Praca Magisterska

Teoriomnogościowe Podejście Do Typów, Formalizacja Typów Za Pomocą Automatów Oraz Nowe Konstrukcje W Typach

Klara Zielińska

Uniwersytet Wrocławski, Wydział Matematyki i Informatyki

15 września 2014*

Promotor: prof. Witold Charatonik

* Poprawki: 9 lutego 2015

Streszczenie

W pracy prezentujemy podejście do typów, w którym identyfikujemy typy nie z termami, jak to się robi zwykle, ale ze zbiorami wartości. Pozwala to na pewne ujednoznacznienie typów i odzyskanie pewnej kongruencji między typami, a ich semantyką, która zostaje złamana w pewnych systemach typów — np. w systemach z typami rekurencyjnymi. Ponadto pokazujemy, jak użyć pewnego wariantu automatów skończonych jako reprezentacji typów, co w naturalny sposób pozwala na implementację systemów typów w komputerach. W szczególności używając tej reprezentacji pokazujemy, jak obliczać podtypowanie dla pewnych (możliwe, że nowych) konstrukcji w typach — to jest teoriomnogościowych sum i typów rekurencyjnych z operatorami μ, ν. Poza tymi konstrukcjami pokazujemy także inne podstawowe konstrukcje w typach osadzone w naszym podejściu.

Poza głównym tematem pracy opisujemy też sposób na sprawdzanie inkluzji pomiędzy językami akceptowanymi przez słabe automaty Büchiego na drzewach – który ma szansę być wydajniejszy niż konstrukcje opisywane dla ogólniejszych automatów – oraz poprawkę pewnej konstrukcji dla automatów opisanej przez Damiana Niwińskiego w „Fixed Point Characterization of Infinite Behavior of Finite-State Systems” w 1997.

Master's Thesis

Set Approach To Types, Automata Formalization Of Types And New Type Constructions

Klara Zielińska

Wrocław University, Department of Mathematics and Computer Science

15th September 2014*

Supervisor: prof. Witold Charatonik

Correction: 9th February 2015

Abstract

In this paper we present a view on types, where we identify types not with terms, as usual, but with sets of values. This leads to some disambiguation of types and it regains some congruence between types and their semantics, that can be seen lost in some type systems; like in the case of recursive types. Moreover we put some finite automata as a possible representation of types, what gives a natural way for computational reasoning about type systems. Particularly using this representation we show how to compute subtyping for two possibly-new type constructions — set-like unions and μ, ν recursive types. Besides of this we also put other basic type constructions in the proposed setting for completeness.

Apart from the main course we give some description of a hopefully-efficient way to calculate inclusion-testing between weak Büchi automata on trees and a correction of some automata constructions given in "Fixed Point Characterization of Infinite Behavior of Finite-State Systems" by Damian Niwiński in 1997.

Contents

1 Introduction **6**

2 Preliminaries **8**

2.1 Accessible pointed graphs 8

2.2 Automata 9

2.2.1 Non-deterministic automata (minimal, maximal, Büchi, Rabin) . . 10

2.2.2 Cascading automata 12

2.2.3 Alternating automata 12

2.2.4 Remark about automata inclusion 13

3 Types **14**

3.1 General terms 14

3.2 General formalization concepts 15

3.3 Example type constructions 17

3.3.1 How do we give constructions 17

3.3.2 Base types 19

3.3.3 Pairs 19

3.3.4 Untagged unions 20

3.3.5 Tagged unions 21

3.3.6 Recursive types . . . 21
3.3.7 Intersection types . . . 25
3.3.8 Inductive and coinductive types . . . 27
3.3.9 Towards functions . . . 32

A Constructing algorithm for weak Büchi non-deterministic automata inclusion checking **36**
A.1 Intersection . . . 36
A.2 Complementation . . . 36
A.3 Alternation removing . . . 37
A.4 Checking emptiness . . . 39
A.5 Checking inclusion . . . 42

B What is wrong with the automata for fixed points in the paper [Niw97] **44**

Bibliography **45**

Notation

ε – empty sequence

$x_1 \dots x_k$ – finite sequence; equivalent of $\{x_i\}_{i=1}^{k}$

$x_1 \dots x_k \dots$ – infinite sequence; equivalent of $\{x_i\}_{i \in \mathbb{N} \setminus \{0\}}$

$|w|$ – length of sequence w; for infinite w it is fixed as ω

$w|_k$ – prefix of k first elements of the sequence w, where w may be finite or infinite — undefined if w has less then k elements

A^* – set of finite sequences of A elements

$A^{*|\omega}$ – set of finite and infinite sequences of A elements; equivalent of $A^* \cup A^\omega$

$[n]$ – the set $\{0, \dots, n-1\}$

$Dom\,(f)$ – the domain of function f

$f|_A$ – function f restricted to the domain A

$f\,[x \mapsto y]$ – a substitution in function f that results in the function $f \setminus \{\, (x,z) \in f \,\} \cup \{(x,\, y)\}$

$A \hookrightarrow B$ – the set of partial functions from A to B

Chapter 1

Introduction

Types are sets. This claim seems correct from semantical point of view. Types generally describe classes of values with some given properties, where the universe of these values is mostly a set. Still we rather identify types with terms, then with sets, where additional relation (usually denoted by :) determines the mentioned classes for each such term. This approach may lead to some ambiguity, as many term-types can represent a semantically equivalent entity. For example two recursive types NatList1 = Nat | Nat ∗ NatList1 and NatList2 = Nat | Nat ∗ (Nat | Nat ∗ NatList2) commonly describe exactly the same set of values — which may be seen as a set of values that represent non-empty sequences of natural numbers. This ambiguity is generally solved by setting types as possibly-infinite terms (represented by possibly-infinite trees, cf. [Pc] Chapter 21) and setting the prior terms as denotations of these. Still this solution is partial. Let's, for example, take a look at the following. If we depict types as infinite terms, there is only one type that satisfies the equality NatList = Nat | Nat ∗ NatList (cf. fig. 1.1). But when we interpret the right side as the set of values that belongs to the class represented by the term Nat | Nat ∗ NatList with respect to the mentioned relation (:) and a valuation of the variable NatList, then there can be more solutions of the equation. For example if we have values that represent finite and infinite sequences of naturals in our universe (e.g., like in lazy programming languages), then the set of all finite sequences of natural numbers is, as well, a solution of this, as the set of all finite and infinite sequences of natural numbers. So in other words there is incongruity between semantical and syntactical interpretations of recursive types definitions in some universes.

However problems with term-like types do not concern only recursive types. For example if we would like to have union types in a type system that act as set unions, we quickly face the earlier problem of ambiguity. Specifically, if we make the type Nat | Bool to describe the class of all natural numbers and boolean values (where booleans are disjoint with naturals), then it should be semantically equivalent to Bool | Nat. But when types are interpreted as terms, the types are different. This, of course, can be dealt by setting types as classes of equivalence on the terms, still these classes are isomorphic to the sets of values represented by these terms, which we mentioned at the beginning. So why do not put types just as these sets?

Figure 1.1.: The infinite term-like-type that solves NatList = Nat | Nat ∗ NatList

In this paper we propose to define types as sets of values within some value universe, which is a set. This solves the issue of ambiguity and it probably follows intuitions about types. However it also makes types syntax-independent and so it allows for freedom in representation. Thus we can represent types by terms as previously for readability, but we can also have other representations for other purposes. Particularly, in the sequel we show how to put automata as type representations, what allows to compute checking of subtyping for types that are usually out of range of current type systems — if the reader is unfamiliar with subtyping, this can be described in our setting by inclusion between types and it is a crucial concept for practical usage of types.

Two new — or at least not common — concepts that become available to computer systems due to the usage of automata will be introduced in this paper. These are set-like union types that we mentioned above, and μ, ν (least, greatest) re-

cursive types. The first are called untagged unions, as opposed to tagged unions that are generally used nowadays in strongly-typed programming languages. They allow to define values in much briefer and more natural way that is known from dynamically-typed programming languages or just from pure mathematics. But except comfort this also allows for attempts to static analysis of dynamically-typed programs. The second type concept make it possible to distinguish finite and infinite values in types. So we can, for example, guarantee sequences of natural numbers that we mentioned earlier to be finite or infinite in some place of a program. This can be useful in existing lazy languages for better static analysis, but it also legitimates introducing some infinite values in strongly-typed eager programming languages, what would be unsafe without warranties on finiteness of values, as we would be able to easily fall in an infinite computation sequence.

Example 1.1. Here we give two definitions of the same function in two different type systems. In the first case we use a type system with tagged unions for type-checking, which enforces additional operators for allowing the definition. In the second case untagged unions are used.

The tagged unions case:

```
fn n  => if n = 0 then left (true)
         else if n = 1 then right (left (1))
         else right (right (1.5))
```

The untagged unions case:

```
fn n  => if n = 0 then true
         else if n = 1 then 1
         else 1.5
```

Example 1.2. The infinite sequence of 1s may be defined in an eager programming language by an expression like $\mathsf{let\ rec\ x} = (1, \mathsf{x})$ and it can be typed with $\nu x.\mathsf{Nat} * x$, that denotes the greatest type that satisfies the equality $x = \mathsf{Nat} * x$. One may note that such a construction allows for circular values known in languages with side effects in languages without them.

Apart from these two possibly-new in strongly-typed programming languages type concepts we also give a brief description of other fundamental type constructions arranged to the proposed type approach. Particularly, we show how to put these constructions in automata setting and check subtyping with them.

In the paper we will not be concerned about checking of membership in types from the practical point of view. Theoretically this problem is straightforward — we can just run automata on tree-like representations of values. Still this, of course, is useless for computer implementations in the case where values may be infinite. A workaround may be here to put additional finite representations of values and emulate automata runs on values or just to check these finite representations against term representations of types in algorithmic way. This issue, however, is much simpler then checking subtyping and we will leave it as some conceptual riddle.

Chapter 2

Preliminaries

In this chapter we describe basics for later use. While this can be moderately interesting, we will try to make it moderately minimalistic without much description.

2.1 Accessible pointed graphs

Due to some strange impulses for accessible pointed graphs (APGs) — which we explain later — we will try to set some general terms about them.

An APG is a directed graph with a distinguished (pointed) node v such that any node in the graph may be accessed from v by the edges. Intuitively, we can see these graphs as some equivalent of possibly-infinite trees. For an APG such a tree can be retrieved by setting a node for each finite path in it starting from the pointed node — assuming that the empty path is also valid. Then an edge from one such node to another is set in the tree if and only if the path corresponding to the former is a prefix of the path corresponding to the latter shorter by 1. The root of the tree is then the node corresponding to the empty path.

Example 2.1. Figures 2.1 and 2.2 show an APG and an equivalent tree obtained from it. The pointed node and the root are marked with underlines.

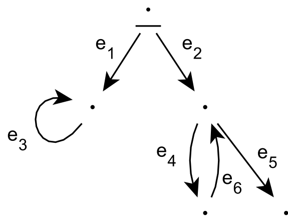


Figure 2.1.: An APG corresponding to the tree in Fig 2.2

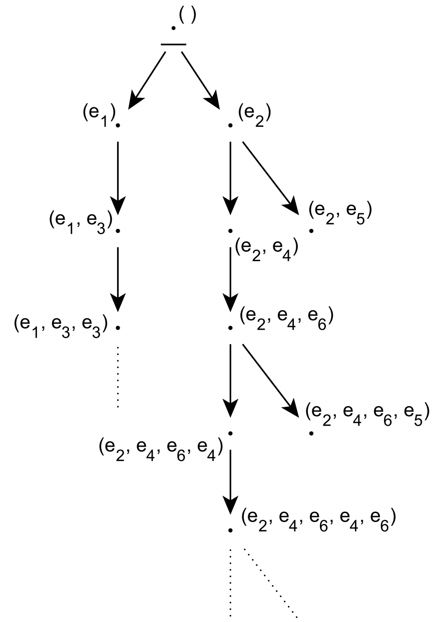


Figure 2.2.: A tree corresponding to the APG in Fig 2.1

Let's formalize it a little bit now.

Definition 2.2. Let $\mathbb{I}$ be some set of indexes and N be some underlying set of nodes. We say a pair $\langle N, \downarrow \rangle$ is an ***accessible pointed graph (APG) environment***, if $\downarrow$ is a partial function $N \times \mathbb{I} \hookrightarrow N$. We call then $\downarrow$ a ***child function*** and write $t_{\downarrow i}$ for $\downarrow (t, i)$.

An APG environment describes a directed unlabeled graph with indexed edges, where N stands for the nodes and $\downarrow$ fixes the edges. More precisely, if $v_{1_{\downarrow i}} = v_2$, then there is an edge indexed by i from the node v_1 to the node v_2 in the graph. We simply say then that v_2 is the i'th child of v_1 and v_1 is a parent of v_2. If for some $v \in N$ and $i_1 \dots i_n \in \mathbb{I}^*$ a node $v_{\downarrow i_1 \downarrow i_2 \dots \downarrow i_n}$ is defined, we say it is a ***descendant*** of v and write $v_{\downarrow i_1 i_2 \dots i_n}$ instead of $v_{\downarrow i_1 \downarrow i_2 \dots \downarrow i_n}$.

Such environment can be seen as a set of APGs, because if we point a node v from N, it determines

an APG — the ***nodes of the APG*** are all descendants of v, the edges are determined by $\downarrow$ and the pointed node is the pointed node v.

Definition 2.3. An ***APG*** is a pair $\langle env, v\rangle$, where $env = \langle N, \downarrow\rangle$ is an APG environment and $v \in N$.

We also say that $t = \langle env, v\rangle$ is an APG in env. The pointed node v is also called a root then and it is denoted by $root\,(t)$. To pop the underlying set from t or from env we write ***carr (t)*** or $carr\,(env)$ (carrier), respectively.

APGs with a finite number of nodes are called ***finite.*** APGs for which a root have no parents and all other nodes have a single parent are called ***trees***.

Definition 2.4. For an APG $t = \langle\langle N, \downarrow\rangle, v\rangle$, ***paths (t)*** is the set of all $w \in \mathbb{I}^*$ such that $v_{\downarrow w}$ is defined. Such a sequence w is then called an (index) ***path*** in t.

If $w \in paths\,(t)$, we write simply $t_{\downarrow w}$ instead of $root\,(t)_{\downarrow w}$. We also set a function $\vartriangle$ which applied to t and w stands for the APG $\langle\langle N, \downarrow\rangle, v_{\downarrow w}\rangle$. We write $t_{\vartriangle i_1 \dots i_n}$ instead of $\vartriangle(t, i_1 \dots i_n)$.

Definition 2.5. For an APG t indexed by $\mathbb{I}$, $\mathcal{P}ath\mathcal{E}nv\,(t)$ is the APG environment $\langle paths\,(t), \downarrow\rangle$ also indexed by $\mathbb{I}$, where the partial function $\downarrow$ is defined as $w_{\downarrow i} = wi$ if and only if $w, wi \in paths\,(t)$. The APG $\mathcal{P}ath\mathcal{T}ree\,(t)$ is then $\langle \mathcal{P}ath\mathcal{E}nv\,(t), \varepsilon\rangle$.

Note that $\mathcal{P}ath\mathcal{T}ree\,(t)$ is, indeed, a tree as each node in it has exactly one parent if it is not ε, and it has no parents if it is ε. We may also note that $\mathcal{P}ath\mathcal{T}ree\,(t)$ stands for the tree corresponding to t that we mentioned at the beginning of this section.

Recall that in the following definition $\mathbb{I}^{*|\omega}$ stands for the set of all finite and infinite sequences of elements form $\mathbb{I}$.

Definition 2.6. A ***branch*** in an APG t is a sequence $\varpi \in \mathbb{I}^{*|\omega}$ such that $\varpi|_n \in paths\,(t)$ for all $n \le |\varpi|$ and if $|\varpi| < \omega$, then $\varpi i \notin paths\,(t)$ for any $i \in \mathbb{I}$.

We say a node $v \in N$ is on a branch ϖ in an APG t if there exists n such that $t_{\downarrow \varpi|_n} = v$.

Definition 2.7. The ***height*** of an APG is the length of the longest branch in it incremented by 1. If the branch is infinite, then its length is ω.

Now we extend the concept of APGs and their environments by adding labelings to them.

Definition 2.8. Let $\mathbb{I}$ be a set of indexes, N be some underlying set and Σ be a set of symbols (labels) — we call it an ***alphabet***. Then $\langle N, \downarrow, \imath\rangle$ is a ***Σ-APG environment***, if $\langle N, \downarrow\rangle$ is an APG environment and $\imath$ is a ***symbol function*** of the type $N \to \Sigma$. A pair $\langle\langle N, \downarrow, \imath\rangle, v\rangle$ is then an ***Σ-APG*** if v is a member of N.

All terms defined for APGs may be used with Σ-APGs by removing $\imath$ from the environment and applying them as before. An exception is $\vartriangle$ that does not remove $\imath$.

$$\langle\langle N, \downarrow, \imath\rangle, v\rangle_{\vartriangle w} = \langle\langle N, \downarrow, \imath\rangle, v_{\downarrow w}\rangle$$

For a Σ-APG $t = \langle\langle N, \downarrow, \imath\rangle, v\rangle$ and a path $w \in paths\,(t)$ we also give a subscript operation t_w that stands for $\imath\,(v_{\downarrow w})$. To extend an APG environment $env = \langle N, \downarrow\rangle$ to $\langle N, \downarrow, \imath\rangle$ we write $env + \imath$.

Definition 2.9. For a Σ-APG t, $\mathcal{L}ab\mathcal{P}ath\mathcal{E}nv\,(t)$ is defined as $\mathcal{P}ath\mathcal{E}nv\,(t) + \imath$, where $\imath$ is set as $\imath\,(w) = t_w$. The Σ-APG $\mathcal{L}ab\mathcal{P}ath\mathcal{T}ree\,(t)$ is then $\langle \mathcal{L}ab\mathcal{P}ath\mathcal{E}nv\,(t), \varepsilon\rangle$, as before.

Definition 2.10. Two Σ-APGs t_1, t_2 are called ***isomorphic*** if $\mathcal{L}ab\mathcal{P}ath\mathcal{T}ree\,(t_1) = \mathcal{L}ab\mathcal{P}ath\mathcal{T}ree\,(t_2)$, that is if $paths\,(t_1) = paths\,(t_2)$ and for each path $w \in paths\,(t_1)$ we have $t_{1_w} = t_{2_w}$.

Definition 2.11. A ***ranked alphabet*** $\mathcal{F}$ is an ordered pair of a set Σ — alphabet — and an arity function $ar : \Sigma \to \mathbb{N}$.

For conciseness, we write $a \in \mathcal{F}$ for $a \in \Sigma$ and we write Σ_n for $\{\, a \in \Sigma \mid ar\,(a) = n \,\}$. For simplicity we also define a ranked alphabet $\mathcal{F}$ by writing $\mathcal{F} = \{a_1/k_1, \dots, a_n/k_n\}$, what means that $\mathcal{F} = \langle \{a_1, \dots, a_n\}, \{a_1 \mapsto k_1, \dots, a_n \mapsto k_n\}\rangle$.

Definition 2.12. Let $\mathcal{F} = \langle \Sigma, ar\rangle$ be a ranked alphabet. We say a Σ-APG environment $\langle N, \downarrow, \imath\rangle$ indexed by $\mathbb{N}$ is an ***$\mathcal{F}$-APG environment***, if each node $v \in N$ has exactly $ar\,(\imath\,(v))$ children $v_{\downarrow 1}, \dots, v_{\downarrow ar(\imath(v))}$. An ***$\mathcal{F}$-APG*** is then any Σ-APG.

