Semantics of a Relational λ-Calculus*

Pablo Barenbaum[1], Federico Lochbaum[2], and Mariana Milicich[3]

[1] Universidad de Buenos Aires and Universidad Nacional de Quilmes, Argentina
pbarenbaum@dc.uba.ar
[2] Universidad Nacional de Quilmes, Argentina
federico.lochbaum@gmail.com
[3] Universidad de Buenos Aires, Argentina
milicichmariana@gmail.com

Abstract. We extend the λ-calculus with constructs suitable for relational and functional–logic programming: non-deterministic choice, fresh variable introduction, and unification of expressions. In order to be able to unify λ-expressions and still obtain a confluent theory, we depart from related approaches, such as λProlog, in that we do not attempt to solve higher-order unification. Instead, abstractions are decorated with a *location*, which intuitively may be understood as its memory address, and we impose a simple *coherence* invariant: abstractions in the same location must be equal. This allows us to formulate a *confluent* small-step operational semantics which only performs first-order unification and does not require *strong* evaluation (below lambdas). We study a simply typed version of the system. Moreover, a denotational semantics for the calculus is proposed and reduction is shown to be sound with respect to the denotational semantics.

Keywords: Lambda Calculus · Semantics · Relational Programming · Functional Programming · Logic Programming · Confluence

1 Introduction

Declarative programming is defined by the ideal that programs should resemble abstract specifications rather than concrete implementations. One of the most significant declarative paradigms is *functional programming*, represented by languages such as Haskell. Some of its salient features are the presence of first-class functions and inductive datatypes manipulated through pattern matching. The fact that the underlying model of computation—the λ-calculus—is *confluent* allows one to reason equationally about the behavior of functional programs.

Another declarative paradigm is *logic programming*, represented by languages such as Prolog. Some of its salient features are the ability to define relations rather than functions, and the presence of existentially quantified *symbolic variables* that become instantiated upon querying. This sometimes allows to use

* Work partially supported by project grants ECOS Sud A17C01, PUNQ 1346/17, and UBACyT 20020170100086BA.

n-ary relations with various patterns of instantiation, *e.g.* `add(3, 2, X)` computes `X := 3 + 2` whereas `add(X, 2, 5)` computes `X := 5 - 2`. The underlying model of computation is based on *unification* and refutation search with *backtracking*.

The idea to marry functional and logic programming has been around for a long time, and there have been many attempts to combine their features gracefully. For example, λProlog (Miller and Nadathur [?,?]) takes Prolog as a starting point, generalizing first-order terms to *λ-terms* and the mechanism of first-order unification to that of *higher-order unification*. Another example is Curry (Hanus et al. [?,?]) in which programs are defined by equations, quite like in functional languages, but evaluation is non-deterministic and evaluation is based on *narrowing*, *i.e.* variables become instantiated in such a way as to fulfill the constraints imposed by equations.

One of the interests of combining functional and logic programming is the fact that the increased expressivity aids declarative programming. For instance, if one writes a parser as a function `parser : String` $\longrightarrow$ `AST`, it should be possible, under the right conditions, to *invert* this function to obtain a pretty-printer `pprint : AST` $\longrightarrow$ `String`:

$$\texttt{pprint ast = } \nu \texttt{ source . ((ast } \overset{\bullet}{=} \texttt{ parse source) ; source)}$$

In this hypothetical functional–logic language, intuitively speaking, the expression $(\nu x. t)$ creates a fresh symbolic variable x and proceeds to evaluate t; the expression $(t \overset{\bullet}{=} s)$ unifies t with s; and the expression $(t; s)$ returns the result of evaluating s whenever the evaluation of t succeeds.

Given that unification is a generalization of pattern matching, a functional language with explicit unification should in some sense generalize *λ-calculi with patterns*, such as the Pure Pattern Calculus [?]. For example, by relying on unification one may build *dynamic* or *functional patterns*, *i.e.* patterns that include operations other than constructors. A typical instance is the following function `last : [a]` $\longrightarrow$ `a`, which returns the last element of a non-empty cons-list:

```
last (xs ++ [x]) = x
```

Note that `++` is not a constructor. This definition may be desugared similarly as for the `pprint` example above:

$$\texttt{last lst = } \nu \texttt{ xs . } \nu \texttt{ x. (lst } \overset{\bullet}{=} \texttt{ (xs ++ [x])); x}$$

Still another interest comes from the point of view of the *proposition-as-types correspondence*. Terms of a λ-calculus with types can be understood as encoding proofs, so for instance the identity function $(\lambda x : A. x)$ may be understood as a proof of the implication $A \to A$. From this point of view, a functional–logic program may be understood as a *tactic*, as can be found in proof assistants such as Isabelle or Coq (see *e.g.* [?]). A term of type A should then be understood as a non-deterministic procedure which attempts to find a proof of A and it may leave holes in the proof or even fail. For instance if P is a property on natural numbers,

p is a proof of $P(0)$ and q is a proof of $P(1)$, then $\lambda n.\, ((n \overset{\bullet}{=} 0); p) \oplus ((n \overset{\bullet}{=} 1); q)$ is a tactic that given a natural number n produces a proof of $P(n)$ whenever $n \in \{0, 1\}$, and otherwise it fails. Here $(t \oplus s)$ denotes the *non-deterministic alternative* between t and s.

The goal of this paper is to **provide a foundation for functional–logic programming** by **extending the λ-calculus with relational constructs**. Recall that the syntactic elements of the λ-calculus are λ-terms $(t, s, \ldots)$, which inductively may be *variables* $(x, y, \ldots)$, *abstractions* $(\lambda x.\, t)$, and *applications* $(t\, s)$. Relational programming may be understood as the purest form of logic programming, chiefly represented by the family of miniKanren languages (Byrd et al. [**?**,**?**]). The core syntactic elements of miniKanren, following for instance Rozplokhas et al. [**?**] are goals $(G, G', \ldots)$ which are inductively given by: *relation symbol invocations*, of the form $R(T_1, \ldots, T_n)$, where R is a relation symbol and $T_1, \ldots, T_n$ are terms of a first-order language, *unification* of first-order terms $(T_1 \overset{\bullet}{=} T_2)$, *conjunction* of goals $(G; G')$, *disjunction* of goals $(G \oplus G')$, and *fresh variable introduction* $(\nu x.\, G)$.

Our starting point is a "chimeric creature"—a functional–logic language resulting from cross breeding the λ-calculus and miniKanren, given by the following abstract syntax:

$$\begin{array}{rlll|lll}
t, s ::= & x & \text{variable} & & \mathbf{c} & \text{constructor} \\
| & \lambda x.\, t & \text{abstraction} & & t\, s & \text{application} \\
| & \nu x.\, t & \text{fresh variable introduction} & & t \oplus s & \text{non-deterministic choice} \\
| & t; s & \text{guarded expression} & & t \overset{\bullet}{=} s & \text{unification}
\end{array}$$

Its informal semantics has been described above. Variables $(x, y, \ldots)$ may be instantiated by unification, while constructors $(\mathbf{c}, \mathbf{d}, \ldots)$ are constants. For example, if $\texttt{coin} \overset{\text{def}}{=} (\mathbf{true} \oplus \mathbf{false})$ is a non-deterministic boolean with two possible values and $\texttt{not} \overset{\text{def}}{=} \lambda x.\, ((x \overset{\bullet}{=} \mathbf{true}); \mathbf{false}) \oplus ((x \overset{\bullet}{=} \mathbf{false}); \mathbf{true})$ is the usual boolean negation, the following non-deterministic computation:

$$(\lambda x.\, \lambda y.\, (x \overset{\bullet}{=} \texttt{not}\, y); \mathbf{pair}\ x\, y)\ \texttt{coin}\, \texttt{coin}$$

should have two results, namely $\mathbf{pair\, true\, false}$ and $\mathbf{pair\, false\, true}$.

Structure of this paper. In Section 2, we discuss some technical difficulties that arise as one intends to provide a formal operational semantics for the informal functional–logic calculus sketched above. In Section 3, we refine this rough proposal into a calculus we call the $\lambda^{\mathtt{U}}$*-calculus*, with a formal small-step operational semantics (Def. 3.1). To do so, we distinguish *terms*, which represent a single choice, from *programs*, which represent a non-deterministic alternative between zero or more terms. Moreover, we adapt the standard first-order unification algorithm to our setting by imposing a *coherence* invariant on programs. In Section 4, we study the operational properties of the $\lambda^{\mathtt{U}}$-calculus: we provide an inductive characterization of the set of normal forms (Prop. 4.1), and we prove that it is confluent (Thm. 4.4) (up to a notion of structural equivalence). In Section 5, we propose a straightforward system of simple types and we show that it enjoys subject reduction (Prop. 5.2). In Section 6, we define a (naive) denotational semantics, and we show that the operational semantics is sound (although it is not complete) with respect to this denotational semantics (Thm. 6.2). In Section 7, we conclude and we lay out avenues of further research.

Note. Most proofs have been ommited from this paper. For details, see the extended version[4].

2 Technical Challenges

This section is devoted to discussing technical stumbling blocks that we encountered as we attempted to define an operational semantics for the functional–logic calculus incorporating all the constructs mentioned in the introduction. These technical issues motivate the design decisions behind the actual $\lambda^{\mathtt{U}}$-calculus defined in Sec. 3. The discussion in this section is thus **informal**. Examples are carried out with their hypothetical or intended semantics.

Locality of symbolic variables. The following program introduces a fresh variable x and then there are two alternatives: either x unifies with $\mathbf{c}$ and the result is x, or x unifies with $\mathbf{d}$ and the result is x. The expected reduction semantics is the following. The constant **ok** is the result obtained after a successful unification:

$$\begin{aligned}
\nu x.\ \Big(((x \overset{\bullet}{=} \mathbf{c}); x) \oplus ((x \overset{\bullet}{=} \mathbf{d}); x)\Big) &\to ((x \overset{\bullet}{=} \mathbf{c}); x) \oplus ((x \overset{\bullet}{=} \mathbf{d}); x) \text{ with } x \text{ fresh} \\
&\to (\mathbf{ok}; \mathbf{c}) \oplus ((x \overset{\bullet}{=} \mathbf{d}); x) \qquad (\bigstar) \\
&\to (\mathbf{ok}; \mathbf{c}) \oplus (\mathbf{ok}; \mathbf{d}) \\
&\twoheadrightarrow \mathbf{c} \oplus \mathbf{d}
\end{aligned}$$

Note that in the step marked with $(\bigstar)$, the variable x becomes instantiated to $\mathbf{c}$, but *only to the left of the choice operator* $(\oplus)$. This suggests that programs should consist of different *threads* fenced by choice operators. Symbolic variables should be local to each thread.

