Function Spaces for Orbit-Finite Sets

Mikołaj Bojańczyk
University of Warsaw, Poland

Lê Thành Dũng (Tito) Nguyễn
École normale supérieure de Lyon, France

Rafał Stefański
University of Warsaw, Poland

Abstract

Orbit-finite sets are a generalisation of finite sets, and as such support many operations allowed for finite sets, such as pairing, quotienting, or taking subsets. However, they do not support function spaces, i.e. if X and Y are orbit-finite sets, then the space of finitely supported functions from X to Y is not orbit-finite. We propose a solution to this problem inspired by linear logic.

2012 ACM Subject Classification Theory of computation → Formal languages and automata theory

Keywords and phrases Orbit-finite sets, automata, linear types, game semantics

Digital Object Identifier 10.4230/LIPIcs.ICALP.2024.130

Category Track B: Automata, Logic, Semantics, and Theory of Programming

Related Version *Full Version with Appendix*: `https://arxiv.org/abs/2404.05265`

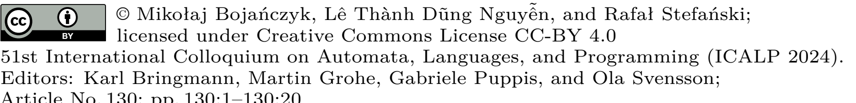
Funding *Mikołaj Bojańczyk*: Supported by Polish NCN Maestro Grant 2022/46/A/ST6/00072.
Lê Thành Dũng (Tito) Nguyễn: Supported by the LABEX MILYON (ANR-10-LABX-0070) of Université de Lyon, within the program "France 2030" (ANR-11-IDEX-0007) operated by the French National Research Agency (ANR).
Rafał Stefański: Supported by EPSRC project EP/V040944/1 "Resources in Computation".

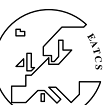
Acknowledgements L. T. D. Nguyễn would like to thank Clovis Eberhart and Cécilia Pradic for their ongoing collaboration on "implicit automata for data words" (cf. [18, §1.4.4]), which inspired some ideas in this paper. R. Stefański would like to thank Samson Abramsky for introducing him to the beautiful world of game semantics and suggesting to study it in the context of this paper.

1 Introduction

The class of orbit-finite sets is a class of sets that contains all finite sets and some infinite sets, but still shares some properties with the class of finite sets. The idea, which dates back to Fraenkel–Mostowski models of set theory, is to begin with an infinite set $\mathbb{A}$ of *atoms* or *urelements*. We think of the atoms as being names, such as Eve or John, and atoms can only be compared with respect to equality. Intuitively speaking, an orbit-finite set is a set that can be constructed using the atoms, such as $\mathbb{A}^2$ or $\mathbb{A}^*$, subject to the constraint that there are finitely many elements up to renaming atoms. For example, $\mathbb{A}^2$ is orbit-finite because it has two elements up to renaming atoms, namely (John, John) and (John, Eve), while $\mathbb{A}^*$ is not orbit-finite, because the length of a sequence is invariant under renaming atoms, and there are infinitely many possible lengths. For a survey on orbit-finite sets, see [5].

The notion of orbit-finiteness can be seen as an attempt to find an appropriate notion of finiteness for the nominal sets of Gabbay and Pitts [20]. This attempt emerged from the study of computational models such as monoids [6] and automata [7] over infinite alphabets. Let us illustrate orbit-finiteness using an automaton example.


© Mikołaj Bojańczyk, Lê Thành Dũng Nguyễn, and Rafał Stefański;
licensed under Creative Commons License CC-BY 4.0
51st International Colloquium on Automata, Languages, and Programming (ICALP 2024).
Editors: Karl Bringmann, Martin Grohe, Gabriele Puppis, and Ola Svensson;
Article No. 130; pp. 130:1–130:20

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany


▶ **Example 1.1** (An orbit-finite automaton). Let $L \subseteq \mathbb{A}^*$ be the language of words in which the letter from the first position does not appear again. This language contains John · Mark · Mark · Eve, because John does not reappear, but it does not contain John · Mark · John. To recognize this language, we can use a deterministic automaton, which uses its state to remember the first letter. In this automaton, the input alphabet is $\Sigma = \mathbb{A}$ and the state space is $Q = 1 + 1 + \mathbb{A}$. In this state space, there are two special states, namely the initial state and a rejecting error state, and furthermore there is one state for each atom $a \in \mathbb{A}$, which represents a situation where the first letter was a but it has not been seen again yet. This state space is infinite; but it is orbit-finite, since each of the three components in Q represents a single orbit. ⌟

Orbit-finite sets have many advantages, which ensure that they are a good setting for automata theory, and discrete mathematics in general. For example, an orbit-finite set can be represented in a finite way [5], which ensures that it becomes meaningful to talk about algorithms that input orbit-finite sets, such as an emptiness check for an automaton. Also, orbit-finite sets are closed under taking disjoint unions and products, which ensures that natural automata constructions, such as the union of two nondeterministic automata or the product of two deterministic automata can be performed.

However, orbit-finite sets do not have all the closure properties of finite sets. Notably missing is the powerset operation, and more generally taking function spaces. For example, if we look at the powerset of $\mathbb{A}$, then this powerset will not be orbit-finite, since already the finite subsets give infinitely many orbits (two finite subsets of different sizes will be in different orbits). The lack of powersets means that one cannot do the subset construction from automata theory, and in particular deterministic and nondeterministic automata are not equivalent. This non-equivalence was known from the early days of automata for infinite alphabets [16], and in fact, some decision problems, such as equivalence, are decidable for deterministic automata but undecidable for nondeterministic automata [17]. Another construction that fails is converting a deterministic automaton into a monoid [6, p. 221]; this is because function spaces on orbit-finite sets are no longer orbit-finite, as explained in the following example.

▶ **Example 1.2** (Failure of the monoid construction). Let us show that the automaton from Example 1.1 cannot be converted into a monoid. The standard construction would be to define the monoid as the subset $M \subseteq Q \to Q$ of all state transformations, namely the subset generated by individual input letters. Unfortunately, this construction does not work. This is because in order for two input words to give the same state transformation, they need to have the same set of letters that appear in them. In particular, the corresponding set of set transformations is not orbit-finite, for the same reason as why the finite powerset is not orbit-finite. Not only does the standard construction not work, but also this language is not recognized by any orbit-finite monoid. ⌟

An attempt to address this problem was provided in [24, 9], based on *single-use* functions. The idea, which originates in linear types and linear logic, is to restrict the functions so that they use each argument at most once. For example, consider the following two functions that input atoms and output Booleans:

$$a \in \mathbb{A} \mapsto \begin{cases} \text{true} & \text{if } a = \text{John} \\ \text{false} & \text{otherwise} \end{cases} \qquad a \in \mathbb{A} \mapsto \begin{cases} \text{true} & \text{if } a = \text{John or } a = \text{Eve} \\ \text{false} & \text{otherwise} \end{cases}$$

Intuitively, the first function is single-use, since it compares the input atom to John only, while the second function is not single-use, since it requires two comparisons, with John and Eve. Here is another example, which shows that the problems with the monoid construction from Example 1.2 could be blamed on a violation of the single-use condition.

▶ **Example 1.3.** Consider the transition function of the automaton in Example 1.1, which inputs a state in $1 + 1 + \mathbb{A}$ together with an input letter from $\mathbb{A}$, and returns a new state. This function is not single use. Indeed, if the state is in $\mathbb{A}$, then the transition function compares it for equality with the input letter; but if the comparison returns true, another copy of the old state must be kept as the new state for future comparisons. ⌟

If one restricts attention to functions that are single-use, much of the usual robustness of automata theory is recovered, with deterministic automata being equivalent to monoids, and both being equivalent to two-way deterministic automata [9].

Despite the success of the single-use restriction in solving automata problems, one would ideally prefer a more principled approach, in which instead of defining single-use automata, we would define a more general object, namely single-use sets and functions. Then the definitions of automata and monoids, as well theorems speaking about them, should arise automatically as a result of suitable closure properties of the sets and functions.