2.2 Automata

APG automata are an equivalent of tree automata. These automata expose a quite big topic, so if the reader finds the following content is not enough for him/her, more information can be found in [TATA] and [Tho90].

To avoid consternation of readers that already know the topic, we shall probably point out here that our automata are not exactly the classical

ones. That is, the general concept of tree automata is extended in this paper by adding variables to them. Such approach was also proposed by Damian Niwiński in [Niw97]. Intuitively, variables allow an automaton to parse only a part of an input tree (APG), associate what is left with some variables and leave the acceptance of it to a valuation of these variables.

2.2.1 Non-deterministic automata (minimal, maximal, Büchi, Rabin)

Definition 2.13. A ***non-deterministic APG automaton (NDA)*** $\mathcal{A}$ is a tuple $\langle \mathcal{F}, Q, V, q_0, V_0, \delta, \alpha \rangle$, where

- $\mathcal{F}$ is a finite ranked alphabet
- Q is a finite set of states
- V is a finite set of variables
- $q_0 \in Q$ is a start state
- $V_0 \subseteq V$ is a set of initial variables
- $\delta \subseteq \bigcup_n Q \times \Sigma_n \times (Q \cup V)^n$ is a transition relation
- $\alpha : 2^Q \to \{true, false\}$ is an acceptance condition

By convention we write $q \xrightarrow{a} (s_1, \ldots, s_n)$ for $(q, a, (s_1, \ldots, s_n)) \in \delta$. We also write δ_a for $\left\{ q \xrightarrow{b} (s_1, \ldots, s_n) \in \delta \;\middle|\; b = a \right\}$, $\delta_{q,a}$ for $\left\{ p \xrightarrow{b} (s_1, \ldots, s_n) \in \delta \;\middle|\; b = a,\ p = q \right\}$ and $\mathcal{F}_\mathcal{A}$, $Q_\mathcal{A}$, $V_\mathcal{A}$, $q_{0\mathcal{A}}$, $V_{0\mathcal{A}}$, $\delta_\mathcal{A}$, $\alpha_\mathcal{A}$ for corresponding elements of an automaton $\mathcal{A}$. For automata with the empty set V we will say they are ***closed***.

Recall that in the following definition $r_{\downarrow w}$ stands for the node at path w from the root of r and r_w stands for the symbol assigned to this node. In the definition there also appear valuations that map variables to sets of APG nodes, however as the set of nodes of all APGs is not restricted we need to allow these valuations to map to sets of any objects. Those objects that are not nodes of a parsed APG will be just of no use then.

Definition 2.14. Let $\mathcal{A}$ be an NDA, t be an $\mathcal{F}_\mathcal{A}$-APG, S be a set, val be a valuation function such that $val|_{V_\mathcal{A}} : V_\mathcal{A} \to 2^S$ and let $s \in Q_\mathcal{A} \cup V_\mathcal{A}$. Then an ***$s$-run*** of $\mathcal{A}$ on t with respect to val is a $(Q_\mathcal{A} \cup V_\mathcal{A})$-APG r indexed by naturals satisfying the following conditions.

- $r_\varepsilon = s$
- for each path w in r
  - ⋆ if r_w is a state, then the node $r_{\downarrow w}$ has exactly n (not necessarily distinct) children $r_{\downarrow w1}, \ldots, r_{\downarrow wn}$ and there is a transition $r_w \xrightarrow{t_w} (r_{w1}, \ldots, r_{wn})$ in $\delta_\mathcal{A}$
  - ⋆ if r_w is a variable, then $t_{\downarrow w} \in val\,(r_w)$

Definition 2.15. We say that a q_0-run r of an NDA $\mathcal{A}$ is ***accepting***, if for each infinite branch ϖ in r the condition $\alpha_\mathcal{A}\left(\left\{ q \mid r_{\varpi|_n} = q \text{ for infinitely many } n \right\}\right)$ is true. We also say that x_0-run of $\mathcal{A}$ is accepting if $x_0 \in V_{0_\mathcal{A}}$.

If for an $\mathcal{F}_\mathcal{A}$-APG $\langle env, v \rangle$ there exists an accepting run of $\mathcal{A}$ with respect to some valuation, then the APG is called ***accepted*** by $\mathcal{A}$ with respect to this valuation. Alternatively we also say that the node v is ***accepted*** by $\mathcal{A}$ in env with respect to this valuation.

When an automaton is closed we can omit the clause "with respect to".

Definition 2.16. The ***semantics of an NDA*** $\mathcal{A}$ within $\mathcal{F}_\mathcal{A}$-APG environment env with respect to a valuation val is the set of nodes $\mathcal{A}^{env}\,[val] = \{v \in carr\,(env) \mid v \text{ is accepted by } \mathcal{A}$ in env with respect to $val\}$.

Again, when an automaton is closed we can omit "with respect to" and define $\mathcal{A}^{env} = \{v \in carr\,(env) \mid v \text{ is accepted by } \mathcal{A} \text{ in } env\}$.

Theorem 2.17. *If APGs t_1, t_2 are isomorphic, then for each NDA $\mathcal{A}$ and valuations val_1, val_2 that are equal up to APG isomorphism, $\mathcal{A}$ accepts t_1 with respect to val_1 if and only it accepts t_2 with respect to val_2.*

Proof. It is obvious, as the definition of runs does not depend on the underlying sets of APGs for which these runs are set. □

Remark 2.18. As for each APG there exists a tree isomorphic to it, Theorem 2.17 shows that our APG automata and common tree automata (where the trees may be infinite) are equivalent.

Definition 2.19. Let $\mathcal{A}$ be an NDA and val be a valuation for it. An APG environment is ***weakly closed over*** $\mathcal{A}$ with respect to val, if for each finite APG accepted by $\mathcal{A}$ wrt val there exists an isomorphic APG in the environment.

For closed automata the clause "with respect to" may again be omitted.

Definition 2.20. ***Minimal condition*** is the NDA's α condition that constantly equals *false*

and the ***maximal condition*** is α that constantly equals *true*. We will denote them later as $\bot$ and $\top$, respectively.

Definition 2.21. A ***Büchi condition*** is an NDA's α condition with a fixed set B of automata states, such that $\alpha(P) = true$ if and only if $P \cap B$ is non-empty. If α is Büchi, we denote this set by B_α. The states in B_α are called then ***accepting states***.

Definition 2.22. A ***Rabin condition*** is an NDA's α condition with a fixed set $\{(L_1, U_1), \ldots, (L_n, U_n)\}$ of pairs of automata states, such that for any state set P the condition $\alpha(P) = true$ if and only if there exists i for which $P \cap U_i$ is non-empty and $P \cap L_i$ is empty. These pairs of states are also called ***accepting pairs***.

Automata with an accepting condition from some of the given classes are named by preceding their names with the name of this class. So we say "a Büchi non-deterministic APG automaton" (Büchi NDA, in short) instead of "an NDA with a Büchi accepting condition", and so on.

In this paper we will not be very concerned about Rabin automata, however we refer to them sometimes, thus we give a definition for them.

One such reference goes here.

Theorem 2.23. *If a closed Rabin NDA accepts any APG, then there exists a finite APG that is accepted by it.*

Proof. It is known that, for any Rabin non-deterministic automaton on trees — and so also on APGs — if there exists a tree accepted by it, then there also exists a so-called regular tree accepted by it (cf. [Tho90]). But for any regular tree we can easily build a finite APG isomorphic to it, so our claim is also true. This is because regular trees are trees that have only finitely many isomorphic subtrees (in our case a subtree of a tree t is any possible $t_{\triangle w}$), thus we can fix the underlying set for the mentioned isomorphic APG as the set of equivalence classes on subtrees of a tree and then fix the APG's edges and the labeling by inheriting them from the tree. □

Note that Büchi NDAs are also Rabin NDAs, so the theorem is also valid for the former ones.

Definition 2.24. Let $\mathcal{A}_0, \mathcal{A}_1, \ldots, \mathcal{A}_n$ be NDAs over an alphabet $\mathcal{F}$ and $x_1, \ldots, x_n$ are pairwise different variables of $\mathcal{A}_0$. Then the automaton $\mathcal{A}_0[x_1 \mapsto \mathcal{A}_1, \ldots, x_n \mapsto \mathcal{A}_n]$ defined below is called a ***composition*** of $\mathcal{A}_0$ with $\mathcal{A}_1, \ldots, \mathcal{A}_n$.

Let $\mathcal{A}_i = \langle \mathcal{F}_i, Q_i, V_i, q_{0i}, V_{0i}, \delta_i, \alpha_i \rangle$ for $i = 0, \ldots, n$. Then $\mathcal{A}[x_1 \mapsto \mathcal{A}_1, \ldots, x_n \mapsto \mathcal{A}_n] =$

$$\langle \mathcal{F}, Q, V, q_0, V_0, \delta, \alpha \rangle$$

where

$$Q = \bigcup_{i=0,\ldots,n} \{i\} \times Q_i$$

$$V = V_i \setminus \{x_1, \ldots, x_n\} \cup \bigcup_{i=1,\ldots,n} V_i$$

$$q_0 = (0, q_{00})$$

$$V_0 = V_{00} \setminus \{x_1, \ldots, x_n\} \cup \bigcup_{x_i \in \{x_1,\ldots,x_n\} \cap V_{00}} V_{0i}$$

$$\begin{aligned}\delta = \bigcup_{i=0,\ldots,n} \Big\{ (i,q) \xrightarrow{a} \overrightarrow{z} \mid q \xrightarrow{a} \left(s_1, \ldots, s_{ar(a)}\right) \in \delta_i \\ \wedge \overrightarrow{z} \in \textstyle\prod_{k=1}^{ar(a)} \sigma_i(s_k) \Big\} \cup \\ \bigcup_{x_i \in \{x_1,\ldots,x_n\} \cap V_{00}} \Big\{ q_0 \xrightarrow{a} \overrightarrow{z} \mid q_{0i} \xrightarrow{a} \left(s_1, \ldots, s_{ar(a)}\right) \in \delta_i \\ \wedge \overrightarrow{z} \in \textstyle\prod_{k=1}^{ar(a)} \sigma_i(s_k) \Big\}\end{aligned}$$

$$\alpha(P) = \begin{cases} true, & \text{if } P = \{i\} \times P' \text{ and} \\ & \alpha_i(P') \text{ is true for some } i, P' \\ false, & \text{otherwise} \end{cases}$$

and for $i = 0, \ldots, n$ the mapping $\sigma_i : Q_i \cup V_i \to 2^{Q \cup V}$ is defined as

$$\sigma_i(s) = \begin{cases} \{(k, q_{0k})\} \cup V_{0k}, & \text{if } i = 0 \wedge s = x_k \\ & \text{for some } k \in \{1, \ldots, n\} \\ \{(i, s)\}, & \text{if } s \in Q_i \\ \{s\}, & \text{otherwise} \end{cases}$$

Note that the "otherwise" condition means that s is any variable for $i > 0$ and $s \in V_0 \setminus \{x_1, \ldots, x_n\}$ for $i = 0$.

Lemma 2.25. *For any NDAs $\mathcal{A}_0, \mathcal{A}_1, \ldots, \mathcal{A}_n$ over $\mathcal{F}$, an $\mathcal{F}$-APG environment env and a valuation val, we have that*

$$\mathcal{A}_0[x_1 \mapsto \mathcal{A}_1, \ldots, x_n \mapsto \mathcal{A}_n]^{env}[val] = \mathcal{A}_0^{env}[val']$$

where

$$val' = val[x_1 \mapsto \mathcal{A}_1^{env}[val], \ldots, x_n \mapsto \mathcal{A}_n^{env}[val]]$$

Proof. We can decompose any accepting run of $\mathcal{A}_0[x_1 \mapsto \mathcal{A}_1, \ldots, x_n \mapsto \mathcal{A}_n]$ to accepting runs of $\mathcal{A}_0, \mathcal{A}_1, \ldots, \mathcal{A}_n$ and vice versa. □

Remark 2.26. If $\mathcal{A}_0, \mathcal{A}_1, \ldots, \mathcal{A}_n$ are Büchi automata, then their composition is also a Büchi automaton. If $\mathcal{A}_0, \mathcal{A}_1, \ldots, \mathcal{A}_n$ are Rabin automata, then their composition is a Rabin automaton.

2.2.2 Cascading automata

Here we propose a subclass of NDAs. This subclass is not common, but it will be helpful as it allows for structural breaking automata to smaller ones and simplify reasoning.

Definition 2.27. An NDA $\langle \mathcal{F}, Q, V, q_0, V_0, \delta, \alpha \rangle$ is a ***cascading automaton***, if there exists a partition $\{Q_i\}_{i \in [n]}$ of Q and a sequence of accepting conditions $\{\alpha_i\}_{i \in [n]}$ such that

- for each $q \in Q_i$, $p \in Q_j$ if $i < j$, then there is no $q \xrightarrow{a} (\ldots, p, \ldots)$ in δ,
- $\alpha_i : 2^{Q_i} \to \{true, false\}$ and
- $\alpha(P) = \begin{cases} true, & \text{if } \alpha_i(P) = true \text{ for some } i \\ false, & \text{otherwise} \end{cases}$

Such an automaton is then denoted by

$$\left\langle \mathcal{F}, \{Q_i\}_{i \in [n]}, V, q_0, V_0, \delta, \{\alpha_i\}_{i \in [n]} \right\rangle$$

Note that cascading automata with only minimal and maximal α_i conditions are Büchi. Commonly these are called ***weak Büchi automata*** and they are generally easier for computations.

2.2.3 Alternating automata

Alternating automata on APGs are an extension of NDAs.

Definition 2.28. An ***alternating APG automaton (AA)*** is a tuple $\langle \mathcal{F}, Q, V, q_0, V_0, \delta, \alpha \rangle$ defined as an NDA except δ, which still is a transition relation, but of the shape $\delta \subseteq \bigcup_n Q \times \Sigma_n \times \left(2^{Q \cup V}\right)^n$ (cf. def. 2.13).

Definition 2.29. Let $\mathcal{A}$ be an AA, t be an $\mathcal{F}_{\mathcal{A}}$-APG, S be a set, *val* be a valuation such that $val|_{V_{\mathcal{A}}} : V_{\mathcal{A}} \to 2^S$ and let $s \in Q_{\mathcal{A}} \cup V_{\mathcal{A}}$. Then an ***s-run*** of $\mathcal{A}$ on t wrt *val* is a $(Q_{\mathcal{A}} \cup V_{\mathcal{A}})$-APG r indexed by $\mathbb{N} \times Q_{\mathcal{A}}$ satisfying the following conditions.

- $r_\varepsilon = s$
- for each path $w = (k_1, q_1) \ldots (k_n, q_n)$ in r
  - ⋆ if $n > 0$, then $r_w = q_n$,
  - ⋆ if r_w is a state, then for $a = t_{k_1 \ldots k_n}$ there is a transition $r_w \xrightarrow{a} \left(P_1, \ldots, P_{ar(a)}\right)$ in $\delta_{\mathcal{A}}$ such that the node $r_{\downarrow w}$ has exactly $\sum_{i=1}^{ar(a)} |P_i|$ (not necessarily distinct) children $r_{\downarrow w(i,q)}$ for all $i = 1, \ldots, ar(a)$ and all $q \in P_i$, and
  - ⋆ if r_w is a variable, then $t_{\downarrow w} \in val(r_w)$

Here we do some work around, as the symbol function of r just copies the symbol from the last element of the path leading to a node, and so it is superfluous. Still we put it there to treat a run as an APG over the alphabet of states and variables of an automaton, as it was done for NDAs. This is handy, because this way we can work with AAs runs similarly to those of NDAs.

Example. Figure 2.3 shows a part of a run of some alternating automaton with transitions $q_0 \xrightarrow{\cdots} (\{q_0, q_1\}, \{q_0\})$, $q_0 \xrightarrow{\cdots} (\{q_0\}, \{q_0, q_1\})$, $q_0 \xrightarrow{\cdots} (\{q_0\}, \{q_0\})$ and $q_1 \xrightarrow{\cdots} ()$. (In practice the second axis in edge labels may be omitted as it is always a copy of the label of a target node.)

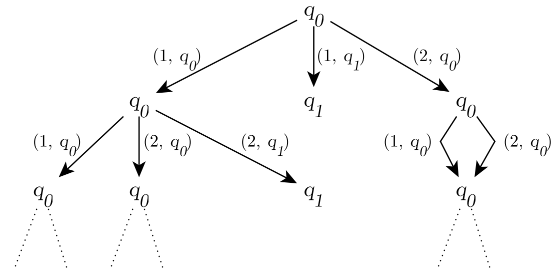


Figure 2.3.

Definition 2.30. Acceptance and semantics for alternating automata is defined as for NDAs (cf. def. 2.15, 2.16).

Note that the classes of accepting conditions defined for non-deterministic automata in Subsection 2.2.1 are still valid for AAs.

Definition 2.31. Any NDA $\mathcal{A}$ can be seen as AA by exchanging each transition $q \xrightarrow{a} \left(q_1, \ldots, q_{ar(a)}\right)$ in it to $q \xrightarrow{a} \left(\{q_1\}, \ldots, \{q_{ar(a)}\}\right)$. We will denote such an alternating automaton by $Alt(\mathcal{A})$.

Definition 2.32. The ***cascading*** concept is defined for AAs by exchanging $q \xrightarrow{a} (\ldots, p, \ldots)$ with $q \xrightarrow{a} (\ldots, \{p, \ldots\}, \ldots)$ in the first clause of Definition 2.27.

Definition 2.33. ***Compositions*** can be given for AAs after changing σ_i in Definition 2.24 to $\sigma_i' : 2^{Q_i \cup V_i} \to 2^{2^{Q \cup V}}$ such that $\sigma_i'(P) = \{\, f(P) \mid f \in \prod_{s \in P} \sigma_i(s) \,\}$, where σ_i is the old mapping.

Remark 2.34. Lemma 2.25 and Remark 2.26 hold for AAs.

2.2.4 Remark about automata inclusion

This section is a conclusion from Appendix A (mostly Section A.5).

First let's say what we understand under automata inclusion. ***Inclusion*** of one automaton in another one is the property saying that each APG accepted by the former is also accepted by the latter for any valuation of their variables. In this paper we will need to check the inclusion with respect to some specific APG environment for testing if a type is a subtype of another type.

The following theorem is a reformulation of a generally known fact that testing inclusion between regular languages of infinite trees is decidable.

Theorem 2.35. *Let $\mathcal{A}_1$, $\mathcal{A}_2$ be closed Rabin NDAs over a ranked alphabet $\mathcal{F}$ and env be an $\mathcal{F}$-APG environment weakly closed over $\mathcal{A}_1$. Then testing the inclusion $\mathcal{A}_1^{env} \subseteq \mathcal{A}_2^{env}$ is decidable.*

Unfortunately this result is not very practical, as the cost of the computation is very high in the case of Rabin automata. That is why we will describe some more effective solution of this problem for closed weak Büchi NDAs in Appendix A and we will show how to restrict our type propositions that we give in the sequel to be representable by these.

The decidability of inclusion-checking for Büchi non-deterministic automata on trees, and so APGs, is a well-known fact. However we give a procedure for this in Appendix A, as most of already described ones are given for stronger automata and so they are more complex — both in idea and computations. Particularly, in our case we can take an advantage of the fact, that for a weak Büchi NDA there exists a Büchi NDA that recognises the complement of the former one and this automaton is a crucial part of evaluating the inclusion-checking. This is yet not true for stronger automata, as for them we mostly need Rabin or other equivalent condition to do the complementation and using it results in heavy constructions.

As the following remark states, the pessimistic complexity of the procedure that we propose is single exponential with a linear exponent (up to the $ar(a)$ factor, that can be usually bound by a small constant), what is a good result in comparison to those for stronger automata.