Need of commutative conversions. Redexes may be *blocked* by the choice operator—for example in the application $((t \oplus \lambda x.\, s)\, u)$, there is a potential β-redex $((\lambda x.\, s)\, u)$ which is blocked. This suggests that *commutative conversions* that distribute the choice operator should be incorporated, allowing for instance a reduction step $(t \oplus \lambda x.\, s)\, u \to t\, u \oplus (\lambda x.\, s)\, u$. In our proposal, we force in the syntax that a program is always written, canonically, in the form $t_1 \oplus \ldots \oplus t_n$, where each t_i is a deterministic program (*i.e.* choice operators may only appear inside lambdas). This avoids the need to introduce commutative rules.

[4] https://arxiv.org/abs/2009.10929

Confluence only holds up to associativity and commutativity. There are two ways to distribute the choice operators in the following example:

$$\begin{tikzcd}
t_1(s_1 \oplus s_2) \oplus t_2(s_1 \oplus s_2) \arrow[d] & (t_1 \oplus t_2)\,(s_1 \oplus s_2) \arrow[l] \arrow[r] & (t_1 \oplus t_2)\,s_1 \oplus (t_1 \oplus t_2)\,s_2 \arrow[d] \\
(t_1\,s_1 \oplus t_1\,s_2) \oplus (t_2\,s_1 \oplus t_2\,s_2) \arrow[rr, dotted, no head, "\equiv" description] & & (t_1\,s_1 \oplus t_2\,s_1) \oplus (t_1\,s_2 \oplus t_2\,s_2)
\end{tikzcd}$$

The resulting programs cannot be equated unless one works up to an equivalence relation that takes into account the associativity and commutativity of the choice operator. As we mentioned, the $\lambda^{\mathtt{U}}$-calculus works with programs in canonical form $t_1 \oplus \ldots \oplus t_n$, so there is no need to work modulo associativity. However, we do need commutativity. As a matter of fact, we shall define a notion of *structural equivalence* ($\equiv$) between programs, allowing the arbitrary reordering of threads. This relation will be shown to be well-behaved, namely, a *strong bisimulation* with respect to the reduction relation, *cf.* Lem. 3.3.

Non-deterministic choice is an effect. Consider the program $(\lambda x.\, x\, x)(\mathbf{c} \oplus \mathbf{d})$, which chooses between $\mathbf{c}$ and $\mathbf{d}$ and then it produces two copies of the chosen value. Its expected reduction semantics is:

$$(\lambda x.\, x\, x)(\mathbf{c} \oplus \mathbf{d}) \to (\lambda x.\, x\, x)\mathbf{c} \oplus (\lambda x.\, x\, x)\mathbf{d} \twoheadrightarrow \mathbf{c}\,\mathbf{c} \oplus \mathbf{d}\,\mathbf{d}$$

This means that the first step in the following reduction, which produces two copies of $(\mathbf{c} \oplus \mathbf{d})$ cannot be allowed, as it would break confluence:

$$(\lambda x.\, x\, x)(\mathbf{c} \oplus \mathbf{d}) \not\to (\mathbf{c} \oplus \mathbf{d})\,(\mathbf{c} \oplus \mathbf{d}) \twoheadrightarrow \mathbf{c}\,\mathbf{c} \oplus \mathbf{c}\,\mathbf{d} \oplus \mathbf{d}\,\mathbf{c} \oplus \mathbf{d}\,\mathbf{d}$$

The deeper reason is that non-deterministic choice is a side-effect rather than a value. Our design decision, consistent with this remark, is to follow a **call-by-value** discipline. Another consequence of this remark is that the choice operator should not commute with abstraction, given that $\lambda x.\,(t \oplus s)$ and $(\lambda x.\, t) \oplus (\lambda x.\, s)$ are not observationally equivalent. In particular, $\lambda x.\,(t \oplus s)$ is a value, which may be *copied*, while $(\lambda x.\, t) \oplus (\lambda x.\, s)$ is not a value. On the other hand, if W is any *weak* context, *i.e.* a term with a hole which does *not* lie below a binder, and we write $\mathsf{W}\langle t \rangle$ for the result of plugging a term t into the hole of W, then $\mathsf{W}\langle t \oplus s \rangle = \mathsf{W}\langle t \rangle \oplus \mathsf{W}\langle s \rangle$ should hold.

Evaluation should be weak. Consider the term $F \overset{\text{def}}{=} \lambda y.\,((y \overset{\bullet}{=} x); x)$. Intuitively, it unifies its argument with a (global) symbolic variable x and then returns x. This poses two problems. First, when x becomes instantiated to y, it may be outside the scope of the abstraction binding y, for instance, the step $F\, x = (\lambda y.\,((y \overset{\bullet}{=} x); x))\, x \to (\lambda y.\,(\mathbf{ok}; y))\, y$ produces a meaningless free occurrence of y. Second, consider the following example in which two copies of F are used with different arguments. If we do **not** allow evaluation under lambdas, this example fails due to a unification clash, *i.e.* it produces no outputs:

$$\begin{aligned}
(\lambda f.\,(f\,\mathbf{c})\,(f\,\mathbf{d}))\, F &\to (F\,\mathbf{c})\,(F\,\mathbf{d}) \\
&\to ((\mathbf{c} \overset{\bullet}{=} x); x)\,((\mathbf{d} \overset{\bullet}{=} x); x) \\
&\to (\mathbf{ok}; \mathbf{c})\,((\mathbf{d} \overset{\bullet}{=} \mathbf{c}); \mathbf{c}) \qquad (\bigstar) \\
&\to \mathtt{fail}
\end{aligned}$$

Note that in the step marked with ($\bigstar$), the symbolic variable x has become instantiated to $\mathbf{c}$, leaving us with the unification goal $\mathbf{d} \overset{\bullet}{=} \mathbf{c}$ which fails. On the other hand, if we were to allow reduction under lambdas, given that there are no other occurrences of x anywhere in the term, in one step F becomes $\lambda y.\,(\mathbf{ok}; y)$, which then behaves as the identity:

$$\begin{aligned}(\lambda f.\,(f\,\mathbf{c})\,(f\,\mathbf{d}))\,F &\not\to (\lambda f.\,(f\,\mathbf{c})\,(f\,\mathbf{d}))\,(\lambda y.\,\mathbf{ok}; y)\\ &\to ((\lambda y.\,\mathbf{ok}; y)\,\mathbf{c})\,((\lambda y.\,\mathbf{ok}; y)\,\mathbf{d})\\ &\twoheadrightarrow \mathbf{c}\,\mathbf{d}\end{aligned}$$

Thus allowing reduction below abstractions in this example would break confluence. This suggests that evaluation should be **weak**, *i.e.* it should not proceed below binders.

Avoiding higher-order unification. The calculus proposed in this paper rests on the design choice to *avoid* attempting to solve higher-order unification problems. Higher-order unification problems can be expressed in the syntax: for example in $(f\mathbf{c} \overset{\bullet}{=} \mathbf{c})$ the variable f represents an unknown value which should fulfill the given constraint. From our point of view, however, this program is stuck and its evaluation cannot proceed—it is a normal form. However, note that we **do** want to allow *pattern matching* against functions; for example the following should succeed, instantiating f to the identity:

$$(\mathbf{c}\,f \overset{\bullet}{=} \mathbf{c}(\lambda x.\,x)); (f \overset{\bullet}{=} f) \to (\lambda x.\,x) \overset{\bullet}{=} (\lambda x.\,x) \to \mathbf{ok}$$

The decision to sidestep higher-order unification is a debatable one, as it severely restricts the expressivity of the language. But there are various reasons to explore alternatives. First, higher-order unification is undecidable [?], and even second order unification is known to be undecidable [?]. Huet's semi-decision procedure [?] does find a solution should it exist, but even then higher-order unification problems do not necessarily possess *most general unifiers* [?], which turns confluence hopeless[5]. Second, there are decidable restrictions of higher-order unification which do have most general unifiers, such as *higher-order pattern unification* [?] used in λProlog, and *nominal unification* [?] used in αProlog. But these mechanisms require *strong* evaluation, *i.e.* evaluation below abstractions, departing from the traditional execution model of eager applicative languages such as in the Lisp and ML families, in which closures are opaque values whose bodies cannot be examined. Moreover, they are formulated in a necessarily typed setting.

The calculus studied in this paper relies on a standard first-order unification algorithm, with the only exception that abstractions are deemed to be equal if and only if they have the same "identity". Intuitively speaking, this means that they are stored in the same memory location, *i.e.* they are represented by the same pointer. This is compatible with the usual implementation techniques

[5] Key in our proof of confluence is the fact that if σ and σ' are most general unifiers for unification problems G and G' respectively, then the most general unifier for $(\mathsf{G} \cup \mathsf{G}')$ is an instance of both σ and σ'. See Ex. 4.5.

of eager applicative languages, so it should allow to use standard compilation techniques for λ-abstractions. Also note that the operational semantics does not require to work with typed terms—in fact the system presented in Sec. 3 is untyped, even though we study a typed system in Sec. 5.

3 The $\boldsymbol{\lambda}^{\mathtt{U}}$-Calculus — Operational Semantics

In this section we describe the operational semantics of our proposed calculus, including its syntax, reduction rules (Def. 3.1), an invariant (*coherence*) which is preserved by reduction (Lem. 3.2), and a notion of structural equivalence which is a *strong bisimulation* with respect to reduction (Lem. 3.3).

