Delta \rightarrow_{SUBST} \Delta'$ then there exists $\Delta''_{\mathcal{U}}$ such that $\Delta'_{\mathcal{U}} \twoheadrightarrow_{SUBST} \Delta''_{\mathcal{U}}$ and $E[\Delta'] \rightarrow_{\mathcal{U}} \Delta''_{\mathcal{U}}$.*



Proof. Split cases on the rule used in $\mathcal{U}$.

Case U-LIT or VAR-SWAP : impossible as no SUBST redex under $k_1 = k_2$ or $x = y$.

Case U-TUP : Here, $\Delta_{\mathcal{U}} \equiv \langle u_1, \ldots, u_n \rangle = \langle v_1, \ldots, v_n \rangle$ and wlog the SUBST redex is $u_1' \rightarrow_{u_1}$ so join at $u\_1' = v_1; \ldots; u_n = v_n$.

Case U-FAIL : Here, $\Delta_{\mathcal{U}} \equiv hnf_i \longrightarrow hnf_i'$ so join at **fail**

Case U-OCCURS : Here, $\Delta_{\mathcal{U}} \equiv x = V[x]$ and the SUBST redex is under $V[x]$, *i.e.* $V^{[}x] \rightarrow_{\text{SUBST}} V^{[}x]'$ as the free variables are preserved by SUBST hence we can join at **fail**.

Case HNF-SWAP : Here, $\Delta_{\mathcal{U}} \equiv hnf = x$ and the SUBST redex is under hnf *i.e.* $hnf \rightarrow_{\text{SUBST}} hnf'$, hence join at $x = hnf'$.

Case SUBST : via Lemma C.14.

Case CHOOSE : via Lemma C.7.

Case SEQ-ASSOC : $\Delta_{\mathcal{U}} \equiv (eq; e_1); e_2 \longrightarrow eq; (e_1; e_2) \equiv \Delta'_{\mathcal{U}}$. Split cases on where Δ occurs.

Case $\Delta \subseteq eq$ *i.e.* $eq \rightarrow_{\mathcal{R}'} eu'$ Join at $\Delta''_{\mathcal{U}} \equiv eu'; (e_1; e_2)$.

Case $\Delta \subseteq e_1$ *i.e.* $e_1 \rightarrow_{\mathcal{R}'} e_1'$ Join at $\Delta''_{\mathcal{U}} \equiv eq; (e_1'; e_2)$.

Case $\Delta \subseteq e_2$ *i.e.* $e_2 \rightarrow_{\mathcal{R}'} e_2'$ Join at $\Delta''_{\mathcal{U}} \equiv eq; (e_1; e_2')$.

Case $\Delta \subseteq (eq; e_1)$ *i.e.* SUBST : $(eq; e_1) \longrightarrow (eu'; e_1')$ Join at $\Delta''_{\mathcal{U}} \equiv eu'; (e_1'; e_2)$.

Case $\Delta \subseteq ((eq; e_1); e_2)$ *i.e.* SUBST : $(eq; e_1); e_2 \longrightarrow (eu'; e_1'); e_2'$ Join at $\Delta''_{\mathcal{U}} \equiv eu'; (e_1'; e_2')$.

Case EQN-FLOAT : $\Delta_{\mathcal{U}} \equiv v = (eq; e_1); e_2 \longrightarrow eq; (v = e_1; e_2) \equiv \Delta'_{\mathcal{U}}$. Split cases on where Δ occurs.

Case $\Delta \subseteq v$ *i.e.* $v \rightarrow_{\mathcal{R}'} v'$ Join at $\Delta''_{\mathcal{U}} \equiv eq; (v' = e_1; e_2)$.

Case $\Delta \subseteq eq$ *i.e.* $eq \rightarrow_{\mathcal{R}'} eu'$ Join at $\Delta''_{\mathcal{U}} \equiv eu'; (v = e_1; e_2)$.

Case $\Delta \subseteq e_1$ *i.e.* $e_1 \rightarrow_{\mathcal{R}'} e_1'$ Join at $\Delta''_{\mathcal{U}} \equiv eq; (v = e_1'; e_2)$.

Case $\Delta \subseteq e_2$ *i.e.* $e_2 \rightarrow_{\mathcal{R}'} e_2'$ Join at $\Delta''_{\mathcal{U}} \equiv eq; (v = e_1; e_2')$.

Case $\Delta \subseteq (eq; e_1)$ *i.e.* SUBST : $v = (eq; e_1); e_2 \longrightarrow v = (eu'; e_1'); e_2$. Join at $\Delta''_{\mathcal{U}} \equiv eu'; (v = e_1'; e_2)$.

Case $\Delta \subseteq v = (eq; e_1); e_2$ *i.e.* SUBST : $(v = eq; e_1); e_2 \longrightarrow (v' = eu'; e_1'); e_2'$ Join at $\Delta''_{\mathcal{U}} \equiv eu'; (v' = e_1'; e_2')$.

□

C.5 Normalization is Confluent

Recall that $\mathcal{N} \equiv$ EXI-SWAP + EXI-FLOAT + VAR-SWAP + SUBST-VAR where

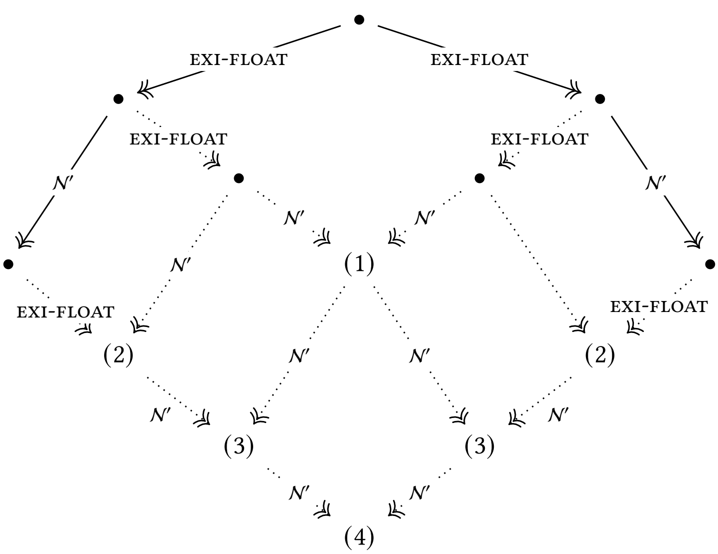
$$\text{SUBST-VAR} \quad X[x = y;\ e] \longrightarrow (X\{y/x\})[x = y;\ e\{y/x\}]$$

It will be convenient to *factor out* EXI-FLOAT so let

$$\mathcal{SS} \doteq \text{SUBST-VAR} + \text{VAR-SWAP}$$
$$\mathcal{N}' \doteq \mathcal{SS} + \text{EXI-SWAP}$$
$$\mathcal{N} \doteq \mathcal{N}' + \text{EXI-FLOAT}$$

Lemma C.31 ($\mathcal{N}$-Confluent). *$\mathcal{N}$ is confluent.*

Proof. The above result follows in two steps. First we show that $\mathcal{N}'$ – *i.e.* normalization-without-EXI-FLOAT – is confluent in Lemma C.35. Second we show that $\mathcal{N}'$ *strongly postpones* after EXI-FLOAT Lemma C.34. Consequently, each $\twoheadrightarrow_{\mathcal{N}}$ can be rewritten as the composition of $\twoheadrightarrow_{\text{EXI-FLOAT}}$ followed by $\twoheadrightarrow_{\mathcal{N}'}$ after which the following diagram completes the proof, where (1) Lemma C.32 (2) Lemma C.33 (3) Lemma C.35. (4) Lemma C.35



□

Lemma C.32. *If $e \twoheadrightarrow_{\text{EXI-FLOAT}} e_1$ and $e \twoheadrightarrow_{\text{EXI-FLOAT}} e_2$ then exists $e_1 \twoheadrightarrow_{\text{EXI-FLOAT}} e_1'$, $e_2 \twoheadrightarrow_{\text{EXI-FLOAT}} e_2'$, such that $e_1' \downarrow_{\text{EXI-SWAP}} e_2'$.*

Proof. On each side add the (missing) EXI-FLOAT steps on the other side, and then use (multiple) EXI-SWAP to join. □

Lemma C.33. *EXI-FLOAT strongly commutes with $\mathcal{N}'$.*

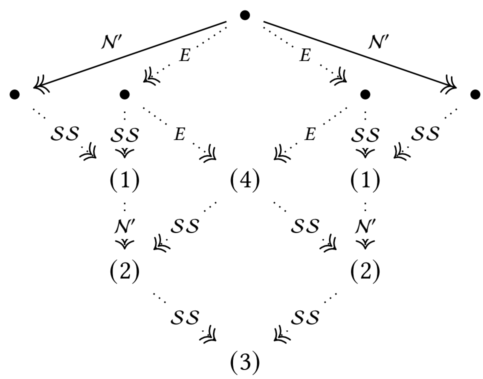
Proof. Split cases on each possible case of $\mathcal{N}'$, the diamond is completed trivially as the rules are non-overlapping. □

Lemma C.34. *$\mathcal{N}'$ strongly postpones after EXI-FLOAT, so $\mathcal{N}^* \equiv \text{EXI-FLOAT}^* \cdot \mathcal{N}'^*$.*

Proof. Split cases on each possible case of $\mathcal{N}'$; the diamond is completed trivially as the rules are non-overlapping. □

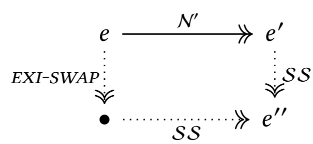
Lemma C.35. *$\mathcal{N}'$ is confluent.*

Proof. Via the following diagram, where: (1) is Lemma C.36; (2) is Lemma C.40; (3) is Lemma C.39; (4) is Lemma C.38.