This approach was pursued in [24], in which a *category* of orbit-finite sets with single-use functions was proposed. In this corresponding category, one can represent the set of all single-use functions between two orbit-finite sets X and Y as a new set, call it $X \Rightarrow Y$, which is also orbit-finite. However, as we will see later in this paper, this proposal is not entirely satisfactory, since it fails to account for standard operations that one would like to perform on function spaces, most importantly partial application (currying). In the language of category theory, the proposal from [24] failed to be a monoidal closed category.

Contributions of this paper. The main contribution of this paper is to propose a notion of single-use sets and functions, which extends the proposal from [24], but which is rich enough to be closed under taking function spaces. More formally, we propose a category of single-use functions between orbit-finite sets equipped with additional metadata, and we prove that a suitable quotient of this category is symmetric monoidal closed (Theorem 4.2).

The main idea is to follow the tradition of linear types, and extend the type system from [24] with a new type constructor &. This way we can distinguish between two kinds of products namely $X \otimes Y$ (in [24] denoted as $X \times Y$) and $X \& Y$. Thanks to this distinction, the function space can be built so that the appropriate operations on functions, namely application and currying, can be implemented in a single-use way.

Our proposed category is strongly inspired by linear types, and the proof that it admits function spaces uses a form of *game semantics* – a tool that we take from programming language theory. However, as far as we know, it is an original idea to have an infinite but orbit-finite base type $\mathbb{A}$, and to observe that all constructions in game semantics are consistent with orbit-finiteness. We believe that the resulting category deserves further study, and that it is an interesting and non-trivial example of a category representing “finite” objects.

Some further adjacent developments – such as an alternative solution to the problem of function spaces, using vector spaces of orbit-finite dimension – are presented in Section 7.

2 Sets with atoms

We begin with a brief introduction to orbit-finite sets. For a more detailed treatment, see [5].

Fix for the rest of this paper a countably infinite set $\mathbb{A}$, whose elements will be called atoms. We assume that $\mathbb{A}$ has no other structure except for equality; we will only be interested in notions which are *equivariant*, i.e. invariant under renaming atoms. For example, $\mathbb{A}$ has only two equivariant subsets, namely the empty and full subsets. On the other hand, the set $\mathbb{A}^2$ has four equivariant subsets ($\emptyset$, $\mathbb{A}^2$, the diagonal and its complement). In order to meaningfully speak about equivariant subsets, we must be able to have an action of atom

renamings on the set, as formalized in the following definition. The finite support condition is a technical condition that ensures that the action is well-behaved; it dates back to the work of Fraenkel and Mostowski, and is explained in the survey texts [20, 5].

▶ **Definition 2.1.** *A* set with atoms *is a set X, equipped with an action of the group of atom renamings, subject to the following* finite support condition: *for every $x \in X$ there is a finite set of atoms, such that if an atom renaming π fixes all atoms in the set, then it also fixes x.*

The idea is that a set with atoms is any kind of object that deals with atoms, such as the set $\mathbb{A}^*$ of all words over the alphabet $\mathbb{A}$, or the family of finite subsets of $\mathbb{A}$. Among such objects, we will be interested in those which are "finite". This will be formalized by saying that there are finitely many orbits, as described below. Define the *orbit* of an element x in a set with atoms to be the elements that can be obtained from x by applying some atom renaming. For example, in the set $\mathbb{A}^2$, the orbit of (John, Eve) contains (Mark, John), but it does not contain (John, John). The orbits form a partition of a set with atoms.

▶ **Definition 2.2.** *A set with atoms is called* orbit-finite *if it has finitely many orbits.*

Typical examples of orbit-finite sets are polynomial expressions such as $\mathbb{A}^4 + \mathbb{A}^3 + \mathbb{A}^3 + 1$. Here, 1 represents the set of zero-length sequences; this set has a unique element which is its own orbit. For example, $\mathbb{A}^3$ has five orbits, because there are five possible ways of choosing a pattern of equalities in a sequence of three names. On the other hand, $\mathbb{A}^*$ has infinitely many orbits, since sequences of different lengths are necessarily in different orbits. The family of finite subsets of $\mathbb{A}$ is also not orbit-finite, because subsets of different sizes are in different orbits. The full powerset $\mathsf{P}\mathbb{A}$ is not even a legitimate object in our setting, because some of its elements, i.e. some subsets of $\mathbb{A}$, violate the finite support condition.

2.1 Finiteness of function spaces

As mentioned above, orbit-finite sets can be seen as a certain generalization of finite sets. They allow some, but not all, operations that can usually be done on finite sets. For example, orbit-finite sets are closed under disjoint unions $X + Y$ and products $X \times Y$. Another good property is that an orbit-finite set has only finitely many equivariant subsets (an equivariant subset is one that is invariant under the action of atom permutations). This is because an equivariant subset is a union of some of the finitely many orbits. This accounts for some of the good computational properties of orbit-finite sets.

If X and Y are orbit-finite sets, the set of equivariant functions $X \to Y$ is always literally finite, because it is an equivariant subset of $X \times Y$. However, as we have seen in Example 1.2, when converting an automaton to a monoid, we want to use the *partial applications* $\delta(-, a)$ of the transition function $\delta : Q \times \Sigma \to Q$; while δ is equivariant, in general, $\delta(-, a)$ is not. But it is *finitely supported*, i.e. invariant under all atom renamings that fix some finite set of atoms that depends only on the function. The issue is that the *finitely supported function space* from X to Y is not orbit-finite.

▶ **Example 2.3.** The function $\mathbb{A} \to \mathbb{A}$ below is finitely supported, because it is invariant under all atom renamings that fix Mark, John, Eve and Bill:

$$f(a) = \begin{cases} \text{Mark} & \text{if } a \in \{\text{John, Eve, Bill}\} \\ a & \text{otherwise} \end{cases}$$

If π is the atom renaming that swaps Mark with Adam, then applying it to the function f defined above gives the function $\pi(f)$ that has the same definition (or source code, if a programming intuition is to be followed), except that Mark is used instead of Adam.

In the case of $\mathbb{A} \to \mathbb{A}$, the finitely supported function space is not orbit-finite. Indeed, the condition $a \in \{\text{John, Eve, Bill}\}$ can be replaced by $a \in X$ for any finite set $X \subset \mathbb{A} \setminus \{\text{Mark}\}$ of exceptional values, and two choices of X of different cardinalities will give us two functions in different orbits. ⌟

3 Single-use sets and functions

The lack of function spaces is the problem addressed in this paper. Our solution builds on the idea from [24, Section 2.2], which is to consider only functions that are single-use. This notion, already illustrated intuitively in the introduction, is formalized in Section 3.1, where we show how it almost, but not quite, achieves function spaces. Then, in the rest of this section, we show how function spaces can be recovered by using a more refined type system.

3.1 Single-use functions over polynomial orbit-finite sets

We do not define the single-use functions on all orbit-finite sets, but only a syntactically defined fragment, namely the *polynomial orbit-finite sets*, which are sets that can be generated from 1 and $\mathbb{A}$ using products $\times$ and disjoint unions $+$. Therefore, we will allow orbit-finite sets like $1 + \mathbb{A}^2$, but we will not allow orbit-finite sets like the set of non-repeating pairs $\{(a, b) \mid a \neq b \in \mathbb{A}\}$ or the set of unordered pairs $\{\{a, b\} \mid a \neq b \in \mathbb{A}\}$. It is an open problem to find a satisfactory definition of single-use functions on all orbit-finite sets. (A simple hack is to use a quotienting construction, similar to Section 4, but what we would really like to do is to identify some extra structure in a set, possibly an action of some yet unknown group or semigroup, which enables us to speak about single-use functions.)

Consider two polynomial orbit-finite sets X and Y. To define which functions $X \to Y$ are single-use, we use an inductive definition. We begin with certain functions that are considered single-use, such as the equality test of type $\mathbb{A} \times \mathbb{A} \to 1 + 1$. These functions are called the *prime functions*, and their full list is given in Figure 1. Next, we combine the prime functions into new ones using three combinators. The first, and most important, combinator is function composition. Then, we have two combinators for the two type constructors: if two functions $f_1 : X_1 \to Y_1$ and $f_2 : X_2 \to Y_2$ are single-use, then the same is true for:

$$\underbrace{f_1 \times f_2 : X_1 \times X_2 \to Y_1 \times Y_2}_{(x_1,x_2) \mapsto (f_1(x_1), f_2(x_2))} \qquad \underbrace{f_1 + f_2 : X_1 + X_2 \to Y_1 + Y_2}_{\text{left}(x_1) \mapsto \text{left}(f_1(x_1)) \quad \text{right}(x_2) \mapsto \text{right}(f_2(x_2))}.$$

Crucially, the list of prime single-use functions does not contain the copying function $a \in \mathbb{A} \mapsto (a, a) \in \mathbb{A}^2$. Therefore, an alternative name for the single-use functions is *copyless*. If we added copying, then we would get all finitely supported functions [24, Lemma 23].