Syntax of terms and programs. Suppose given denumerably infinite sets of *variables* $\mathsf{Var} = \{x, y, z, \dots\}$, *constructors* $\mathsf{Con} = \{\mathbf{c}, \mathbf{d}, \mathbf{e}, \dots\}$, and *locations* $\mathsf{Loc} = \{\ell, \ell', \ell'', \dots\}$. We assume that there is a distinguished constructor $\mathbf{ok}$. The sets of *terms* $t, s, \dots$ and *programs* $P, Q, \dots$ are defined mutually inductively as follows:

$t ::= x$	variable	$\mid \mathbf{c}$	constructor
$\mid \lambda x.\, P$	abstraction	$\mid \lambda^{\ell} x.\, P$	allocated abstraction
$\mid t\, t$	application	$\mid \nu x.\, t$	fresh variable introduction
$\mid t; t$	guarded expression	$\mid t \overset{\bullet}{=} t$	unification

$P ::= \mathtt{fail}$	empty program
$\mid t \oplus P$	non-deterministic choice

The set of *values* $\mathsf{Val} = \{\mathtt{v}, \mathtt{w}, \dots\}$ is a subset of the set of terms, given by the grammar $\mathtt{v} ::= x \mid \lambda^{\ell} x.\, P \mid \mathbf{c}\, \mathtt{v}_1 \dots \mathtt{v}_n$. Values of the form $\mathbf{c}\, \mathtt{v}_1 \dots \mathtt{v}_n$ are called *structures*.

Intuitively, an (unallocated) abstraction $\lambda x.\, P$ represents the static code to create a closure, while $\lambda^{\ell} x.\, P$ represents the closure created in runtime, stored in the memory cell ℓ. When the abstraction is evaluated, it becomes decorated with a location (*allocated*). We will have a rewriting rule like $\lambda x.\, P \to \lambda^{\ell} x.\, P$ where ℓ is fresh.

Notational conventions. We write $\mathsf{C}, \mathsf{C}', \dots$ for *arbitrary contexts*, *i.e.* terms with a single free occurrence of a hole $\Box$. We write $\mathsf{W}, \mathsf{W}', \dots$ for *weak contexts*, which do not enter below abstractions nor fresh variable declarations, *i.e.* $\mathsf{W} ::= \Box \mid \mathsf{W}\, t \mid t\, \mathsf{W} \mid \mathsf{W}; t \mid t; \mathsf{W} \mid \mathsf{W} \overset{\bullet}{=} t \mid t \overset{\bullet}{=} \mathsf{W}$. We write $\oplus_{i=1}^{n} t_i$ or also $t_1 \oplus t_2 \dots \oplus t_n$ to stand for the program $t_1 \oplus (t_2 \oplus \dots (t_n \oplus \mathtt{fail}))$. In particular, if t is a term, sometimes we write t for the *singleton* program $t \oplus \mathtt{fail}$. The set of free variables $\mathsf{fv}(t)$ (resp. $\mathsf{fv}(P)$) of a term (resp. program) is defined as expected, noting that fresh variable declarations $\nu x.\, t$ and both kinds of abstractions $\lambda x.\, P$ and $\lambda^{\ell} x.\, P$ bind the free occurrences of x in the body. Expressions are considered up to α-equivalence, *i.e.* renaming of all bound variables. Given a context or weak context C and a term t, we write $\mathsf{C}\langle t \rangle$ for the (capturing) substitution of $\Box$ by t in C. The set of locations $\mathsf{locs}(t)$ (resp. $\mathsf{fv}(P)$) of a term (resp. program) is defined as the set of all locations ℓ decorating any abstraction on t. We write $t\{\ell := \ell'\}$ for the term that results from replacing all occurrences of the location ℓ in t by ℓ'. The program being evaluated is called the *toplevel program*. The toplevel program is always of the form $t_1 \oplus t_2 \ldots \oplus t_n$, and each of the t_i is called a *thread*.

Operations with programs. We define the operations $P \oplus Q$ and $\mathsf{W}\langle P\rangle$ by induction on the structure of P as follows; note that the notation "$\oplus$" is overloaded both for consing a term onto a program and for concatenating programs:

$$\begin{array}{rclrcl} \mathtt{fail} \oplus Q & \overset{\text{def}}{=} & Q & \qquad \mathsf{W}\langle\mathtt{fail}\rangle & \overset{\text{def}}{=} & \mathtt{fail} \\ (t \oplus P) \oplus Q & \overset{\text{def}}{=} & t \oplus (P \oplus Q) & \qquad \mathsf{W}\langle t \oplus P\rangle & \overset{\text{def}}{=} & \mathsf{W}\langle t\rangle \oplus \mathsf{W}\langle P\rangle \end{array}$$

Substitutions. A *substitution* is a function $\sigma : \mathsf{Var} \to \mathsf{Val}$ with *finite support*, *i.e.* such that the set $\mathrm{supp}(\sigma) \overset{\text{def}}{=} \{x \mid \sigma(x) \neq x\}$ is finite. We write $\{x_1 \mapsto \mathtt{v}_1, \ldots, x_n \mapsto \mathtt{v}_n\}$ for the substitution σ such that $\mathrm{supp}(\sigma) = \{x_1, \ldots, x_n\}$ and $\sigma(x_i) = \mathtt{v}_i$ for all $i \in 1..n$. A *renaming* is a substitution mapping each variable to a variable, *i.e.* a substitution of the form $\{x_1 \mapsto y_1, \ldots, x_n \mapsto y_n\}$.

If $\sigma : \mathsf{Var} \to \mathsf{Val}$ is a substitution and t is a term, t^σ denotes the capture-avoiding substitution of each occurrence of a free variable x in t by $\sigma(x)$. Capture-avoiding substitution of a single variable x by a value $\mathtt{v}$ in a term t is written $t\{x := \mathtt{v}\}$ and defined by $t^{\{x \mapsto \mathtt{v}\}}$. Subsitutions ρ, σ may be *composed* as follows: $(\rho \cdot \sigma)(x) \overset{\text{def}}{=} \rho(x)^\sigma$. Substitutions can also be applied to weak contexts, taking $\Box^\sigma \overset{\text{def}}{=} \Box$. A substitution σ is *idempotent* if $\sigma \cdot \sigma = \sigma$. A substitution σ is *more general* than a substitution ρ, written $\sigma \lesssim \rho$ if there is a substitution τ such that $\rho = \sigma \cdot \tau$.

Unification. We describe how to adapt the standard first-order unification algorithm to our setting, in order to deal with unification of λ-abstractions. As mentioned before, our aim is to solve only first-order unification problems. This means that the unification algorithm should only deal with equations involving terms which are already *values*. Note that unallocated abstractions ($\lambda x. P$) are *not* considered values; abstractions are only values when they are allocated ($\lambda^\ell x. P$). Allocated abstractions are to be considered equal if and only if they are decorated with the same location. Note that terms of the form $x\, t_1 \ldots t_n$ are *not* considered values if $n > 0$, as this would pose a higher-order unification problem, possibly requiring to instantiate x as a function of its arguments.

We expand briefly on why a naive approach to first-order unification would not work. Suppose that we did not have locations and we declared that two abstractions $\lambda x. P$ and $\lambda y. Q$ are equal whenever their bodies are equal, up to α-renaming (*i.e.* $P\{x := y\} = Q$). The problem is that this notion of equality is not preserved by substitution, for example, the unification problem given by the equation $\lambda x. y \overset{\bullet}{=} \lambda x. z$ would *fail*, as $y \neq z$. However, the variable y may become instantiated into z, and the equation would become $\lambda x. z \overset{\bullet}{=} \lambda x. z$, which succeeds. This corresponds to the following critical pair in the calculus, which cannot be closed:

$$\mathtt{fail} \leftarrow (\lambda x. y \overset{\bullet}{=} \lambda x. z); (y \overset{\bullet}{=} z) \to (\lambda x. z \overset{\bullet}{=} \lambda x. z); \mathbf{ok} \to \mathbf{ok}; \mathbf{ok}$$

This is where the notion of *allocated abstraction* plays an important role. We will work with the invariant that if $\lambda^{\ell}x.\,P$ and $\lambda^{\ell'}y.\,Q$ are two allocated abstractions in the same location ($\ell = \ell'$) then their bodies will be equal, up to α-renaming. This ensures that different allocated abstractions are still different after substitution, as they must be decorated with different locations.

Unification goals and unifiers. A *goal* is a term of the form $\mathtt{v} \overset{\bullet}{=} \mathtt{w}$. A *unification problem* is a finite set of goals $\mathsf{G} = \{\mathtt{v}_1 \overset{\bullet}{=} \mathtt{w}_1, \dots, \mathtt{v}_n \overset{\bullet}{=} \mathtt{w}_n\}$. If σ is a substitution we write G^{σ} for $\{{\mathtt{v}_1}^{\sigma} \overset{\bullet}{=} {\mathtt{w}_1}^{\sigma}, \dots, {\mathtt{v}_n}^{\sigma} \overset{\bullet}{=} {\mathtt{w}_n}^{\sigma}\}$. A *unifier* for $\mathsf{G} = \{\mathtt{v}_1 \overset{\bullet}{=} \mathtt{w}_1, \dots, \mathtt{v}_n \overset{\bullet}{=} \mathtt{w}_n\}$ is a substitution σ such that ${\mathtt{v}_i}^{\sigma} = {\mathtt{w}_i}^{\sigma}$ for all $1 \leq i \leq n$. A unifier σ for G is *most general* if for any other unifier ρ one has $\sigma \lesssim \rho$.

Coherence invariant. As mentioned before, we impose an invariant on programs forcing that allocated abstractions decorated with the same location must be syntactically equal. Moreover, we require that allocated abstractions do not refer to variables bound outside of their scope, *i.e.* that they are in fact closures. Note that the source program trivially satisfies this invariant, as it is expected that allocated abstractions are not written by the user but generated at runtime.

More precisely, a set X of terms is **coherent** if the two following conditions hold. **(1)** Consider any allocated abstraction under a context C, *i.e.* let $t \in X$ such that $t = \mathsf{C}\langle \lambda^{\ell}x.\,P\rangle$. Then the context C does not bind any of the free variables of $\lambda^{\ell}x.\,P$. **(2)** Consider any two allocated abstractions in t and s with the same location, *i.e.* let $t, s \in X$ be such that $t = \mathsf{C}\langle \lambda^{\ell}x.\,P\rangle$ and $s = \mathsf{C}'\langle \lambda^{\ell}y.\,Q\rangle$, Then $P\{x := y\} = Q$.