Remark 2.36. The complexity of the solution for weak Büchi NDAs inclusion-testing given in the appendix is $\mathcal{O}(K + L + LM)$ in time and $\mathcal{O}(K + L)$ in space, where pessimistically

- $K \le |Q_1| \, 3^{|Q_2|}$
- $L \le \sum_{a\in\mathcal{F}} |\delta_{1_a}| \, 3^{|Q_2|} \left(ar(a) + 1\right)^{|\delta_{2\,a}|+1}$
- $M \le |B_{\alpha_1}| \, 2^{|Q_2|}$

and indexes 1, 2 point out from which automaton, $\mathcal{A}_1$ or $\mathcal{A}_2$ respectively, the symbols come.

This complexity may be still too high for practical use even for small automata, which we are interested in. But the bound do not take into account the structure of tested automata and in our case this works enough in our favour to give a credit to these automata as a computation model in this paper. Particularly, the complexity significantly decreases with raise of the level of determinism of the automaton $\mathcal{A}_2$ to finally reach the polynomial class for fully deterministic automata; and automata that we are going to propose seem unlikely to hit a high level of non-determinism.

Remark 2.37. For deterministic $\mathcal{A}_2$ the constants from the previous remark may be decreased to

- $K \le |Q_1| \left(|Q_2| + 1\right)$
- $L \le \sum_{a\in\mathcal{F}} ar^2(a) \, |\delta_{1_a}| \, |Q_2|$
- $M \le |B_{\alpha_1}| \left(|Q_2| - |B_{\alpha_2}|\right)$

One more thing that we should mention here is that, the inclusion-checking for alternating automata is also decidable, and that, there is still a hope for using weak Büchi AAs for our case.

Proposition 2.38. *Theorem 2.35 holds also for AAs.*

Proof. This is straightforward as translations of AAs to equivalent NDAs are well-known. □

Remark 2.39. For closed weak Büchi AAs the complexity of the procedure for inclusion-checking given in the appendix is as in Remark 2.36 with the difference that $L \le \sum_{a\in\mathcal{F}} ar(a) \, |\delta_{1_a}| \, 3^{|Q_2|} \left(ar(a) \, |Q_2| + 1\right)^{|\delta_{2\,a}|}$.

This seems again too much, but in our case the factor $|Q_2|$ is bound by the number of occurrences of some special operation on types, and as this operation is rather rare in practice the procedure should have close complexity to the one for NDAs.

The results described in this section are explained in more detail in Appendix A.

Chapter 3

Types

In the introduction we announced, we are going to propose an alternative view on types. In this chapter we will give a description of this view. We also give a few hints about how to represent some constructions considered at the field of type systems and about computable ways of implementation of type-checking with respect to the proposed approach.

3.1 General terms

In formal systems — e.g., such as programming languages — we usually work on some objects from a fixed ***universe***. In programming languages these objects are commonly called values. Types then describe some sets of these values. So we have, e.g., a type Nat which describes natural numbers, Bool which describes boolean values, record types which describe mappings of some finite sets of identifiers to values, like {a : Nat, b : Bool}, which describes all mappings associating a natural number to the identifier a and a boolean value to b, and others. These types are usually defined as expressions. But due to the problems appointed in the introduction, we are going to step aside a little bit from this.

What we propose is to define types as subsets of the universe.

Definition 3.1. Let $\mathcal{U}$ be a set representing a universe of values. A ***type*** in $\mathcal{U}$ is then any set $T \subseteq \mathcal{U}$ and a value v is said to be of this type if $v \in T$.

Of course this definition is extremely general and without restrictions checking membership in some types is undecidable. E.g., we can fix a set of values which encode programs that stop as a type. Then checking if a value belongs to this type is equivalent of checking the halting problem what is purely undecidable. Still such a class of values exists, thus it can be considered a type. So to deal with this we will have to put some restrictions on type sets. For this we use type systems.

Definition 3.2. Let $\mathcal{U}$ be a universe. Then a ***type system*** T_{sys} over $\mathcal{U}$ is any subset of the set of types $2^{\mathcal{U}}$.

Type systems generally allow to satisfy a desired level of computational complexity of checking types in a formal statement (like in a computer program) and sometimes they establish a correspondence between these statements and some other entities (cf. Curry-Howard isomorphism).

For types given as sets a very natural definition of subtyping flows.

Definition 3.3. A type T_1 is said to be a ***subtype*** of T_2 if $T_1 \subseteq T_2$.

This concept will be of high importance in the sequel, as next to testing membership in types it is a key part of checking types in formal statements. However, while testing the membership mostly does not expose troubles, testing subtyping may be unclear and so we generally omit the first one in favour to the second.

The above definitions give a semantical foundation to types, yet they do not give a formal way of checking them and representing them in physical space. Thus we put the following concept to handle this — we may see it, somehow, as a syntax of types.

Definition 3.4. A ***formalization*** of a type system T_{sys} over $\mathcal{U}$ is a set T_{form} of ***representations of types*** together with relations $:$ and $<:$, where the former is a subset of $\mathcal{U} \times T_{form}$ and the latter is a subset of $T_{form} \times T_{form}$, that is epimorphic with T_{sys} and relations $\in, \subseteq$. A term $v : t$ is read then as "the value v is of the type t" and a term $t_1 <: t_2$ as "the type t_1 is a subtype of the type t_2". The epimorphism is denoted by $[\![\,]\!]$ and called semantics.

Note that we do not need to give relations $:$ and $<:$ explicitly if we fix a semantic function, as the epimorphism determines them — that is $v : t \equiv v \in [\![t]\!]$ and $t_1 <: t_2 \equiv [\![t_1]\!] \subseteq [\![t_2]\!]$.

Formalizations can be interpreted as sets of encodings of types — for example for use in computer systems. We set an epimorphism instead of isomorphism here, as it may be impractical to set encodings which map one-to-one to type systems. E.g., if we have a formalization that is a set of expressions which contain a semantically commut-

ative symbol, like $\cup$ representing a set-union, then expressions like $exp_1 \cup exp_2$ and $exp_2 \cup exp_1$ should map to the same type. Then, to establish the isomorphism we would need to disallow one of these expressions in the formalization, what softly-saying would be at least unhandy in most cases.

Example 3.5. Let the values universe be $\mathcal{U} = \mathbb{Q} \cup \mathcal{U} \times \mathcal{U}$, where $\mathbb{Q}$ is the set of rational numbers (there is only one such $\mathcal{U}$ by the axiom of regularity). So in other words the universe is a set of nested pairs of rational numbers. An example type system can be set then as $\times_{sys} = \{\mathbb{N}, \mathbb{Q}\} \cup \{ T_1 \times T_2 \mid T_1, T_2 \in \times_{sys} \}$, where we assume that $\mathbb{N} \subseteq \mathbb{Q}$ (again there is only one such $\times_{sys}$ by the axiom of regularity). By doing so the type system allows to work with pairs of values in formal statements and to treat natural numbers in a privileged way. Specifically, we can have a natural-number-only operations — e.g. modulus — in a formal system and checking of types should guarantee us that we never apply these operation to non-natural values. A formalization for such a system can be fixed as the set of expressions $\times_{exp} ::= \mathsf{Nat} \mid \mathsf{Rat} \mid \times_{exp} * \times_{exp}$ with the natural semantics: $[\![\mathsf{Nat}]\!] = \mathbb{N}$, $[\![\mathsf{Rat}]\!] = \mathbb{Q}$ and $[\![t_1 * t_2]\!] = [\![t_1]\!] \times [\![t_2]\!]$. Such representation of types can be easily used then to compute membership and subtype checking in our type system by doing structural inductions on values and expressions.

3.2 General formalization concepts

When we use types in formal systems, we mostly use formalizations to operate on them. Formalizations give a way to expose formal and decidable rules to check type membership and subtyping, which the set theory's apparatus, that can be used for raw type systems, does not provide. Thus when we design a type system, in many cases we should be concerned to be able to provide a good formalization for it. This is, however, not a golden rule, as some systems may not need to be decidable.

Below we propose some general choices for formalizations that can be made.

Expressions

A commonly used approach are expressions. They are intuitive, readable and nicely fit computer implementations, as they are easy to input and output. Moreover, in simple type systems we can easily check types for such representations using structural induction. And on top of these all, with expressions we can formalize any type system that can ever be written, what makes them probably the most remarkable formalizations.

Nevertheless, expressions are not perfect. The induction has some limits and it fails as soon as we put recursive types in a system. Additionally, if we put set-like union types next to recursive ones, it become even more unclear how to check types with this formalization. This makes a place for other approaches.

But let's adhere to expressions right now.

Generally we give an expression formalization by defining a set of expressions, for example with a BNF formula, and then we set a semantical function from them into a type system. However, sometimes it may be more convenient to give a superset of expressions formalising a system and to set semantics as a partial function. Then we can retrieve a proper formalization from the set of all expressions by finding the domain of the semantic function.

Sometimes it may be hard to give a semantics of expressions directly into types. A good example here are expressions containing variables. Particularly, if we have the expression $x \times y$, we can suspect that it does not describe any single type, but rather a function that for a given valuation of variables x and y returns a type (in this case, it will probably be the Cartesian product function). That is why in some cases we may want to give semantics of expressions as a function not only from expressions by them own, but also from some contexts that determinize their meaning.

Putting it more formally the following simple observation may be used to give expression formalizations and possibly type systems.

Observation 3.6. *Let $\mathcal{U}$ be a value universe, $\mathcal{E}xp$ be a set of expressions, $\mathcal{C}$ be a set of contexts, $C_0 \in \mathcal{C}$ be some initial context and $[\![]\!] : \mathcal{E}xp \times \mathcal{C} \hookrightarrow 2^{\mathcal{U}}$. Then $T_{exp} = \{ e \mid (e, C_0) \in Dom([\![]\!]) \}$ is an expression formalization of the type system $\{ [\![e]\!]_{C_0} \mid e \in T_{exp} \}$, where the semantic function is given as $[\![e]\!] = [\![e]\!]_{C_0}$ and $[\![e]\!]_C$ stands for $[\![]\!](e, C)$.*

In practice the only issue that requires contexts

is probably introducing variables to expressions, so without losing much sight we may see $\mathcal{C}$ as a set of valuations of variables to types with C_0 being the empty valuation.

APGs

Some generalisation of expressions are accessible pointed graphs. These graphs may be seen as expressions that allow cycles in their structure and reusing parts of a type definition in many places. They seem proper for modeling recursive types — like lists, trees, etc. — as recursion corresponds to cycles. This, however, works fine only until more then one recursion kinds are introduced. So if we have, for example, inductive and coinductive recursions in one type system, it may be not straight any more how to describe types with this approach. Additionally, a disadvantage of these formalizations in comparison to expressions is that we cannot perform structural induction on graphs (due to cycles).

Nevertheless, we put this concept here, despite of questionable attractiveness, as graphs may be a nice graphical representation of some types.

Example 3.7. The following figures present some types formalized by APGs. Intuitively $\times$ represents a Cartesian product, | a union, Nat the set of naturals and pointed nodes are marked with underlines. (Note that the nested Cartesian product in Fig 3.2 may be seen as a set of triples of a natural and two child trees.)

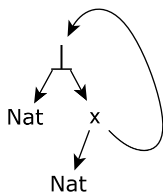


Figure 3.1.: Type for non-empty lists of natural numbers

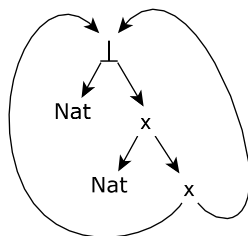


Figure 3.2.: Type for non-empty binary trees where each node is labeled with a natural number

Automata

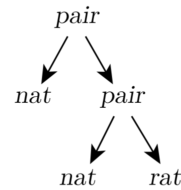
Typically values that occure in formal systems can be expressed as trees, or more accurately, as APGs. In such case types correspond to sets of trees/APGs and so they can be represented as automata on trees/APGs. This concept seems promising as much work has been already done in terms of computation of problems and operations on automata. Moreover, it can be further generalized to other objects then trees and APGs by the proposition given by Damian Niwiński in [Niw97] of automata working on algebras. Still automata are not too readable and printable, so when using them we probably should also provide some other formalization, like expressions, for proper presentation of types.

To use this approach first we need to set an $\mathcal{F}$-APG environment for some ranked alphabet $\mathcal{F}$ where the universe of values is the underlying set. Children of a value in the environment can be seen then as subvalues from which the parent value is composed and the symbol with which this value is labeled says, how to compose these children to obtain the parent. In such a setting a closed $\mathcal{F}$-APG automaton obviously describes a type. It is the type of all values that are accepted by the automaton in the environment. In other words if we name our universe environment $env_{\mathcal{U}}$ and the automaton $\mathcal{A}$, the type is $\mathcal{A}^{env_{\mathcal{U}}}$.

Example 3.8. Let's take $\mathcal{U} = \mathbb{Q} \cup \mathcal{U} \times \mathcal{U}$ as in Example 3.5. An APG representation of it can be set then as the $\mathcal{F}$-APG environment $env_{\mathcal{U}} = \langle \mathcal{U}, \downarrow, \imath \rangle$ such that

- $\mathcal{F} = \{nat/0, rat_{-}/0, pair/2\}$,
- for $i \in \{1, 2\}$ we have that $v_{\downarrow i} = v'$ if and only if there exist $u_1, u_2 \in \mathcal{U}$ such that $v = (u_1, u_2)$ and $v' = u_i$, and
- $\imath(n) = nat$, $\imath(x) = rat_{-}$, $\imath((u, u')) = pair$, for any $n \in \mathbb{N}$, $x \in \mathbb{Q} \setminus \mathbb{N}$, $u, u' \in \mathcal{U}$

Note we need some disambiguation here, as if we try to put a symbol rat that represents all rationals instead of rat_{-}, then $\imath(1)$ should be both nat and rat, what cannot be. To illustrate how the environment represent values we can take the value $(1, (2, 2.5))$. The APG representation of it is then



There are two things worthy to note here. One is that, there can be more then one value with the same symbol and with the same children assigned

to it in an APG environment. Thus the methods of constructing a value from its subvalues appointed by symbols within an environment can be non-deterministic. The most common example for this are constant symbols, like *nat* that is assigned to all leaf (0-chidren) nodes representing natural numbers, what says that from the empty set of subvalues we can construct the value 0, the value 1, the value 2 and so on. The other thing is that, there can be only one symbol assigned to a value in an APG environment, so each value may have only one way of construction. This causes that, if the universe countenances for constructing a value in many ways, some disambiguation is needed before using automata as a formalism (cf. ex. 3.8).

Now, when we have such an automata setting, the relation : is just testing acceptance of values by automata representing types in environment $env_{\mathcal{U}}$ and the relation <: is testing inclusion between these automata with respect to $env_{\mathcal{U}}$. We may remark here that for testing the second we usually will want the environment to be weakly closed over automata from a formalization, to make use of Theorem 2.35.

Example 3.9. Taking $env_{\mathcal{U}}$ from Example 3.8 the type $\mathbb{N} \times \mathbb{Q}$ may be given as the automaton $\mathcal{A} = \langle \mathcal{F}, \{q_0, q_1, q_2\}, \varnothing, q_0, \varnothing, \delta, \bot \rangle$ where

$$\delta = \left\{ q_0 \xrightarrow{pair} (q_1, q_2), q_1 \xrightarrow{nat} (), \right.$$
$$\left. q_2 \xrightarrow{nat} (), q_2 \xrightarrow{rat_-} () \right\}$$

To show some example of subtyping it is enough now to add a transition to $\mathcal{A}$. Like this: $\mathcal{A}' = \left\langle \mathcal{F}, \{q_0, q_1, q_2\}, \varnothing, q_0, \varnothing, \delta \cup \left\{ q_1 \xrightarrow{rat_-} () \right\}, \bot \right\rangle$. Then $\mathcal{A} <: \mathcal{A}'$, as any accepting run of the first automaton on an APG is also an accepting run of the second one on this APG.

Example 3.10. An automata formalization $\times_{aut}$ of the system $\times_{sys}$ may be the set of all automata $\langle \mathcal{F}, Q, \varnothing, q_0, \varnothing, \delta, \bot \rangle$ such that $Q \subseteq \mathcal{S}tates$ for some fixed infinite set $\mathcal{S}tates$ and $q \xrightarrow{rat_-} () \in \delta \implies q \xrightarrow{nat} () \in \delta$ where $v : \mathcal{A}$ is defined, as mentioned, by $v \in \mathcal{A}^{env_{\mathcal{U}}}$ and $\mathcal{A} <: \mathcal{A}'$ by $\mathcal{A}^{env_{\mathcal{U}}} \subseteq \mathcal{A}'^{env_{\mathcal{U}}}$.

Now it is probably a good moment to recall that we announced, we would explain our impulses for APGs. These impulses come from the observation that values in formal systems often can be described as APGs. Of course, we can also use trees for this, however the correspondence of trees to values is not so straight as in the case of APGs. The reason for this is that, one value may be used many times in construction of another value. When we would like to use trees, this leads to the situation where in a tree representing some value multiple subtrees representing a single subvalue may occur. This way we lose a structure of values if we want to represent them as trees. But, what is worse, this also makes the size of representations may blow and this blow can be very significant. Particularly, in the world of coinductive values (that we explain later), there are values that can be represented as finite APGs, but if we represent them as trees, these trees needs to be infinite (cf. ex. 3.26).

3.3 Example type constructions

We close the chapter with a description, how to put some basic constructions used in common type systems into our setting.

3.3.1 How do we give constructions

We will use Observation 3.6 here. Saying more precisely, we define a type system over a value universe $\mathcal{U}$ by giving a set of expressions, a set of contexts and a semantic function. The expression set $\mathcal{E}xp$ is defined with a single ABNF-style rule that we extend for each new construction we add to the system. The context set $\mathcal{C}$ is fixed as the set of all valuations $val : \mathcal{V}ar \hookrightarrow 2^{\mathcal{U}}$ where $\mathcal{V}ar$ is some infinite countable set of variables. The initial context C_0 is set as the empty valuation. Finally, the semantics $[\![]\!] : \mathcal{E}xp \times \mathcal{C} \hookrightarrow 2^{\mathcal{U}}$ is given by a set of recursive equalities, which we extend in parallel with $\mathcal{E}xp$.

Of course to extend a type system with some new construction we need also to guarantee some corresponding values to be in $\mathcal{U}$. E.g., if we introduce pair types to the system, there should be pair values in the universe that are described by these types. These prerequisites will be formed as conditions on $\mathcal{U}$ that must be satisfied before adding a construction.

Next to the expression formalization we also give an automata formalism. For this we assume that we have an $\mathcal{F}$-APG environment $env_{\mathcal{U}}$ over the set of values $\mathcal{U}$, where $\mathcal{F}$ corresponds to our type constructions. This correspondence is described per each construction by a condition on $\mathcal{F}$. The automata in the formalization are then given with a partial algebra, such that for each construction we

add one or more operations to the algebra, in a way similar to extending the ABNF rule for expressions. The automata formalization is then fixed as the set of all closed automata generated by the empty set through the algebra.

As for each new expression construction we will have a corresponding operation on automata, it is going to be straightforward that the automata formalise the same system as the expressions.

Still the big picture lack its key item — the underlying set for our algebra — and this exposes some problems. First of all, we need to decide what kind of automata is going to be represented by the set and this choice is not obvious, as some of the type constructions suit more to one kind then to another. The second problem is that, even if we choose the class of automata, we still cannot take the set of all automata in it and put it as the underlying set, as sets of states and variables are unlimited in automata and ZFC theory does not allow for sets of all sets.