□

Lemma C.36. *If* $e \twoheadrightarrow_{\mathcal{N}'} e'$ *there exists* e'' *such that* $e' \twoheadrightarrow_{\mathcal{SS}} e''$ *and* $e \twoheadrightarrow_{\text{EXI-SWAP}} \cdot \twoheadrightarrow_{\mathcal{SS}} e''$.

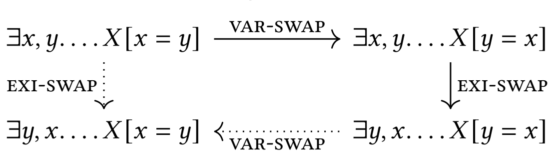


Proof. By using Lemma B.25 with the facts that $\mathcal{SS}$ is confluent (Lemma C.39) and $\mathcal{SS}$ *hops after* exi-swap (Lemma C.37). □

Lemma C.37. *$\mathcal{SS}$ (resp. $\mathcal{U}$) hops after* exi-swap.

Proof. By splitting cases on the $\mathcal{SS}$(resp. $\mathcal{U}$) reduction that precedes the exi-swap.

Case var-swap Let the $\Delta_{\text{SWAP}} \equiv X[x = y]$. If the exi-swap preserves the order of x and y then the result follows trivially (as the reductions are non-overlapping.) If the exi-swap toggles the order then the result follows via the diagram



Case *non*-var-swap An $\mathcal{U}$ reduction *other than* var-swap is variable-order independent, so the sequence of $\mathcal{U}$-step followed by exi-swap is equivalent to first doing the exi-swap and then the $\mathcal{U}$step.

□

Lemma C.38. *exi-swap is confluent.*

Proof. Trivial, via the diamond property. □

Lemma C.39. *$\mathcal{SS}$ = subst-var + swap is confluent.*

Proof. Note that $\mathcal{SS}$ is a subset of $\mathcal{U}$; the proof follows similar to the proof of Lemma C.19 (ignoring the bits about u-tup and u-lit and u-fail and substituting HNF values.) □

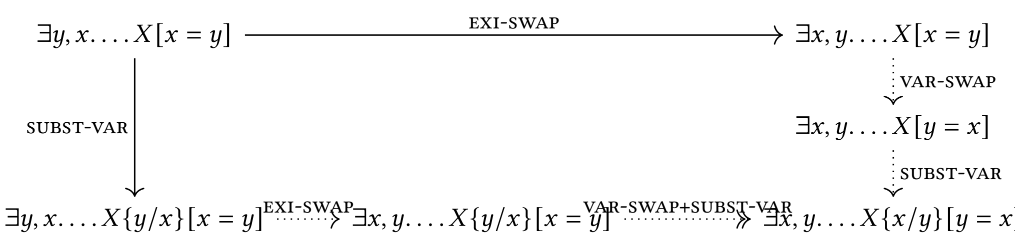
Lemma C.40. *$\mathcal{SS}$ commutes with $\mathcal{N}'$.*

Proof. Recall that $\mathcal{N}' \equiv \mathcal{SS} + $ exi-swap. The proof follows by observing that $\mathcal{SS}$ *half-commutes* with exi-swap Lemma C.41, recalling that $\mathcal{SS}$ is confluent Lemma C.39, after which Lemma B.27 yields the conclusion $\mathcal{SS}$ commutes with $\mathcal{SS} + $ exi-swap $\equiv \mathcal{N}'$. □

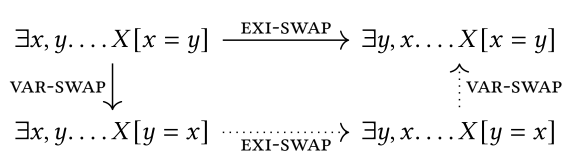
Lemma C.41. *$\mathcal{SS}$ half-commutes with exi-swap.*

Proof. Recall that $\mathcal{SS} \equiv$ subst-var + var-swap. Split cases and show each reduction half-commutes with exi-swap.

Case subst-var If the exi-swap occurs under subst-var then they trivially commute as the variable order is unaffected by the exi-swap. If the subst-var occurs under exi-swap the proof is completed by the following diagram.



Case var-swap (under exi-swap) The non-trivial cases are where the *same* variables x, y are being swapped by both rules (otherwise, the reductions half-commutes trivially via the diamond property). For the variables to be the same, the var-swap *must* occur under the exi-swap (as otherwise the same variables are not in scope.) Hence, the proof is completed by the following diagram.



□

C.6 Unification + Normalization is Confluent

Recall that

$$\mathcal{N} \doteq \text{EXI-FLOAT} + \text{EXI-SWAP} + \mathcal{SS}$$

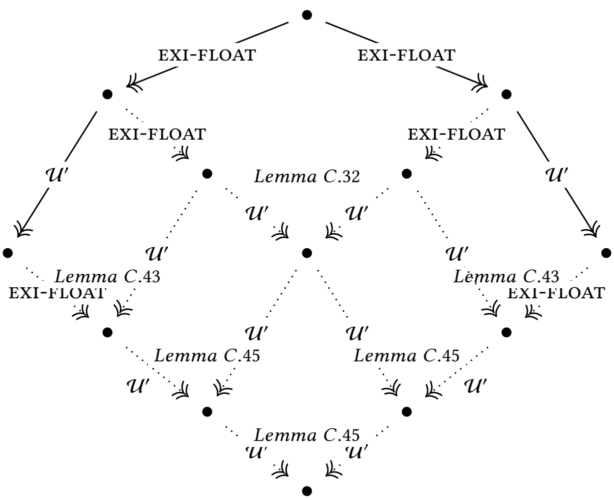
and define

$$\mathcal{U}' \doteq \mathcal{U} + \text{EXI-SWAP}$$

Lemma C.42. *$\mathcal{U} \cup \mathcal{N}$ is confluent.*

Proof. We prove $\mathcal{U} \cup \mathcal{N}$ is confluent by a generalization of the proof of Lemma C.31 where we use the full $\mathcal{U}$ relation (instead of the subset $\mathcal{SS}$). First we show that $\mathcal{U}'$ – *i.e.* $\mathcal{U} \cup \mathcal{N}$ without-exi-float – is confluent Lemma C.45. Second we show that $\mathcal{U}'$ *strongly postpones* after exi-float Lemma C.44. Consequently, each $\twoheadrightarrow_{\mathcal{U} \cup \mathcal{N}}$ can be rewritten as the composition of $\twoheadrightarrow_{\text{EXI-FLOAT}}$ followed by $\twoheadrightarrow_{\mathcal{U}'}$

after which the following diagram completes the proof.



□

Lemma C.43. *EXI-FLOAT strongly commutes with $\mathcal{U}'$.*

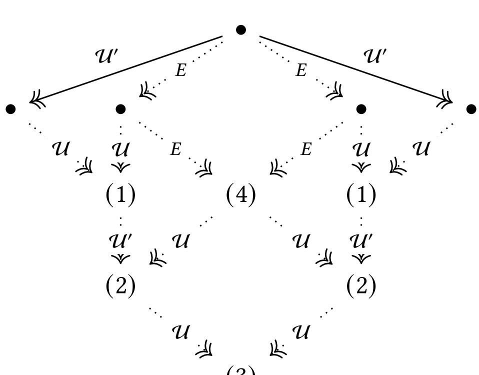
Proof. Split cases on each possible case of $\mathcal{U}'$; the diamond is completed trivially as the rules are non-overlapping. □

Lemma C.44. *Let $\mathcal{U}' \doteq \mathcal{U} + $ EXI-SWAP. $\mathcal{U}$ strongly postpones after EXI-FLOAT, so $\mathcal{U}'^{*} \equiv$ EXI-FLOAT$^{*} \cdot \mathcal{U}^{*}$.*

Proof. Same as Lemma C.34. □

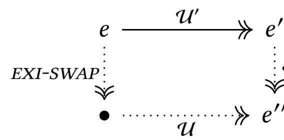
Lemma C.45. *Let $\mathcal{U}' \doteq \mathcal{U} + $ EXI-SWAP. $\mathcal{U}'$ is confluent.*

Proof. Via the following diagram, where: (1) is Lemma C.46; (2) is Lemma C.47; (3) is Lemma C.19; (4) is Lemma C.38.