We extend the notion of coherence to other syntactic categories as follows. A term t is coherent if $\{t\}$ is coherent. A program $P = t_1 \oplus \ldots \oplus t_n$ is *coherent* if each thread t_i is coherent. A unification problem G is *coherent* if it is coherent seen as a set. Note that a program may be coherent even if different abstractions in different threads have the same location. For example, $(\lambda^{\ell}x.\,x\,x \overset{\bullet}{=} \lambda^{\ell}y.\,\mathbf{c}) \oplus (\lambda^{\ell'}y.\,y)$ is not coherent, whereas $(\lambda^{\ell}x.\,x\,x \overset{\bullet}{=} \lambda^{\ell}y.\,y\,y) \oplus (\lambda^{\ell}y.\,\mathbf{c})$ is coherent.

Unification algorithm. The standard Martelli–Montanari [**?**] unification algorithm can be adapted to our setting. In particular, there is a computable function $\mathsf{mgu}(-)$ such that if G is a coherent unification problem then either $\mathsf{mgu}(\mathsf{G}) = \sigma$, *i.e.* $\mathsf{mgu}(\mathsf{G})$ returns a substitution σ which is an idempotent most general unifier for G, or $\mathsf{mgu}(\mathsf{G}) = \bot$, *i.e.* $\mathsf{mgu}(\mathsf{G})$ fails and G has no unifier. Moreover, it can be shown that if the algorithm succeeds, the set $\mathsf{G}^{\sigma} \cup \{\sigma(x) \mid x \in \mathsf{Var}\}$ is coherent. The algorithm, formal statement and proofs are detailed in the appendix .

Operational semantics. The $\lambda^{\mathtt{U}}$**-calculus** is the rewriting system whose objects are programs, and whose reduction relation is given by the union of the following six rules:

Definition 3.1 (Reduction rules).

$$\begin{array}{rcll}
P_1 \oplus \mathsf{W}\langle \lambda x.\, P\rangle \oplus P_2 & \xrightarrow{\mathtt{alloc}} & P_1 \oplus \mathsf{W}\langle \lambda^{\ell} x.\, P\rangle \oplus P_2 & \textit{if } \ell \notin \mathsf{locs}(\mathsf{W}\langle \lambda x.\, P\rangle) \\
P_1 \oplus \mathsf{W}\langle (\lambda^{\ell} x.\, P)\, \mathtt{v}\rangle \oplus P_2 & \xrightarrow{\mathtt{beta}} & P_1 \oplus \mathsf{W}\langle P\{x := \mathtt{v}\}\rangle \oplus P_2 & \\
P_1 \oplus \mathsf{W}\langle \mathtt{v}; t\rangle \oplus P_2 & \xrightarrow{\mathtt{guard}} & P_1 \oplus \mathsf{W}\langle t\rangle \oplus P_2 & \\
P_1 \oplus \mathsf{W}\langle \nu x.\, t\rangle \oplus P_2 & \xrightarrow{\mathtt{fresh}} & P_1 \oplus \mathsf{W}\langle t\{x := y\}\rangle \oplus P_2 & \textit{if } y \notin \mathsf{fv}(\mathsf{W}) \\
P_1 \oplus \mathsf{W}\langle \mathtt{v} \overset{\bullet}{=} \mathtt{w}\rangle \oplus P_2 & \xrightarrow{\mathtt{unif}} & P_1 \oplus \mathsf{W}\langle \mathbf{ok}\rangle^{\sigma} \oplus P_2 & \textit{if } \mathsf{mgu}(\{\mathtt{v} \overset{\bullet}{=} \mathtt{w}\}) = \sigma \\
P_1 \oplus \mathsf{W}\langle \mathtt{v} \overset{\bullet}{=} \mathtt{w}\rangle \oplus P_2 & \xrightarrow{\mathtt{fail}} & P_1 \oplus P_2 & \textit{if } \mathsf{mgu}(\{\mathtt{v} \overset{\bullet}{=} \mathtt{w}\}) \textit{ fails}
\end{array}$$

Note that all rules operate on a single thread and they are **not** closed under any kind of evaluation contexts. The `alloc` rule allocates a closure, *i.e.* whenever a λ-abstraction is found below an evaluation context, it may be assigned a fresh location ℓ. The `beta` rule applies a function to a value. The `guard` rule proceeds with the evaluation of the right part of a guarded expression when the left part is already a value. The `fresh` rule introduces a fresh symbolic variable. The `unif` and `fail` rules solve a unification problem, corresponding to the success and failure cases respectively. If there is a unifier, the substitution is applied to the affected thread. For example:

$$\begin{array}{rl}
(\lambda x.\, x \oplus (\nu y.\, ((x \overset{\bullet}{=} \mathbf{c}\, y); y)))\, (\mathbf{c}\, \mathbf{d}) & \xrightarrow{\mathtt{alloc}} (\lambda^{\ell} x.\, x \oplus (\nu y.\, ((x \overset{\bullet}{=} \mathbf{c}\, y); y)))\, (\mathbf{c}\, \mathbf{d}) \\
& \xrightarrow{\mathtt{beta}} \mathbf{c}\, \mathbf{d} \oplus \nu y.\, ((\mathbf{c}\, \mathbf{d} \overset{\bullet}{=} \mathbf{c}\, y); y) \\
& \xrightarrow{\mathtt{fresh}} \mathbf{c}\, \mathbf{d} \oplus ((\mathbf{c}\, \mathbf{d} \overset{\bullet}{=} \mathbf{c}\, z); z) \\
& \xrightarrow{\mathtt{unif}} \mathbf{c}\, \mathbf{d} \oplus (\mathbf{ok}; \mathbf{d}) \\
& \xrightarrow{\mathtt{guard}} \mathbf{c}\, \mathbf{d} \oplus \mathbf{d}
\end{array}$$

Structural equivalence. As already remarked in Sec. 2, we will not be able to prove that confluence holds strictly speaking, but only *up to reordering of threads* in the toplevel program. Moreover the `alloc` and `fresh` rules introduce fresh names, and, as usual the most general unifier is unique only *up to renaming.* These conditions are expressed formally by means of the following relation of structural equivalence.

Formally, **structural equivalence** between programs is written $P \equiv Q$ and defined as the reflexive, symmetric, and transitive closure of the three following axioms:

1. $\equiv$-`swap`: $P \oplus t \oplus s \oplus Q \equiv P \oplus s \oplus t \oplus Q$.
2. $\equiv$-`var`: If $y \notin \mathsf{fv}(t)$ then $P \oplus t \oplus Q \equiv P \oplus t\{x := y\} \oplus Q$.
3. $\equiv$-`loc`: If $\ell' \notin \mathsf{locs}(t)$, then $P \oplus t \oplus Q \equiv P \oplus t\{\ell := \ell'\} \oplus Q$.

In short, $\equiv$-`swap` means that threads may be reordered arbitrarily, $\equiv$-`var` means that symbolic variables are local to each thread, and $\equiv$-`loc` means that locations are local to each thread.

The following lemma establishes that the coherence invariant is closed by reduction and structural equivalence, which means that the $\lambda^{\mathtt{U}}$-calculus is well-defined if restricted to coherent programs. In the rest of this paper, we always assume that **all programs enjoy the coherence invariant**.

Lemma 3.2. *Let P be a coherent program. If $P \equiv Q$ or $P \to Q$, then Q is also coherent.*

The following lemma establishes that reduction is well-defined modulo structural equivalence (*i.e.* it lifts to $\equiv$-equivalence classes):

Lemma 3.3. *Structural equivalence is a strong bisimulation with respect to $\to$. Precisely, let $P \equiv P' \xrightarrow{\mathtt{x}} Q$ with $\mathtt{x} \in \{\mathtt{alloc}, \mathtt{beta}, \mathtt{guard}, \mathtt{fresh}, \mathtt{unif}, \mathtt{fail}\}$. Then there exists a program Q' such that $P \xrightarrow{\mathtt{x}} Q' \equiv Q$.*

Example 3.4 (Type inference algorithm). As an illustrative example, the following translation $\mathbb{W}[-]$ converts an untyped λ-term t into a $\lambda^{\mathtt{U}}$-term that calculates the principal type of t according to the usual Hindley–Milner [?] type inference algorithm, or fails if it has no type. Note that an arrow type $(A \to B)$ is encoded as $(\mathbf{f}\,A\,B)$:

$$\mathbb{W}[x] \overset{\text{def}}{=} a_x \qquad \mathbb{W}[\lambda x.\, t] \overset{\text{def}}{=} \nu a_x.\, \mathbf{f}\, a_x\, \mathbb{W}[t] \qquad \mathbb{W}[t\, s] \overset{\text{def}}{=} \nu a.\, ((\mathbb{W}[t] \overset{\bullet}{=} \mathbf{f}\, \mathbb{W}[s]\, a); a)$$

For instance, $\mathbb{W}[\lambda x.\, \lambda y.\, y\, x] = \nu a.\, \mathbf{f}\, a\, (\nu b.\, \mathbf{f}\, b\, (\nu c.\, (b \overset{\bullet}{=} \mathbf{f}\, a\, c); c)) \twoheadrightarrow \mathbf{f}\, a\, (\mathbf{f}\, (\mathbf{f}\, a\, c)\, c)$.

4 Operational Properties

In this section we study some properties of the operational semantics. First, we characterize the set of *normal forms* of the $\lambda^{\mathtt{U}}$-calculus syntactically, by means of an inductive definition (Prop. 4.1). Then we turn to the main result of this section, proving that it enjoys confluence up to structural equivalence (Thm. 4.4).