In the first case we decide to use non-deterministic automata (NDAs). This is because, all our constructions except intersection types — which are not very common — can be easily given within this class. Intersection types are going to be presented with some variant of alternating automata (AAs). This will expose a minor problem, that we describe in the corresponding section. However, as the basic concept stays valid, we put the construction here. Besides intersection types we put one more construction, namely function types, in terms of AAs, as making it with NDAs causes very serious blow of automata size. Still, the previous statement is not a lie and doing it with NDAs is striaightforward.

To deal with the second problem we simply restrict sets of states and variables that automata can range through. So we put the set $\mathcal{V}ar$ that we settled before for expressions as the universe of automata variables and the set $\mathcal{S}tates = (\mathbb{N} \cup \mathcal{V}ar) \times \mathbb{N}^*$ as the universe of automata states. Then we fix the underlying set $\mathbb{A}$ of our algebra as the set of all NDAs $\langle \mathcal{F}, Q, V, q_0, V_0, \delta, \alpha \rangle$ such that $Q \subseteq \mathcal{S}tates$ and $V \subseteq \mathcal{V}ar$. Our $\mathbb{A}$ is now, indeed, a set as $\mathcal{F}$ is fixed by the APG environment $env_{\mathcal{U}}$ and q_0, V_0, δ, α are limited by $\mathcal{F}$, $\mathcal{S}tates$, $\mathcal{V}ar$. In fact, the state universe may be chosen here as any infinite set, however we decide for this to be easily able to compose automata according our definition and sometimes to switch variables into states.

The partial algebra on $\mathbb{A}$ is now defined as $\langle \mathbb{A}, \Phi \rangle$ where Φ stands for a set of operations that will be exposed by following type constructions.

Let's put it all together. We start giving our constructions with the following initial setting.

$\boldsymbol{\mathcal{U}}$ is any value universe

$\boldsymbol{env_{\mathcal{U}}}$ is any $\mathcal{F}$-APG environment (where $\mathcal{F}$ is a ranked alphabet)

$\boldsymbol{\mathcal{E}xp}$ =
(in other words, the initial set of expressions is empty)

$[\![]\!] = \varnothing$

$\boldsymbol{\Phi} = \varnothing$

Then by putting a similar frame per each construction we extend our type system as follows. In first two paragraphs we give some restrictions on $\mathcal{U}$ and $env_{\mathcal{U}}$ (that includes restrictions on $\mathcal{F}$). Then we assume that there is an ABNF rule $\mathcal{E}xp = \phi$, an automata algebra $\langle \mathbb{A}, \Phi \rangle$ and a set of equalities Ψ on $\mathcal{E}xp$ that fixes the semantic function $[\![]\!]$ given for the type system that we are about to extend. So in the three paragraphs we extend this items by putting $\mathcal{E}xp = \ldots$ / *some clause*, an equalities Eq on $[\![]\!]$ and $\Phi = \cdots \cup$ *some operations*, respectively, what means that we define the extended system where the new set of expressions is $\mathcal{E}xp = \phi$ / *some clause*, the new expression semantics $[\![]\!]$ is given by the equalities $\Psi \cup Eq$ and the new automata algebra is $\langle \mathbb{A}, \Phi \cup \textit{some operations} \rangle$.

In parallel with introducing type constructions, we also perform a "step-by-step" proof of equivalence between expression and automata formalization. Saying more precisely, we give the following theorem and per each type construction we perform a step of two inductions that prove it.

Theorem 3.11. *Let $\mathcal{U}$, $env_{\mathcal{U}}$, $\mathcal{E}xp$, $[\![]\!]$, Φ be given with type constructions in the following sections. Then for each $T \in 2^{\mathcal{U}}$ and $val \in \mathcal{C}$, an expression e such that $[\![e]\!]_{val} = T$ exists if and only if there exists an automaton $\mathcal{A}$ generated by $\varnothing$ in $\langle \mathbb{A}, \Phi \rangle$ such that $\mathcal{A}^{env_{\mathcal{U}}}[val] = T$.*

A "blueprint" of the proof may be described like this.

We do one induction on the expression e, to show the left-to-right implication and we do second induction on the algebraic expression describing the automaton $\mathcal{A}$, to show the opposite implication. Then per each type construction we give a lemma that proves the corresponding cases of the inductions assuming the theorem is true for smaller expressions.

Corollary 3.12. *If $\mathcal{U}$, $env_{\mathcal{U}}$, $\mathcal{E}xp$, $[\![\,]\!]$, Φ are given with our type constructions, then the type system*

$$\left\{ T \;\middle|\; \exists_e \; [\![e]\!]_{\varnothing} = T \right\}$$

is equal to the system

$$\{ \mathcal{A}^{env_{\mathcal{U}}} \mid \mathcal{A} \text{ is generated by } \varnothing \text{ in } \langle \mathbb{A}, \Phi \rangle \}$$

3.3.2 Base types

Base types are the concept to give a predefined fixed sets of values as types in a type system. Prevalent ones of these are e.g. the sets of natural, integer and rational numbers, or subsets of them[1], the set of boolean values, some sets of characters or sets of strings of characters.

In the expression formalization these types are commonly represented as constant symbols — like Nat, Int, Bool, etc. — with the natural semantics that maps them to corresponding predefined sets. In case of automata things may be more tricky, as sometimes we can end up in an urge of infinite alphabets. For example, if we have a type for all natural numbers, doing things straightforward may come with setting all of them as symbols in the alphabet. Still it is not a big threat, as commonly all base types are disjoint and in such case we can deal with that easily. Precisely, we can set the symbol function $\imath$ for all entities of a base type to the same 0-ary symbol and thus narrow the number of symbols to the number of base types in a system. In the case where base types are not disjoint, it is also possible to deal with the problem, similarly as we did in Ex 3.8, but we will leave it as an exercise for readers.

Definition 3.13. We extend a type system with a new base type B as follows.

$B \subseteq \mathcal{U}$

Some 0-ary symbol $b \in \mathcal{F}$, and $\imath(v) = b$ in $env_{\mathcal{U}}$ if and only if $v \in B$

$\mathcal{E}xp = \ldots \;/\; \mathsf{b}$
where b is a constant symbol

$[\![\mathsf{b}]\!]_{val} = B$

$\Phi = \cdots \cup \{B_{aut}\}$
where $B_{aut} = \left\langle \mathcal{F}, \{0\}, \varnothing, 0, \varnothing, \left\{ 0 \xrightarrow{b} () \right\}, \bot \right\rangle$

Note that in this case B_{aut} is a constant in the automata algebra.

Lemma 3.14. *The cases of Theorem 3.11 for $e = \mathsf{b}$ and $\mathcal{A} = B_{aut}$.*

Proof. This flows trivially from $[\![\mathsf{b}]\!]_{val} = B_{aut}^{env_{\mathcal{U}}}[val]$. ☐

3.3.3 Pairs

Pairs are the most basic composed values. We can see them as ordered pairs of values in the set theoretical meaning. Still it is not always proper, while in some systems, we can have values which are infinitely nested pairs, e.g. $(1, (1, (1, \ldots)))$ and this is not allowed by the ZFC theory — precisely by the axiom of regularity. However, such a point of view seems proper as an intuition.

For the purpose of this paper, we give a workaround of the problem of infinitely deep pairs by introducing an injection $\langle\, , \,\rangle : S \times S \to S$, for some set S that includes $\mathcal{U}$, which mimic the set theory $(\, , \,)$. The set S is unveiled here to remark that we can construct pair-like objects not only from values in $\mathcal{U}$. In this case the set of all pair values in $\mathcal{U}$ is described as $\{ \langle x, y \rangle \mid x, y \in \mathcal{U} \}$.

As we can expect, types for pair values correspond intuitively to Cartesian products. Still, as the values may not be set-like pairs, we need to give a workaround also here to avoid a collision with the regularity axiom. We are doing it by introducing another injection $\bar{\times} : 2^S \times 2^S \to 2^S$

[1] The subsets are caused by implementational reasons in computer systems. In computers each base type has often some fixed finite bound on space, which can be used to represent a value of this type, and so only finitely many of values can be arranged per type.

that, of course, mimic the set theory $\times$ and so $A\bar{\times}B = \{ \langle x, y\rangle \mid x \in A, y \in B \}$. The pair type of types T_1, T_2 is then just $T_1\bar{\times}T_2$. In type expressions the symbol $\bar{\times}$ is usually denoted by $*$.

Definition 3.15. We extend a type system with pair types as follows.

$\mathcal{U}\bar{\times}\mathcal{U} \subseteq \mathcal{U}$

The 2-ary symbol $\langle\, ,\, \rangle \in \mathcal{F}$, and $\imath(v) = \langle\, ,\, \rangle$ and $v_{\downarrow 1} = u_1$, $v_{\downarrow 2} = u_2$ in $env_{\mathcal{U}}$ if and only if $v = \langle u_1, u_2\rangle \in \mathcal{U}\bar{\times}\mathcal{U}$

$\mathcal{E}xp = \dots \,/\, \mathcal{E}xp * \mathcal{E}xp$

$[\![e_1 * e_2]\!]_{val} = [\![e_1]\!]_{val} \bar{\times} [\![e_2]\!]_{val}$

$\Phi = \dots \cup \{\times_{aut}\}$
where $\times_{aut}$ is the 2-ary operation $\times_{aut}(\mathcal{A}_1, \mathcal{A}_2) = \mathcal{A}_{\times}[x \mapsto \mathcal{A}_1, y \mapsto \mathcal{A}_2]$ and $\mathcal{A}_{\times} = \left\langle \mathcal{F}, \{0\}, \{x, y\}, 0, \varnothing, \left\{0 \xrightarrow{\langle\,,\,\rangle} (x, y)\right\}, \bot \right\rangle$ for some distinct variables x, y

---

Lemma 3.16. *The cases of Theorem 3.11 for* $e = e_1 * e_2$ *and* $\mathcal{A} = \times_{aut}(\phi_1, \phi_2)$*, where* ϕ_1, ϕ_2 *are algebraic expressions describing some automata* $\mathcal{A}_1, \mathcal{A}_2$*.*

Proof. By the hypothesis we have that for e_1, e_2 there exist $\mathcal{A}_1, \mathcal{A}_2$ such that $[\![e_1]\!]_{val} = \mathcal{A}_1^{env_{\mathcal{U}}}[val]$ and $[\![e_2]\!]_{val} = \mathcal{A}_2^{env_{\mathcal{U}}}[val]$, and inversely for $\mathcal{A}_1, \mathcal{A}_2$ there exist such e_1, e_2. Then if we note that for any $val' = \{x \mapsto T_1, y \mapsto T_2, \dots\}$ we have $\mathcal{A}_{\times}^{env_{\mathcal{U}}}[val'] = T_1\bar{\times}T_2$, the equality $[\![e_1 * e_2]\!]_{val} = \times_{aut}(\mathcal{A}_1, \mathcal{A}_2)^{env_{\mathcal{U}}}[val]$ flows straightly from the lemma about the automata composition semantics 2.25. ☐

3.3.4 Untagged unions

Here we propose some non-standard concept. An untagged union is a type standing for a set union of two (or more) types. Such unions, in opposition to tagged ones, cause subtyping computations to be not obvious, what may be the cause of resignation from them in general usage. However, the proposed automaton formalization can beat this problem.

The common symbols for unions in type expressions are | and +. We will use the first one here and the second for tagged unions to distinguish them.

Definition 3.17. We extend a type system with untagged unions as follows.

There are no additional requirements on $\mathcal{U}$ and $env_{\mathcal{U}}$.

$\mathcal{E}xp = \dots \,/\, \mathcal{E}xp \mid \mathcal{E}xp$

$[\![e_1|e_2]\!]_{val} = [\![e_1]\!]_{val} \cup [\![e_2]\!]_{val}$

$\Phi = \dots \cup \{|_{aut}\}$
where $|_{aut}$ is the 2-ary operation $|_{aut}(\mathcal{A}_1, \mathcal{A}_2) = \mathcal{A}_{|}[x \mapsto \mathcal{A}_1, y \mapsto \mathcal{A}_2]$ and $\mathcal{A}_{|} = \langle \mathcal{F}, \{0\}, \{x, y\}, 0, \{x, y\}, \varnothing, \bot\rangle$ for some distinct variables x, y

---

Lemma 3.18. *The cases of Theorem 3.11 for* $e = e_1|e_2$ *and* $\mathcal{A} = |_{aut}(\phi_1, \phi_2)$ *where* ϕ_1, ϕ_2 *are algebraic expressions describing some automata* $\mathcal{A}_1, \mathcal{A}_2$*.*

Proof. Again by the hypothesis we have that for e_1, e_2 there exist $\mathcal{A}_1, \mathcal{A}_2$ such that $[\![e_1]\!]_{val} = \mathcal{A}_1^{env_{\mathcal{U}}}[val]$ and $[\![e_2]\!]_{val} = \mathcal{A}_2^{env_{\mathcal{U}}}[val]$, and inversely for $\mathcal{A}_1, \mathcal{A}_2$ there exist such e_1, e_2. Then if we note that for any $val' = \{x \mapsto T_1, y \mapsto T_2, \dots\}$ we have $\mathcal{A}_{|}^{env_{\mathcal{U}}}[val'] = T_1 \cup T_2$, we again get the straight equality $[\![e_1|e_2]\!]_{val} = |_{aut}(\mathcal{A}_1, \mathcal{A}_2)^{env_{\mathcal{U}}}[val]$ by the composition semantics lemma 2.25. ☐

The computation problem of type-checking expressions with untagged unions comes from the fact that we cannot perform a structural induction on type expressions to test <: . Why? Let's take a look at the following example. Suppose that we want to check $e_1 * (e_2|e_3)$ <: $(e_1 * e_2)|(e_1 * e_3)$ for some expressions e_1, e_2, e_3. This is surely true as the both sides of <: stand for the same type. But now, if we would like to use the mentioned induction, then, briefly saying, we cannot perform it on the left expression, as the right one is not a product, and if we do it on the right one, we have to compare $e_1 * (e_2|e_3)$ with $e_1 * e_2$ and $e_1 * e_3$ in the result, where both expressions $e_1 * e_2$ and $e_1 * e_3$ represent strictly smaller types then the former one. Hence the induction would fail. Fortunately, we can cope with this problem easily by the automata formalization, as untagged unions map straightly to non-determinism of automata and checking inclusion between non-deterministic automata is decidable (cf. sec. 2.2.4).

Remark 3.19. If we resign from this kind of unions, then automata representing types become determ-

inistic.

It is worth to note that untagged unions may be supportive in static type-checking of so-called dynamically typed programming languages, as we mentioned in the introduction. For example, the python expression `sqrt (x) if x >= 0 else None` , which stands for `None` if `x` is negative and for the square root of `x` otherwise, is not typable in most common type systems, but it can have a type `float | NoneType` when we use untagged unions. Another advantage of these unions is that they can significantly shrink expressions describing values in formal statements (e.g., programs) in comparison to their tagged version — we will put an example of this in the next section.

3.3.5 Tagged unions

Tagged unions are generally used alternative for the untagged ones. They are much easier to type-check, but require adding extra constructions to a system. More precisely, we have to mark each value of such a type. Assuming that we have only binary unions and the marks are set as *Left* and *Right*, and they are included in the set S given for the pair operator $\langle\,,\,\rangle$, then the union type of types T_1, T_2 is the set of pair values $\{Left\} \bar{\times} T_1 \cup \{Right\} \bar{\times} T_2$ — again, we cannot use simple set-theory pairs here, due to the same reason as in Section 3.3.3.

Definition 3.20. We extend a type system with tagged unions as follows.

$\{Left\} \bar{\times} \mathcal{U} \cup \{Right\} \bar{\times} \mathcal{U} \subseteq \mathcal{U}$ and
$Left, Right \notin \mathcal{U}$

The unary symbols $InL, InR \in \mathcal{F}$,
$\imath(v) = InL$ and $v_{\downarrow 1} = u$ in $env_{\mathcal{U}}$ if and only if
$v = \langle Left, u\rangle \in \{Left\} \bar{\times} \mathcal{U}$,
$\imath(v) = InR$ and $v_{\downarrow 1} = u$ in $env_{\mathcal{U}}$ if and only if
$v = \langle Right, u\rangle \in \{Right\} \bar{\times} \mathcal{U}$

$\mathcal{E}xp = \ldots \;/\; \mathcal{E}xp \;+\; \mathcal{E}xp$

$$[\![e_1 + e_2]\!]_{val} = \{Left\} \bar{\times} [\![e_1]\!]_{val} \cup \{Right\} \bar{\times} [\![e_2]\!]_{val}$$

$\Phi = \cdots \cup \{+_{aut}\}$
where $+_{aut}$ is the 2-ary operation $+_{aut}(\mathcal{A}_1, \mathcal{A}_2) = \mathcal{A}_+[x \mapsto \mathcal{A}_1, y \mapsto \mathcal{A}_2]$ and $\mathcal{A}_+ = \left\langle \mathcal{F}, \{0\}, \{x, y\}, 0, \varnothing, \left\{0 \xrightarrow{InL} x,\ 0 \xrightarrow{InR} y\right\}, \bot \right\rangle$
for some distinct variables x, y

Note that the definition does not collide with the definition of pair types as $Left, Right \notin \mathcal{U}$ and so $\imath(\langle Left, u\rangle), \imath(\langle Right, u\rangle)$ do not need to be $\langle\,,\,\rangle$.

Lemma 3.21. *The cases of Theorem 3.11 for* $e = e_1 + e_2$ *and* $\mathcal{A} = +_{aut}(\phi_1, \phi_2)$ *where* ϕ_1, ϕ_2 *are algebraic expressions describing automata* $\mathcal{A}_1, \mathcal{A}_2$.

Proof. This is analogous to the previous proofs. □

Tagged unions are good for algorithmic reasons, as we can do a structural induction on expressions to check subtyping. In this approach a type $e_1 + e_2$ can be a subtype only of another type of the form $e_3 + e_4$, so we can perform an induction and compare e_1 with e_3 and e_2 with e_4 to check $<:$. The example from the previous section is not a threat any more, as $exp_1 * (exp_2 + exp_3)$ and $(exp_1 * exp_2) + (exp_1 * exp_3)$ represent now two completely distinct types, where the first describes the set $[\![exp_1]\!] \bar{\times} (\{Left\} \bar{\times} [\![exp_2]\!] \cup \{Right\} \bar{\times} [\![exp_3]\!])$ and the second $\{Left\} \bar{\times} ([\![exp_1]\!] \bar{\times} [\![exp_2]\!]) \cup \{Right\} \bar{\times} ([\![exp_1]\!] \bar{\times} [\![exp_3]\!])$. Note also that such a construction is deterministic in terms of automata, what seriously improves sybtyping computations in the case of this formalism when compared to the untagged unions.

These things surely stand in favour for tagged unions. But on the other hand, these unions are not good for practical reasons, as they blow length of expressions that describe values of union types. For example, an expression in a formal statement that stands for a value of the type `Bool | Nat | Int` in the untagged approach can be just the boolean `true`, while when using the tagged one, it must be something like `left (left (true))`. This issue should be especially taken in mind when we are going to give a type system for a programming language, where longer code is generally less maintainable and makes a higher chance of mistakes.

3.3.6 Recursive types

Here we go with a heavier construction. Intuitively saying, recursive types are limits of iterations of some functions $F : 2^{\mathcal{U}} \to 2^{\mathcal{U}}$ on some initial types that satisfy the equation $X = F(X)$. These limits can be depicted in our case as

$$\bigcup_i F^i(T_0) \equiv \bigcup_i \overbrace{F \circ \cdots \circ F}^{i}(T_0) \tag{3.1}$$

and

$$\bigcap_i F^i\left(T_0'\right) \equiv \bigcap_i \overbrace{F \circ \cdots \circ F}^{i}\left(T_0'\right) \tag{3.2}$$

where T_0, T_0' stand for initial types. This is, however, not very precise, as the domain of i is not specified. Naturally, we would like to say that i ranges through natural numbers. Still for some functions this is not enough and to obtain their fixed points, we need to perform a transfinite iteration, where i ranges through ordinal numbers — we will explain this in more detail in a moment. Moreover, even equipped with transfinite iterations we cannot give recursive types for all Fs, as for some of them there may be no fixed points in the mentioned limit form, or there may be no fixed point at all. So to guarantee existence of these types we will restrict ourselves here only to Fs that are monotone wrt to $\subseteq$. We will see why it is going to work in the following subsection.