□

Lemma C.46. *Let $\mathcal{U}' = \mathcal{U} + \text{EXI-SWAP}$. If $e \twoheadrightarrow_{\mathcal{U}'} e'$ there exists e'' such that $e' \twoheadrightarrow_{\mathcal{U}} e''$ and $e \twoheadrightarrow_{\text{EXI-SWAP}} \cdot \twoheadrightarrow_{\mathcal{U}} e''$.*



Proof. (Similar to Lemma C.36), By using Lemma B.25 with the facts that $\mathcal{U}$ is confluent (Lemma C.19) and $\mathcal{U}$ *hops after* EXI-SWAP (Lemma C.37). □

Lemma C.47. *Let $\mathcal{U}' = \mathcal{U} + \text{EXI-SWAP}$. $\mathcal{U}$' commutes with $\mathcal{U}$.*

Proof. The proof follows by observing that $\mathcal{U}$ *half-commutes* with EXI-SWAP Lemma C.48, recalling that $\mathcal{U}$ is confluent Lemma C.19, after which Lemma B.27 yields the conclusion $\mathcal{U}$ commutes with $\mathcal{U} + \text{EXI-SWAP} \equiv \mathcal{U}'$. □

Lemma C.48. *$\mathcal{U}$ half-commutes with EXI-SWAP.*

Proof. Same as Lemma C.41; the rules in $\mathcal{U}$*other than those* in the subset $\mathcal{SS}$ trivially half-commutes as they do not overlap with EXI-SWAP. □

C.7 $\mathcal{U} \cup \mathcal{N}$ Commute With $\mathcal{A} \cup \mathcal{G} \cup \mathcal{C}$

Lemma C.49 ($\mathcal{U}$-$\mathcal{A}$-Comm). *$\mathcal{U}$ and $\mathcal{A}$ commute.*

Proof. We show that $\mathcal{U}$ $*$-commutes with $\mathcal{A}$ and hence commutes via Lemma B.34. Let $\Delta_{\mathcal{U}} \rightarrow_{\mathcal{U}} \Delta'_{\mathcal{U}}$ and $\Delta_{\mathcal{A}} \rightarrow_{\mathcal{A}} \Delta'_{\mathcal{A}}$ denote the reducts for $\mathcal{U}$ and $\mathcal{A}$ respectively.

Case: $\Delta_{\mathcal{U}}$ and $\Delta_{\mathcal{A}}$ disjoint via Lemma C.4.
Case: $\Delta_{\mathcal{U}} \subseteq \Delta_{\mathcal{A}}$ via Lemma C.5.
Case: $\Delta_{\mathcal{A}} \subseteq \Delta_{\mathcal{U}}$ via Lemma C.6.

□

Lemma C.50 ($\mathcal{U} - \mathcal{G}$-Comm). *$\mathcal{U}$ and $\mathcal{G}$ commute.*

Proof. We show that $\mathcal{U}$ $*$-commutes commutes wth $\mathcal{G}$, and hence by Lemma B.34, $\mathcal{U}$ commutes wth $\mathcal{G}$. Let $\Delta_{\mathcal{U}} \rightarrow_{\mathcal{U}} \Delta'_{\mathcal{U}}$ and $\Delta_{\mathcal{G}} \rightarrow_{\mathcal{G}} \Delta'_{\mathcal{G}}$ denote the reducts for $\mathcal{U}$ and $\mathcal{G}$ respectively. If the reducts are disjoint then terms can be trivially joined. Let us split cases on whether $\Delta_{\mathcal{U}}$ occurs under $\Delta_{\mathcal{G}}$ or vice versa.

Case $\Delta_{\mathcal{U}} \subseteq \Delta_{\mathcal{G}}$: via Lemma C.9.
Case $\Delta_{\mathcal{G}} \subseteq \Delta_{\mathcal{U}}$: via Lemma C.6.

□

Lemma C.51 ($\mathcal{U} - \mathcal{C}$-Comm). *$\mathcal{U}$ and $\mathcal{C}$ commute.*

Proof. We show that $\mathcal{U}$ $*$-commutes with $\mathcal{C}$, and hence by Lemma B.34, $\mathcal{U}$ commutes wth $\mathcal{C}$. Let $\Delta_{\mathcal{U}} \rightarrow_{\mathcal{U}} \Delta'_{\mathcal{U}}$ and $\Delta_{\mathcal{C}} \rightarrow_{\mathcal{C}} \Delta'_{\mathcal{C}}$ denote the reducts for $\mathcal{U}$ and $\mathcal{C}$ respectively. If the reducts are disjoint then terms can be trivially joined. Let us split cases on whether $\Delta_{\mathcal{U}}$ occurs under $\Delta_{\mathcal{C}}$ or vice versa.

Case $\Delta_{\mathcal{U}} \subseteq \Delta_{\mathcal{C}}$ via Lemma C.10.
Case $\Delta_{\mathcal{C}} \subseteq \Delta_{\mathcal{U}}$ via Lemma C.6.

□

Lemma C.52. *$\mathcal{N}$ and $\mathcal{A}$ commute.*

Proof. We show that $\mathcal{N}$ strongly commutes with $\mathcal{A}$, hence commutes via Lemma B.19. Let $\Delta_{\mathcal{A}} \rightarrow_{\mathcal{A}} \Delta'_{\mathcal{A}}$ and $\Delta_{\mathcal{N}} \rightarrow_{\mathcal{N}} \Delta'_{\mathcal{N}}$ denote the reducts for $\mathcal{A}$ and $\mathcal{N}$ respectively. If the reducts are disjoint then terms can be trivially joined in a single step. Let us split cases on whether $\Delta_{\mathcal{A}}$ occurs under $\Delta_{\mathcal{N}}$ or vice versa.

Case $\Delta_{\mathcal{A}} \subseteq \Delta_{\mathcal{N}}$ via Lemma C.8.
Case $\Delta_{\mathcal{N}} \subseteq \Delta_{\mathcal{A}}$ via Lemma C.5.

□

Lemma C.53. *$\mathcal{N}$ and $\mathcal{G}$ commute.*

Proof. We show that $\mathcal{N}$ strongly commutes with $\mathcal{G}$, hence commutes via Lemma B.19. Let $\Delta_{\mathcal{G}} \rightarrow_{\mathcal{G}} \Delta'_{\mathcal{G}}$ and $\Delta_{\mathcal{N}} \rightarrow_{\mathcal{N}} \Delta'_{\mathcal{N}}$ denote the reducts for $\mathcal{G}$ and $\mathcal{N}$ respectively. If the reducts are disjoint then terms can be trivially joined in a single step. Let us split cases on whether $\Delta_{\mathcal{G}}$ occurs under $\Delta_{\mathcal{N}}$ or vice versa.

Case $\Delta_{\mathcal{G}} \subseteq \Delta_{\mathcal{N}}$ via Lemma C.8.
Case $\Delta_{\mathcal{N}} \subseteq \Delta_{\mathcal{G}}$ via Lemma C.9.

□

Lemma C.54. *$\mathcal{N}$ and C commute.*

Proof. We show that $\mathcal{N}$ strongly commutes with C, hence commutes via Lemma B.19. Let $\Delta_C \rightarrow_C \Delta'_C$ and $\Delta_{\mathcal{N}} \rightarrow_{\mathcal{N}} \Delta'_{\mathcal{N}}$ denote the reducts for C and $\mathcal{N}$ respectively. If the reducts are disjoint then terms can be trivially joined in a single step. Split cases on whether Δ_C occurs under $\Delta_{\mathcal{N}}$ or vice versa.

Case $\Delta_C \subseteq \Delta_{\mathcal{N}}$ via Lemma C.8.
Case $\Delta_{\mathcal{N}} \subseteq \Delta_C$ via Lemma C.10.

□

C.8 Application

Lemma C.55. *$\mathcal{A}$ is confluent.*

Proof. We show that $\mathcal{A}$ satisfies the diamond property and hence, is confluent by Lemma B.10. Suppose that $e \rightarrow_{\mathcal{A}} e_1$ via the redux $\Delta_1 \rightarrow_{\mathcal{A}} \Delta'_1$, and $e \rightarrow_{\mathcal{A}} e_2$ via the redux $\Delta_2 \rightarrow_{\mathcal{A}} \Delta'_2$. If Δ_1 and Δ_2 are disjoint in e, the terms e_1 and e_2 can be trivially joined in a single step. If $\Delta_1 \subseteq \Delta_2$ (or $\Delta_2 \subseteq \Delta_1$) then Lemma C.5 completes the proof. □

Lemma C.56. *$\mathcal{A}$ and $\mathcal{G}$ commute.*

Proof. We show that $\mathcal{A}$ strongly commutes with $\mathcal{G}$, hence commutes via Lemma B.19. Let $\Delta_{\mathcal{A}} \rightarrow_{\mathcal{A}} \Delta'_{\mathcal{A}}$ and $\Delta_{\mathcal{G}} \rightarrow_{\mathcal{G}} \Delta'_{\mathcal{G}}$ denote the reducts for $\mathcal{A}$ and $\mathcal{G}$ respectively. If the reducts are disjoint then terms can be trivially joined in a single step. Let us split cases on whether $\Delta_{\mathcal{A}}$ occurs under $\Delta_{\mathcal{N}}$ or vice versa.

Case $\Delta_{\mathcal{A}} \subseteq \Delta_{\mathcal{G}}$ via Lemma C.9.
Case $\Delta_{\mathcal{G}} \subseteq \Delta_{\mathcal{A}}$ via Lemma C.5.

□

Lemma C.57. *$\mathcal{A}$ and C commute.*

Proof. We show that $\mathcal{A}$ strongly commutes with C, hence commutes via Lemma B.19. Let $\Delta_{\mathcal{A}} \rightarrow_{\mathcal{A}} \Delta'_{\mathcal{A}}$ and $\Delta_C \rightarrow_C \Delta'_C$ denote the reducts for $\mathcal{A}$ and C respectively. If the reducts are disjoint then terms can be trivially joined in a single step. Let us split cases on whether $\Delta_{\mathcal{A}}$ occurs under Δ_{C} or vice versa.

Case $\Delta_{C} \subseteq \Delta_{\mathcal{A}}$ via Lemma C.5.

Case $\Delta_{\mathcal{A}} \subseteq \Delta_{C}$ via Lemma C.10.