Characterization of normal forms. The set of **normal terms** $t^\star, s^\star, \dots$ and **stuck terms** $S, S', \dots$ are defined mutually inductively as follows. A normal term is either a value or a stuck term, *i.e.* $t^\star ::= \mathtt{v} \mid S$. A term is stuck if the judgment $t\triangledown$ is derivable with the following rules:

$$\frac{n > 0}{x\, t_1^\star \dots t_n^\star \triangledown}\ \texttt{stuck-var} \qquad \frac{t_i^\star \triangledown \ \text{ for some } i \in \{1, 2, \dots, n\}}{\mathbf{c}\, t_1^\star \dots t_n^\star \triangledown}\ \texttt{stuck-cons}$$

$$\frac{t_1^\star \triangledown \qquad n \geq 0}{(t_1^\star; t_2^\star)\, s_1^\star \dots s_n^\star \triangledown}\ \texttt{stuck-guard} \qquad \frac{t_i^\star \triangledown \ \text{ for some } i \in \{1, 2\} \qquad n \geq 0}{(t_1^\star \overset{\bullet}{=} t_2^\star)\, s_1^\star \dots s_n^\star \triangledown}\ \texttt{stuck-unif}$$

$$\frac{t^\star \triangledown \qquad n \geq 0}{(\lambda^\ell x.\, P)\, t^\star\, s_1^\star \dots s_n^\star \triangledown}\ \texttt{stuck-lam}$$

The set of **normal programs** $P^\star, Q^\star, \dots$ is given by the following grammar: $P^\star ::= \mathtt{fail} \mid t^\star \oplus P^\star$. For example, the program $(\lambda^\ell x.\, x \overset{\bullet}{=} x) \oplus ((y\, \mathbf{c} \overset{\bullet}{=} \mathbf{d}); \mathbf{e}) \oplus z\, (z\, \mathbf{c})$ is normal, being the non-deterministic alternative of a value and two stuck terms. Normal programs capture the notion of normal form:

Proposition 4.1. *The set of normal programs is exactly the set of $\to$-normal forms.*

Confluence. In order to prove that the $\lambda^{\mathtt{U}}$-calculus has the Church–Rosser property, we adapt the method due to Tait and Martin-Löf [**?**, Sec. 3.2] by defining a *simultaneous reduction relation* $\Rightarrow$, and showing that it verifies the diamond property (*i.e.* $\Leftarrow\Rightarrow \subseteq \Rightarrow\Leftarrow$) and the inclusions $\to \subseteq \Rightarrow \subseteq \twoheadrightarrow$, where $\twoheadrightarrow$ denotes the reflexive–transitive closure of $\to$. Actually, these properties only hold up to structural equivalence, so our confluence result, rather than the usual inclusion $\twoheadleftarrow\twoheadrightarrow \subseteq \twoheadrightarrow\twoheadleftarrow$, expresses the weakened inclusion $\twoheadleftarrow\twoheadrightarrow \subseteq \twoheadrightarrow\equiv\twoheadleftarrow$.

To define the relation of simultaneous reduction, we use the following notation, to lift the binary operations of unification ($t \overset{\bullet}{=} s$), guarded expression ($t; s$), and application ($t\, s$) from the sort of terms to the sort of programs. Let $\star$ denote a binary term constructor (*e.g.* unification, guarded expression, or application). Then we write $(\bigoplus_{i=1}^{n} t_i) \star (\bigoplus_{j=1}^{m} s_j) \overset{\text{def}}{=} \bigoplus_{i=1}^{n} \bigoplus_{j=1}^{m} (t_i \star s_j)$.

First, we define a judgment $t \xRightarrow{\mathsf{G}} P$ of simultaneous reduction, relating a term and a program, parameterized by a set G of unification goals representing pending constraints:

$$\dfrac{}{x \xRightarrow{\varnothing} x}\,\texttt{Var} \qquad \dfrac{}{\mathbf{c} \xRightarrow{\varnothing} \mathbf{c}}\,\texttt{Cons} \qquad \dfrac{}{\nu x.\, t \xRightarrow{\varnothing} \nu x.\, t}\,\texttt{Fresh}_1 \qquad \dfrac{t \xRightarrow{\mathsf{G}} P \qquad x \text{ fresh}}{\nu x.\, t \xRightarrow{\mathsf{G}} P}\,\texttt{Fresh}_2$$

$$\dfrac{}{\lambda x.\, P \xRightarrow{\varnothing} \lambda x.\, P}\,\texttt{Abs}_1^{\mathtt{C}} \qquad \dfrac{\ell \text{ fresh}}{\lambda x.\, P \xRightarrow{\varnothing} \lambda^{\ell} x.\, P}\,\texttt{Abs}_2^{\mathtt{C}} \qquad \dfrac{}{\lambda^{\ell} x.\, P \xRightarrow{\varnothing} \lambda^{\ell} x.\, P}\,\texttt{Abs}^{\mathtt{A}}$$

$$\dfrac{t \xRightarrow{\mathsf{G}} P \qquad s \xRightarrow{\mathsf{H}} Q}{t\, s \xRightarrow{\mathsf{G}\cup\mathsf{H}} P\, Q}\,\texttt{App}_1 \qquad \dfrac{}{(\lambda^{\ell} x.\, P)\, \mathtt{v} \xRightarrow{\varnothing} P\{x := \mathtt{v}\}}\,\texttt{App}_2 \qquad \dfrac{t \xRightarrow{\mathsf{G}} P \qquad s \xRightarrow{\mathsf{H}} Q}{t; s \xRightarrow{\mathsf{G}\cup\mathsf{H}} P; Q}\,\texttt{Guard}_1$$

$$\dfrac{t \xRightarrow{\mathsf{G}} P}{\mathtt{v}; t \xRightarrow{\mathsf{G}} P}\,\texttt{Guard}_2 \qquad \dfrac{t \xRightarrow{\mathsf{G}} P \qquad s \xRightarrow{\mathsf{H}} Q}{t \overset{\bullet}{=} s \xRightarrow{\mathsf{G}\cup\mathsf{H}} P \overset{\bullet}{=} Q}\,\texttt{Unif}_1 \qquad \dfrac{}{\mathtt{v} \overset{\bullet}{=} \mathtt{w} \xRightarrow{\{\mathtt{v}\overset{\bullet}{=}\mathtt{w}\}} \mathbf{ok}}\,\texttt{Unif}_2$$

As usual, most term constructors have two rules, the rule decorated with “1” is a congruence rule which chooses not to perform any evaluation on the root of the term, while the rule decorated with “2” requires that there is a redex at the root of the term, and contracts it. Note that rule $\texttt{Unif}_2$ does *not* perform the unification of $\mathtt{v}$ and $\mathtt{w}$ immediately; it merely has the effect of propagating the unification constraint.

Using the relation defined above, we are now able to define the relation of **simultaneous reduction** between programs:

$$\dfrac{}{\mathtt{fail} \Rightarrow \mathtt{fail}}\,\texttt{Fail} \qquad \dfrac{t \xRightarrow{\mathsf{G}} P \qquad Q \Rightarrow Q' \qquad P' = \begin{cases} P^{\sigma} & \text{if } \sigma = \mathsf{mgu}(\mathsf{G}) \\ \mathtt{fail} & \text{if } \mathsf{mgu}(\mathsf{G}) \text{ fails} \end{cases}}{t \oplus Q \Rightarrow P' \oplus Q'}\,\texttt{Alt}$$

The following lemma summarizes some of the key properties of simultaneous reduction. Most are straightforward proofs by induction, except for item 3.:

Lemma 4.2 (Properties of simultaneous reduction).

1. **Reflexivity.** *$t \overset{\varnothing}{\Longrightarrow} t$ and $P \Rightarrow P$.*
2. **Context closure.** *If $t \overset{\mathsf{G}}{\Longrightarrow} P$ then $\mathsf{W}\langle t\rangle \overset{\mathsf{G}}{\Longrightarrow} \mathsf{W}\langle P\rangle$.*
3. **Strong bisimulation.** *Structural equivalence is a strong bisimulation with respect to $\Rightarrow$,* i.e. *if $P \equiv P' \Rightarrow Q$ then there is a program Q' such that $P \Rightarrow Q' \equiv Q$.*
4. **Substitution.** *If $t \overset{\mathsf{G}}{\Longrightarrow} P$ then $t^{\sigma} \overset{\mathsf{G}^{\sigma}}{\Longrightarrow} P^{\sigma}$.*

The core argument is the following adaptation of Tait–Martin-Löf's technique, from which confluence comes out as an easy corollary. See in the appendix for details.

Proposition 4.3 (Tait–Martin-Löf's technique, up to $\equiv$).
1. $\to \subseteq \Rightarrow\equiv$
2. $\Rightarrow \subseteq \twoheadrightarrow\equiv$
3. $\Rightarrow$ has the diamond property, up to $\equiv$, that is:
If $P_1 \Rightarrow P_2$ and $P_1 \Rightarrow P_3$ then $P_2 \Rightarrow\equiv P_4$ and $P_3 \Rightarrow\equiv P_4$ for some P_4.

Theorem 4.4 (Confluence). *The reduction relation $\to$ is confluent, up to $\equiv$. More precisely, if $P_1 \twoheadrightarrow P_2$ and $P_1 \twoheadrightarrow P_3$ then there is a program P_4 such that $P_2 \twoheadrightarrow\equiv P_4$ and $P_3 \twoheadrightarrow\equiv P_4$.*

Example 4.5. Suppose that $\sigma = \mathsf{mgu}(\mathtt{v}_1 \overset{\bullet}{=} \mathtt{v}_2)$ and $\tau = \mathsf{mgu}(\mathtt{w}_1 \overset{\bullet}{=} \mathtt{w}_2)$. Consider:

$$({\mathtt{v}_1}^{\tau} \overset{\bullet}{=} {\mathtt{v}_2}^{\tau})\,\mathbf{ok}\,t^{\tau} \leftarrow (\mathtt{v}_1 \overset{\bullet}{=} \mathtt{v}_2)\,(\mathtt{w}_1 \overset{\bullet}{=} \mathtt{w}_2)\,t \to \mathbf{ok}\,({\mathtt{w}_1}^{\sigma} \overset{\bullet}{=} {\mathtt{w}_2}^{\sigma})\,t^{\sigma}$$

Then both $\sigma' = \mathsf{mgu}({\mathtt{v}_1}^{\tau} \overset{\bullet}{=} {\mathtt{v}_2}^{\tau})$ and $\tau' = \mathsf{mgu}({\mathtt{w}_1}^{\sigma} \overset{\bullet}{=} {\mathtt{w}_2}^{\sigma})$ must exist, and the peak may be closed as follows:

$$({\mathtt{v}_1}^{\tau} \overset{\bullet}{=} {\mathtt{v}_2}^{\tau})\,\mathbf{ok}\,t^{\tau} \to \mathbf{ok}\,\mathbf{ok}\,(t^{\tau})^{\sigma'} \equiv \mathbf{ok}\,\mathbf{ok}\,(t^{\sigma})^{\tau'} \leftarrow \mathbf{ok}\,({\mathtt{w}_1}^{\sigma} \overset{\bullet}{=} {\mathtt{w}_2}^{\sigma})\,t^{\sigma}$$

the equivalence relies on the fact that $\tau' \circ \sigma$ and $\sigma' \circ \tau$ are both most general unifiers of $\{\mathtt{v}_1 \overset{\bullet}{=} \mathtt{v}_2, \mathtt{w}_1 \overset{\bullet}{=} \mathtt{w}_2\}$, hence $(t^{\tau})^{\sigma'} \equiv (t^{\sigma})^{\tau'}$, up to renaming.