Transfinite iterations

For the purpose of this paper we present here two kinds of transfinite iterations — that is, increasing and decreasing ones. For both we need a monotonic function F on a complete lattice A. Now if we denote the least upper bound of $X \subseteq A$ by $\bigvee X$, the greatest lower bound by $\bigwedge X$, and the least and the greatest elements in A by, respectively, $\bot$ and $\top$, the iterations can be given as follows.

Definition 3.22. The increasing (respectively, decreasing) ***transfinite iteration*** of an monotonic function $F : A \to A$ is the sequence of elements $a_\alpha = F^\alpha(\bot)$ (resp., $a_\alpha = F^\alpha(\top)$) indexed by ordinal numbers α, where $F^0(x) = x$, $F^{\alpha+1}(x) = F(F^\alpha(x))$ for $x = \bot, \top$ and $F^\eta(\bot) = \bigvee_{\beta<\eta} F^\beta(\bot)$, $F^\eta(\top) = \bigwedge_{\beta<\eta} F^\beta(\top)$ when η is a limit ordinal.

We may note here that the sequences are not sets, but in our case it does not matter — we may just think about them as classes.

In this place we should probably mention that the above definition is not the only proper one. Actually, even though the described iterations are, in fact, often called increasing and decreasing ones, they are rather a normalized form of these. In general we can put any a such that $a \le F(a)$ in the place of $\bot$ and any a such that $F(a) \le a$ in the place of $\top$, and the defined classes of transfinite iterations stay the same up to the first elements of sequences in them. We state this more precisely in the following observation.

Observation 3.23. *Let's assume that we use the definition of transfinite iterations with arbitrary initial elements as stated above and let $\{a_\alpha\}_\alpha$ be any increasing (resp., decreasing) iteration of a function F. Then the increasing (resp., decreasing) iteration $\{b_\alpha\}_\alpha$ of the function G defined as $G(x) = F(x \vee a_0)$ (resp., $G(x) = F(x \wedge a_0)$) where the initial element b_0 is set to $\bot$ (resp., $\top$) is equal to $\{a_\alpha\}_\alpha$ for all $\alpha > 0$.*

Now, an important for us fact about increasing and decreasing iterations is that they always stabilize after some ordinal number of steps and so they appoint fixed points of the iterated functions. We put it precisely in the following piece of general knowledge.

Theorem 3.24. *For each increasing and decreasing transfinite iteration of a function F there exists an ordinal β such that the element a_β is a fixed point of F. Moreover, in the case of an increasing iteration we have that a_β is the least fixed point of F and in the case of a decreasing one we have that a_β is the greatest fixed point of F.*

Later we will refer to the least β satisfying the theorem as, respectively, "the increasing ***fixing number*** of F" and "the decreasing fixing number of F".

Recursive types definition

Let's go back to the previous unfinished definition of recursive types. To ensure existence of types that satisfies it, we restrict F to monotonic functions, as mentioned, and then treat $F^i(T_0)$ as elements of the increasing transfinite iteration of F; and similarly $F^i(T_0')$ as elements of the decreasing iteration. Of course, in this case these sequences follow rather the more general definition, where T_0 and T_0' correspond to the arbitrary initial element a. So to proceed with our definition we put another restriction and we fix T_0 and T_0' to be, respectively, $\varnothing$ and $\mathcal{U}$ — later we will show that if a type system posses some type constructions (untagged unions and intersections) we do not lose anything by doing so.

Now, if we set the range of i to $\{0, \ldots, \beta_1\}$ in Equation 3.1 and to $\{0, \ldots, \beta_2\}$ in Equation 3.2, where β_1, β_2 are, respectively, the least and the greatest fixing number of F, the previous incomplete definition of recursive types becomes valid. This means that our recursive types are nothing

more then just the least and the greatest fixed points of monotonic functions $F: 2^{\mathcal{U}} \to 2^{\mathcal{U}}$ (in inclusion order). To put it simple in symbols we will use μF to denote the least fixed point of F and νF to denote the greatest one.

But let's going back on Earth, as we still have a very primary issue to take care about. That is, we need to somehow represent functions F in our formalizations. This is fortunately not very hard, as in the case of expressions we can use the common approach of defining a function by writing an expression with variables that describes it; and in the case of automata we can use the concept of variables that we introduced to them to achieve the same effect.

This is enough for a general definition, but, unfortunately, our automata formalization still makes an urge to clarify one thing. As we have mentioned in preliminaries, when we want to compute inclusion-checking between automata, the environment $env_{\mathcal{U}}$ must be weakly closed over those in our formalization. Until we have automata that generate greatest fixed points this may be easily shown by induction on depth of automata's runs. However, with the introduction of greatest recursive types we lose this possibility, as depth of runs may become infinite. To fix it we must guarantee some additional values to be in $\mathcal{U}$. As a direct restriction on $\mathcal{U}$ may be quite technical, we will do this by the brutal request on $env_{\mathcal{U}}$ to include all finite $\mathcal{F}$-APGs up to APG isomorphism. A practical consequence of this is that, some values with infinite tree representations — like the one depicted in Example 3.26 — will appear in $\mathcal{U}$. For the purpose of this paper we will call such values ***coninductive*** (which is more or less proper, but it somehow reflects the idea of using an entity to define the same entity, what can be seen a foundation of these values).

In the below construction we write $\{\langle L_1, U_1\rangle, ..., \langle L_n, U_n\rangle\}_\alpha$ for a Rabin accepting condition fixed by the listed state pairs. Note that a Rabin condition $\{\langle L_1, U_1\rangle, ..., \langle L_n, U_n\rangle\}_\alpha$ is equivalent to $\{\langle L_1, U_1\rangle, ..., \langle L_n, U_n\rangle, \langle \varnothing, \varnothing\rangle\}_\alpha$, so in the case of νx_{aut} operations, if $L_n \neq \varnothing$, we may always add the empty accepting pair to the accepting condition of $\mathcal{A}$. Note also that the fixed points $\mu F, \nu F$ in the following definition always exist as the semantics of expressions for each type construction that we give in this paper is monotonic wrt valuations of variables.

Definition 3.25. We extend a type system with the least and greatest recursive types as follows.

There is no requirement on $\mathcal{U}$.

For any finite $\mathcal{F}$-APG there is an isomorphic $\mathcal{F}$-APG in $env_{\mathcal{U}}$.[2]

$\mathcal{E}xp = \ldots \;/ \mathcal{V}ar \;/\; \mu\mathcal{V}ar.\mathcal{E}xp \;/\; \nu\mathcal{V}ar.\mathcal{E}xp$

$$[\![x]\!]_{val} = \begin{cases} val(x), & \text{if } x \in Dom(val) \\ \text{undefined}, & \text{otherwise} \end{cases}$$

$[\![\mu x.e]\!]_{val} = \mu F$
$[\![\nu x.e]\!]_{val} = \nu F$
where $F: 2^{\mathcal{U}} \to 2^{\mathcal{U}}$ and $F(X) = [\![e]\!]_{val[x \mapsto X]}$

$\Phi = \ldots \cup \bigcup_{x \in \mathcal{V}ar} \{x_{aut}, \mu x_{aut}, \nu x_{aut}\}$
where $x_{aut} = \langle \mathcal{F}, \{0\}, \{x\}, 0, \{x\}, \varnothing, \bot\rangle$ and $\mu x_{aut}, \nu x_{aut}$ are unary operations as follows.

For any automaton $\mathcal{A}$ and variable x let

$$\delta'_{\mathcal{A},x} = \Big\{ q \xrightarrow{a} (p_1, ..., p_{ar(a)}) \;\Big|\; q \xrightarrow{a} \overrightarrow{s} \in \delta_{\mathcal{A}} \wedge \forall_{i=1}^{ar(a)} (s_i \neq x \Rightarrow p_i = s_i) \wedge (s_i = x \Rightarrow p_i \in \{x\} \cup V_{0_{\mathcal{A}}}) \Big\}.$$

Let $\{\langle L_1, U_1\rangle, ..., \langle L_n, U_n\rangle\}_\alpha$ be the Rabin acc. condition of $\mathcal{A}$. Then

$$\mu x_{aut}(\mathcal{A}) = \langle \mathcal{F}, Q_{\mathcal{A}} \cup \{x\}, V_{\mathcal{A}} \setminus \{x\}, q_{0_{\mathcal{A}}}, V_{0_{\mathcal{A}}} \setminus \{x\}, \delta'_{\mathcal{A},x} \cup \Big\{ x \xrightarrow{a} \overrightarrow{q} \mid q_{0_{\mathcal{A}}} \xrightarrow{a} \overrightarrow{q} \in \delta'_{\mathcal{A},x} \Big\}, \{\langle L_1 \cup \{x\}, U_1\rangle, ..., \langle L_n \cup \{x\}, U_n\rangle\}_\alpha \rangle.$$

If $x \in V_{0_{\mathcal{A}}}$, then

$$\nu x_{aut}(\mathcal{A}) = \Big\langle \mathcal{F}, \{0\}, \varnothing, 0, \varnothing, \Big\{ 0 \xrightarrow{a} \overbrace{(0, ..., 0)}^{ar(a)} \mid a \in \mathcal{F} \Big\}, \top \Big\rangle.$$

If $x \notin V_{0_{\mathcal{A}}}$, then let $\{\langle L_1, U_1\rangle, ..., \langle \varnothing, U_n\rangle\}_\alpha$ be the Rabin acc. condition of $\mathcal{A}$ and

$$\nu x_{aut}(\mathcal{A}) = \langle \mathcal{F}, Q_{\mathcal{A}} \cup \{x\}, V_{\mathcal{A}} \setminus \{x\}, q_{0_{\mathcal{A}}}, V_{0_{\mathcal{A}}}, \delta'_{\mathcal{A},x} \cup \Big\{ x \xrightarrow{a} \overrightarrow{q} \mid q_{0_{\mathcal{A}}} \xrightarrow{a} \overrightarrow{q} \in \delta'_{\mathcal{A},x} \Big\}, \{\langle L_1, U_1\rangle, ..., \langle \varnothing, U_n \cup \{x\}\rangle\}_\alpha \rangle.$$

---

[2] This retrogradely enforces coinductive values to be in $\mathcal{U}$.

Example 3.26. Two figures illustrating a coinductive value, which in this case is the circular list of 1s.

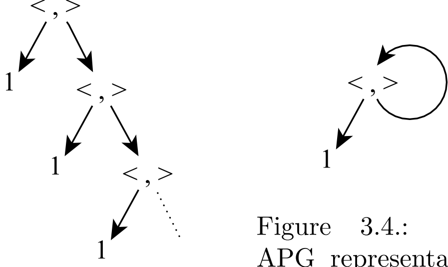


Figure 3.3.: A tree representation of a circular list of 1s

Figure 3.4.: An APG representation of a circular list of 1s

Saying it in a more natural way — the case of $\nu x_{aut}(\mathcal{A})$ when $x \in V_{0\mathcal{A}}$ is trivial. The greatest set x is simply the set of all values in the universe. In the case of $\mu x_{aut}(\mathcal{A})$ and $\nu x_{aut}(\mathcal{A})$ with $x \notin V_{0\mathcal{A}}$ we make x a state of the automaton and intuitively add ε-transitions from x to the initial state and initial variables of $\mathcal{A}$, and then we properly put x into the acceptance condition. If we depict the acc. condition of an automaton $\mathcal{A}$ as a formula over operators $\wedge$, $\vee$, $\neg$ and literals "$q \in P$", where q are states of $\mathcal{A}$ and P is the argument of $\alpha_{\mathcal{A}}$, then this "proper putting" literally means setting $\alpha_{\mu x_{aut}(\mathcal{A})}(P) = \alpha_{\mathcal{A}}(P) \wedge \neg x \in P$ and $\alpha_{\nu x_{aut}(\mathcal{A})}(P) = \alpha_{\mathcal{A}}(P) \vee x \in P$.

Lemma 3.27. *The cases of Theorem 3.11 for* $e = x$ *and* $\mathcal{A} = x_{aut}$.

Proof. Trivial. □

Lemma 3.28. *The cases of Theorem 3.11 for* $e = \mu x.e'$ *and* $\mathcal{A} = \mu x_{aut}(\phi)$*, and respectively for* $e = \nu x.e'$ *and* $\mathcal{A} = \nu x_{aut}(\phi)$*, where* ϕ *is algebraic expressions describing some automaton* $\mathcal{A}'$.

Proof. In the paper [Niw97] (Lemma 3.4, 3.5) it was shown that for any NDA $\mathcal{A}$ and variable x we have automata $\mathcal{A}_{\mu x}$, $\mathcal{A}_{\nu x}$ (respectively, $\mathcal{A}'$, $\mathcal{A}''$ in [Niw97]) that evaluate the least and the greatest fixed points of the function described by $\mathcal{A}$ wrt a valuation of the variable x. Precisely saying, the paper states that for any $\mathcal{F}$-APG environment env with an underlying set N and any valuation of variables $V_{\mathcal{A}} \setminus \{x\}$ into 2^N, we have the equalities $\mathcal{A}_{\mu x}^{env}[val] = \mu F$ and $\mathcal{A}_{\nu x}^{env}[val] = \nu F$, where $F : 2^N \to 2^N$ is the monotonic function defined as $F(X) = \mathcal{A}^{env}[val[x \mapsto X]]$. Still, it appears that there is a missed case in the construction of $\mathcal{A}_{\mu x}$, $\mathcal{A}_{\nu x}$ — what we put more precisely in Appendix B — and to fix it we need to inject our $\delta'_{\mathcal{A},x}$ into these automata. So in the result we obtain the above claim for $\mu x_{aut}(\mathcal{A})$ in the place of $\mathcal{A}_{\mu x}$ and $\nu x_{aut}(\mathcal{A})$ in the place of $\mathcal{A}_{\nu x}$.

Now the lemma may be shown in a similar manner as the previous ones. That is, by the hypothesis we have that for e' there exists $\mathcal{A}'$ such that $[\![e']\!]_{val[x \mapsto T]} = \mathcal{A}'^{env_{\mathcal{U}}}[val[x \mapsto T]]$ for any type T, and inversely for $\mathcal{A}'$ there exist such e'. But then both e' and $\mathcal{A}'$ describe the same function $F(X) = [\![e']\!]_{val[x \mapsto X]} = \mathcal{A}'^{env_{\mathcal{U}}}[val[x \mapsto X]]$. So by the Niwiński's lemmas we have that $[\![\mu x.e']\!]_{val} = \mu x_{aut}(\mathcal{A})^{env_{\mathcal{U}}}[val]$ and $[\![\nu x.e']\!]_{val} = \nu x_{aut}(\mathcal{A})^{env_{\mathcal{U}}}[val]$. □

Of course we can resign from the μ or ν operation in the above construction, when we do not need one. By doing so we will obtain type systems with recursive types corresponding to those usually used today (June 2014) in programming languages. Precisely, systems with only μ will correspond to those of eager languages and systems with only ν to those of lazy ones.

Unfortunately the proposed construction is very heavy for type-checking computations, as we mostly require automata with Rabin condition here. That is, it was shown in [Niw97] (Theorem 3.4) that in our case for any Rabin NDA over the alphabet $\{\langle\,,\,\rangle/2\}$, there exists an equivalent type expressions build out of pair, untagged union and recursive operators. Thus, if only we have pairs, untagged unions and recursive types in a system, then the accepting condition for automata in our formalization cannot be weakened. Moreover, such recursive types are hard in understanding their semantics, what is actually a serious issue in practical usage of types. That is why in Section 3.3.8 we will propose a weaker version of recursive types that are easier in both, understanding and computations.

Example 3.29. Why do recursive types are hard to understand? Let's take a look at the types $\mu x.\nu y.(\mathsf{Unit}|x) * y$ and $\nu y.\mu x.(\mathsf{Unit}|x) * y$, where Unit stands for the singleton of a value unit, the precedence of μ, ν is like this of quantifiers and we have coinductive values in the universe. Despite the little difference in terms the semantics of both are not the same. Still it may be not obvious, how these types do differ. To figure it out we will take a look on the automata representations of these

type, below.

(Recall that states of automata in our automata universe $\mathbb{A}$ are sequences — what is caused by the usage of automata substitutions. So 00 stands for the 2-element sequence of 0 and 0, 100 stands for the 3-element sequence of 1, 0 and 0, and so on. Thus, if a state comes from a result of an automata composition, then its first digit is the number of the origin automaton and the rest of the sequence is just a state in this automaton.)

The automata for the types $\mu x.\nu y.(\mathsf{Unit}|x) * y$, $\nu y.\mu x.(\mathsf{Unit}|x) * y$ are build according to the automata algebra in a similar way. First we set

$$\begin{aligned}
\mathcal{F} &= \{unit/0, \langle\,,\,\rangle/2\} \\
\mathsf{Unit}_{aut} &= \left\langle \mathcal{F}, \{0\}, \varnothing, 0, \varnothing, \left\{0 \xrightarrow{unit} ()\right\}, \bot \right\rangle \\
x_{aut} &= \langle \mathcal{F}, \{0\}, \{x\}, 0, \{x\}, \varnothing, \bot \rangle \\
y_{aut} &= \langle \mathcal{F}, \{0\}, \{y\}, 0, \{y\}, \varnothing, \bot \rangle
\end{aligned}$$

Then from the first two automata we produce by the operation $|_{aut}$

$$\begin{aligned}
\mathcal{A}_{\mathsf{Unit}|x} = \Big\langle & \mathcal{F}, \{00, 10, 20\}, \{x\}, 00, \{x\}, \\
& \left\{00 \xrightarrow{unit} (), 10 \xrightarrow{unit} ()\right\}, \bot \Big\rangle
\end{aligned}$$

Then from $\mathcal{A}_{\mathsf{Unit}|x}$ and y_{aut} by the operation $*_{aut}$ we produce

$$\begin{aligned}
\mathcal{A}_{(\mathsf{Unit}|x)*y} = \Big\langle & \mathcal{F}, \{00, 100, 110, 120, 20\}, \{x, y\}, 0, \varnothing, \\
& \Big\{100 \xrightarrow{unit} (), 110 \xrightarrow{unit} (), \\
& 00 \xrightarrow{\langle\,,\,\rangle} (100, 20), 00 \xrightarrow{\langle\,,\,\rangle} (100, y), \\
& 00 \xrightarrow{\langle\,,\,\rangle} (x, 20), 00 \xrightarrow{\langle\,,\,\rangle} (x, y)\Big\}, \bot \Big\rangle
\end{aligned}$$

Finally after applying μx_{aut} and νx_{aut} in the proper order to $\mathcal{A}_{(\mathsf{Unit}|x)*y}$, removing superfluous states and some human-friendly states renaming we receive two automata of the form

$$\begin{aligned}
& \langle \mathcal{F}, \{q_0, q_1, x, y\}, \varnothing, q_0, \varnothing, \\
& \Big\{q_0 \xrightarrow{\langle\,,\,\rangle} (q_1, y), q_0 \xrightarrow{\langle\,,\,\rangle} (x, y), q_1 \xrightarrow{unit} (), \\
& y \xrightarrow{\langle\,,\,\rangle} (q_1, y), y \xrightarrow{\langle\,,\,\rangle} (x, y), \\
& x \xrightarrow{\langle\,,\,\rangle} (q_1, y), x \xrightarrow{\langle\,,\,\rangle} (x, y)\Big\}, \alpha \Big\rangle
\end{aligned}$$

where α for the first one is the Rabin condition $\{\langle\{x\}, \{y\}\rangle\}_\alpha$ and $\{\langle\varnothing, \{y\}\rangle\}_\alpha$ for the second one.