□

C.9 Garbage Collection

Lemma C.58. *$\mathcal{G}$ is confluent.*

Proof. We show that $\mathcal{G}$ satisfies the diamond property and hence, is confluent by Lemma B.10. Suppose that $e \longrightarrow_{\mathcal{G}} e_1$ via the redux $\Delta_1 \longrightarrow_{\mathcal{G}} \Delta_1'$ $e \longrightarrow_{\mathcal{G}} e_2$ via the redux $\Delta_2 \longrightarrow_{\mathcal{G}} \Delta_2'$. If Δ_1 and Δ_2 are disjoint, the terms e_1 and e_2 can be trivially joined in a single step. If $\Delta_1 \subseteq \Delta_2$ (or $\Delta_2 \subseteq \Delta_1$) then Lemma C.9 completes the proof. □

Lemma C.59. *$\mathcal{G}$ and C commute.*

Proof. We show that $\mathcal{G}$ and C strongly commute. Let $\Delta_{\mathcal{G}} \longrightarrow_{\mathcal{G}} \Delta_{\mathcal{G}}'$ and $\Delta_{C} \longrightarrow_{C} \Delta_{C}'$ denote the reducts for $\mathcal{G}$ and $\mathcal{G}$ respectively. If the reducts are disjoint then terms can be trivially joined in a single step. Let us split cases on whether $\Delta_{\mathcal{G}}$ occurs under Δ_{C} or vice versa.

Case $\Delta_{\mathcal{G}} \subseteq \Delta_{C}$**:** via Lemma C.10.

Case $\Delta_{C} \subseteq \Delta_{\mathcal{G}}$**:** via Lemma C.9.

□

C.10 Choice

Lemma C.60. *C is confluent.*

Proof. Lemma C.10 shows that C has the diamond property (as $C \subseteq \mathcal{R}$), and hence C is confluent via Lemma B.10. □

D SKEW CONFLUENCE

[This is a sketch of an incomplete proof of skew confluence.]

We now consider a version of $\mathcal{VC}$ that fully supports recursive substitution, and lifts the pesky no-recursion precondition on the confluence theorem. Rather than lifting the side condition $x \notin \mathsf{fvs}(v)$ on rule SUBST, which avoids using a recursive equation for substitution, we take another approach that we believe leads to a simpler proof: we introduce the familiar, conventional fixpoint operator $\mu x.\, hnf$, but allow it to be applied only to head values, not to general expressions. A new unification rule U-OCCURS-WRAP can turn a recursive equation into one that is not recursive (by packaging up the right-hand side within a fixpoint), after which rule SUBST may be applied. A corresponding new rule U-UNWRAP can expand a fixpoint by applying the conventional rewrite rule $\mu x.\, hnf \longrightarrow hnf\{\mu x.\, hnf/x\}$. While this rule allows infinite application, a sensible evaluation strategy would apply this rule "only when needed"—when it is on either side of an equation, or when it is in the function position of an application. If we regard any $\mathcal{VC}$ data structure as tree, the fixpoint construct in effect can label any subtree in such a way that any node beneath it can have a "back pointer" up to the labeled node by referring to the bound variable that serves as the label.

D.1 Free Variables

We use the conventional notation $\mathsf{fvs}(e)$ to denote the set of variables that occur free in the expression e. Variables are bound by the constructs $\exists x.\, e$, $\lambda x.\, e$, and $\mu x.\, hnf$. Figure 15 contains a formal definition of this function for $\mathcal{VC}$.

We use the variation $\mathsf{fvsol}(e)$ to denote the set of variables that occur free in the expression e in at least one position that is not within the body of a lambda expression[13]. As an example, $\mathsf{fvsol}(\exists x.\, \langle x, f, g, \lambda y.\, \langle x, g, y, z\rangle\rangle) = \{f, g\}$ because:

- x is bound by $\exists$, so it is not free.
- f is free in a position not within the body of a lambda expression.
- g is free in at least one position not within the body of a lambda expression (it also happens to be free in a second position that is within the body of a lambda expression).
- y is bound by λ, so it is not free.
- z is free, but appears only in a position that is within the body of a lambda expression.

Figure 16 contains a formal definition of this function. Unlike $\mathsf{fvs}(e)$, $\mathsf{fvsol}(v)$ is only ever appplied to a value v.

D.2 Substitution

We use the notation $e\{v/x\}$ to denote the expression that results from performing standard capture-avoiding substitution of the value v for every occurrence of the variable x within the expression e (it turns out that, for $\mathcal{VC}$, substitution of general expressions for variables is not required, only substitution of values for variables). Figure 17 contains a formal definition of how this notation applies to the $\mathcal{VC}$ grammar (compare [Barendregt 1984, §2.1.15]).

D.3 Modified grammar and rewrite rules

Let us modify the grammar for $\mathcal{VC}$ to have one additional kind of value, a *fixpoint* value $\mu x.\, hnf$:

$$\text{Values} \quad v ::= x \mid hnf \mid \mu x.\, hnf$$

and a modify the set of Unification rewrite rules $\mathcal{U}$ so that rule U-OCCURS

$$\text{U-OCCURS} \quad x = V[\,x\,] \longrightarrow \mathbf{fail} \quad \text{if } V \neq \square$$

[13] "$\mathsf{fvsol}(\cdot)$" abbreviates "free variables outside lambda"

$$
\begin{array}{rcl}
\mathsf{fvs}(x) & = & \{x\} \\
\mathsf{fvs}(k) & = & \{\,\} \\
\mathsf{fvs}(op) & = & \{\,\} \\
\mathsf{fvs}(\langle v_1, \ldots, v_n \rangle) & = & \mathsf{fvs}(v_1) \cup \cdots \cup \mathsf{fvs}(v_n) \\
\mathsf{fvs}(\lambda x.\, e) & = & \mathsf{fvs}(e) \setminus \{x\} \\
\mathsf{fvs}(\mu x.\, e) & = & \mathsf{fvs}(e) \setminus \{x\} \\
\mathsf{fvs}(eq;\, e) & = & \mathsf{fvs}(eq) \cup \mathsf{fvs}(e) \\
\mathsf{fvs}(v = e) & = & \mathsf{fvs}(v) \cup \mathsf{fvs}(e) \\
\mathsf{fvs}(\exists x.\, e) & = & \mathsf{fvs}(e) \setminus \{x\} \\
\mathsf{fvs}(\mathbf{fail}) & = & \{\,\} \\
\mathsf{fvs}(e_1 \,\|\, e_2) & = & \mathsf{fvs}(e_1) \cup \mathsf{fvs}(e_2) \\
\mathsf{fvs}(v_1\ v_2) & = & \mathsf{fvs}(v_1) \cup \mathsf{fvs}(v_2) \\
\mathsf{fvs}(\mathbf{one}\{e\}) & = & \mathsf{fvs}(e) \\
\mathsf{fvs}(\mathbf{all}\{e\}) & = & \mathsf{fvs}(e)
\end{array}
$$

Fig. 15. Definition of the free-variables function $\mathsf{fvs}(e)$

$$
\begin{array}{rcl}
\mathsf{fvsol}(x) & = & \{x\} \\
\mathsf{fvsol}(k) & = & \{\,\} \\
\mathsf{fvsol}(op) & = & \{\,\} \\
\mathsf{fvsol}(\langle v_1, \ldots, v_n \rangle) & = & \mathsf{fvsol}(v_1) \cup \cdots \cup \mathsf{fvsol}(v_n) \\
\mathsf{fvsol}(\lambda x.\, e) & = & \{\,\}
\end{array}
$$

Fig. 16. Definition of the free-variables-outside-lambdas function $\mathsf{fvsol}(v)$

is replaced by the two rules[14][15]

$$
\begin{array}{llll}
\textsc{u-occurs-fail} & x = hnf;\ e \longrightarrow \mathbf{fail} & & \text{if } x \in \mathsf{fvsol}(hnf) \\
\textsc{u-occurs-wrap} & x = hnf;\ e \longrightarrow x = \mu x.\, hnf;\ e & & \text{if } x \in \mathsf{fvs}(hnf) \text{ and } x \notin \mathsf{fvsol}(hnf)
\end{array}
$$

Let us also add this rewrite rule:

$$
\textsc{u-unwrap} \quad \mu x.\, hnf \longrightarrow hnf\{\mu x.\, hnf / x\}
$$

As we will see, rule u-unwrap is confluent but not Noetherian.

The resulting grammar is not confluent; in particular, it suffers from the even-odd problem described in Section 4.1 [Ariola and Blom 2002, Example 4.1]. Therefore we will modify the proof of confluence for $\mathcal{U}$ to become a proof of *skew confluence* [Ariola and Blom 2002], and then go on to prove that $\mathcal{VC}$ itself, with this modification, is skew confluent.

D.4 Unwrapping of Fixpoints is Confluent but not Noetherian

Lemma D.1. *The rule u-unwrap is confluent.*

[14] These two rules allow equations to be recursive through lambda expressions and possibly also tuples, but not through tuples only; thus equations such as $f = \lambda y.\, \langle y, f \rangle$ and $x = \langle 1, \lambda y.\, \langle y, x \rangle \rangle$ can be processed by rule u-occurs-wrap, but the equation $x = \langle 1, x \rangle$ can be processed only by rule u-occurs-fail. An alternate design using the single rule

$$
\textsc{u-occurs-wrap} \quad x = hnf \longrightarrow x = \mu x.\, hnf \quad \text{if } x \in \mathsf{fvs}(hnf)
$$

plus rule u-unwrap could be used instead to support recursion through tuples only as well as through lambda expressions.