5 Simple Types for $\lambda^{\mathtt{U}}$

In this section we discuss a simply typed system for the $\lambda^{\mathtt{U}}$-calculus. The system does not present any essential difficulty, but it is a necessary prerequisite to be able to define the denotational semantics of Sec. 6. The main result in this section is subject reduction (Prop. 5.2).

Note that, unlike in the simply typed λ-calculus, reduction may create free variables, due to fresh variable introduction. For instance, in the reduction step $\mathbf{c}(\nu x.\,x) \to \mathbf{c}\,x$, a new variable x appears free on the right-hand side. Therefore the subject reduction lemma has to *extend* the typing context in order to account for freshly created variables. This may be understood only as a matter of notation, *e.g.* in a different presentation of the $\lambda^{\mathtt{U}}$-calculus the step above could be written as $\mathbf{c}(\nu x.\,x) \to \nu x.\,(\mathbf{c}\,x)$, using a *scope extrusion* rule reminiscent of the rule to create new channels in process calculi (*e.g.* π-calculus), avoiding the creation of free variables.

Types and typing contexts. Suppose given a denumerable set of *base types* $\alpha, \beta, \gamma, \ldots$. The sets of *types* $\mathsf{Type} = \{A, B, \ldots\}$ and *typing contexts* $\Gamma, \Delta, \ldots$ are given by:

$$A, B, \ldots ::= \alpha \mid A \to B \qquad \Gamma ::= \varnothing \mid \Gamma, x : A$$

we assume that no variable occurs twice in a typing context. Typing contexts are to be regarded as finite sets of assumptions of the form $(x : A)$, *i.e.* we work implicitly modulo contraction and exchange. We assume that each constructor $\mathbf{c}$ has an associated type $\mathcal{T}_{\mathbf{c}}$.

Typing rules. Judgments are of the form. "$\Gamma \vdash X : A$" where X may be a term or a program, meaning that X has type A under Γ. The typing rules are the following:

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A}\,\texttt{t-var} \qquad \frac{}{\Gamma \vdash \mathbf{c} : \mathcal{T}_{\mathbf{c}}}\,\texttt{t-cons} \qquad \frac{\Gamma, x : A \vdash P : B}{\Gamma \vdash \lambda^{(\ell)} x. P : A \to B}\,\texttt{t-lam(l)}$$

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash s : A}{\Gamma \vdash t \overset{\bullet}{=} s : \mathcal{T}_{\mathbf{ok}}}\,\texttt{t-unif} \qquad \frac{\Gamma \vdash t : \mathcal{T}_{\mathbf{ok}} \qquad \Gamma \vdash s : A}{\Gamma \vdash t; s : A}\,\texttt{t-guard}$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \nu x. t : B}\,\texttt{t-fresh} \qquad \frac{}{\Gamma \vdash \texttt{fail} : A}\,\texttt{t-fail} \qquad \frac{\Gamma \vdash t : A \qquad \Gamma \vdash P : A}{\Gamma \vdash t \oplus P : A}\,\texttt{t-alt}$$

Note that all abstractions are typed in the same way, regardless of whether they are allocated or not. A unification has the same type as the constructor $\mathbf{ok}$, as does t in the guarded expression $(t; s)$. A freshly introduced variable of type A represents, from the logical point of view, an unjustified assumption of A. The empty program `fail` can also be given any type. All the threads in a program must have the same type. The following properties of the type system are routine:

Lemma 5.1. *Let X stand for either a term or a program. Then:*

1. **Weakening.** *If $\Gamma \vdash X : A$ then $\Gamma, x : B \vdash X : A$.*
2. **Strengthening.** *If $\Gamma, x : A \vdash X : B$ and $x \notin \mathsf{fv}(X)$, then $\Gamma \vdash X : B$.*
3. **Substitution.** *If $\Gamma, x : A \vdash X : B$ and $\Gamma \vdash s : A$ then $\Gamma \vdash X\{x := s\} : B$.*
4. **Contextual substitution.** *$\Gamma \vdash \mathsf{W}\langle t \rangle : A$ holds if and only if there is a type B such that $\Gamma, \Box : B \vdash \mathsf{W} : A$ and $\Gamma \vdash t : B$ hold.*
5. **Program composition/decomposition.** *$\Gamma \vdash P \oplus Q : A$ holds if and only if $\Gamma \vdash P : A$ and $\Gamma \vdash Q : A$ hold.*

Proposition 5.2 (Subject reduction). *Let $\Gamma \vdash P : A$ and $P \to Q$. Then $\Gamma' \vdash Q : A$, where $\Gamma' = \Gamma$ if the step is derived using any reduction rule other than* `fresh`*, and $\Gamma' = (\Gamma, x : B)$ if the step introduces a fresh variable $(x : B)$.*

Proof. By case analysis on the transition $P \to Q$, using Lem. 5.1. The interesting case is the `unif` case, which requires proving that the substitution σ returned by $\mathsf{mgu}(\mathsf{G})$ preserves the types of the instantiated variables.

6 Denotational Semantics

In this section we propose a *naive* denotational semantics for the $\lambda^{\mathtt{U}}$-calculus. The semantics is naive in at least three senses: first, types are interpreted merely as sets, rather than as richer structures (*e.g.* complete partial orders) or in a more abstract (*e.g.* categorical) framework. Second, since types are interpreted as sets, the *multiplicities* of results are not taken into account, so for example $[\![x \oplus x]\!] = [\![x]\!] \cup [\![x]\!] = [\![x]\!]$. Third, and most importantly, the denotation of abstractions $(\lambda x.\, P)$ is conflated with the denotation of allocated abstractions $(\lambda^{\ell} x.\, P)$. This means that the operational semantics cannot be complete with respect to the denotational one, given that for example $\lambda^{\ell} x.\, x$ and $\lambda^{\ell'} x.\, x$ have the same denotation but they are not observationally equivalent[6]. Nevertheless, studying this simple denotational semantics already presents some technical challenges, and we regard it as a first necessary step towards formulating a better behaved semantics[7].

Roughly speaking, the idea is that a type A shall be interpreted as a set $[\![A]\!]$, while a program P of type A shall be interpreted as a subset $[\![P]\!] \subseteq [\![A]\!]$. For example, if $[\![\mathtt{Nat}]\!] = \mathbb{N}$, then given constructors $\mathbf{1} : \mathtt{Nat}$, $\mathbf{2} : \mathtt{Nat}$ with their obvious interpretations, and if $\mathit{add} : \mathtt{Nat} \to \mathtt{Nat} \to \mathtt{Nat}$ denotes addition, we expect that:

$$[\![(\lambda f : \mathtt{Nat} \to \mathtt{Nat}.\, \nu y.\, ((y \overset{\bullet}{=} \mathbf{1}); \mathit{add}\, y\, (f\, y)))(\lambda x.\, x \oplus \mathbf{2})]\!] = \{1+1, 1+2\} = \{2, 3\}$$

The soundness result that we shall prove states that if $P \twoheadrightarrow Q$ then $[\![P]\!] \supseteq [\![Q]\!]$. Intuitively, the possible behaviors of Q are among the possible behaviors of P.

To formulate the denotational semantics, for ease of notation, we work with an *à la Church* variant of the type system. That is, we suppose that the set of variables is partitioned in such a way that each variable has an intrinsic type. More precisely, for each type A there is a denumerably infinite set of variables $x^A, y^A, z^A, \ldots$ of that type. We also decorate each occurrence of `fail` with its type, *i.e.* we write $\mathtt{fail}^A$ for the empty program of type A. Sometimes we omit the type decoration if it is clear from the context. Under this assumption, it is easy to show that the system enjoys a strong form of *unique typing*, *i.e.* that if X is a typable term or program then there is a unique derivation $\Gamma \vdash X : A$, up to weakening of Γ with variables not in $\mathsf{fv}(X)$. This justifies that we may write $\vdash X : A$ omitting the context.

Domain of interpretation. We suppose given a **non-empty** set S_α for each base type α. The *interpretation* of a type A is a set written $[\![A]\!]$ and defined recursively as follows, where $\mathcal{P}(X)$ is the usual set-theoretic power set, and Y^X is the set of functions with domain X and codomain Y:

$$[\![\alpha]\!] \overset{\text{def}}{=} \mathsf{S}_\alpha \qquad [\![A \to B]\!] \overset{\text{def}}{=} \mathcal{P}([\![B]\!])^{[\![A]\!]}$$

[6] *E.g.* $\lambda^{\ell} x.\, x \overset{\bullet}{=} \lambda^{\ell} x.\, x$ succeeds but $\lambda^{\ell} x.\, x \overset{\bullet}{=} \lambda^{\ell'} x.\, x$ fails.

[7] We expect that a less naive semantics should be stateful, involving a *memory*, in such a way that abstractions $(\lambda x.\, P)$ allocate a memory cell and store a closure, whereas allocated abstractions $(\lambda^{\ell} x.\, P)$ denote a memory location in which a closure is already stored.