Now, when we study these automata we can settle the difference in types. (Recall that we index children in our APGs from 1, thus in the following 1 denotes a left-most child.) So, the both types stands for sets of values represented with APGs such that each node is labeled with *unit* or $\langle\,,\,\rangle$, each right child of a node is labeled with $\langle\,,\,\rangle$ and there are no branches ended with infinitely many 1s (that is there are no branches where from some point we always turn left). The difference between them is that in the first case the APGs additionally cannot have branches with infinitely many 1s on them at all.

As we can see by this example understanding general recursive types may be not very obvious and the analysis it requires is probably not what we expect to do when we use a type system.

3.3.7 Intersection types

Intersections are mostly a secondary concept in types. Shortly saying, we may define them just as set intersections of two (or more) types. Adding such types to a system generally does not bring new expressive power, but rather it allows to shrink expressions for recursive types. However, it appears that it is not a golden rule and there may be exceptions. Specifically, if we incorporate intersection types in a type system containing function types proposed in Section 3.3.9 the expressive power of the system grows — we remark this in more detail in Section 3.3.9.

In the following definition of intersection types we set $\mathbb{A}$ to be a universe of some variant of alternating automata. Apparently we cannot use here ordinary AAs as the starting condition of them appears to be too weak for this case. So we strengthen the definition as follows.

Enhancement of alternating automata

Let alternating automata be defined as before, but let q_0, V_0 in automata tuples be replaced with a collection of initial sets of states and variables. Precisely, let an AA be now a tuple $\langle \mathcal{F}, Q, V, I, \delta, \alpha \rangle$, where $\mathcal{F}, Q, V, \delta, \alpha$ are defined as before and $I \subseteq 2^{Q \cup V}$ is an initial condition. An APG is said to be accepted by such an automaton wrt some valuation if there exists an initial set $S \in I$ such that for each $s \in S$ there exists an accepting s-run of the automaton on the APG wrt the valuation, where runs and acceptance are again defined as before.

Beyond the above changes other terms related to AAs, like compositions or alternation removing (that we describe in the appendix), together with most of our automata operations stay valid after

some minor fixes. The only operations that do not apply straightly to such AAs are these that cannot be given in terms of automata composition. That is, recursive operations. To cope with this problem we present the following procedure that constructs equivalents of $\mu x_{aut}(\mathcal{A})$ and $\nu x_{aut}(\mathcal{A})$ for an enhanced alternating automaton $\mathcal{A}$.

Observation 3.30. *Let* $I_{\mathcal{A}} = \left\{ \{s_{i,j}\}_{j \in [m_i]} \right\}_{i \in [n]}$. *Then for any valuation* val *of* $\mathcal{A}$*'s variables*

$$\mathcal{A}^{env_{\mathcal{U}}}[val] = \bigcup_{i \in [n]} \bigcap_{j \in [m_i]} \mathcal{A}'_{i,j}[val]$$

where $\mathcal{A}'_{i,j} = \langle \mathcal{F}, Q, V, \{\{s_{i,j}\}\}, \delta, \alpha \rangle$.

The above observation allows to express the functions described by $\mathcal{A}$ wrt its variables as combinations of functions described by automata with a single initial state or a variable. Precisely, it shows that the function $F(X) = \mathcal{A}^{env_{\mathcal{U}}}[val, x \mapsto X]$ is equal to the function $G(X) = \bigcup_{i \in [n]} \bigcap_{j \in [m_i]} F'_{i,j}(X)$ where $F'_{i,j}(X) = \mathcal{A}'_{i,j}[val, x \mapsto X]$. Now by iterating the following lemma we can reduce $\mathcal{A}$ to an automaton $\mathcal{A}'$ where for all $S \in I_{\mathcal{A}'}$ we have $x \notin S$ or $S = \{x\}$ and it describes a function that has the same least or greatest fixed point as F, depending on a variant of the lemma that we use. (In the following, recall that functions described by our automata are monotonic.)

Lemma 3.31. *Let* $F : 2^{\mathcal{U}} \to 2^{\mathcal{U}}$ *be a function such that* $F(X) = X \cap F_1(X) \cup F_2(X)$ *for some monotonic* F_1, F_2. *Then each element* $F^{\alpha}(\varnothing)$ *of the increasing iteration of* F *is equal to the element* $F_2^{\alpha}(\varnothing)$ *of the increasing iteration of* F_2*; and each element* $F^{\alpha}(\mathcal{U})$ *of the decreasing iteration of* F *is equal to the element* $H^{\alpha}(\mathcal{U})$ *of the decreasing iteration of* H *given as* $H(X) = F_1(X) \cup F_2(X)$.

Proof. The first equality may be shown by a simple transfinite induction and the second by a transfinite induction strengthened by the two theses

$$F_1(H^{\alpha}(\mathcal{U})) \subseteq H^{\alpha}(\mathcal{U}),$$
$$F_2(H^{\alpha}(\mathcal{U})) \subseteq H^{\alpha}(\mathcal{U})$$

□

After such reduction of $\mathcal{A}$ we may attempt to use the following enhanced versions of μx_{aut}, νx_{aut} to obtain automata evaluating desired fixed points.

Let $P = Q_{\mathcal{A}'} \cap \bigcup I_{\mathcal{A}}$ be all states of $\mathcal{A}'$ that appear in some initial set and $p_1, \ldots, p_n$ be all elements of P. Then the enhanced operation μx_{aut} is defined as

$$\mu x_{aut}(\mathcal{A}') = \Big\langle \mathcal{F},\ Q_{\mathcal{A}'} \cup \{ix\}_{i=1}^{n},\ V_{\mathcal{A}'} \setminus \{x\},\ I_{\mathcal{A}'} \setminus \{\{x\}\},\ \delta'_{\mathcal{A}',x} \cup \bigcup_{i=1}^{n} \left\{ ix \xrightarrow{a} \vec{q} \mid p_i \xrightarrow{a} \vec{q} \in \delta'_{\mathcal{A}',x} \right\},\ \left\{ \left\langle L_1 \cup \{ix\}_{i=1}^{n}, U_1 \right\rangle, \ldots, \left\langle L_n \cup \{ix\}_{i=1}^{n}, U_n \right\rangle \right\}_{\alpha} \Big\rangle$$

where $\{\langle L_1, U_1 \rangle, \ldots, \langle L_n, U_n \rangle\}_{\alpha}$ is the Rabin acc. condition of $\mathcal{A}'$.

If $\{x\} \in I_{\mathcal{A}'}$ then the enhanced operation νx_{aut} is defined as

$$\nu x_{aut}(\mathcal{A}) = \Big\langle \mathcal{F}, \{0\}, \varnothing, \{\{0\}\},\ \Big\{ 0 \xrightarrow{a} \overbrace{(\{0\}, \ldots, \{0\})}^{ar(a)} \mid a \in \mathcal{F} \Big\}, \top \Big\rangle$$

otherwise

$$\nu x_{aut}(\mathcal{A}') = \Big\langle \mathcal{F},\ Q_{\mathcal{A}'} \cup \{ix\}_{i=1}^{n},\ V_{\mathcal{A}'} \setminus \{x\},\ I_{\mathcal{A}'},\ \delta'_{\mathcal{A}',x} \cup \bigcup_{i=1}^{n} \left\{ ix \xrightarrow{a} \vec{q} \mid p_i \xrightarrow{a} \vec{q} \in \delta'_{\mathcal{A}',x} \right\},\ \left\{ \langle L_1, U_1 \rangle, \ldots, \left\langle \varnothing, U_n \cup \{ix\}_{i=1}^{n} \right\rangle \right\}_{\alpha} \Big\rangle$$

where $\{\langle L_1, U_1 \rangle, \ldots, \langle \varnothing, U_n \rangle\}_{\alpha}$ is the Rabin acc. condition of $\mathcal{A}'$.
Finally

$$\delta'_{\mathcal{A}',x} = \Big\{ q \xrightarrow{a} \left(P_1, \ldots, P_{ar(a)}\right) \Bigm| q \xrightarrow{a} \vec{S} \in \delta_{\mathcal{A}'} \wedge \forall_{i=1}^{ar(a)} \left(x \notin S_i \Rightarrow P_i = S_i \right) \wedge \left(x \in S_i \Rightarrow \exists_{I \in I_{\mathcal{A}'}} P_i = S_i \setminus \{x\} \cup I \cap \mathcal{V}ar \cup \{ix \mid p_i \in I\} \right) \Big\}$$

Intuitions about these versions of the operations are similar to those given previously for them, but now we have multiple initial states. So we introduce new states ix — in place of single x — that simulate each of the initial states. Alternatively we may describe this as putting an ε-transition from x to each initial set S in $I_{\mathcal{A}'}$ and perform a bottom-up ε-elimination. Such transition means that to accept an APG from x we need to accept it from all elements of S.

In this paper we are not going to give a proof of correctness of these new μx_{aut}, νx_{aut} operations, thus we mark them as an attempt. Still it seems that the Niwiński's proofs for their NDA versions can be extended to cover them.

Intersection types definition

Definition 3.32. We extend a type system with intersections as follows.

There are no additional requirements on $\mathcal{U}$ and $env_{\mathcal{U}}$.

$\mathcal{E}xp = \ldots$ / $\mathcal{E}xp$ & $\mathcal{E}xp$

$[\![e_1\&e_2]\!]_{val} = [\![e_1]\!]_{val} \cap [\![e_2]\!]_{val}$

$\Phi = \cdots \cup \{\&_{aut}\}$
where $\&_{aut}$ is a 2-ary operation $\&_{aut}(\mathcal{A}_1, \mathcal{A}_2) = \mathcal{A}_{\&}[x \mapsto \mathcal{A}_1, y \mapsto \mathcal{A}_2]$ and $\mathcal{A}_{\&} = \langle \mathcal{F}, \varnothing, \{x, y\}, \{\{x, y\}\}, \varnothing, \bot \rangle$ for some distinct variables x, y.

Lemma 3.33. *The cases of Theorem 3.11 for* $e = e_1\&e_2$ *and* $\mathcal{A} = \&_{aut}(\phi_1, \phi_2)$ *where* ϕ_1, ϕ_2 *are algebraic expressions describing automata* $\mathcal{A}_1, \mathcal{A}_2$.

Proof. This is analogous to the previous proofs. □

Example 3.34. Below we show how intersections may shrink type expressions. Both below expressions represent the same type of finite natural number sequences of length divisible by 20.

$$\mu L.\mathsf{Unit} \mid \mathsf{Nat} \times L$$

$$(\mu L.\mathsf{Unit} \mid \mathsf{Nat} \times \mathsf{Nat} \times \mathsf{Nat} \times \mathsf{Nat} \times L)\ \&$$
$$(\mu L.\mathsf{Unit} \mid \mathsf{Nat} \times \mathsf{Nat} \times \mathsf{Nat} \times \mathsf{Nat} \times \mathsf{Nat} \times L)$$

3.3.8 Inductive and coinductive types

Inductive and coinductive types are recursive types that do not need a transfinite iteration. In other words they are those recursive types that can be expressed as $\bigcup_{i\in\mathbb{N}} F^i(\varnothing)$ and $\bigcap_{i\in\mathbb{N}} F^i(2^{\mathcal{U}})$. To determine which recursive types are these special ones, we can use the Kleene fixed-point theorem, which states that, if A is a complete partial order and $F: A \to A$ is a Scott-continuous function, then $\bigvee_{i\in\mathbb{N}} F^i(\bot)$ is the least fixed point of F. By this claim we obtain that, if our F is Scott-continuous according to $\subseteq$, then the type μF is exactly $\bigcup_{i\in\mathbb{N}} F^i(\varnothing)$, and if F is Scott-continuous according to $\supseteq$, then the type of νF is $\bigcap_{i\in\mathbb{N}} F^i(2^{\mathcal{U}})$. Thus if we want to build a type system with only inductive and coinductive recursive types it is sufficient to restrict Fs to Scott-continuous functions wrt to $\subseteq$ and $\supseteq$.

In the case of our construction of recursive types such condition may be easily enforced by requesting operators μ and ν not to alternate. However we need few new terms for saying formally what it does mean. Let's go through them.

We assume below that $\mathcal{E}xp$ stands for the one given in Definition 3.25.

Definition 3.35. If $e_0, \ldots, e_n \in \mathcal{E}xp$ and $x_1, \ldots, x_n \in \mathcal{V}ar$, then the ***substitution*** $e_0[x_1 \mapsto e_1, \ldots, x_n \mapsto e_n]$ of $e_1, ..., e_n$ for $x_1, ..., x_n$ in e_0 is defined recursively as follows.

Let χ be any of μ, ν, then

- $x_i[x_1 \mapsto e_1, \ldots, x_n \mapsto e_n] = e_i$,
- $(\chi x_i.e)[x_1 \mapsto e_1, ..., x_n \mapsto e_n] = \chi x_i.e\,[x_1 \mapsto e_1, ..., x_{i-1} \mapsto e_{i-1}, x_{i+1} \mapsto e_{i+1}, ..., x_n \mapsto e_n]$,
- $(\chi y.e)[x_1 \mapsto e_1, ..., x_n \mapsto e_n] = \chi y.e[x_1 \mapsto e_1, ..., x_n \mapsto e_n]$ if $y \neq x_i$ for all i, and
- $(H(e'_1, ..., e'_m))[x_1 \mapsto e_1, ..., x_n \mapsto e_n] = H(e'_1[x_1 \mapsto e_1, ..., x_n \mapsto e_n], \ldots, e'_m[x_1 \mapsto e_1, ..., x_n \mapsto e_n])$ for any other m-ary expression symbol H

Definition 3.36. For $e \in \mathcal{E}xp$ the set of ***free variables*** in it, denoted by $FV(e)$, is defined recursively as follows.

- $FV(x) = \{x\}$ for any variable x,
- $FV(\chi x.e') = FV(e') \setminus \{x\}$ for $\chi \in \{\mu, \nu\}$,
- $FV(H(e'_1, ..., e'_m)) = \bigcup_{i=1}^m FV(e'_i)$ for any other m-ary expression symbol H

An expression e is called ***closed*** if $FV(e) = \varnothing$ and it is called ***open*** otherwise.

Lemma 3.37. *For any* $e_0, \ldots, e_n \in \mathcal{E}xp$, $x_1, \ldots, x_n \in \mathcal{V}ar$ *and* $val \in \mathcal{C}$, *such that* $e_1, \ldots, e_n$ *are closed, we have*

$$[\![e_0[x_1 \mapsto e_1, \ldots, x_n \mapsto e_n]]\!]_{val} = [\![e_0]\!]_{val[x_1 \mapsto [\![e_1]\!], \ldots, x_n \mapsto [\![e_n]\!]]}$$

or both sides of the equation are undefined.

Proof. This may be simply proved by considering inductively the substitution definition and equations on $[\![\,]\!]$ that we give for each construction. □

Definition 3.38. Let's say that $e \in \mathcal{E}xp$ can be split to a ***head*** h and a ***tail*** $e_1, \ldots, e_n$ if $e = h \in \mathcal{V}ar$ and $n = 0$ or $h \notin \mathcal{V}ar$ and $e = h[x_1 \mapsto e_1, ..., x_n \mapsto e_n]$ for some $x_1, \ldots, x_n \in FV(h)$.

For conciseness we will say that a ***tail is closed*** if all expressions in it are closed.

Example 3.39. The type expression $\mu x.\mathsf{Unit}|(\nu y.\mathsf{Nat} * y) * x$ — that describes a type for finite sequences of infinite sequences of naturals — can be split to a head $\mu x.\mathsf{Unit}|z * x$ and a tail $\nu y.\mathsf{Nat} * y$.

Note that if we split an expression, then all expressions in the tail are smaller in a well-founded sense (as they contain less symbols), so we can use them to perform induction.

Now if we are able to split a recursive type expression to a head that contain only one kind of recursive operators and a closed tail, then the type described by it is inductive or coinductive.

Theorem 3.40. *For any expression $\mu x.e \in \mathcal{E}xp$ such that it can be split to an open head h not-containing ν operators and a closed tail, and for any valuation $val \in \mathcal{C}$ of variables $FV(\mu x.e)$, we have that the function $F : 2^{\mathcal{U}} \to 2^{\mathcal{U}}$, $F(X) = [\![e]\!]_{val[x\mapsto X]}$ is continuous wrt $\subseteq$.*

Proof. First we decompose $\mu x.e$ by the definition to $h = \mu x.e_0$ and a closed tail $e_1, \ldots, e_n$ such that $\mu x.e = (\mu x.e_0)[x_1 \mapsto e_1, ..., x_n \mapsto e_n]$. Without losing generality we may assume that $x \neq x_i$ for all i. Note that in such case by the definition of substitutions and Lemma 3.37 we have that $F(X) = [\![e_0]\!]_{val'[x\mapsto X]}$ where $val' = val[x_1 \mapsto [\![e_1]\!], \ldots, x_n \mapsto [\![e_n]\!]]$. Now to show F is continuous wrt $\subseteq$ we perform an induction on e_0 as follows.

- If e_0 is a variable, F must be an identity or a constant function and it is trivially continuous.
- If $e_0 = H(e'_1, \ldots, e'_m)$, where H is a symbol given in another type construction, we may wrap each of $e'_1, \ldots, e'_m$ with μx and obtain the inductive hypothesis for them. Then if we take a look at semantics of our type construction, we see that $[\![H(e'_1, \ldots, e'_m)]\!]_{val'[x\mapsto X]}$ is given with $\bar{\times}$, $\cup$, $\cap$, some fixed types and $[\![e'_1]\!]_{val'[x\mapsto X]}$, $\ldots$, $[\![e'_n]\!]_{val'[x\mapsto X]}$. So if we treat the last as functions on X, we get that F is a composition of continuous functions and so it is continuous.
- If $e_0 = \mu x.e'_0$, then it is trivial as F is constant.
- If $e_0 = \mu y.e'_0$ where $y \neq x$, then by the induction hypothesis we have that $F(X) = \bigcup_{i\in\mathbb{N}} G^i_X(\emptyset)$ for the continuous functions $G_X(Y) = [\![e'_0]\!]_{val'[x\mapsto X, y\mapsto Y]}$. But as we can wrap e'_0 also with μx and use the induction hypothesis on it, we have that the functions $G'_Y(X) = [\![e'_0]\!]_{val'[y\mapsto Y, x\mapsto X]}$ are also continuous for any type Y. (Simply saying we have here the 2-ary function $G(X,Y) = [\![e'_0]\!]_{val'[y\mapsto Y, x\mapsto X]}$ that is continuous on both arguments.) What we have to do now is showing that $\bigcup_{i\in\mathbb{N}} G^i_{\bigcup_{j\in J} X_j}(\emptyset) = \bigcup_{j\in J}\bigcup_{i\in\mathbb{N}} G^i_{X_j}(\emptyset)$ for any directed set of types $\{X_j\}_{j\in J}$. This can be easily done by rewriting the left side, first by the continuousity of G'_Ys, to

$$\bigcup_{i\in\mathbb{N}} \overbrace{\bigcup_{j\in J} G_{X_j}\left(\ldots\left(\bigcup_{j\in J} G_{X_j}\left(\bigcup_{j\in J} G_{X_j}(\emptyset)\right)\right)\right)}^{i}$$

and then by the continuity of G_Xs to what we want. □

Analogously we have the following.