[15] u-occurs-fail is identical to u-occurs in its effect, but it is re-expressed using $\mathsf{fvsol}(\cdot)$, which we need anyway for u-occurs-wrap. Now we can drop the context V, which was only used in u-occurs.

$$
\begin{array}{rcll}
x\{v/x\} & \equiv & v & \\
y\{v/x\} & \equiv & y & \text{if } y \neq x \\
k\{v/x\} & \equiv & k & \\
op\{v/x\} & \equiv & op & \\
\langle v_1, \ldots, v_n \rangle\{v/x\} & \equiv & \langle v_1\{v/x\}, \ldots, v_n\{v/x\}\rangle & \\
(\lambda y.\, e)\{v/x\} & \equiv & \lambda y.\, e\{v/x\} & \text{if } y \notin \text{fvs}(x, v) \text{ [use } \alpha] \\
(\mu y.\, v')\{v/x\} & \equiv & \mu y.\, v'\{v/x\} & \text{if } y \notin \text{fvs}(x, v) \text{ [use } \alpha] \\
(eq;\, e)\{v/x\} & \equiv & eq\{v/x\};\, e\{v/x\} & \\
(v' = e)\{v/x\} & \equiv & v'\{v/x\} = e\{v/x\} & \\
(\exists y.\, e)\{v/x\} & \equiv & \exists y.\, e\{v/x\} & \text{if } y \notin \text{fvs}(x, v) \text{ [use } \alpha] \\
\mathbf{fail}\{v/x\} & \equiv & \mathbf{fail} & \\
(e_1 \mid e_2)\{v/x\} & \equiv & e_1\{v/x\} \mid e_2\{v/x\} & \\
(v_1\ v_2)\{v/x\} & \equiv & v_1\{v/x\}\ v_2\{v/x\} & \\
(\mathbf{one}\{e\})\{v/x\} & \equiv & \mathbf{one}\{e\{v/x\}\} & \\
(\mathbf{all}\{e\})\{v/x\} & \equiv & \mathbf{all}\{e\{v/x\}\} &
\end{array}
$$

Fig. 17. Definition of the substitution notation $e\{v/x\}$

Proof. Suppose that $e \longrightarrow_{\text{U-UNWRAP}} e_1$ and $e \longrightarrow_{\text{U-UNWRAP}} e_2$ for distinct redexes within e.

If the redexes are disjoint, then Lemma C.4 applies.

Otherwise, without loss of generality assume the redex for $e \longrightarrow_{\text{U-UNWRAP}} e_1$ contains the redex for $e \longrightarrow_{\text{U-UNWRAP}} e_2$; let e must be of the form $C_1[\mu x.\, C_2[\mu y.\, hnf]]$ (α-conversion may be used to ensure that x and y are distinct variables).

Then $e_1 = C_1[C_2[\mu y.\, hnf]\{\mu x.\, C_2[\mu y.\, hnf]/x\}]$ and $e_2 = C_1[\mu x.\, C_2[hnf\{\mu y.\, hnf/y\}]]$.

From e_2 we can take just one more U-UNWRAP step, using the outermost redex $\mu x.\, C_2[cdots]$, obtaining $e' = C_1[(C_2[hnf\{\mu y.\, hnf/y\}])\{\mu x.\, C_2[hnf\{\mu y.\, hnf/y\}]/x\}]$.

[More to come.]

Thus we have $e_1 \twoheadrightarrow_{\text{U-UNWRAP}} e'$ and $e_2 \longrightarrow_{\text{U-UNWRAP}} e'$, so U-UNWRAP is strongly confluent, and therefore by Lemma B.16 is confluent. □

To see that U-UNWRAP is not Noetherian, observe that

$$\mu x.\, \langle 1, x\rangle \longrightarrow_{\text{U-UNWRAP}} \langle 1, \mu x.\, \langle 1, x\rangle\rangle \longrightarrow_{\text{U-UNWRAP}} \langle 1, \langle 1, \mu x.\, \langle 1, x\rangle\rangle\rangle \longrightarrow_{\text{U-UNWRAP}} \cdots$$

is an unending sequence of reduction steps.

D.5 Information Content

We define a second grammar, for a second language $\mathcal{VC}_\Omega$, by adding one more value Ω, which indicates a lack of information as to just what will be computed. When Ω appears in a context where only a value is permitted, it indicates uncertainty as to what value will be provided; when Ω appears in a context where any expression permitted, it furthermore indicates uncertainty as to how many values will be provided (possibly none).

$$\text{Values} \quad v ::= x \mid hnf \mid \mu x.\, hnf \mid \Omega$$

For every term of $\mathcal{VC}$ there is a corresponding term of $\mathcal{VC}_\Omega$, identical in structure and appearance and containing no occurrence of Ω; we identify such terms and regard the set of terms of $\mathcal{VC}$ as simply a subset of the terms of $\mathcal{VC}_\Omega$.

The definition of substitution (Fig. 17) is extended in the expected trivial manner: $\Omega\{v/x\} \equiv \Omega$.

Information Content: $\mathcal{I}$

INFO-FIX	$\mu x.\, v \longrightarrow \Omega$
INFO-SEQ	$eq;\ e \longrightarrow \Omega$
INFO-EXI	$\exists x.\, e \longrightarrow \Omega$
INFO-FAIL-L	$\mathbf{fail} \mid e \longrightarrow \Omega$
INFO-FAIL-R	$e \mid \mathbf{fail} \longrightarrow \Omega$
INFO-CHOICE-OMEGA	$\Omega \mid e \longrightarrow \Omega$
INFO-CHOICE-ASSOC	$(e_1 \mid e_2) \mid e_3 \longrightarrow \Omega$
INFO-APP-OMEGA	$\Omega\ v \longrightarrow \Omega$
INFO-APP-HNF	$hnf\ v \longrightarrow \Omega$
INFO-ONE	$\mathbf{one}\{e\} \longrightarrow \Omega$
INFO-ALL	$\mathbf{all}\{e\} \longrightarrow \Omega$

Fig. 18. Rewrite rules on $\mathcal{VC}_\Omega$ for defining the function $\omega_{\text{VC}}(e)$

Definition D.2. (after [Ariola and Blom 2002, Definition 2.3]) Let T be a set of terms over a signature that includes the constant Ω. Define $\leqslant_\omega{}^T$ be the relation such that $\Omega \leqslant_\omega{}^T M$ for every term $M \in T$; then define $\leqslant_\omega$ to be the transitive, reflexive, and compatible closure of $\leqslant_\omega{}^T$.

Figure 18 shows a system $\mathcal{I}$ of rewrite rules on $\mathcal{VC}_\Omega$. These may be compared to similar rules for the $\lambda\circ$ calculus [Ariola and Blom 2002, Definition 5.20].

LEMMA D.3. *[Huet 1980, Lemma 3.1] The relation $\longrightarrow_\mathcal{R}$ is locally confluent iff for every critical pair (e_1, e_2) of $\mathcal{R}$, e_1 and e_2 can be* joined*—that is, there exists e_3 such that $e_1 \longrightarrow_\mathcal{R} e_3$ and $e_2 \longrightarrow_\mathcal{R} e_3$.*

LEMMA D.4 ($\mathcal{I}$-CONFLUENT). *$\mathcal{I}$ is locally confluent and Noetherian; therefore $\mathcal{I}$ is confluent.*

PROOF. Consider all critical pairs of $\mathcal{I}$:

- Rules INFO-FAIL-L and INFO-FAIL-R produce the critical pair (Ω, Ω).
- Rules INFO-FAIL-L and INFO-CHOICE-OMEGA produce no critical pairs.
- Rules INFO-FAIL-L and INFO-CHOICE-ASSOC produce the critical pair $(\Omega \mid e, \Omega)$.
- Rules INFO-FAIL-R and INFO-CHOICE-OMEGA produce the critical pair (Ω, Ω).
- Rules INFO-FAIL-R and INFO-CHOICE-ASSOC produce the critical pairs (Ω, Ω) and $(\Omega \mid e, \Omega)$.
- Rules INFO-CHOICE-OMEGA and INFO-CHOICE-ASSOC produce the critical pair $(\Omega \mid e, \Omega)$.
- No other pairs of rules produce any critical pairs.

The critical pair (Ω, Ω) can be trivially joined at Ω. The critical pair $(\Omega \mid e, \Omega)$ can be joined at Ω in one step by using rule INFO-CHOICE-OMEGA on $\Omega \mid e$.

All critical pairs can be joined; therefore by Lemma D.3 $\mathcal{I}$ is locally confluent.

Let the size of a term of $\mathcal{VC}_\Omega$ be the number of tokens it contains other than parentheses. Each of the rewrite rules in Fig. 18 strictly decreases the number of such tokens. Because any given term has a finite number n of such tokens, and the number of tokens cannot be less than zero, for every term every rewriting sequence from that term can have no more than n steps. So $\mathcal{I}$ is bounded and therefore Noetherian.

Because $\mathcal{I}$ is locally confluent and Noetherian, it is confluent by Newman's lemma. □

Because $\mathcal{I}$ is confluent and Noetherian, it follows that $\mathcal{I}$ defines unique normal forms for $\mathcal{VC}_\Omega$. Therefore we are justified in defining $\omega_{\text{VC}}(e)$ to be the function that takes any term in $\mathcal{VC}_\Omega$ and returns the term that is its normal form under $\mathcal{I}$.

Definition D.5. The comparison $e \leqslant_{\omega_{\text{VC}}} e'$ is defined to mean $\omega_{\text{VC}}(e) \leqslant_\omega \omega_{\text{VC}}(e')$.