▶ **Example 3.1.** Consider the function of type $\mathbb{A}^3 \to \mathbb{A}$ which inputs a triple (a, b, c) of atoms and returns a if c is equal to Mark, and b otherwise. This function is a single-use function. It is obtained by composing the six functions listed below:

Function	Type after function
Append 1.	$\mathbb{A} \times \mathbb{A} \times \mathbb{A} \times 1$
Replace added 1 with Mark using the constant function.	$\mathbb{A} \times \mathbb{A} \times \mathbb{A} \times \mathbb{A}$
Apply the equality test to the last two components.	$\mathbb{A} \times \mathbb{A} \times (1 + 1)$
Distribute.	$\mathbb{A} \times \mathbb{A} \times 1 + \mathbb{A} \times \mathbb{A} \times 1$
Project to first and second components, respectively.	$\mathbb{A} + \mathbb{A}$
Co-diagonal	$\mathbb{A}$

Table 1 The prime single-use functions for polynomial orbit-finite sets X, Y and Z.

Function	Type	Definition
Functions about $\mathbb{A}$		
equality test	$\mathbb{A} \times \mathbb{A} \to 1 + 1$	$a, b \mapsto$ if $a = b$ then true else false
constant a	$1 \to \mathbb{A}$	$x \mapsto a$
identity	$\mathbb{A} \to \mathbb{A}$	$x \mapsto x$
Functions about $\times$		
commutativity of $\times$	$X \times Y \to Y \times X$	$x \times y \mapsto y \times x$
first projection	$X \times Y \to X$	$x \times y \mapsto x$
second projection	$X \times Y \to Y$	$x \times y \mapsto y$
append 1	$X \to X \times 1$	$x \mapsto x \times ()$
associativity of $\times$	$(X \times Y) \times Z \to X \times (Y \times Z)$	$(x \times y) \times z \mapsto x \times (y \times z)$
Functions about $+$		
first co-projection	$X \to X + Y$	$x \mapsto \text{left}(x)$
second co-projection	$Y \to X + Y$	$y \mapsto \text{right}(y)$
co-diagonal	$X + X \to X$	$\begin{cases} \text{left}(x) \mapsto x \\ \text{right}(x) \mapsto x \end{cases}$
commutativity of $+$	$X + Y \to Y + X$	$\begin{cases} \text{left}(x) \mapsto \text{right}(x) \\ \text{right}(y) \mapsto \text{left}(y) \end{cases}$
associativity of $+$	$(X + Y) + Z \to X + (Y + Z)$	$\begin{cases} \text{left}(\text{left}(x)) \mapsto \text{left}(x) \\ \text{left}(\text{right}(y)) \mapsto \text{right}(\text{left}(y)) \\ \text{right}(z) \mapsto \text{right}(\text{right}(z)) \end{cases}$
Distributivity		
$+$ distributes over $\times$	$X \times (Y + Z) \to (X \times Y) + (X \times Z)$	$\begin{cases} x \times (\text{left}(y)) \mapsto \text{left}(x \times y) \\ x \times (\text{right}(z)) \mapsto \text{right}(x \times z) \end{cases}$

To justify this description, one should also show that the six functions are single-use. Appending 1, distributivity and co-diagonal are prime functions. The other three are obtained by combining prime functions using the combinators. For example, the equality test is paired, using the combinator for $\times$, with the identity on the remaining two atoms. ⌟

The design goal of the single-use restriction is to have orbit-finite function spaces. The rough idea is that a single-use function can only use a bounded number of atoms in its source code, which guarantees orbit-finiteness of the function space.

▶ **Example 3.2.** Consider the functions of type $\mathbb{A} \to 1 + 1$, which can be seen as subsets of the atoms, with $1 + 1$ representing the Booleans.

- The finitely supported functions $\mathbb{A} \to 1 + 1$ correspond to the finite or co-finite subsets of $\mathbb{A}$. Therefore, the set of such functions admits an equivariant bijection with a disjoint union of two copies of the finite powerset $\mathsf{P}_{\text{fin}}\mathbb{A}$, in particular it is not orbit-finite.
- There are four possible single-use functions $\mathbb{A} \to 1 + 1$: (a) always return true; (b) always return false; (c) check for equality with some fixed atom a; (d) check for inequality ($\neq$) with some fixed atom a. Therefore, the set of single-use functions admits an equivariant bijection with the orbit-finite set $1 + 1 + \mathbb{A} + \mathbb{A}$. ⌟

The above example shows that the space of single-use functions of some type $X \to Y$ is orbit-finite, and in fact it can be described using a polynomial orbit-finite set. This is true for every choice of polynomial orbit-finite sets X and Y, as proved in [24, Theorem 5], and illustrated in the following example.

▶ **Example 3.3.** Assume that the input type X is some power of the atoms $\mathbb{A}^k$, and the output type Y does not use atoms, e.g. it is $Y = 1 + 1$. The assumption on the input type can be made without loss of generality using distributivity, while the assumption on the output type is a proper restriction, but it will allow us to skip some technical details of the general construction while retaining the important intuitions. We describe below a type that represents all single-use functions from $\mathbb{A}^k$ to Y; we shall denote it by $\mathbb{A}^k \Rightarrow Y$. Note that $\Rightarrow$ is *not* a primitive type constructor in our grammar of types; it is a notation that stands for the inductive construction below.

This type is defined by induction on k. In the base case of $k = 0$ we simply need to give a value from the output type, and therefore $\mathbb{A}^0 \Rightarrow Y$ is the same as Y. Consider now the induction step of $k > 0$. We observe that a single-use function that inputs $\mathbb{A}^k$ must begin with some equality test, and then continue with one of two single-use functions that have fewer arguments (one for the case when the equality test returns true, and one for the other case). This observation leads to the following definition of the type $\mathbb{A}^k \Rightarrow Y$:

$$\underbrace{\coprod_{i \in \{1,\ldots,k\}} \mathbb{A} \times (\mathbb{A}^{k-1} \Rightarrow Y) \times (\mathbb{A}^{k-1} \Rightarrow Y)}_{\substack{\text{starts by comparing } i\text{-th} \\ \text{coordinate to some constant}}} \quad + \quad \underbrace{\coprod_{i,j \in \{1,\ldots,k\}} (\mathbb{A}^{k-2} \to Y) \times (\mathbb{A}^{k-2} \to Y)}_{\substack{\text{starts by comparing the} \\ i\text{-th and } j\text{-th coordinates}}}.$$

Note that the above representation of the function space is not necessarily unique, i.e. the same function can be represented in several different ways. For example, the order in which equality tests are performed will matter for the representation, but might not matter for the function. This is not something that we worry about; it is dealt with in Section 4. ⌟

Unfortunately, the proposal illustrated in Example 3.3 and described in more detail in [24] does not give a satisfactory solution to the problem of function spaces. The problem is that the set of representations $X \Rightarrow Y$ should also support operations on functions. More specifically, we should be able to indicate single-use operations which do the following:

evaluation: a single-use function from $(X \Rightarrow Y) \times X$ to Y which inputs a representation of a function and applies it to an argument;

composition: a single-use function from $(X \Rightarrow Y) \times (Y \Rightarrow Z)$ to $(X \Rightarrow Z)$ which inputs the representations of two functions and returns a representation of their composition;

currying (i.e. partial application): for each single-use function from $X \times Y$ to Z, there should be a single-use function from X to $Y \Rightarrow Z$ which inputs a first argument and returns a representation of the partially applied function.

Only in the presence of all of these operations can we speak of a function space, and the corresponding category can be called closed. (Composition can be obtained through evaluation and currying, so the essential operations are evaluation and currying.) The following example shows that the currying operation is not single-use, and therefore the space of single-use functions as defined in this section is not closed.