Theorem 3.41. *For any expression $\nu x.e \in \mathcal{E}xp$ such that it can be split to an open head not-containing μ operators and a closed tail, and for any valuation $val \in \mathcal{C}$ of variables $FV(\nu x.e)$, we have that the function $F : 2^{\mathcal{U}} \to 2^{\mathcal{U}}$, $F(X) = [\![e]\!]_{val[x\mapsto X]}$ is continuous wrt $\supseteq$.*

The above theorems allow us to simplify reasoning for particular recursion occurrences in a type expression. If all these occurrences in an expression satisfy one of the theorems, we say that the whole expression is alternation free.

Definition 3.42. An expression $e \in \mathcal{E}xp$ is ***μ,ν-alternation free*** if it can be split to a head that contains at most one of the operators μ, ν and a closed tail such that all expressions in it are μ,ν-alternation free.

Example 3.43. The type expressions from Example 3.29 are not alternation free. Consider the first one, $\mu x.\nu y.(\mathsf{Unit}|x) * y$. If the expression was alternation-free, then the head would be of the form $\mu x.e_0$, where e_0 cannot posses ν. So e_0 would need to be a variable. But then the tail would be $\nu y.(\mathsf{Unit}|x) * y$ and it would not be closed. So there is no proper partition of $\mu x.\nu y.(\mathsf{Unit}|x) * y$ to make it be μ,ν-alternation free.

By Theorems 3.40, 3.41 we can see that types given by μ,ν-alternation free expressions are easier to reason about, as in their case we do not need transfinite iterations. But what is also important, we can significantly simplify automata represent-

ations for such types. Saying it more precisely, we can show that types given by μ, ν-alternation free expressions can be represented by weak Büchi automata, instead of Rabin automata that are required in the general case. We give it in more detail below.

First let's put some helper terms.

Definition 3.44. An automaton (both NDA and AA) is ***head-only open*** if it is a cascading automaton $\left\langle \mathcal{F}, \{Q_i\}_{i\in[n]}, V, ..., \delta, \{\alpha_i\}_{i\in[n]}\right\rangle$, $V \neq \varnothing$ and for each transition $q \xrightarrow{...} (..., x, ...) \in \delta$ where $x \in V$ – that is a transition containing a variable – we have that $q \in Q_0$.

Definition 3.45. Let $\mathcal{A} = \left\langle \mathcal{F}, \{Q_i\}_{i\in[n]}, V, q_0, V_0, \delta, \{\alpha_i\}_{i\in[n]}\right\rangle$ and $0 \le k \le l < n$, $\alpha_k, \alpha_{k+1} \dots, \alpha_l = \top$. ***Collapsion of maximal cascades*** $Q_k, ..., Q_l$ in $\mathcal{A}$ is defined as $collapse^{\top}\left(\mathcal{A}, \bigcup_{i=k}^{l} Q_i\right) = \left\langle \mathcal{F}, \{P_i\}_{i\in[n-l+k+1]}, V, q_0, V_0, \delta, \{\beta_i\}_{i\in[n-l+k+1]}\right\rangle$
where $P_i = Q_i, \quad \beta_i = \alpha_i \quad$ for $i < k$

$$P_k = \bigcup_{i=k}^{l} Q_i, \quad \beta_k : \bigcup_{i=k}^{l} Q_i \to \{true\}$$

$$P_i = Q_{l+i-k}, \quad \beta_i = \alpha_{l+i-k} \quad \text{for } i > k$$

Observation 3.46. *If $collapse^{\top}(\mathcal{A}, P)$ is defined, then $\mathcal{A}^{env\mathcal{U}}[val] = collapse^{\top}(\mathcal{A}, P)^{env\mathcal{U}}[val]$ for all valuations val of $V_{\mathcal{A}}$.*

In other words, adjacent maximal cascades in an automaton can be merged. The same applies to adjacent minimal cascades, but in their case we do not need a special operation for this, as automata with merged and split minimal cascades are literally equal.

Observation 3.47. *Any cascading automaton $\langle \mathcal{F}, (Q_0, ..., Q_k, ...Q_n), V, q_0, V_0, \delta, (\bot, ..., \bot, \alpha_{k+1}, ..., \alpha_n)\rangle$ is equal to $\langle \mathcal{F}, (Q_0 \cup ... \cup Q_k, ...Q_n), V, q_0, V_0, \delta, (\bot, \alpha_{k+1}, ..., \alpha_n)\rangle$.*

Now to show that we can build the automata as stated, we add new automata operations. These operations will not allow to define new types, but they will give a way to build weak Büchi automata for some already existing ones.

Recall that operations $\times_{aut}, |_{aut}, +_{aut}$ are given by automata compositions of $\mathcal{A}_{\times}, \mathcal{A}_{|}, \mathcal{A}_{+}$ with argument automata and the only state of the former is 0. Then, as $\mathcal{A}_{\times}^{\top}, \mathcal{A}_{|}^{\top}, \mathcal{A}_{+}^{\top}$ will be copies of their $\top$-less versions up to α in the following definition, the state 00 will be a copy of their only state in the results of operations $\times_{aut}^{\top}, |_{aut}^{\top}, +_{aut}^{\top}$. Moreover, note that each such result will be able to be arranged as a cascading automaton such that HD will be a sum of 1, 2 or 3 succeeding top cascades of it with maximal accepting conditions.

Definition 3.48. In pair, untagged union, tagged union and recursive type constructions — Definitions 3.15, 3.17, 3.20 and 3.25 — we add, respectively, $\times_{aut}^{\top}, |_{aut}^{\top}, +_{aut}^{\top}$ and $\mu x_{aut}^{w}, \nu x_{aut}^{w}$ operations to Φ that are given as follows. (We put w for "weak".)

For $o = \times, |, +$ the operation $o_{aut}^{\top}$ is binary

$$o_{aut}^{\top}(\mathcal{A}_1, \mathcal{A}_2) = collapse^{\top}\left(\mathcal{A}_o^{\top}[x \mapsto \mathcal{A}_1, y \mapsto \mathcal{A}_2], HD\right)$$

where $\mathcal{A}_o^{\top} = \langle \mathcal{F}, Q_{\mathcal{A}_o}, V_{\mathcal{A}_o}, q_{0\mathcal{A}_o}, V_{0\mathcal{A}_o}, \delta_{\mathcal{A}_o}, \top\rangle$ and HD is given as follows.

- If $\mathcal{A}_1$, $\mathcal{A}_2$ are cascading with $\alpha_{0\mathcal{A}_1} = \top$, $\alpha_{0\mathcal{A}_2} = \top$, then $HD = \{00\} \cup \{1\} \times Q_{0\mathcal{A}_1} \cup \{2\} \times Q_{0\mathcal{A}_2}$.
- If only $\mathcal{A}_1$ is cascading with $\alpha_{0\mathcal{A}_1} = \top$, then $HD = \{00\} \cup \{1\} \times Q_{0\mathcal{A}_1}$.
- If only $\mathcal{A}_2$ is cascading with $\alpha_{0\mathcal{A}_2} = \top$, then $HD = \{00\} \cup \{2\} \times Q_{0\mathcal{A}_2}$.
- Otherwise $HD = \{00\}$.

The state substitution $\delta'_{\mathcal{A},x}$ is defined as previously, but with the singleton $\{x\}$ replaced by $\{q_0\}$ (Def. 3.49).

For a head-only open automaton $\mathcal{A}$ with the first accepting condition $\alpha_0 = \bot$ and $q_{0\mathcal{A}}$ being in the first cascade

$$\mu x_{aut}^{w}(\mathcal{A}) = \left\langle \mathcal{F}, Q_{\mathcal{A}}, V_{\mathcal{A}} \setminus \{x\}, q_{0\mathcal{A}}, V_{0\mathcal{A}} \setminus \{x\}, \delta'_{\mathcal{A},x}, \alpha_{\mathcal{A}}\right\rangle$$

For a head-only open automaton $\mathcal{A}$ with its $\alpha_0 = \top$, $q_{0\mathcal{A}}$ being in the first cascade and $x \in V_{0\mathcal{A}}$

$$\nu x_{aut}^{w}(\mathcal{A}) = \left\langle \mathcal{F}, \{0\}, \varnothing, 0, \varnothing, \left\{0 \xrightarrow{a} \overbrace{(0, ..., 0)}^{ar(a)} \mid a \in \mathcal{F}\right\}, \top\right\rangle$$

For a head-only open automaton $\mathcal{A}$ with its $\alpha_0 = \top$, $q_{0\mathcal{A}}$ being in the first cascade and $x \notin V_{0\mathcal{A}}$

$$\nu x_{aut}^{w}(\mathcal{A}) = \left\langle \mathcal{F}, Q_{\mathcal{A}}, V_{\mathcal{A}} \setminus \{x\}, q_{0\mathcal{A}}, V_{0\mathcal{A}} \setminus \{x\}, \delta'_{\mathcal{A},x}, \alpha_{\mathcal{A}}\right\rangle$$

The operations are undefined for other arguments.

Definition 3.49. New state substitution:

$$\delta'_{\mathcal{A},x} = \left\{ q \xrightarrow{a} \left(p_1, ..., p_{ar(a)}\right) \,\middle|\, q \xrightarrow{a} \overrightarrow{s} \in \delta_{\mathcal{A}} \wedge \forall_{i=1}^{ar(a)} \left(s_i \neq x \Rightarrow p_i = s_i\right) \wedge \left(s_i = x \Rightarrow p_i \in \{q_0\} \cup V_{0_{\mathcal{A}}}\right) \right\}$$

In the given construction we collapse top cascades in the $\top$-ed operations to make variables from first cascades of argument automata be available for $\mu x^w_{aut}, \nu x^w_{aut}$ operations. Note that we do not need to do it in the case of $\top$-less operations by Observation 3.47.

Take in mind that when we allow for intersections in a type system, and so $\mathbb{A}$ becomes a universe of alternating automata given in Section 3.3.7, then the issue with recursive automata operations that we mentioned also apply to $\mu x^w_{aut}, \nu x^w_{aut}$.

Observation 3.50. *Automata generated by $\varnothing$ through any our algebra $\langle \mathbb{A}, \Phi \smallsetminus \{\mu x_{aut}, \nu x_{aut}\}_{x \in \mathcal{V}ar} \rangle$, where Φ is enhanced by Definition 3.48, are weak Büchi.*

Proof. This flows from that all our automata operations except the recursive ones preserve cascades of their arguments eventually reorganising the top ones; and $\mu x^w_{aut}, \nu x^w_{aut}$ may be applied only to automata such that the application will not break the weak Büchi condition (the only place where $\mu x^w_{aut}, \nu x^w_{aut}$ may break the condition is switching $q \xrightarrow{\cdots} (..., x, ...)$ to $q \xrightarrow{\cdots} (..., q_{0_{\mathcal{A}}}, ...)$, but $q, q_{0_{\mathcal{A}}}$ must be members of the first cascade of $\mathcal{A}$).

The following lemma says that if weak recursive operations are applicable, then they result in semantically equivalent automata as they stronger versions. □

Lemma 3.51. *If $\mathcal{A} \in Dom\left(\mu x^w_{aut}\right)$, then*

$$\mu x^w_{aut}\left(\mathcal{A}\right)^{env_{\mathcal{U}}}[val] = \mu x_{aut}\left(\mathcal{A}\right)^{env_{\mathcal{U}}}[val]$$

and if $\mathcal{A} \in Dom\left(\nu x^w_{aut}\right)$, then

$$\nu x^w_{aut}\left(\mathcal{A}\right)^{env_{\mathcal{U}}}[val] = \nu x_{aut}\left(\mathcal{A}\right)^{env_{\mathcal{U}}}[val]$$

for all valuations val of variables $V_{\mathcal{A}} \smallsetminus \{x\}$.

Proof. Let's show that $\mu x^w_{aut}(\mathcal{A})$ is equivalent to $\mu x_{aut}(\mathcal{A})$ wrt *val*. First we take any accepting run r of the first. If $\mathcal{A}$ does not contain x, there is nothing to do, as r is also an accepting run of the latter automaton. Then we assume that $\mathcal{A}$ is a head-only open automaton with the first states cascade Q_0 and the first accepting condition $\alpha_0 = \bot$. In this case, we exchange appropriate labels $q_{0_{\mathcal{A}}}$ to x in r to obtain a valid run r' of the automaton $\mu x_{aut}(\mathcal{A})$ (we can do it, as for each transition in the first automaton we have exactly the same transition in the second with the difference that some occurrences of $q_{0_{\mathcal{A}}}$ are replaced by x). Now by the fact that μx^w_{aut} does not change the acceptance condition of its argument, we have that the the conditions of the automata $\mathcal{A}$ and $\mu x^w_{aut}(\mathcal{A})$ are the same and by the mentioned constrain $\alpha_0 = \bot$ they can be written as a Rabin condition of the form $\{\langle Q_0 \cup L_1, U_1\rangle, \ldots, \langle Q_0 \cup L_n, U_n\rangle\}_\alpha$. Then by the definition the condition of $\mu x_{aut}(\mathcal{A})$ is the Rabin $\{\langle Q_0 \cup L_1 \cup \{x\}, U_1\rangle, \ldots, \langle Q_0 \cup L_n \cup \{x\}, U_n\rangle\}_\alpha$. The acceptance condition of $\mu x^w_{aut}(\mathcal{A})$ says now that each branch in r cannot have infinitely many $q_{0_{\mathcal{A}}}$ on it and so the corresponding branch in r' cannot have infinitely many x on it. Then each branch in r' is accepted by the accepting pair corresponding to the one that accepted it in r.

To show that $\mu x^w_{aut}(\mathcal{A})$ accepts if $\mu x_{aut}(\mathcal{A})$ accepts we do almost the same, but in the first step we exchange all xs in an accepting run of $\mu x_{aut}(\mathcal{A})$ to $q_{0_{\mathcal{A}}}$.

The case for $\nu x^w_{aut}(\mathcal{A})$ and $\nu x_{aut}(\mathcal{A})$ is similar, with the difference we have a Rabin condition of the form $\{\langle L_1, U_1\rangle, \ldots, \langle \varnothing, U_n \cup Q_0\rangle\}_\alpha$ for the automata $\mathcal{A}$, $\nu x^w_{aut}(\mathcal{A})$ and $\{\langle L_1, U_1\rangle, \ldots, \langle \varnothing, U_n \cup Q_0 \cup \{x\}\rangle\}$ for $\nu x_{aut}(\mathcal{A})$. □

Now we show that the new automata operations are, indeed, semantically redundant, so we do not extended our type systems by adding them.

Proposition 3.52. *For any automaton $\mathcal{A}$ generated by $\varnothing$ through any our algebra $\langle \mathbb{A}, \Phi \rangle$ enhanced by Definition 3.48, there exists an automaton $\mathcal{A}'$ generated by $\varnothing$ through $\langle \mathbb{A}, \Phi \smallsetminus \{\times^\top_{aut}, |^\top_{aut}, +^\top_{aut}\} \smallsetminus \{\mu x^w_{aut}, \nu x^w_{aut}\}_{x \in \mathcal{V}ar} \rangle$, such that $V_{\mathcal{A}} = V_{\mathcal{A}'}$ and $\mathcal{A}^{env_{\mathcal{U}}}[val] = \mathcal{A}'^{env_{\mathcal{U}}}[val]$ for any valuation val of $V_{\mathcal{A}}$.*

Proof. We do an induction on an expression ϕ that gives the automaton $\mathcal{A}$. If the top operation in ϕ is not among those given in Definition 3.48, things are trivial. If the top operation is one of $\times^\top_{aut}, |^\top_{aut}, +^\top_{aut}$, things are easy, as $o_{aut}(\mathcal{A}_1, \mathcal{A}_2)^{env_{\mathcal{U}}}[val] = o^\top_{aut}(\mathcal{A}_1, \mathcal{A}_2)^{env_{\mathcal{U}}}[val]$ for all $\mathcal{A}_1, \mathcal{A}_2$ and $o = \times, |, +$ — this is because the only thing changed by adding $\top$ to the automata operations is switching the cascade $\{00\}$ in $o_{aut}(\mathcal{A}_1, \mathcal{A}_2)$ from minimal to maximal, where the state 00 can appear only once in any run of this

automaton. Finally, in the cases for $\phi = \mu x^w_{aut}(\psi)$ and $\phi = \nu x^w_{aut}(\psi)$, by Lemma 3.51 we have that that $\mathcal{A}$ is semantically equal to the automaton given by $\mu x_{aut}(\psi)$ or respectively by $\nu x_{aut}(\psi)$. Then by the induction hypothesis we have a proper automaton $\mathcal{A}''$ for the expression ψ. So $\mathcal{A}''$ and ψ defines the same function wrt to x, or in other words, we have a function F such that $F(X) = \psi^{env\mathcal{U}}[val[x \mapsto X]] = \mathcal{A}''^{env\mathcal{U}}[val[x \mapsto X]]$. Then it is obvious that $\mu x_{aut}(\psi)$, $\mu x_{aut}(\mathcal{A}'')$ and $\nu x_{aut}(\psi)$, $\nu x_{aut}(\mathcal{A}'')$ describes the same type, as they compute the same fixed points of the same functions (wrt val). □

Finally we show that for any μ,ν-alternation free type expression we have an automaton that represents the same type, and that can be build with our automata operations extended by Definition 3.48, but without μx_{aut}, νx_{aut}; and vice versa, for each such an automaton we have such a μ,ν-alternation free type expression.

Theorem 3.53. *For any type T given with our type constructions extended by Definition 3.48 and for any valuation $val \in \mathcal{C}$,*

- *there exists an open μ,ν-alternation free expression $e \in \mathcal{E}xp$ such that $[\![e]\!]_{val} = T$ and e can be split to a head without μ and a closed tail if and only if there exists a head-only open automaton $\mathcal{A}$ with $\alpha_0 = \top$ generated by $\varnothing$ in the algebra $\langle \mathbb{A}, \Phi \setminus \{\mu x_{aut}, \nu x_{aut}\}_{x \in \mathcal{V}ar}\rangle$ such that $\mathcal{A}^{env\mathcal{U}}[val] = T$,*
- *there exists an open μ,ν-alternation free expression $e \in \mathcal{E}xp$ such that $[\![e]\!]_{val} = T$ and e can be split to a head without ν and a closed tail if and only if there exists a head-only open automaton $\mathcal{A}$ with $\alpha_0 = \bot$ generated by $\varnothing$ in the algebra $\langle \mathbb{A}, \Phi \setminus \{\mu x_{aut}, \nu x_{aut}\}_{x \in \mathcal{V}ar}\rangle$ such that $\mathcal{A}^{env\mathcal{U}}[val] = T$, and*
- *there exists a closed μ,ν-alternation free expression $e \in \mathcal{E}xp$ such that $[\![e]\!]_{val} = T$ if and only if there exists a closed automaton $\mathcal{A}$ generated by $\varnothing$ in the algebra $\langle \mathbb{A}, \Phi \setminus \{\mu x_{aut}, \nu x_{aut}\}_{x \in \mathcal{V}ar}\rangle$ such that $\mathcal{A}^{env\mathcal{U}}[val] = T$*

Below we assume that we work with NDAs, but the reasoning goes the same for AA, with cosmetic changes.

Proof. For all three equivalences the left-to-right implications may be done with simple inductive constructions performed on e. In the first case we do it by building $\mathcal{A}$ with the operations $\times^{\top}_{aut}$, $|^{\top}_{aut}$, $+^{\top}_{aut}$, $(\&_{aut},)$ x_{aut}, νx^w_{aut}, in the second case with $\times_{aut}$, $|_{aut}$, $+_{aut}$, $(\&_{aut},)$ x_{aut}, μx^w_{aut} and in the last one we may use any operations except x_{aut}. The correctness of such construction can be justified as follows.