[Need to prove that $\leqslant_{\omega_{VC}}$ is monotonic with respect to $\leqslant_\omega$; this should be routine [Ariola and Blom 2002, Proposition 5.21].]

Lemma D.6 ($\mathcal{VC}$ monotonic). *Every rewrite rule in $\mathcal{VC}$ is monotonic with respect to $\leqslant_{\omega_{VC}}$.*

Proof. For every rewrite rule in $\mathcal{A} \cup \mathcal{U} \cup \mathcal{N} \cup \mathcal{G} \cup \mathcal{C}$, the left-hand side is an expression that is mapped to Ω by the function ω_{VC}, and no matter what expression e is the result of applying ω_{VC} to the right-hand side, we have $\Omega \leqslant_\omega e$. □

D.6 Preliminaries about Skew Confluence

Why use skew confluence? Ordinary confluence is useful because if term e has an $\mathcal{R}$-normal form, then that normal form is *unique* if $\mathcal{R}$ is confluent. Ariola and Blom define a related notion, which we will refer to as $\mathcal{R}$-*skew-normal form*[16], and prove that under certain conditions, if term e has an $\mathcal{R}$-skew-normal form, then that normal form is unique if $\mathcal{R}$ is skew confluent.

A $\mathcal{R}$-skew-normal form is not a single term, but rather a possibly infinite set of *erased* terms. We summarize this idea, using our own terminology, as follows:

- Let an *erasure* of a term be a copy in which some number of subterms (possibly zero, and possibly the entire term) have been replaced with Ω, a special term that means "unknown" or "we don't know yet."
- We can compare erased terms with the partial order $\leqslant_\omega$, which is the transitive, reflexive, and compatible closure of the relation in which Ω is less than any other term. Observe that if e' is any erasure of e (including e itself) then $e' \leqslant_\omega e$.
- Define the $\mathcal{R}$-*information content* $\omega_\mathcal{R}(e)$ of a term e to be the unique erasure of e in which *every* redex has been replaced by Ω. Any non-Ω structure in the result is therefore "permanent": no further reductions under $\mathcal{R}$ can alter that structure.
- Define the *downward closure* $\Downarrow_{\leqslant_\omega} A$ of a set of terms A to be the set of all elements of A and all possible erasures of those elements, that is, $\Downarrow_{\leqslant_\omega} A = \{e' \mid e \in A, e' \leqslant_\omega\}$.
- Define the $\mathcal{R}$-*skew-normal form* of e to be the downward closure of the set of information contents of all possible $\mathcal{R}$-reducts of e, that is, $\Downarrow_{\leqslant_\omega}\{\omega(e') \mid e \twoheadrightarrow_\mathcal{R} e'\}$.

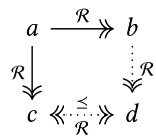
Taking the downward closure with respect to $\leqslant_\omega$ is crucial; without that step, it would not be possible to prove that skew-normal forms are unique for certain reduction relations.

Skew confluence is a natural extension of confluence, in this sense: if $\mathcal{R}$ is skew confluent, then an expression e has a unique $\mathcal{R}$-normal form if and only if its $\mathcal{R}$-skew-normal form is the (finite) set consisting of all possible erasures of that $\mathcal{R}$-normal form. (For example, $\langle 1, 2\rangle$ is the unique normal form of $\exists x.\ x = 2;\ \langle 1, x\rangle$, and the $\mathcal{R}$-skew-normal form of that same term is $\{\langle 1, 2\rangle, \langle \Omega, 2\rangle, \langle 1, \Omega\rangle, \langle \Omega, \Omega\rangle, \Omega\}$.)

But working with possibly infinite sets directly is tricky. Fortunately, there is a simpler path, because Ariola and Blom prove an important theorem: Define the partial order $e \preceq_{\omega_\mathcal{R}} e'$ to mean $\omega_\mathcal{R}(e) \leqslant_\omega \omega_\mathcal{R}(e')$; then a reduction relation that is *monotonic* in $\preceq_{\omega_\mathcal{R}}$ ($e \rightarrow_\mathcal{R} e' \implies e \preceq_{\omega_\mathcal{R}} e'$) has unique skew-normal forms if and only if the reduction relation is skew confluent [Ariola and Blom 2002, Theorem 5.4]—and skew confluence is much easier to prove, using techniques that do not involve possibly infinite sets, but are fairly similar to proofs of ordinary confluence that use commutative diagrams and case analysis. They also prove that if a reduction relation is confluent and monotonic, then it is skew confluent [Ariola and Blom 2002, Corollary 5.5].

Definition D.7. Reduction relation $\mathcal{R}$ over the set of terms T is *skew confluent* using quasi order $\preceq$ if for all $a, b, c \in T$, if $a \twoheadrightarrow_\mathcal{R} b$ and $a \twoheadrightarrow_\mathcal{R} c$, there exists $d \in T$ such that $b \twoheadrightarrow_\mathcal{R} d$ and $c \preceq d$. As a diagram:

[16] Ariola and Blom call it the "infinite normal form"; this is a bit misleading because in fact not all such forms are infinite.

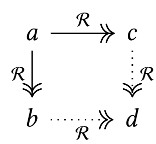


[More to come.]

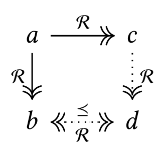
D.7 New Lemmas about Skew Confluence

Lemma D.8. *If relation $\mathcal{R}$ is monotonic in some quasi order $\preceq$ and confluent, then it is skew confluent using $\preceq$.*

Proof. By the definition of confluence,



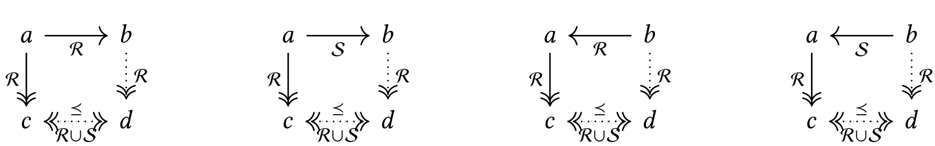
Because $\mathcal{R}$ is monotonic, $\twoheadrightarrow_{\mathcal{R}} \subset \underset{\mathcal{R}}{\overset{\preceq}{\twoheadleftarrow\twoheadrightarrow}}$, therefore



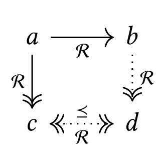
□

Definition D.9. Let reduction relation $\mathcal{R}$ be monotonic in quasi order $\preceq$ and skew confluent using $\preceq$. Let reduction relation $\mathcal{R}^{\leftarrow}_{\preceq}$ be defined by a set of rewrite rules that are converses of those rewrite rules of $\mathcal{R}$ whose converses are used in the proof that $\mathcal{R}$ is skew confluent. Then we say that $\mathcal{R}$ is *skew confluent using $\preceq$ and $\mathcal{R}^{\leftarrow}_{\preceq}$*.

Lemma D.10. *Let relation $\mathcal{R}$ be monotonic in quasi order $\preceq$ and skew confluent using $\preceq$ and $\mathcal{R}^{\leftarrow}_{\preceq}$; similarly let $\mathcal{S}$ be monotonic in the same quasi order $\preceq$ and skew confluent using $\preceq$ and $\mathcal{S}^{\leftarrow}_{\preceq}$. Suppose furthermore that $\mathcal{R}$ and $\mathcal{S}$ commute and that $\mathcal{R}$ and $\mathcal{S}^{\leftarrow}_{\preceq}$ commute. Then the following four diagrams hold:*

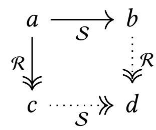


Proof. (1) Because $\mathcal{R}$ is skew confluent, we have



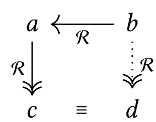
Because $\underset{\mathcal{R}}{\overset{\preceq}{\twoheadleftarrow\twoheadrightarrow}} \subset \underset{\mathcal{R}\cup\mathcal{S}}{\overset{\preceq}{\twoheadleftarrow\twoheadrightarrow}}$, the first diagram follows.

(2) Because $\mathcal{R}$ and $\mathcal{S}$ commute, we have



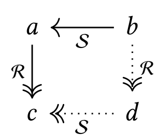
Because $\twoheadrightarrow_{\mathcal{S}} \subset \overset{\leq}{\twoheadleftarrow\!\!\twoheadrightarrow}_{\mathcal{R}\cup\mathcal{S}}$, the second diagram follows.

(3) The following diagram clearly holds if the sequence of reduction steps from b to d is the same as the sequence of reduction steps from b to a to d:



Because $\equiv \subset \overset{\leq}{\twoheadleftarrow\!\!\twoheadrightarrow}_{\mathcal{R}\cup\mathcal{S}}$, the third diagram follows.

(4) Because $\mathcal{R}$ and $\mathcal{S}^{\leftarrow}_{\leq}$ commute, we have

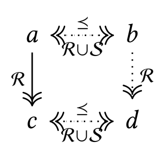


Because $\twoheadleftarrow_{\mathcal{S}} \subset \overset{\leq}{\twoheadleftarrow\!\!\twoheadrightarrow}_{\mathcal{R}\cup\mathcal{S}}$, the fourth diagram follows.

[It may turn out to be impossible to prove for our specific application that $\mathcal{R}$ and $\mathcal{S}^{\leftarrow}_{\leq}$ commute. In that case, it may be necessary to use a more complicated or more subtle precondition. The important thing is to prove the fourth diagram somehow.]