▶ **Example 3.4.** Consider the single-use function

$$f : \mathbb{A} \times \mathbb{A} \to 1 + 1 \qquad (a, b) \mapsto \begin{cases} \text{result of test } a = \text{Mark} & \text{if } b = \text{Eve} \\ \text{result of test } a = \text{John} & \text{otherwise.} \end{cases}$$

The currying of this function, is a new function which maps a first argument $a \in \mathbb{A}$ to the partially applied function $f(a, _)$. This currying is

$$a \mapsto \begin{cases} b \mapsto b = \text{Eve} & \text{if } a = \text{Mark} \\ b \mapsto b \neq \text{Eve} & \text{if } a = \text{John} \\ b \mapsto \text{false} & \text{otherwise} \end{cases}$$

Recall that in Example 3.2 we showed that the space of single-use functions of type $\mathbb{A} \to 1+1$ can be represented as $1+1+\mathbb{A}+\mathbb{A}$. If we use this representation, then the currying of the function f is not single-use, because we need to compare the input atom a to two constants, Mark and John. If we use the representation from Example 3.3, then the corresponding type will be $\mathbb{A} \otimes (1+1)^2$, but the problems with currying will persist. ⌟

For similar reasons, the function space we proposed above does not support function composition either; it cannot be used to convert automata into single-use monoids since the resulting monoid would need to use function composition as its monoid operation.[1]

3.2 Linear types and single-use functions on them

To solve the problems above, we introduce a more refined type system, which is based on linear types. The main idea is to pay more attention to types in Example 3.3, where we describe a single-use function by specifying the first equality test that it makes, and then giving two descriptions of the functions that will be used in each of the two possible outcomes of the equality test. The main observation is that these two outcomes are mutually exclusive, and therefore we intend to use only one of the two descriptions. For this reason, we will use a type constructor & that comes from linear logic. The intended meaning is that an object of type $X\&Y$ consists of two objects, but with the ability to use only one of them. Since linear logic uses $\otimes$ for the product that we have so far denoted by $\times$, we will also follow that convention. Using these two products, the appropriate type for Example 3.3 becomes:

$$\coprod_{i \in \{1,\dots,k\}} \mathbb{A} \otimes ((\mathbb{A}^{k-1} \Rightarrow Y)\&(\mathbb{A}^{k-1} \Rightarrow Y)) \quad + \quad \coprod_{i,j \in \{1,\dots,k\}} (\mathbb{A}^{k-2} \to Y)\&(\mathbb{A}^{k-2} \to Y).$$

Under this definition, the problems from Example 3.4 will be solved, at least for the particular type considered in that example. However, by introducing a new type constructor, we will have to redefine the single-use functions, and then we will have to give a representation of functions that allow this new type constructor, without incurring the need to add any other new type constructors. This is what we will do now.

▶ **Definition 3.5** (Linear types). *A* linear type *is any expression constructed from the atomic types* 1 *and* $\mathbb{A}$ *using three*[2] *binary type constructors* $+, \&$ *and* $\otimes$.

---

[1] This problem is solved in [9] and [24] in a different way, namely by showing that every orbit-finite monoid necessarily divides a single-use monoid, using a Krohn-Rhodes construction. However, this construction is difficult and delicate, in particular it does not work for atoms that have more structure than equality alone. In contrast, the proposal that we give in this paper works for other kinds of atoms, as discussed in Section 6.

[2] We set up our type system without using the multiplicative disjunction ⅋ of linear logic. This is a common practice in linear type systems, as they are often based on intuitionistic linear logic, rather than classical linear logic (see e.g. [1]). It is also worth mentioning that our type system is in fact *affine* [13, §1.2.1], as we are going to allow discarding the unused values of $\mathbb{A}$. However, since there is no risk of confusion, we have decided to use the name *linear types* for the sake of simplicity.

In our linear types, it is only the products that are differentiated, while $+$ comes in only one version. Here is the intuitive explanation of the difference between the two kinds of products, following Girard [13, §1.1.2]. Having a pair $x \otimes y$ is like having the ability of using both components x and y. On the other hand, having a pair $x \& y$ is like having the ability to use one of the two components, at our choice, but not both at once. For example, the input type of the equality test will be $\mathbb{A} \otimes \mathbb{A}$ not $\mathbb{A}\&\mathbb{A}$, since the test will need to consume both arguments. This intuition can only go so far; for example, it is not entirely clear what "our choice" means. We revisit this intuition in Section 5, where game semantics will be used to indicate who makes which choices.

We think of each linear type X as representing a set $[\![X]\!]$, as defined below:

$$[\![1]\!] = 1 \quad [\![\mathbb{A}]\!] = \mathbb{A} \quad [\![X + Y]\!] = [\![X]\!] + [\![Y]\!] \quad [\![X \otimes Y]\!] = [\![X\&Y]\!] = [\![X]\!] \times [\![Y]\!].$$

All sets that arise in this way will be polynomial orbit-finite sets. Note that the two kinds of product represent the same set, namely the set of pairs in the usual set-theoretic sense. However, the two type constructors will be different, because different functions will be allowed to operate on them. As the expression goes, "the proof of the pudding is in the eating"; in this case the pudding is the types and the eating is the functions.

As we did in Section 3.1, the single-use functions will be defined in terms of prime functions and combinators. The combinators are the same, except that instead of $f_1 \times f_2$ we now have two ways of pairing functions, using $\otimes$ and $\&$. The prime functions are inherited from the previous system, with $\times$ understood as $\otimes$, together with a few new functions for $\&$, as described in Table 2. This is summarized in the following definition.

▶ **Definition 3.6** (Single-use functions). *The class of single-use functions is the least class of functions with the following properties:*
1. *It contains the functions from Tables 1 and 2, with $\times$ in Table 1 understood as $\otimes$;*
2. *It is closed under composition, as well as under combining functions using $+$, $\otimes$ and $\&$.*

Table 2 Prime single-use functions that involve &.

Function	Type	Definition
diagonal	$X \to X\&X$	$x \mapsto x\&x$
first projection	$X\&Y \to X$	$x\&y \mapsto x$
second projection	$X\&Y \to Y$	$x\&y \mapsto y$
& distributes over $\otimes$	$X \otimes (Y\&Z) \to (X \otimes Y)\&(X \otimes Z)$	$x \otimes (y\&z) \mapsto (x \otimes y)\&(x \otimes z)$
& distributes over $+$	$X + (Y\&Z) \to (X\&Y) + (X\&Z)$	$\begin{cases} x\&\mathrm{left}(y) \mapsto \mathrm{left}(x\&y) \\ x\&\mathrm{right}(z) \mapsto \mathrm{right}(x\&z) \end{cases}$

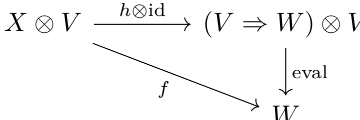
Formally speaking, a single-use function consists of an input linear type X, an output linear type Y, and a function between the sets $[\![X]\!]$ and $[\![Y]\!]$ that is generated using the prime functions and combinators from the above definition. As was the case in Section 3.1, all single-use functions are finitely supported. Therefore, one can think of the single-use functions of type $X \to Y$ as being a subset of the set of all finitely supported functions from $[\![X]\!]$ to $[\![Y]\!]$. This subset is strict: as we will see, the space of single-use functions will be orbit-finite, unlike the space of all finitely supported functions. We will be thinking of the single-use functions as a category.

▶ **Definition 3.7** (Category of single-use sets). *The category of single-use sets is:*
1. *The objects are linear types, as per Definition 3.5.*
2. *The morphisms between types X and Y are single-use functions from $[\![X]\!]$ to $[\![Y]\!]$.*

In the very definition of the above category, there is a faithful functor to the category of orbit-finite sets with finitely supported functions. This functor maps objects X to their underlying sets $[\![X]\!]$, which are orbit-finite sets, and it maps morphisms to the corresponding functions. The functions seen to be finitely supported, and the functor is faithful by definition, since the morphisms in Definition 3.7 are defined to be single-use functions.

The main technical result of this paper is that the category of single-use sets has function spaces, as stated in the following theorem. The appropriate product will be $\otimes$, and not &. Since the Cartesian product in our category is & and not $\otimes$, this means that the result we are targeting is symmetric monoidal closed with respect to $\otimes$, and not Cartesian closed. For now, our theorem stops a bit short of saying that the category is monoidal closed, since several different elements of the function space might represent the same function; but we will come back to this in Section 4.