Note that, for every type A, the set $[\![A]\!]$ is non-empty, given that we require that S_α be non-empty. This decision is not arbitrary; rather it is necessary for soundness to hold. For instance, operationally we have that $x^A; y^B \xrightarrow{\texttt{guard}} y^B$, so denotationally we would expect $[\![x^A; y^B]\!] \supseteq [\![y^B]\!]$. This would not hold if $[\![A]\!] = \varnothing$ and $[\![B]\!] \neq \varnothing$, as then $[\![x^A; y^B]\!] = \varnothing$ whereas $[\![y^B]\!]$ would be a non-empty set.

Another technical constraint that we must impose is that *the interpretation of a value should always be a singleton.* For example, operationally we have that $(\lambda x : \texttt{Nat}.\, x + x)\,\texttt{v} \twoheadrightarrow \texttt{v} + \texttt{v}$, so denotationally, by soundness, we would expect that $[\![(\lambda x : \texttt{Nat}.\, x + x)\,\texttt{v}]\!] \supseteq [\![\texttt{v} + \texttt{v}]\!]$. If we had that $[\![\texttt{v}]\!] = \{1, 2\}$ is not a singleton, then we would have that $[\![(\lambda x.\, x + x)\,\texttt{v}]\!] = \{1 + 1, 2 + 2\}$ whereas $[\![\texttt{v} + \texttt{v}]\!] = \{1 + 1, 1 + 2, 2 + 1, 2 + 2\}$.

Following this principle, given that terms of the form $\mathbf{c}\,\texttt{v}_1 \ldots \texttt{v}_n$ are values, their denotation $[\![\mathbf{c}\,\texttt{v}_1 \ldots \texttt{v}_n]\!]$ must always be a singleton. This means that constructors must be interpreted as singletons, and constructors of function type should always return singletons (which in turn should return singletons if they are functions, and so on, recursively). Formally, any element $a \in [\![\alpha]\!]$ is declared to be α**-unitary**, and a function $f \in [\![A \to B]\!]$ is $(A \to B)$**-unitary** if for each $a \in [\![A]\!]$ the set $f(a) = \{b\} \subseteq [\![B]\!]$ is a singleton and b is B-unitary. Sometimes we say that an element a is *unitary* if the type is clear from the context. If f is $(A \to B)$-unitary, and $a \in [\![A]\!]$ sometimes, by abuse of notation, we may write $f(a)$ for the unique element $b \in f(a)$.

Interpretation of terms. For each constructor $\mathbf{c}$, we suppose given a $\mathcal{T}_\mathbf{c}$-unitary element $\underline{\mathbf{c}} \in [\![\mathcal{T}_\mathbf{c}]\!]$. Moreover, we suppose that the interpretation of constructors is *injective*, *i.e.* that $\underline{\mathbf{c}}(a_1) \ldots (a_n) = \underline{\mathbf{c}}(b_1) \ldots (b_n)$ implies $a_i = b_i$ for all $i = 1..n$.

An *environment* is a function $\rho : \mathsf{Var} \to \bigcup_{A \in \mathsf{Type}} [\![A]\!]$ such that $\rho(x^A) \in [\![A]\!]$ for each variable x^A of each type A. If ρ is an environment and $a \in [\![A]\!]$, we write $\rho[x^A \mapsto a]$ for the environment that maps x^A to a and agrees with ρ on every other variable. We write Env for the set of all environments.

Let $\vdash t : A$ (resp. $\vdash P : A$) be a typable term (resp. program) and let ρ be an environment. If $\vdash X : A$ is a typable term or program, we define its *denotation under the environment* ρ, written $[\![X]\!]_\rho$ as a subset of $[\![A]\!]$ as follows:

$$\begin{aligned}
[\![x^A]\!]_\rho &\stackrel{\text{def}}{=} \{\rho(x^A)\} \\
[\![\mathbf{c}]\!]_\rho &\stackrel{\text{def}}{=} \{\underline{\mathbf{c}}\} \\
[\![\lambda x^A.\, P]\!]_\rho &\stackrel{\text{def}}{=} \{f\} \quad \text{ where } f : [\![A]\!] \to \mathcal{P}([\![B]\!]) \text{ is given by } f(a) = [\![P]\!]_{\rho[x^A \mapsto a]} \\
[\![\lambda^\ell x^A.\, P]\!]_\rho &\stackrel{\text{def}}{=} \{f\} \quad \text{ where } f : [\![A]\!] \to \mathcal{P}([\![B]\!]) \text{ is given by } f(a) = [\![P]\!]_{\rho[x^A \mapsto a]} \\
[\![t\, s]\!]_\rho &\stackrel{\text{def}}{=} \{b \mid \exists f \in [\![t]\!]_\rho,\ \exists a \in [\![s]\!]_\rho,\ b \in f(a)\} \\
[\![t \stackrel{\bullet}{=} s]\!]_\rho &\stackrel{\text{def}}{=} \{\underline{\mathbf{ok}} \mid \exists a \in [\![t]\!]_\rho,\ \exists b \in [\![s]\!]_\rho,\ a = b\} \\
[\![t; s]\!]_\rho &\stackrel{\text{def}}{=} \{a \mid \exists b \in [\![t]\!]_\rho,\ a \in [\![s]\!]_\rho\} \\
[\![\nu x^A.\, t]\!]_\rho &\stackrel{\text{def}}{=} \{b \mid \exists a \in [\![A]\!],\ b \in [\![t]\!]_{\rho[x^A \mapsto a]}\} \\
[\![\texttt{fail}^A]\!]_\rho &\stackrel{\text{def}}{=} \varnothing \\
[\![t \oplus P]\!]_\rho &\stackrel{\text{def}}{=} [\![t]\!]_\rho \cup [\![P]\!]_\rho
\end{aligned}$$

The denotation of a toplevel program is written $[\![P]\!]$ and defined as the union of its denotations under all possible environments, *i.e.* $[\![P]\!] \overset{\text{def}}{=} \bigcup_{\rho \in \mathsf{Env}} [\![P]\!]_\rho$.

Proposition 6.1 (Properties of the denotational semantics).

1. **Irrelevance.** *If ρ and ρ' agree on $\mathsf{fv}(X)$, then $[\![X]\!]_\rho = [\![X]\!]_{\rho'}$. Here X stands for either a program or a term.*
2. **Compositionality.**
   2.1 $[\![P \oplus Q]\!]_\rho = [\![P]\!]_\rho \cup [\![Q]\!]_\rho$.
   2.2 If W is a context whose hole is of type A, then $[\![\mathsf{W}\langle t\rangle]\!]_\rho = \{b \mid a \in [\![t]\!]_\rho, b \in [\![\mathsf{W}]\!]_{\rho[\Box^A \mapsto a]}\}$.
3. **Interpretation of values.** *If $\mathtt{v}$ is a value then $[\![\mathtt{v}]\!]_\rho$ is a singleton.*
4. **Interpretation of substitution.**
   Let $\sigma = \{x_1^{A_1} \mapsto \mathtt{v}_1, \dots, x_n^{A_n} \mapsto \mathtt{v}_n\}$ be a substitution such that $x_i \notin \mathsf{fv}(\mathtt{v}_j)$ for all i, j. Let $[\![\mathtt{v}_i]\!]_\rho = \{a_i\}$ for each $i = 1..n$ (noting that values are singletons, by the previous item of this lemma). Then for any program or term X we have that $[\![X^\sigma]\!]_\rho = [\![X]\!]_{\rho[x_1 \mapsto a_1]\dots[x_n \mapsto a_n]}$.

To conclude this section, the following theorem shows that the operational semantics is sound with respect to the denotational semantics.

Theorem 6.2 (Soundness). *Let $\Gamma \vdash P : A$ and $P \to Q$. Then $[\![P]\!] \supseteq [\![Q]\!]$. The inclusion is an equality for all reduction rules other than the* `fail` *rule.*

Example 6.3. Consider the reduction $\nu x.\ \Big((\lambda z.\, \nu y.\, ((z \overset{\bullet}{=} \mathbf{t}\,\mathbf{1}\,y); (\mathbf{t}\,y\,x)))\,(\mathbf{t}\,x\,\mathbf{2})\Big) \twoheadrightarrow \mathbf{t}\,\mathbf{2}\,\mathbf{1}$. If $[\![\mathtt{Tuple}]\!] = [\![\mathtt{Nat}]\!] \times [\![\mathtt{Nat}]\!] = \mathbb{N} \times \mathbb{N}$, the constructors $\mathbf{1} : \mathtt{Nat}$, $\mathbf{2} : \mathtt{Nat}$ are given their obvious interpretations and $\mathbf{t} : \mathtt{Nat} \to \mathtt{Nat} \to \mathtt{Tuple}$ is the pairing function[8], then for any environment ρ, if we abbreviate $\rho' := \rho[x \mapsto n][z \mapsto p][y \mapsto m]$, we have:

$$\begin{aligned}
&[\![\nu x.\ \Big((\lambda z.\, \nu y.\, ((z \overset{\bullet}{=} \mathbf{t}\,\mathbf{1}\,y); (\mathbf{t}\,y\,x)))\,(\mathbf{t}\,x\,\mathbf{2})\Big)]\!]_\rho \\
&= \{[\![(\lambda z.\, \nu y.\, ((z \overset{\bullet}{=} \mathbf{t}\,\mathbf{1}\,y); (\mathbf{t}\,y\,x)))\,(\mathbf{t}\,x\,\mathbf{2})]\!]_{\rho[x\mapsto n]} \mid n \in \mathbb{N}\} \\
&= \{r \mid n \in \mathbb{N}, f \in [\![\lambda z.\, \nu y.\, ((z \overset{\bullet}{=} \mathbf{t}\,\mathbf{1}\,y); (\mathbf{t}\,y\,x))]\!]_{\rho[x\mapsto n]}, p \in [\![\mathbf{t}\,x\,\mathbf{2}]\!]_{\rho[x\mapsto n]}, r \in f(p)\} \\
&= \{r \mid n, m \in \mathbb{N}, p \in [\![\mathbf{t}\,x\,\mathbf{2}]\!]_{\rho[x\mapsto n]}, r \in [\![(z \overset{\bullet}{=} \mathbf{t}\,\mathbf{1}\,y); (\mathbf{t}\,y\,x)]\!]_{\rho'}\} \\
&= \{r \mid n, m \in \mathbb{N}, p \in \{(n, 2)\}, r \in [\![(z \overset{\bullet}{=} \mathbf{t}\,\mathbf{1}\,y); (\mathbf{t}\,y\,x)]\!]_{\rho'}\} \\
&= \{r \mid n, m \in \mathbb{N}, p \in \{(n, 2)\}, b \in [\![z \overset{\bullet}{=} \mathbf{t}\,\mathbf{1}\,y]\!]_{\rho'}, r \in [\![\mathbf{t}\,y\,x]\!]_{\rho'}\} \\
&= \{r \mid n, m \in \mathbb{N}, p \in \{(n, 2)\}, p = (1, m), r \in [\![\mathbf{t}\,y\,x]\!]_{\rho'}\} \\
&= \{r \mid n \in \{1\}, m \in \{2\}, p \in \{(1, 2)\}, r \in [\![\mathbf{t}\,y\,x]\!]_{\rho'}\} \\
&= \{(2, 1)\} \\
&= [\![\mathbf{t}\,\mathbf{2}\,\mathbf{1}]\!]_\rho
\end{aligned}$$