□

Lemma D.11. *Let relation $\mathcal{R}$ be monotonic in quasi order $\leq$ and skew confluent using $\leq$ and $\mathcal{R}^{\leftarrow}_{\leq}$; similarly let $\mathcal{S}$ be monotonic in the same quasi order $\leq$ and skew confluent using $\leq$ and $\mathcal{S}^{\leftarrow}_{\leq}$. Suppose furthermore that $\mathcal{R}$ and $\mathcal{S}$ commute and that $\mathcal{R}$ and $\mathcal{S}^{\leftarrow}_{\leq}$ commute. Then*

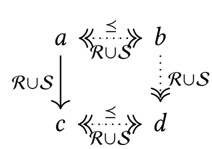


Proof. By induction on the size of the top edge of the diagram. At each step one of the four diagrams from Lemma D.10 will be used.

[More to come.]

□

Lemma D.12. *Let relation $\mathcal{R}$ be monotonic in quasi order $\leq$ and skew confluent using $\leq$ and $\mathcal{R}^{\leftarrow}_{\leq}$; similarly let $\mathcal{S}$ be monotonic in the same quasi order $\leq$ and skew confluent using $\leq$ and $\mathcal{S}^{\leftarrow}_{\leq}$. Then*

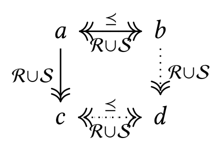


Proof. By case analysis on whether left edge uses $\mathcal{R}$ or $\mathcal{S}$; then project that left edge into $\mathcal{R}^*$ or $\mathcal{S}^*$ respectively and apply Lemma D.11.

[More to come.]

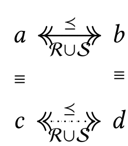
□

Lemma D.13. *Let relation $\mathcal{R}$ be monotonic in quasi order $\preceq$ and skew confluent using $\preceq$ and $\mathcal{R}^{\leftarrow}_{\preceq}$; similarly let $\mathcal{S}$ be monotonic in the same quasi order $\preceq$ and skew confluent using $\preceq$ and $\mathcal{S}^{\leftarrow}_{\preceq}$. Then*

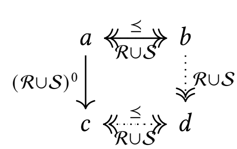


Proof. By induction on the size of the left edge of the diagram.

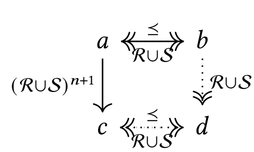
Base case This diagram clearly holds by letting the bottom edge be the same as the top edge:



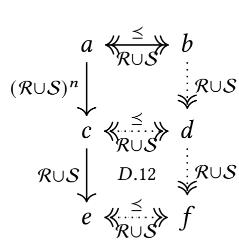
and it implies this diagram:



Inductive case Assume the diagram holds for left edges of all sizes up to n. Then this diagram:



follows from this diagram:



where the top half is the inductive hypothesis and the bottom half follows from Lemma D.12.

□

Lemma D.14. *Let relation $\mathcal{R}$ be monotonic in quasi order $\preceq$ and skew confluent using $\preceq$ and $\mathcal{R}^{\leftarrow}_{\preceq}$; similarly let $\mathcal{S}$ be monotonic in the same quasi order $\preceq$ and skew confluent using $\preceq$ and $\mathcal{S}^{\leftarrow}_{\preceq}$. If $\mathcal{R}$ commutes with $\mathcal{S}$, then $\mathcal{T} = \mathcal{R} \cup \mathcal{S}$ is monotonic in $\preceq$ and skew confluent using $\preceq$ and $\mathcal{T}^{\leftarrow}_{\preceq}$.*

Proof. □

D.8 Proof that $\mathcal{VC}$ Is Skew Confluent

[This is just a brief proof sketch.]

First prove that the modified $\mathcal{U}$ is skew confluent. (In doing so we will define $\mathcal{U}^{\leftarrow}_{\leqslant_{\omega_{\mathrm{VC}}}}$.)

Then use existing proofs to demonstrate that $\mathcal{A} \cup \mathcal{N} \cup \mathcal{G} \cup \mathcal{C}$ is confluent. Because they are also monotonic, they are therefore skew confluent, and $(\mathcal{A} \cup \mathcal{N} \cup \mathcal{G} \cup \mathcal{C})^{\leftarrow}_{\leqslant_{\omega_{\mathrm{VC}}}}$ is trivial.

Prove that $\mathcal{A} \cup \mathcal{N} \cup \mathcal{G} \cup \mathcal{C}$ commutes with $\mathcal{U}^{\leftarrow}_{\leqslant_{\omega_{\mathrm{VC}}}}$.

Then apply Lemma D.14 to show that $\mathcal{U} \cup (\mathcal{A} \cup \mathcal{N} \cup \mathcal{G} \cup \mathcal{C})$ is skew confluent.

Domains

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

Semantics of expressions and values

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

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

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

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

Fig. 19. Expression semantics

E A DENOTATIONAL SEMANTICS FOR $\mathcal{VC}$

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

Domains

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

Operations

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

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

$$
\begin{array}{rcll}
s_1 \Cup s_2 & = & \bot & \text{if } s_1 = \bot \text{ or } s_2 = \bot \\
 & = & \text{WRONG} & \text{if } (s_1 = \text{WRONG and } s_2 \neq \bot) \text{ or } (s_2 = \text{WRONG and } s_1 \neq \bot) \\
 & = & s_1 \cup s_2 & \text{otherwise}
\end{array}
$$

Fig. 20. Set semantics for W^*

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

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

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

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

E.1 A first attempt at a denotational semantics

Our denotational semantics for $\mathcal{VC}$ is given in Fig. 19.

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

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

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

Our first interpretation, given in Figure 20, is a little more refined: W^* includes $\bot$ and WRONG as well as a set of values. Our second interpretation is given in Figure 21, and discussed in Appendix E.4.

The equations themselves, in Fig. 19 are beautifully simple and compositional, as a denotational semantics should be.

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

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

- Values $\mathcal{E}[\![v]\!]\,\rho$: compute the single value for v, and return a singleton sequence of results. The auxiliary function *unit* is defined at the bottom of Fig. 19.
- In particular, values include lambdas. The semantics says that a lambda evaluates to a *singleton* collection, whose only element is a function value. But that function value has type $W \to W^*$; that is, it is a function that takes a single value and returns a *collection* of values.
- Function application $\mathcal{E}[\![v_1\ v_2]\!]\,\rho$ is easy, because $\mathcal{V}$ returns a single value: just apply the meaning of the function to the meaning of the argument. The *apply* function is defined in Figure 19.
- Choice $\mathcal{E}[\![e_1 \mid e_2]\!]\,\rho$: take the union (written $\Cup$) of the values returned by e_1 and e_2 respectively. For bags this union operator is just bag union (Figure 20).
- Unification $\mathcal{E}[\![e_1 \mid e_2]\!]\,\rho$: take the *intersection* of the values returned by e_1 and e_2 respectively. For bags, this "intersection" operator $\Cap$ is defined in Fig. 20. In this definition, the equality is mathematical equality of functions; which we can't implement for functions; see Appendix E.1.
- Sequencing $\mathcal{E}[\![e_1;\ e_2]\!]\,\rho$. Again we use an auxiliary function $\fatsemi$ to combine the meanings of e_1 and e_2. For bags, the function $\fatsemi$ (Fig. 20 again) uses a bag comprehension. Again it does a cartesian product, but without the equality constraint of $\Cap$.
- The semantics of $(\mathbf{one}\{e\})$ simply applies the semantic function $one : W^* \to W^*$ to the collection of values returned by e. If e returns no values, so does $(\mathbf{one}\{e\})$; but if e returns one or more values, $(\mathbf{one}\{e\})$ returns the first. Of course that begs the question of what "the first" means – for bags it would be non-deterministic. We will fix this problem in Appendix E.4, but for now we simply ignore it.
- The semantics of $(\mathbf{all}\{e\})$ is similar, but it always returns a singleton collection (hence the *unit* in the semantics of **all**) whose element is a (possibly-empty) tuple that contains all the values in the collection returned by e.

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

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

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

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

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

E.2 The denotational semantics is un-implementable

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

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

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

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

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

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

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

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

So our touchstone must be:

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

E.3 Getting WRONG right

Getting WRONG right is a bit tricky.

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

There is probably more than one possible choice here.

E.4 An order-sensitive denotational semantics

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

Figure 21 give a new denotational semantics that *does* account for order. The key idea (due to Joachim Breitner) is this: return a sequence of *labelled* values; and then sort that sequence (in *one* and *all*) into canonical order before exposing it to the programmer.