▶ **Theorem 3.8.** *Let V and W be objects (i.e. linear types). There exists an object, denoted by $V \Rightarrow W$, and a morphism (i.e. a single-use function)* $\text{eval} : (V \Rightarrow W) \otimes V \to W$ *with the following property. For every morphism $f : X \otimes V \to W$ there is a (not necessarily unique) morphism $h : X \to (V \Rightarrow W)$ such that the following diagram commutes:*



The above theorem is the main technical contribution of this paper. The difficulty in its proof is finding a representation of the single-use functions that is rich enough to capture all functions, but simple enough to be described by a linear type (in particular, the corresponding set will be orbit-finite). In Section 3.1, when the types did not have "&", we could pull off a relatively simple construction, which was possible mainly due to the strong distributivity rules that allowed converting each type into a normal like $\mathbb{A}^{n_1} + \cdots + \mathbb{A}^{n_\ell}$. In the presence of "&", the distributivity rules are not as strong, and the way in which a single-use program can interact with its input is rather subtle. Our solution is presented in Section 5.

4 Quotienting by partial equivalence relations

A drawback of Theorem 3.8 is that the function space $V \Rightarrow W$ can contain different representations of the same function; this will mean that currying is not unique. To overcome this issue, we use a simple quotient construction, inspired by a classical technique used in realizability semantics of typed λ-calculi (see e.g. [3, Chapter 15]).

Define a *partial equivalence relation* to be a relation that is symmetric and transitive, but not necessarily reflexive. This is the same as (complete) equivalence relation on some subset, hence we may speak of the quotient of a set by a partial equivalence relation (which is an usual quotient of a subset). We will use a partial equivalence on the function space $X \Rightarrow Y$ to: (1) remove objects that do not represent any morphism; (2) identify two objects if they represent the same morphism. After such a quotient, the function space will have unique representations for functions.

▶ **Definition 4.1.** *The* quotiented single-use category *is:*

- *Objects are pairs (linear type X, equivariant partial equivalence relation on $[\![X]\!]$);*
- *The set of morphisms between objects $(X, \sim_X)$ and $(Y, \sim_Y)$ is the quotient of the set of single-use functions from X to Y by*

$$f \sim g \quad :\Longleftrightarrow \quad \forall x, x' \in [\![X]\!],\ x \sim_X x' \Rightarrow f(x) \sim_Y g(x')$$

We can then define the function space object from $(X, \sim_X)$ to $(Y, \sim_Y)$ as $(X \Rightarrow Y, \sim_{X \Rightarrow Y})$ where two elements of $X \Rightarrow Y$ are related by the partial equivalence relation $\sim_{X \Rightarrow Y}$ when the single-use functions they represent are related by the above-defined $\sim$. The quotiented single-use category is also equipped with a tensor product $\otimes$ on its objects.

▶ **Theorem 4.2.** *The quotiented single-use category, equipped with the tensor product $\otimes$, is a monoidal closed category, i.e. it satisfies the conclusions of Theorem 3.8, but, furthermore, the morphism h is unique.*

5 Game semantics

This section is devoted to the proof of Theorem 3.8. To construct the function space $X \Rightarrow Y$, we use game semantics to identify a certain normal form of programs that compute single-use functions. The presentation in this section is self-contained, and does not assume any knowledge of game semantics. We base our notation on [2].[3]

Let us begin with a brief motivation for why game semantics will be useful.

While it is intuitively clear which functions should be allowed as single-use for simple types such as $\mathbb{A} \to 1 + 1$ or $\mathbb{A} \otimes \mathbb{A} \to \mathbb{A} + \mathbb{A}$, these intuitions start to falter when considering more complex types. How can one show that a function is *not* single-use? If one were to use the definition of single-use functions alone, then one would need to rule out any possibility of constructing the function from the primes using the combinators, including constructions that use composition many times, and with unknown intermediate types.

This is the reason why we consider game semantics. It will allow us to give a more principled description of the intuition that pairs of type $X \otimes Y$ can be used on both coordinates, while pairs of type $X \& Y$ can be used on a chosen coordinate only. The idea behind game semantics is to give the description in terms of an interaction between two players:

1. System, who represents the function (we will identify with this player); and
2. Environment, who supplies inputs and requests outputs of the function.

One of the intuitions behind the setup is that if a type $X \& Y$ appears in the input of the function, then it is the System who can choose to use X or Y, while if the type appears in the output, then it is the Environment who makes the choice. (In this paper, we consider functions of first-order types of the form $X \to Y$, where X and Y are linear types that do not use $\to$, and therefore there will be a clear distinction between input and output values.) Before giving a formal definition of game semantics, we give simple example of the interaction.

▶ **Example 5.1.** Consider the two types $X \otimes (Y \& Z)$ and $(X \otimes Y) \& (X \otimes Z)$. Among the prime functions in Table 2, we find distributivity in the direction $\to$, but not in the direction $\leftarrow$. We explain this asymmetry using the interaction between two players System and Environment[4].

Let us first consider the interaction in the direction $\to$. The player Environment begins by requesting an output. Since this output is of type $(X \otimes Y) \& (X \otimes Z)$, this means that Environment can choose to request either of the two types $X \otimes Y$ and $X \otimes Z$. Suppose that Environment requests $X \otimes Y$. Now System needs to react, and produce two elements: one of type X and one of type Y. Both can be obtained from the input; for the second one player System can choose how to resolve the input $Y \& Z$ to get the appropriate value.

[3] Another standard reference for the category of "simple games" upon which we build is [15]. For a recent survey of modern game semantics, see [10].

[4] Observe that in Table 1 the $(+, \otimes)$-distributivity is also only given in one direction. In that case the inverse is, in fact, a single-use function, as it can be constructed from prime functions.

Consider now the interaction in the opposite direction $\leftarrow$. As we will see, System will be unable to react to the behavior of Environment, which demonstrates that there is no distributivity in this direction. The problem is that Environment can begin by requesting an element of type X, since the output type is $X \otimes (Y \& Z)$, while still reserving the possibility to request $Y \& Z$ in the future (because the tensor product $\otimes$ means that both output values need to be produced). To produce this element, System will need choose one of the two coordinates in the input type $(X \otimes Y) \& (X \otimes Z)$, and any of these two choices will be premature, since Environment can then request the opposite choice in the output type. ⌟

As illustrated in the above example, we will use a game to describe the possible behaviours of a function $X \to Y$. The game will be played in an arena, arising from the linear types X and Y, which will tell us what moves are possible for the two players. In each arena, we will be interested in strategies for System, telling us how System should react to Environment's moves. Morally, such a strategy will say how the function reacts to requests in the output type Y and values in the input type X.

Proof scheme for Theorem 3.8. In the remainder of this section, we define the arenas and strategies of our game semantics. Our main Theorem 3.8 will then follow from the two key properties below, that we establish in the technical appendix of the full version.

Representation of single-use functions by strategies: To a strategy in the arena for $X \to Y$, we will assign a single-use function of type $X \to Y$ that it represents. This mapping will be partial, i.e. some ill-behaved strategies will not represent any functions. We will show that the set of strategies in the arena is large enough to represent all single-use functions.

Representation of strategies by a linear type: We shall build a linear type $X \Rightarrow Y$ which can represent all well-formed strategies on the arena for $X \to Y$. In particular, this implies that such strategies form an orbit-finite set. Furthermore, this linear type $X \Rightarrow Y$ will be equipped with single-use evaluation and currying functions, as required.

5.1 Arenas and strategies without constants and equality tests

We begin with a simpler version of the game semantics, in which the arenas and strategies will describe functions that are not allowed to perform equality tests, nor to use constants. These strategies will model functions such as the identity function $\mathbb{A} \to \mathbb{A}$, which directly passes its input to its output, but they will not model the equality test $\mathbb{A} \otimes \mathbb{A} \to 1 + 1$, or the constant functions of type $1 \to \mathbb{A}$. The general idea is to use standard game semantics for linear logic, with an extra feature that we call *register operations*. The register operations will be used to model the way in which atoms are passed from the input to output. For example, in the identity function, Environment will write the input atom into the register, and then System will read the output atom from that register. The following definition of an arena is based on the definition from [2, p. 4], slightly adapted for the context of this paper:

▶ **Definition 5.2** (Arena). *An* arena *consists of:*

1. *A set of* moves*, with each move having an assigned owner, who is either "System" or "Environment", and one of three register operations, which are "none", "read", or "write".*
2. *A set of plays, which a set of finite sequences of moves that is closed under prefixes, and such that in every play, the owner of the first move is Environment, and then the owners alternate between the two players.*

▶ Remark 5.3. In all arenas that we consider, the "read" moves will be owned by System and the "write" moves will be owned by Environment. Therefore, we could simplify the register operations and have just one, called "read/write", whose status is determined by its owner.