An example in which the inclusion is proper is the reduction step $\lambda^\ell x.\, x \overset{\bullet}{=} \lambda^{\ell'} x.\, x \xrightarrow{\mathtt{fail}} \mathtt{fail}$. Note that $[\![\lambda^\ell x.\, x \overset{\bullet}{=} \lambda^{\ell'} x.\, x]\!] = \{\underline{\mathbf{ok}}\} \supsetneq \varnothing = [\![\mathtt{fail}]\!]$, given that our naive semantics equates the denotations of the abstractions, *i.e.* $[\![\lambda^\ell x.\, x]\!] = [\![\lambda^{\ell'} x.\, x]\!]$, in spite of the fact that their locations differ.

[8] Precisely, $\underline{\mathbf{t}}(n) = \{f_n\}$ with $f_n(m) = \{(n, m)\}$.

7 Conclusion

In this work, we have proposed the **$\lambda^{\mathtt{U}}$-calculus** (Def. 3.1) an extension of the λ-calculus with relational features, including non-deterministic choice and first-order unification. We have studied some of its operational properties, providing an inductive **characterization of normal forms** (Prop. 4.1), and proving that it is **confluent** (Thm. 4.4) up to structural equivalence, by adapting the technique by Tait and Martin-Löf. We have proposed a system of simple types enjoying **subject reduction** (Prop. 5.2). We have also proposed a naive denotational semantics, in which a program of type A is interpreted as a set of elements of a set $[\![A]\!]$, for which we have proven **soundness** (Thm. 6.2). The denotational semantics is not complete.

As of the writing of this paper, we are attempting to formulate a refined denotational semantics involving a notion of *memory*, following the ideas mentioned in footnote 7. One difficulty is that in a term like $((x \overset{\bullet}{=} \lambda z.\, z); y)((y \overset{\bullet}{=} \lambda z.\, z); x)$, there seems to be a cyclic dependency between the denotation of the subterm on the left and denotation of the subterm on the right, so it is not clear how to formulate the semantics compositionally.

We have attempted to prove normalization results for the simply typed system, until now unsuccessfully. Given a constructor $\mathbf{c} : (A \to A) \to A$, a self-looping term $\omega(\mathbf{c}\,\omega)$ with $\omega \overset{\text{def}}{=} \lambda x^A.\, \nu y^{A\to A}.\, ((\mathbf{c} y \overset{\bullet}{=} x); y\, x)$ can be built, so some form of *positivity condition* should be imposed. Other possible lines for future work include studying the relationship between calculi with patterns and $\lambda^{\mathtt{U}}$ by means of translations, and formulating richer type systems. For instance, one would like to be able to express *instantiation restrictions*, in such a way that a fresh variable representing a natural number is of type $\mathtt{Nat}^-$ while a term of type $\mathtt{Nat}^+$ represents a fully instantiated natural number.

Related Work. On **functional–logic** programming, we have mentioned λProlog [?,?] and Curry [?,?]. Other languages combining functional and logic features are Mercury [?] and Mozart/Oz [?]. There is a vast amount of literature on functional–logic programming. We mention a few works which most resemble our own. Miller [?] proposes a language with lambda-abstraction and a decidable extension of first-order unification which admits most general unifiers. Chakravarty et al. [?] and Smolka [?] propose languages in which the functional–logic paradigm is modeled as a concurrent process with communication. Albert et al. [?] formulate a *big-step* semantics for a functional–logic calculus with narrowing. On pure **relational** programming (without λ-abstractions), recently Rozhplokas et al. [?] have studied the operational and denotational semantics of miniKanren. On **λ-calculi with patterns** (without full unification), there have been many different approaches to their formulation [?,?,?,?,?]. On **λ-calculi with non-deterministic choice** (without unification), we should mention works on the λ-calculus extended with *erratic* [?] as well as with *probabilistic* choice [?,?].

Acknowledgements. To Alejandro Díaz-Caro for supporting our interactions. To Eduardo Bonelli, Delia Kesner, and the anonymous reviewers for their feedback and suggestions.

References

1. Albert, E., Hanus, M., Huch, F., Oliver, J., Vidal, G.: Operational semantics for functional logic languages. Electronic Notes in Theoretical Computer Science **76**, 1–19 (2002)
2. Arbiser, A., Miquel, A., Ríos, A.: A lambda-calculus with constructors. In: International Conference on Rewriting Techniques and Applications. pp. 181–196. Springer (2006)
3. Ayala-Rincón, M., Bonelli, E., Edi, J., Viso, A.: Typed path polymorphism. Theoretical Computer Science **781**, 111–130 (2019)
4. Barendregt, H.: The Lambda Calculus: Its Syntax and Semantics, vol. 103. Elsevier (1984)
5. Byrd, W.E.: Relational programming in miniKanren: techniques, applications, and implementations. [Bloomington, Ind.]: Indiana University (2010)
6. Chakravarty, M.M., Guo, Y., Köhler, M., Lock, H.C.: Goffin: Higher-order functions meet concurrent constraints. Science of Computer Programming **30**(1-2), 157–199 (1998)
7. Faggian, C., Rocca, S.R.D.: Lambda calculus and probabilistic computation. In: 34th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2019, Vancouver, BC, Canada, June 24-27, 2019. pp. 1–13. IEEE (2019)
8. Friedman, D.P., Byrd, W.E., Kiselyov, O.: The Reasoned Schemer. The MIT Press (July 2005)
9. Gould, W.E.: A Matching Procedure for Omega-Order Logic. Ph.D. thesis, Princeton University (1966)
10. Hanus, M.: Functional logic programming: From theory to Curry. In: Programming Logics - Essays in Memory of Harald Ganzinger. pp. 123–168. Springer LNCS 7797 (2013)
11. Hanus, M.: A unified computation model for functional and logic programming. In: Lee, P., Henglein, F., Jones, N.D. (eds.) Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997. pp. 80–93. ACM Press (1997)
12. Huet, G.P.: The undecidability of unification in third order logic. Information and control **22**(3), 257–267 (1973)
13. Huet, G.P.: A unification algorithm for typed λ-calculus. Theoretical Computer Science **1**(1), 27–57 (1975)
14. Jay, B., Kesner, D.: Pure pattern calculus. In: European Symposium on Programming. pp. 100–114. Springer (2006)
15. Klop, J.W., Van Oostrom, V., De Vrijer, R.: Lambda calculus with patterns. Theoretical Computer Science **398**(1-3), 16–31 (2008)
16. Levy, J., Veanes, M.: On the undecidability of second-order unification. Information and Computation **159**(1-2), 125–150 (2000)
17. Martelli, A., Montanari, U.: An efficient unification algorithm. ACM Transactions on Programming Languages and Systems (TOPLAS) **4**(2), 258–282 (1982)
18. Miller, D.: A logic programming language with lambda-abstraction, function variables, and simple unification. Journal of logic and computation **1**(4), 497–536 (1991)
19. Miller, D.: Unification of simply typed lambda-terms as logic programming. Tech. Rep. MS-CIS-91-24, University of Pennsylvania (1991)

20. Miller, D., Nadathur, G.: Programming with higher-order logic. Cambridge University Press (2012)
21. Milner, R.: A theory of type polymorphism in programming. Journal of computer and system sciences **17**(3), 348–375 (1978)
22. Nadathur, G., Miller, D.: Higher-order logic programming. Proceedings of the Third International Logic Programming Conference pp. 448–462 (1984)
23. Petit, B.: Semantics of typed lambda-calculus with constructors. Log. Methods Comput. Sci. **7**(1) (2011)
24. Ramsey, N., Pfeffer, A.: Stochastic lambda calculus and monads of probability distributions. In: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 154–165 (2002)
25. Rozplokhas, D., Vyatkin, A., Boulytchev, D.: Certified semantics for minikanren. In: Proceedings of the 2019 miniKanren and Relational Programming Workshop. pp. 80–98 (2019)
26. Schmidt-Schauß, M., Huber, M.: A lambda-calculus with letrec, case, constructors and non-determinism. arXiv preprint cs/0011008 (2000)
27. Smolka, G.: A foundation for higher-order concurrent constraint programming. In: Mathematical Methods in Program Development, pp. 433–458. Springer (1997)
28. Somogyi, Z., Henderson, F., Conway, T.: The execution algorithm of mercury, an efficient purely declarative logic programming language. The Journal of Logic Programming **29**(1-3), 17–64 (1996)
29. The Coq Development Team: The Coq proof assistant reference manual. LogiCal Project (2004), http://coq.inria.fr, version 8.0
30. Urban, C., Pitts, A.M., Gabbay, M.J.: Nominal unification. Theoretical Computer Science **323**(1-3), 473–497 (2004)
31. Van Roy, P.: Multiparadigm Programming in Mozart/Oz: Second International Conference, MOZ 2004, Charleroi, Belgium, October 7-8, 2004, Revised Selected Papers, vol. 3389. Springer (2005)