We do not change the equations for $\mathcal{E}$, $\mathcal{V}$, and $\mathcal{O}$ at all; they remain precisely as they are in Figure 19. However the semantics of a collection of values, W^*, does change, and is given in Figure 21:

Domains

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

Operations

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

$$
\begin{array}{rrcll}
\text{Sort} & \mathit{sort} & : & LW^* \to ([W] + \text{WRONG})_\perp & \\
 & \mathit{sort}(s) & = & [] & \text{if } s \text{ is empty} \\
 & & = & \text{WRONG} & \text{if } \mathit{ws} \text{ has more than one element} \\
 & & = & \mathit{ws} & \text{otherwise} \\
 & & & \bowtie \mathit{sort}\{(l, w) \mid (\mathbf{L} : l, w) \in s\} & \\
 & & & \bowtie \mathit{sort}\{(l, w) \mid (\mathbf{R} : l, w) \in s\} & \\
 & & \multicolumn{3}{l}{\text{where } \mathit{ws} = [w \mid ([], w) \in s]}
\end{array}
$$

Fig. 21. Labelled set semantics for W^*

- A collection of values W^* is now $\bot$ or WRONG (as before), or a *set of labelled values*, each of type LW.
- A labelled value (of type LW) is just a pair $([L] \times W)$ of a *label* and a value.
- A label is a sequence of tags L, where a tag is just **L** or **R**, similar to Section 5.1.
- The union (or concatentation) operation $\Cup$, defined in Fig. 21, adds a **L** tag to the labels of the values in the left branch of the choice, and a **R** tag to those coming from the right. So the labels specify where in the tree the value comes from.
- Sequencing $;$ and $\Cap$ both concatenate the labels from the values they combine.
- Finally *sort* puts everything in the "right" order: first the values with an empty label, then the values whose label starts with **L** (notice the recursive sort of the trimmed-down sequence), and then those that start with **R**. Notice that *sort* removes all the labels, leaving just a bare sequence of values W^*.
- Note that if *sort* encounters a set with more than one unlabelled element then this considered WRONG. This makes ambiguous expressions, like $\textbf{one}\{\exists x.\, x\}$, WRONG.

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

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

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

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

E.5 Related work

[Christiansen et al. 2011] gives another approach to a denotational semantics for a functional logic language. We are keen to learn of others.

F UPDATEABLE REFERENCES

The full Verse language has updatable references (à la ML). There are three new primitive operations, **alloc**, **read**, and **write**. The **alloc** creates a new reference with an initial value, **read** extracts the value from a reference, and **write** sets the value of a referene.

Modifying these references is transactional in the sense that if a computation fails, then any updates will not be visible outside the construct that handles the failures. *E.g.*,
$r := \mathbf{alloc}(0);\ (\mathbf{if}\ (\mathbf{write}\langle r, 1\rangle;\ \mathbf{fail})\ \mathbf{then}\ 1\ \mathbf{else}\ 2);\ \mathbf{read}(r)$
will have the value 0, because the **write** is part of an expression that fails, and so its effect is not visible.

To add updateable references we extend the system with syntax and rules from figure 22. The **store** h **in** $\{e\}$ indicates that e should be reduced using the heap h. A heap is simply a mapping from references to values (one mapping being $r \mapsto v$). A reference is some opaque type that supports equality (unification) and creation of a new reference.

The interaction of the new primitives with the store can be seen from the axioms. The **alloc**(v) operation creates a new reference and adds a binding with v to the store. The **read**(r) operation retrieves the value for reference r from the store, and **write**$\langle r, v\rangle$ updates the reference r with v in the store. All of these operations use the context S which ensures that there are no store operations to the left of the hole, *i.e.*, a store operation in the hole is the next one that should execute.

The interesting rules involve choice and **split** because store operations are transactional in the sense that when an expressions fails, none of its store operations will happen.

When reducing **split**$(e)\{f, g\}$ in an S hole, rule ST-SPLIT-DUP, the store is duplicated. Any store operations inside the **split** will happen in this local copy of the store. Note the two occurrences of h in the right hand side of ST-SPLIT-DUP. If the reduction of e results in **fail** then rule FAIL-ELIM is used, and the store from the failing computation is simply thrown away. If the reduction of e results in a value (with or without more alternatives) then rule ST-SPLIT is used. This rule replaces the outer store with the inner store, since we know the inner computation has succeeded.

Similarely, the reduction of $e_1 \parallel e_2$ will duplicate the store into the first branch, ST-CHOICE-DUP. Here e_1 must not contain any store operation nor be a value. And again, similarely, ST-CHOICE commits the new store and throws away the old.

The use of *oe* in the rules is to ensure that the rules cannot get stuck in a loop. Using e instead of *oe* would mean that failing or committing would make the expression match the duplication rule again. It also prevents the duplication rule from repeatedly duplicating the **store**.

Note that **store** is part of the X context, which means that the **store** can float inside existentials. This is necessary for the store rules to fire since the S context does not allow going under existentials.

The semantics of **for**(d) **do** e with respect to store effects is somewhat intricate. The expression d is possibly multi-valued; any effects that happens when computing the first value of d will be visible the first time e is computed. Both these effects are then visible when computing the second value of d, and so on. If any iteration of d fails, then the effects of that computation are not visible outside d. This means that the desugaring of **for** into **split** needs to be more elaborate.

$\mathbf{for}(\exists x_1 \cdots x_n.\ d)\ \mathbf{do}\ e$

means

$f\ \langle\rangle := \langle\rangle;$
$g(v)(r) := (v = \langle x_1, \cdots, x_n\rangle;\ cons\langle e, \mathbf{split}(r\ \langle\rangle)\{f, g\}\rangle;$
$\mathbf{split}(\exists x_1 \cdots x_n.\ d;\ \langle x_1, \cdots, x_n\rangle)\{f, g\}$

To support limited store operations (*e.g.*, **read**, but not **write**) we can equip the store with a set of currently allowed operations. We also need some extra primitives that modify this set.

Syntax extension

References	r		
Expressions	e	$::=$	$\cdots \mid \textbf{store}\ h\ \textbf{in}\ \{e\}$
Primops	op	$::=$	$\cdots \mid \textbf{alloc} \mid \textbf{read} \mid \textbf{write}$
Head values	hnf	$::=$	$\cdots \mid r$
Execution contexts	X	$::=$	$\cdots \mid \textbf{store}\ h\ \textbf{in}\ \{X\}$
Scope contexts	SC	$::=$	$\cdots \mid \textbf{store}\ h\ \textbf{in}\ \{SC\}$
Heap	h	$::=$	$\epsilon \mid r \mapsto v, h$
Heap context	H	$::=$	$\square, h \mid r \mapsto v, H$
Store contexts	S	$::=$	$\square \mid v = S \mid S;\ e \mid se;\ S \mid \exists x.\ S$
Store-op free exprs	se	$::=$	$v \mid se_1 = se_2 \mid se_1;\ se_2 \mid \exists x.\ se \mid sp(v)$
Results	w	$::=$	$v \mid v \,\|\, e$
Non-store primops	sp	$::=$	any, except **alloc**, **read**, **write**
Non-store expression	oe	$::=$	like e, but not w, **store**, or **fail**

Axiom extensions

Normalization change

EXI-FLOAT $\quad X[\exists x.\ e] \longrightarrow \exists x.\ X[e] \quad$ if $X \neq \square$, $x \notin \text{fvs}(X)$, use α
if there is **store** in X then $e \in ce$

Reference ops

REF-ALLOC $\quad \textbf{store}\ h\ \textbf{in}\ \{S[\textbf{alloc}(v)]\} \longrightarrow \textbf{store}\ r \mapsto v, h\ \textbf{in}\ \{S[r]\}$
$\text{fvs}(v) \# \text{bvs}(S)$, r fresh

REF-READ $\quad \textbf{store}\ H[r \mapsto v]\ \textbf{in}\ \{S[\textbf{read}(r)]\} \longrightarrow \textbf{store}\ H[r \mapsto v]\ \textbf{in}\ \{S[v]\}$
$\text{fvs}(v) \# \text{bvs}(S)$, use α

REF-WRITE $\quad \textbf{store}\ H[r \mapsto v_1]\ \textbf{in}\ \{S[\textbf{write}\langle r, v_2\rangle]\} \longrightarrow \textbf{store}\ H[r \mapsto v_2]\ \textbf{in}\ \{S[\langle\rangle]\}$
$\text{fvs}(v_2) \# \text{bvs}(S)$

Store duplication

ST-SPLIT-DUP $\quad \textbf{store}\ h\ \textbf{in}\ \{S[\textbf{split}(oe)\{f, g\}]\} \longrightarrow \textbf{store}\ h\ \textbf{in}\ \{S[\textbf{split}(\textbf{store}\ h\ \textbf{in}\ \{oe\})\{f, g\}]\}$
$\text{fvs}(h) \# \text{bvs}(S)$, use α

ST-CHOICE-DUP $\quad \textbf{store}\ h\ \textbf{in}\ \{oe \,\|\, e\} \longrightarrow \textbf{store}\ h\ \textbf{in}\ \{\textbf{store}\ h\ \textbf{in}\ \{oe\} \,\|\, e\}$

Store commit

ST-SPLIT $\quad \textbf{store}\ h_1\ \textbf{in}\ \{S[\textbf{split}(\textbf{store}\ h_2\ \textbf{in}\ \{w\})\{f, g\}]\} \longrightarrow \textbf{store}\ h_2\ \textbf{in}\ \{S[\textbf{split}(w)\{f, g\}]\}$
$\text{fvs}(h_2) \# \text{bvs}(S)$

ST-CHOICE $\quad \textbf{store}\ h_1\ \textbf{in}\ \{S[(\textbf{store}\ h_2\ \textbf{in}\ \{w\}) \,\|\, e]\} \longrightarrow \textbf{store}\ h_2\ \textbf{in}\ \{S[w \,\|\, e]\}$
$\text{fvs}(h_2) \# \text{bvs}(S)$

Unification

Extension with the obvious axioms making equal references unify, and anything else fail.

Top level

Start top level reduction of e with $\textbf{store}\ \epsilon\ \textbf{in}\ \{e\}$.

Fig. 22. The Verse calculus: store axioms

F.1 Examples

[LA: Not yet]