In this paper, all the strategies that we consider shall be for player System.

▶ **Definition 5.4.** *A* strategy *in an arena is a subset of plays in the arena, which:*
1. *is closed under prefixes;*
2. *if the strategy contains a play p that ends with a move owned by System, then it also contains all possible plays in the arena that extend p with one move of Environment;*
3. *if the strategy contains a play p that ends with a move owned by Environment, then it contains exactly one play in the arena that extends p with one move of System;*
4. *there is some k such that all plays in the strategy have length at most k;*
5. *every "read" move is directly preceded by a "write" move (in particular a play cannot begin with "read"), and every "write" move is either the last move, or directly succeeded by a "read" move.*

Conditions 2 and 3, which are standard in game semantics (cf. [2, p. 5]), guarantee that the strategy is deterministic and only "ends" when Environment has no moves to play. Let us now comment on the last two conditions, which are not standard.
- The fourth condition is motivated by the idea that we study "finite" types, and there will be no need for unbounded computations[5]
- The last condition will be called the *immediate read condition.* It ensures that there is matching between "read" and "write" moves in plays that do not end with write. Since "write" will always be owned by Environment, the immediate read condition will ensure a matching between "write" and "read" moves.

We now show how to associate to each linear type a corresponding arena, and also how to associate an arena to a function type $X \to Y$ (which is not a linear type in the sense of Definition 3.5) between linear types X and Y. The arenas that we construct so far will not be our final proposal, since the corresponding strategies will not be able to use constants or perform equality tests; this will be fixed in Section 5.2. Before giving a formal definition, we begin with a simple example.

▶ **Example 5.5.** We define an arena for $\mathbb{A} \to \mathbb{A}$. It will be rather impoverished, since the only allowed strategy in it will correspond to the identity function. However, this is consistent with the stage that we are at, where we only consider functions that do not use constants or perform equality tests; for such functions of type $\mathbb{A} \to \mathbb{A}$ the only possibility is the identity.

The arena describes the following interaction between the two players: Environment requests an output, then System requests an input, then Environment grants the input, and finally System grants the output by forwarding the input that was granted. It has 4 moves:

move	owner	register operation
request output	Environment	none
request input	System	none
grant input	Environment	write
grant output	System	read

The plays are defined as all sequences that begin with a move of Environment, alternate between players, use each move only once, and have the following condition: "grant output" can only be played after "request output", and likewise for "grant input" and "request input".

[5] This condition is only meaningful, if there are arenas that admit plays of unbounded length. Since, in this section, all the arenas will be finite this condition will be vacuously true. It will only become relevant in Section 5.2, once we introduce arenas with constants and equality tests.

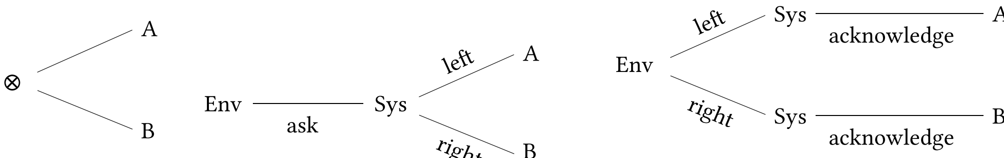


Figure 1 Pictures of the operations $A \otimes B$, $A + B$, $A \& B$ and $A \otimes B$ respectively on arenas. The picture for $\otimes$ is less useful than the next two, since the root node of the tree is not a player, but a node labelled by $\otimes$. The intuition is that the game is played in parallel on both arenas, and therefore a position in it can be visualized as a pair of positions in the two arenas.

A quick inspection of this definition reveals that there is a unique maximal play, where the moves are played in the order from the table, and all other plays are prefixes of this maximal play. Because of the uniqueness of responses, the set of plays is also a strategy. As mentioned at the beginning of this example, the strategy describes the identity function. ⌟

We hope that the above example explains some intuitions about how arenas describe types and strategies describe functions. We now give a formal definition which is compositional: we define arenas for the basic types 1 and $\mathbb{A}$, and then we define operations on arenas that correspond to the type constructors $+$, $\&$, and $\otimes$. We begin with the basic types.

▶ **Definition 5.6** (Arenas for 1 and $\mathbb{A}$)**.**

1. *The arena for the type* 1 *is empty: there are no moves, the only play is the empty sequence.*
2. *The arena for type* $\mathbb{A}$ *has two moves, which must be played one after the other: first player Environment makes a move called "request" that has no register operation, and then player System responds with a move called "grant" that has register operation "read".*

In the above definition, we only described the behaviour of $\mathbb{A}$ when viewed as an output type. To get the input type, where the players are swapped and read is swapped with write, we will use duality, which is another operation on arenas. This operation is defined below, together with other operations that correspond to the type constructors.

▶ **Definition 5.7.** *Let A and B be arenas. We define the following arenas (see also Figure 1):*

$\bar{A}$ *This is called the* dual arena *of A. The moves and plays are the same as in A, except the owners are swapped, and the "read" and "write" register operations are swapped.*

$A + B$ *The moves in this arena are the disjoint union of the moves of A and B, with inherited owners and register operations, plus three extra moves: "ask" owned by Environment, and "left", "right" owned by System. The plays are defined as follows. Player Environment begins with an ask move, then System responds with a left or right move, and the remaining sequence is a play in the arena A or B, depending on whether System chose left or right.*

$A \& B$ *The moves in this arena are the disjoint union of the moves of A and B, with inherited owners and register operations, plus three extra moves: "acknowledge" owned by System, and "left", "right" owned by Environment. The plays are defined as follows. Environment begins by choosing left or right, then System responds with an acknowledge move, and the remaining sequence is a play in either A or B, depending on Environment's choice. (This construction differs slightly from the one from [2, Exercise 1.10] – this is because we want to keep it analogous to the construction for $A + B$.)*

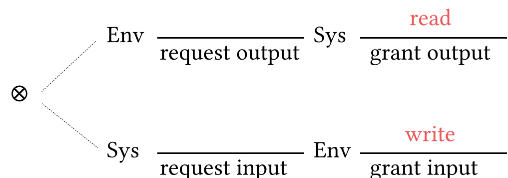
$A \otimes B$ *The moves in this arena are the disjoint union of the moves of A and B, with inherited owners and register operations. A play in this arena is any shuffle of plays in the two arenas A and B. (A shuffle of two words is any word obtained by interleaving them, e.g. shuffles of "abc" and "123" include "a1b23c" and "12ab3c"). By Definition 5.6, we require that the owners of the move alternate in the interleaved sequences. See [2, p.7].*

Equipped with the above definitions, we present our first attempt at assigning arenas to types. In the second item of the following definition, we use the name *library-less*, because our second and final definition of the arena for a function type, as presented in the next section, will be equipped with an extra feature that will be called a library.

▶ **Definition 5.8.** *Let X and Y be linear types.*

- *The* arena for X *is defined by inductively applying the constructions from Definition 5.6 and Definition 5.7 according to the structure of the type.*
- *The* library-less arena for $X \to Y$ *is defined to be (dual of arena of X) $\otimes$ (arena of Y).*

▶ **Example 5.9.** The library-less arena for the identity type $\mathbb{A} \to \mathbb{A}$ is the same as the arena that we explicitly defined in Example 5.5. It can be drawn as the following picture:



As discussed previously, our notion of arenas does not yet take into account some structure on the atoms. This will be fixed in the next section, by modifying the second item in Definition 5.8. On the other hand, the arenas from the first item, for linear types without function types, are already in their final form.

In principle the construction from the second item in Definition 5.8 can be nested, and thus used to assign arenas to higher-order types that can nest $\to$ with the other type constructors. This is how it is usually done in linear logic. However, the construction that we will describe in the next section will be less amenable to nesting, and will use it only to describe functions between types that do not use $\to$.

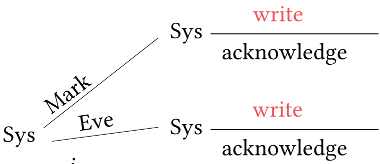
5.2 Arenas and strategies with constants and equality tests

In Section 5.1, we described arenas for functions that did not use the structure of the atoms, i.e. constants and equality tests. We now show how these arenas can be extended to cover this structure. The general idea is to equip the arenas with an extra part, which we call the *library*, that describes the allowed operations on the atoms. (The library as we present it here only contains functions for equality and constants, but in the proof of Theorem 6.2, we will use a library that has other relations beyond equality.)

▶ **Definition 5.10.** *The* library arena *and its parts are defined as follows (cf. Figure 2).*

1. *The* constant choice arena *is the following arena* $\mathbb{A} + 1$ *moves: first player System chooses an atom, then player Environment plays move with register operation "write".*
2. *The* equality test arena *is an arena which the plays are as follows:*
   a. *first System plays a move with register operation "read";*
   b. *then Environment plays an move with no register operation;*
   c. *then System plays a move with register operation "read";*
   d. *then Environment plays one of two moves, called $=$ and $\neq$, with no register operation.*
3. *The* library arena *is obtained by applying $\otimes$ to infinitely[6] many copies of the constant choice arena and infinitely many copies of the equality test arena.*

The constant choice arena:



The equality test arena:

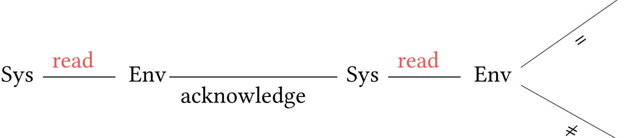


Figure 2 Pictures of the library arenas. We use the convention that the register operations are in red, and the names of the moves, which have no other role than to distinguish them, are in black. Note that the first move in this arena is owned by System, and we assume in Definition 5.2 that the first move is owned by Environment. This is because this arena, like all arenas in Definition 5.10, is not intended to be a stand-alone arena, but only as part of the bigger arena from Definition 5.11 where the first move is indeed owned by Environment.

The library arena is infinite. Taking the tensor product of infinitely many copies of the two arenas ensures that the library arena satisfies the following property, which corresponds to the ! operation from linear logic:

$$\text{library arena} \quad \equiv \quad (\text{constant choice arena}) \otimes (\text{equality test arena}) \otimes (\text{library arena}). \tag{1}$$

In the above, $\equiv$ refers to isomorphism of arenas, which is defined in the natural way: this is a bijection between the moves, which is consistent with the owners, register operations and plays in the expected way. Another property is that the library arena is isomorphic to a tensor product of itself:

$$\text{library arena} \quad \equiv \quad \text{library arena} \otimes \text{library arena}. \tag{2}$$

We are now ready to give the final definition of arenas for functions between linear types, which takes into account the structure of the atoms.

▶ **Definition 5.11** (Function arena). *For linear types X and Y, the arena of $X \to Y$ is*

$$\textit{(library arena)} \otimes \textit{(dual arena of } X\textit{)} \otimes \textit{(arena of } Y\textit{)}.$$

This completes the game semantics of linear types and functions between them. We do not intend to give game semantics for higher-order types, such as functions on functions etc. As a result, we will only be using the dual once, namely for the arena of the input type. Also, note that the read/write operations will be used in a restricted way, as announced in Remark 5.3,

[6] Observe that the shuffle operation from Definition 5.7 can also be used for infinite families of arenas.

namely that the "read" moves will be owned by System and the "write" moves will be owned by Environment. This is because the library arena has this property, the arena for $\mathbb{A}$ has this property, and all operations on arenas that we have defined preserve this property.

As said before, the rest of the proof of Theorem 3.8 takes place in the technical appendix.

6 Beyond equality

So far, we have studied the case when the atoms are equipped with equality only. Consider now a more general case: let $\mathbb{A}$ be any relational structure, i.e. any set equipped with relations (but not functions). For example, we could use the rational numbers with their linear order. Another example would be Presburger arithmetic, i.e. $(\mathbb{N}, +)$. Since we want to have relations only, we think of addition as a ternary relation $x + y = z$. The construction from Section 3 of the single-use category can be generalized to this case; and not only is the definition of the category generic, but the same is true for the proof that function spaces exist.

▶ **Definition 6.1** (Single-use functions over a relational structure)**.** *Let $\mathbb{A}$ be a relational structure. The* single-use category over $\mathbb{A}$ *is defined in the same way as in Definition 3.7, except that the list of prime functions is extended with one prime function $\mathbb{A}^n \to 1 + 1$ for every n-ary relation in the vocabulary of $\mathbb{A}$. (Here, the power $\mathbb{A}^n$ uses $\otimes$.)*

▶ **Theorem 6.2.** *Consider a relational structure $\mathbb{A}$, in which for every $k \in \{0, 1, \ldots\}$ there are finitely many relations of arity k. Then the single-use category over $\mathbb{A}$ satisfies the weak universal property stated in Theorem 3.8.*

In the above statement, "weak" alludes to the non-uniqueness of currying; we could again get a symmetric monoidal closed category by performing the quotient construction of Section 4.

The theorem is proved in the same way as Theorem 3.8. The assumption on the vocabulary is used to ensure that in the corresponding game semantics (cf. Section 5), there are finitely available moves in any given moment, because the vocabulary can only be queried up to the maximal number of atoms in the input, due to the single-use restriction.

Note that this theorem can be applied to any relational structure, including undecidable ones. Clearly, there must be some benefit from assuming that the structure has a decidable first-order theory, which means that there is an algorithm which checks if a first-order sentence is true in the structure. The benefit is that we can check if two morphisms are equal, as expressed in the following theorem, whose assumption applies to structures such as Presburger arithmetic, or the real field $(\mathbb{R}, +, \cdot, \leq)$.

▶ **Theorem 6.3.** *Consider a relational structure $\mathbb{A}$, in which for every $k \in \{0, 1, \ldots\}$ there are finitely many relations in the vocabulary that have arity k. If this structure has a decidable first-order theory, then there is an algorithm for the following problem:*
Input: *Two morphisms, described by expressions using prime functions and combinators.*
Question: *Are they the same morphism?*

7 Further topics

Two-way automata. Theorem 6.3 gives us a reasonable category of single-use functions over a relational structure with a decidable first-order theory. However, the latter property is not the only one needed to ensure that the category is appropriate to automata. If we want to model automata and their decision procedures, we will also need to execute certain fixpoint algorithms, as explained in [5]. In order to allow it, we assume that $\mathbb{A}$ is an *oligomorphic* (see

[5, Definition 3.9]) structure. It is a standard assumption in the theory of sets with atoms, used to ensure that the notion of orbit-finite set is meaningful. Examples of oligomorphic structures include the atoms with equality only, the rational numbers with their linear order, and the Rado graph. Non-examples include Presburger arithmetic and the real field.

▶ **Theorem 7.1.** *Let* $\mathbb{A}$ *be an* oligomorphic *structure with a decidable first-order theory. Then the emptiness problem is decidable for single-use deterministic two-way automata over* $\mathbb{A}$*.*

In the proof of this theorem, we show that the single-use category over an oligomorphic structure $\mathbb{A}$ is *traced* with respect to the coproduct $+$, and use this to model deterministic two-way automata over $\mathbb{A}$. The idea that traced monoidal categories are a natural setting for two-way automata comes from [14].

Orbit-finite dimensional vector spaces. An alternative solution for the problem of function spaces – our central motivation in this paper – is to use vector spaces of orbit-finite dimension. The technical tools for this were developed already in [8]; our contribution here is mainly one of perspective, namely framing it as a symmetric monoidal closed category.

▶ **Theorem 7.2.** *The category of orbit-finitely spanned vector spaces [8, Section VI] is symmetric monoidal closed, with respect to the tensor product.*

▶ Remark 7.3. A similar result was observed in [22, Theorem 3.8], but using the smaller category of vector spaces that admit an orbit-finite basis.

An advantage of the vector space category is its simplicity, and the fact that it is “bigger” in the following sense: The two solutions for function spaces discussed in this paper, namely the single-use solution and the vector space solution, sit on both sides of the classical category of orbit-finite sets, as witnessed by two faithful functors, one from the single-use category to the orbit-finite category, and one from the orbit-finite category to the vector space category. However, there are two limitations of the orbit-finitely spanned vector spaces.

First, the existence of function spaces is dependent on the choice of atoms. Theorem 7.2 works when the atoms have equality only, and it also works when the atoms are equipped with a total order. This is because the dual spaces are orbit-finitely spanned in these cases, as proved in [8, Corollary VI.5]. However, this is no longer the case for other choices of atoms, such as the Rado graph, see [8, Example 9]. This is in contrast to the single-use category, where the existence of function spaces is independent of the choice of atoms.

A second limitation is that unlike the single-use category (cf. Theorem 7.1), the vector space category does not support two-way automata. This is because this category generalizes the orbit-finite category (i.e. it admits a faithful functor from it), and in the orbit-finite category emptiness is undecidable for deterministic two-way automata [17, Theorem 5.3]. This precludes the kind of traced construction that we did in the single-use category. This issue appears already without atoms: the category of finite-dimensional vector spaces is not traced with respect to the sum $\oplus$ of vector spaces.

Related work: categories and λ-calculus. There have been several works using category theory to generalize classical operations on automata, such as the coalgebraic “generalized powerset construction” [23]. The closest to the philosophy that this paper might be the work of Colcombet and Petrişan [11]: it introduces a setting where automata over different categories may be studied and compared (see e.g. [4] for applications). Within this setting, Pradic and the second author have investigated some properties of automata over symmetric monoidal closed categories [18, Sections 1.2.3 and 4.7–4.8].

The latter emerged as part of their research on "implicit automata" [19, 18, 21], which is about relating the expressive power of automata and typed λ-calculi. In [18, Chapter 4], a monoidal closed category of single-use assignments on string-valued registers is built and used to relate a register-based string transducer model to a λ-calculus with linear types. Indeed, symmetric monoidal closed categories are famous for providing denotational semantics for the linear λ-calculus. Similarly, our results here could serve to characterize the languages of words with atoms studied by the first and third author in [9] via some typed λ-calculus.

Conversely, our Theorem 3.8 might also be provable by representing single-use functions as λ-terms (or programs in some richly structured syntactic formalism) instead of strategies over games. Indeed, it is a classical fact that a simple type is inhabited by finitely many linear λ-terms up to β-conversion (when there are no primitive constants), and variations on this fact have been used in the literature to relate automata and λ-calculus [19, 12].

References

1 Samson Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111(1&2):3–57, 1993. doi:10.1016/0304-3975(93)90181-R.

2 Samson Abramsky. Semantics of interaction. In Peter Dybjer and Andrew Pitts, editors, *Proceedings of the 1996 CLiCS Summer School, Isaac Newton Institute*. Cambridge University Press, 1997. arXiv:1312.0121.

3 Roberto M. Amadio and Pierre-Louis Curien. *Domains and Lambda-Calculi*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1998. doi:10.1017/CBO9780511983504.

4 Quentin Aristote. Active learning of deterministic transducers with outputs in arbitrary monoids. In Aniello Murano and Alexandra Silva, editors, *32nd EACSL Annual Conference on Computer Science Logic, CSL 2024, February 19-23, 2024, Naples, Italy*, volume 288 of *LIPIcs*, pages 11:1–11:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPICS.CSL.2024.11.

5 Mikołaj Bojańczyk. Slightly infinite sets. URL: https://www.mimuw.edu.pl/~bojan/paper/atom-book.

6 Mikołaj Bojańczyk. Nominal Monoids. *Theory of Computing Systems*, 53(2):194–222, 2013. doi:10.1007/S00224-013-9464-1.

7 Mikołaj Bojańczyk, Bartek Klin, and Slawomir Lasota. Automata theory in nominal sets. *Logical Methods in Computer Science*, 10(3), 2014. doi:10.2168/LMCS-10(3:4)2014.

8 Mikołaj Bojańczyk, Bartek Klin, and Joshua Moerman. Orbit-finite-dimensional vector spaces and weighted register automata. In *36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 - July 2, 2021*, pages 1–13. IEEE, 2021. doi:10.1109/LICS52264.2021.9470634.

9 Mikołaj Bojańczyk and Rafał Stefański. Single-Use Automata and Transducers for Infinite Alphabets. In Artur Czumaj, Anuj Dawar, and Emanuela Merelli, editors, *International Colloquium on Automata, Languages, and Programming (ICALP 2020)*, volume 168 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 113:1–113:14, Dagstuhl, Germany, 2020. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ICALP.2020.113.

10 Pierre Clairambault. *Causal Investigations in Interactive Semantics*. Habilitation à diriger des recherches, Université Aix-Marseille, February 2024. URL: https://theses.hal.science/tel-04523273.

11 Thomas Colcombet and Daniela Petrişan. Automata Minimization: a Functorial Approach. *Logical Methods in Computer Science*, 16(1), March 2020. doi:10.23638/LMCS-16(1:32)2020.

12 Paul Gallot, Aurélien Lemay, and Sylvain Salvati. Linear high-order deterministic tree transducers with regular look-ahead. In Javier Esparza and Daniel Král', editors, *45th International Symposium on Mathematical Foundations of Computer Science, MFCS 2020, August 24-28, 2020, Prague, Czech Republic*, volume 170 of *LIPIcs*, pages 38:1–38:13. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.MFCS.2020.38.

13 Jean-Yves Girard. Linear logic: its syntax and semantics. In Jean-Yves Girard, Yves Lafont, and Laurent Regnier, editors, *Advances in Linear Logic*, volume 222 of *London Mathematical Society Lecture Notes*, pages 1–42. Cambridge University Press, 1995. doi:10.1017/CBO9780511629150.002.

14 Peter Hines. A categorical framework for finite state machines. *Mathematical Structures in Computer Science*, 13(3):451–480, 2003. doi:10.1017/S0960129503003931.

15 Martin Hyland. Game semantics. In Andrew M. Pitts and P. Dybjer, editors, *Semantics and Logics of Computation*, Publications of the Newton Institute, pages 131–184. Cambridge University Press, 1997. doi:10.1017/CBO9780511526619.005.

16 Michael Kaminski and Nissim Francez. Finite-Memory Automata. *Theoretical Computer Science*, 134(2):329–363, 1994. doi:10.1016/0304-3975(94)90242-9.

17 Frank Neven, Thomas Schwentick, and Victor Vianu. Finite state machines for strings over infinite alphabets. *ACM Transactions on Computational Logic*, 5(3):403–435, 2004. doi:10.1145/1013560.1013562.

18 Lê Thành Dũng Nguyễn. *Implicit automata in linear logic and categorical transducer theory*. PhD thesis, Université Paris XIII (Sorbonne Paris Nord), December 2021. URL: https://theses.hal.science/tel-04132636.

19 Lê Thành Dũng Nguyễn and Cécilia Pradic. Implicit automata in typed λ-calculi I: aperiodicity in a non-commutative logic. In Artur Czumaj, Anuj Dawar, and Emanuela Merelli, editors, *47th International Colloquium on Automata, Languages, and Programming, ICALP 2020, July 8-11, 2020, Saarbrücken, Germany (Virtual Conference)*, volume 168 of *LIPIcs*, pages 135:1–135:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.ICALP.2020.135.

20 Andrew M. Pitts. *Nominal Sets: Names and Symmetry in Computer Science*, volume 57 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2013. doi:10.1017/CBO9781139084673.

21 Cécilia Pradic and Ian Price. Implicit automata in λ-calculi iii: affine planar string-to-string functions, 2024. arXiv:2404.03985.

22 Michał R. Przybyłek. A note on stone-Čech compactification in zfa, 2024. arXiv:2304.09986.

23 Alexandra Silva, Filippo Bonchi, Marcello M. Bonsangue, and Jan J. M. M. Rutten. Generalizing determinization from automata to coalgebras. *Logical Methods in Computer Science*, 9(1), 2013. doi:10.2168/LMCS-9(1:9)2013.

24 Rafał Stefański. *The single-use restriction for register automata and transducers over infinite alphabets*. PhD thesis, University of Warsaw, 2023.