First we consider the case of $e = e_1 * e_2$ that satisfies one of the lemma hypotheses. By the induction hypothesis we have proper automata $\mathcal{A}_1$, $\mathcal{A}_2$ for expressions e_1, e_2. Then we show that if e satisfies the first lemma case, then $\times^{\top}_{aut}(\mathcal{A}_1, \mathcal{A}_2)$ is the desired $\mathcal{A}$, if e satisfies the second, then $\times_{aut}(\mathcal{A}_1, \mathcal{A}_2)$ is a proper $\mathcal{A}$, and if e satisfies the third, $\mathcal{A}$ can by any of these automata.

The semantical equality is obvious, as $[\![e_1]\!]_{val} = \mathcal{A}_1^{env\mathcal{U}}[val]$, $[\![e_2]\!]_{val} = \mathcal{A}_2^{env\mathcal{U}}[val]$ by the hypothesis and we have already stated that in such case $[\![e_1 * e_2]\!]_{val} = \times_{aut}(\mathcal{A}_1, \mathcal{A}_2)^{env\mathcal{U}}[val]$ $= \times^{\top}_{aut}(\mathcal{A}_1, \mathcal{A}_2)^{env\mathcal{U}}[val]$. So the only thing to prove is that $\times_{aut}(\mathcal{A}_1, \mathcal{A}_2)$, $\times^{\top}_{aut}(\mathcal{A}_1, \mathcal{A}_2)$ are of the desired – respectively, head-only open or closed – form. This is straightforward. In the case when e is open and it does not contain μ, automata $\mathcal{A}_1, \mathcal{A}_2$ must be closed or head-only open with the top cascades being maximal. So $\times^{\top}_{aut}(\mathcal{A}_1, \mathcal{A}_2)$ makes the whole part of the result automaton with accessible variables collapse into one top maximal cascade. The case when e is open and it does not contain ν is symmetric (recall that the top minimal cascades of $\times_{aut}(\mathcal{A}_1, \mathcal{A}_2)$ collapses implicitly). Finally, the case for closed e is trivial.

The cases for $e = e_1|e_2$ and $e = e_1 + e_2$ looks the same. The case for $e = x$ is simpler, and the case for $e = \mathsf{b}$ is obvious. So what is left are $e = \mu x.e'$ and $e = \nu x.e'$. These are analogous, so we will take a look only at $e = \nu x.e'$.

In this case by the induction hypothesis we have an equivalent automaton $\mathcal{A}'$ for e' and so $\nu x^w_{aut}(\mathcal{A}')$ is semantically equivalent to e. So what is left to show that $\nu x^w_{aut}(\mathcal{A}')$ is our desired $\mathcal{A}$ is again the form of the automaton. In the case when e is closed, this is straightforward as $\nu x^w_{aut}(\mathcal{A}')$ must only be closed. In the other case e' matches the first point of the theorem, so we have that $\mathcal{A}'$ is a head-only open with a maximal top accepting condition. Then as νx^w_{aut} preserves cascades of its argument (what we have already pointed out) $\nu x^w_{aut}(\mathcal{A}')$ ia a proper $\mathcal{A}$.

The right-to-left implications of the theorem can be proven by an inductive construction of e based on algebraic expression describing $\mathcal{A}$ — this is symmetric to what we have done above. □

By the above claims we get an important co-

rollary that for types given with μ, ν-alternation free expressions we can perform subtype checking much more effectively then for general recursive types — by what we mentioned in Section 2.2.4. A conclusion from this and the problems with understanding semantics of general recursive types may be that, it seems reasonable to exclude types given with expressions where μ, ν operators alternate (and remove $\mu x_{aut}, \nu x_{aut}$ operations from the automata algebra) from practically applicable type systems.

3.3.9 Towards functions

Function types commonly describe sets of representations of computable partial functions (still this is not a golden rule and both computability and partialness may be dropped in some cases). In this paper the functions representations will be set-like functions — sets of pairs of arguments and values — or some equivalent if we do not believe in not-well-founded sets.

Setting function values as mathematical functions may seem inconvenient here, as from the point of view of computer systems — which are the main background of this work — in most cases such values would intuitively need infinite space to be written. However this is not true. In computers we may still represent functions in the ordinary way, that is as lambda terms or programs, because a value and its representation do not have to be the same thing. This is similar to writing $f : \mathbb{N} \to \mathbb{N}$, $f(n) = n + 1$ to define a function. In such case, the writing corresponds to a lambda term or a program, but it is clearly not the function by its own.

Now let's go to the statement about seeing functions in set-like style or some equivalent. The "equivalent" is caused by the same problem that we faced in the Section 3.3.3. That is, treating functions straightly as sets of pairs leads to breaking the regularity axiom if we allow for coinductive function values in our universe. E.g., using the ML syntax we can define a function that takes a natural number and then return itself

```
let rec f (x: Nat) = f
```

Such a function may be represented in computers, but it straightly breaks the axiom when we try to put it in the set-like style — as it would need to satisfy the equality $f = \{(0, f), (1, f), \dots\}$. More or less fortunately functions of this kind require alternation of recursive types with function types to have a type in a type system and in a minute we will show that it may be not a good idea to allow for such types, so in this paper we will proceed with function values identified with sets of pairs.

Why do function types not match recursive types?

First let's take a look at the case when function types represent sets of total functions from some given domain to some given codomain. In such case the semantics of the type expression $x \to x$ wrt to any valuation $\{x \mapsto T, \dots\}$ is the set of all total functions from T to T. So what will happen if we wrap the expression with the recursive operator μx? Of course, we will request for the least fixed point of the function $F(X) = X \to X$ (where $X \to X$ is the set of all total functions from X to X) wrt $\subseteq$. But F is not monotonic, so generally, we do not even know if the fixed point exists. Thus such interpretation of function types is rather not good in our case.

In another case, we may treat function types as sets of partial functions from given domain to given codomain. Doing so regain monotonicity of functions that are under fixed-point operators (like the previous F). But still these functions are not continuous and recursions over them may expose problems in understanding. Particularly, if we have the expression $x \hookrightarrow x$ that is interpreted as before with the change the functions may now be partial, then it may be hard to settle the difference between $\mu x.x \hookrightarrow x$ and $\nu x.x \hookrightarrow x$. Moreover, another problem that concerns this interpretation of function types — as well as the previous one — is that, in this case the subtyping fixed by $\subseteq$ relation does not cover the generally used subtyping of functions that says a type $T_1 \to T_2$ is a subtype of $T_3 \to T_4$ if T_3 is a subtype of T_1 and T_2 is a subtype of T_4 — below we will call this subtyping "nice". This is not good, as it forces us to redefine functions in formal statements (programs) that do not need to be redefined. Thus choosing this interpretation seems also non-satisfying.

The nice subtyping may be regained by the third interpretation of function types. In this case, function types are seen as sets of all (partial or total) functions in some universe such that for all arguments from a given domain they return values from a given codomain and they may return anything for arguments out of the domain. To set a focus let's assume that we go with partial functions. So formally saying for two types

T_1, T_2 the function type $T_1 \dot{\hookrightarrow} T_2$ stands now for $\{ f : \mathcal{U} \cap (\mathcal{U} \hookrightarrow \mathcal{U}) \mid f(T_1) \subseteq T_2 \}$. This seems intuitive as if we have a function of such a type, then we expect to apply it only to values of the type T_1 and so the fact that it can be applied to something else does not concern us. Then with such function types our nice function subtyping is straightforward, as if $T_3 \subseteq T_1$ and $T_2 \subseteq T_4$, then obviously $T_1 \dot{\hookrightarrow} T_2 \subseteq T_3 \dot{\hookrightarrow} T_4$. Moreover, the inverse implication is also true except some edge cases ($T_1 = \varnothing$, $T_4 = \mathcal{U}$), so, in fact, we established an equivalent of the nice subtyping. Unfortunately, our old friend $F(X) = X \dot{\hookrightarrow} X$ becomes again non-monotonic in this case and thus we again get the problem with recursive types.

Thus, by the above considerations the conclusion comes that it may be reasonable to resign from alternating recursive types and function types. As such mixes are not very common anyway and they do not seem very useful, this seems rather justified.

Definition of function types

Assuming that we use the symbol $\rightarrow$ to represent function types and we have $\mathcal{E}xp = \dots \mid \mathcal{E}xp \rightarrow \mathcal{E}xp$, the not-mixing of recursive and function types is given as follows.

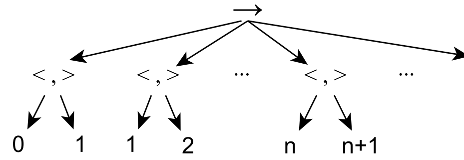
Definition 3.54. An expression $e \in \mathcal{E}xp$ is ***recursion-function not-mixing*** if for each x, e_0, e_1, e_2 such that $e = e_0[x \mapsto (e_1 \rightarrow e_2)]$ we have that e_1, e_2 are closed.

Particularly, the type expression $\mu x.x \rightarrow x$ — which can be used to describe the function that was given above — is not not-mixing, as it can be split to $e_0 = \nu x.f$, $e_1 = x$, $e_2 = x$ where x is obviously not closed.

Now, the thing that we still need to do before giving our frame with the definition of function types is an extension of ranked labeled APGs and automata, as many of our function values will not be able to be expressed as finitely ranked APGs (cf. ex. 3.55). So to fix it we extend the definition of ranked alphabets by switching the type of *ar* function to $\Sigma \rightarrow \mathbb{N} \cup \{*\}$ and we extend the definition of $\mathcal{F}$-APGs such that if a node is labeled with a symbol of the arity $*$ then it has any number of children indexed by some set $\mathbb{S}$.

Example 3.55. The figure shows an APG representation of the function $f : \mathbb{N} \rightarrow \mathbb{N}$, $f(n) = n + 1$.

Note, this is probably not what we want to read with automata, as in their case we would rather want to map all $0, 1, \dots$ to *nat* first to make the alphabet finite and so the graph would collapse to finitely many nodes. However, the idea shown here may be applied to build other functions where we cannot do such a trick. E.g., by replacing the natural numbers in the picture with graphs representing growing sequences of 0s.



To read ranked APGs with $*$ by automata, we extend now the definition of NDAs, by introducing two new kinds of transitions

$$q \xrightarrow{a} \forall q' \qquad q \xrightarrow{a} \exists q'$$

where a is a symbol in $\mathcal{F}$ of the arity $*$ and q, q' are states. Informally, the first kind of the transitions means that if an automaton is in the state q and it reads a node v with the label a, then it may move to all children of this node with the state q'; the second kind means that the automaton in such situation may choose non-deterministically one child of v and move to it with q'. So in other words, a "for all" transition says that an automaton accepts a node labeled with a from a state q if it accepts all children of the node from a state q'; and an "exists" transition says that an automaton accepts such a node from q if there exists a child of this node that the automaton accepts from q'.

Analogically to the transitions for NDAs we also introduce "for all" and "exists" transitions for AAs. The difference in these is only that single quantified states ($\forall q'$,$\exists q'$) on the right sides are replaced with sets of quantified states ($\{\forall q'_1, \exists q'_2, \dots\}$). If the reader is familiar with expressing transition relations of AAs as logical formulas, our transitions though $*$-ary symbols a may be depicted as $q \xrightarrow{a} \phi$, where ϕ are DNF formulas $\bigvee_{i=1}^{n} \bigwedge_{j=1}^{m} l_{i,j}$

such that each $l_{i,j} = \forall_k (k, q')$ or $l_{i,j} = \exists_k (k, q')$ for some state q'. In such case, the range of k is specified in the moment when a transition is about to be applied on a node v and it becomes the set of all indexes of v's children. The semantics of such transitions is then natural.

Unfortunately, the algorithm for checking inclusion between automata cannot be generalised easily to such an extension and further work is needed here. Saying it more precisely, we probably can do complementation of automata by a similar construction as the one given in Section A.2, but it appears that we cannot dealternate the result. Still the automata formalization for function types that we are going to propose here allows only to generate a subset of automata with quantified transitions that is significantly simpler then the general case. Thus it seems that an effective inclusion checking is not without chances. Particularly, if we disallow connecting function types by untagged unions it seems simple.

Remark 3.56. A dualisation of the new kind of transitions in the construction for automata complementation may be seen as switching formulas $\bigvee_{i=1}^{n} \bigwedge_{j=1}^{m} l_{i,j}$ in transitions to $\bigwedge_{i=1}^{n} \bigvee_{j=1}^{m} \overline{l_{i,j}}$, where $\overline{\forall_k (k, q')} = \exists_k (k, q')$, $\overline{\exists_k (k, q')} = \forall_k (k, q')$, and then translating them back to DNF. The proof of correctness from the paper [MuSch95, appendix C] presents a chance to be still valid after such extension.

Now we can propose a definition for function types. For this we assume that $\mathbb{A}$ is the AA universe. This is not necessary, but making it NDA will cause some parts of automata to be complemented and dealternated once during construction of automata and second time during subtype testing, what is obviously very expensive. In the definition we use $\to$ to represent semantical $\dot{\to}$ (for simplicity).

Definition 3.57. We extend a type system with function types as follows.

There is no requirement on $\mathcal{U}$.

$*$-ary symbol $\to \in \mathcal{F}$, and
$\imath(v) = \to$ and v' is a child of v in $env_{\mathcal{U}}$ if and only if $v \in \mathcal{U} \hookrightarrow \mathcal{U}$, $v' = \langle v_1, v_2 \rangle$ for some v_1, v_2 and $v(v_1) = v_2$, and
for any finite $\mathcal{F}$-APG there is an isomorphic $\mathcal{F}$-APG in $env_{\mathcal{U}}$.[3]

$\mathcal{E}xp = \ldots \;/\; \mathcal{E}xp \;\to\; \mathcal{E}xp$

$$[\![e_1 \to e_2]\!]_{val} = \{ f : \mathcal{U} \cap (\mathcal{U} \hookrightarrow \mathcal{U}) \mid f([\![e_1]\!]_{val}) \subseteq [\![e_2]\!]_{val} \}$$

if e_1, e_2 are closed, and it is undefined otherwise.

$\Phi = \cdots \cup \{\to_{aut}\}$
where $\to_{aut}$ is a 2-ary operation

$$\to_{aut}(\mathcal{A}_1, \mathcal{A}_2) = Alt(\mathcal{A}_{\to})\left[x \mapsto \mathcal{A}_1, \overline{x} \mapsto \overline{\mathcal{A}_1}, y \mapsto \mathcal{A}_2, z \mapsto \mathcal{A}_{\top}\right]$$

and

$$\mathcal{A}_{\to} = \left\langle \mathcal{F}, \{0,1\}, \{x, \overline{x}, y, z\}, 0, \varnothing, \left\{0 \xrightarrow{\to} \forall 1, 1 \xrightarrow{\langle\,,\,\rangle} (x, y), 1 \xrightarrow{\langle\,,\,\rangle} (\overline{x}, z)\right\}, \bot \right\rangle$$

for some distinct variables $x, \overline{x}, y, z$, where $\overline{\mathcal{A}_1}$ means complementation of automaton $\mathcal{A}_1$ and $\mathcal{A}_{\top}$ a total automaton that accepts every APG.

Note that because of our restriction on not mixing function types with recursive types, all automata in the formalization are cascading $\langle \mathcal{F}, \{Q_0, Q_1\}, \varnothing, q_0, \varnothing, \delta, \{\alpha_0, \alpha_1\} \rangle$ where transitions through $\to$ symbol belong only to the first cascade (that is, if $q \xrightarrow{\to} \cdots \in \delta$, then $q \in Q_0$) and there are no loops in this cascade (that is, there is no transition sequences $q_1 \xrightarrow{a_1} l_2$, $q_2 \xrightarrow{a_2} l_3$, $\ldots$, $q_n \xrightarrow{a_n} l_1$ such that $q_1, ..., q_n \in Q_0$ and each $l_i = (..., \{q_i, ...\}, ...)$, $\{\forall q_i, ...\}$ or $\{\exists q_i, ...\}$). In such case we can split any of these automata to $\mathcal{A} = \langle \mathcal{F}, Q_0, Q_1, q_0, \varnothing, \delta|_{Q_0}, \alpha_0 \rangle$ (Q_1becomes a set of variables) and $\mathcal{A}_q = \langle \mathcal{F}, Q_1, \varnothing, q, \varnothing, \delta|_{Q_1}, \alpha_1 \rangle$ for all $q \in Q_1$, where $\delta|_P = \left\{ q \xrightarrow{a} \cdots \in \delta \;\middle|\; q \in P \right\}$. Then, we can attempt to check subtyping by inductive

[3]Intuitively, this retrogradely enforces that for any finite sets A, B of finite APGs in $env_{\mathcal{U}}$ and any function $f : A \to B$ there exists a function in $\mathcal{U}$ that realizes f.

analyse of $\mathcal{A}$ automata, finding constraints on $\mathcal{A}_q$ automata and check the constraints without involving $\mathcal{A}$s. Still we leave this idea to further work.

Remark about function and intersection types

An interesting thing about intersection types is using them together with function types. This allows to define types for so called ad hoc polymorphic functions, that are generally used in popular programming languages.

Intuitively, the ad hoc polymorphism allows a function to have many function types at once. Then when we apply such a function to some argument, we need to find one of these types such that the argument matches its domain, and if we have one the type of returned value is the codomain. An example may be here the single argument minus function, that returns the negation of its argument. If we have a type for integer and a type for rational numbers in a system, this function should intuitively be able to work with arguments of both kinds, and return integers for the first and rationals for the second. But without intersections we can set a type for this function only as $\mathsf{Int} \,|\, \mathsf{Rat} \rightarrow \mathsf{Int} \,|\, \mathsf{Rat}$, what do not allow to preserve the constraint. The intersection types can solve this problem as the type $(\mathsf{Int} \rightarrow \mathsf{Int}) \,\&\, (\mathsf{Rat} \rightarrow \mathsf{Rat})$ describes exactly what we want. Such behaviour in type-checking is generally out of the scope of today strongly typed programming languages (like ML family).

Appendix A

Constructing algorithm for weak Büchi non-deterministic automata inclusion checking

A.1 Intersection

Here we give a construction of an intersection automaton that accepts an APG exactly when it is accepted by two other automata given as parameters. We restrict ourselves to do it only for a Büchi and a weak Büchi NDA. As one of these automata is weak, we can give a slightly better solution here, then the general one. (However using the standard construction for an intersection of Büchi NDAs combined with non-reachable states removing should give the same result. But still we give it for completeness.)

Theorem A.1. *For a closed weak Büchi NDA $\mathcal{A}_1$ and a closed Büchi NDA $\mathcal{A}_2$ over the same alphabet $\mathcal{F}$, there exists a closed Büchi NDA $\mathcal{A}_1 \cap \mathcal{A}_2$ over $\mathcal{F}$ that accepts an APG if and only if this APG is accepted by both $\mathcal{A}_1$ and $\mathcal{A}_2$.*

Proof. Let $\mathcal{A}_1 = \langle \mathcal{F}, Q_1, \varnothing, q_{0_1}, \varnothing, \delta_1, \alpha_1 \rangle$, $\mathcal{A}_2 = \langle \mathcal{F}, Q_2, \varnothing, q_{0_2}, \varnothing, \delta_2, \alpha_2 \rangle$ and let Q_1, Q_2 be disjoint (we can always rename states to satisfy this). We set $\mathcal{A}_1 \cap \mathcal{A}_2$ as

$$\langle \mathcal{F}, Q_1 \times Q_2, \varnothing, (q_{0_1}, q_{0_2}), \varnothing, \delta, \alpha \rangle$$

where δ is given as

$$(q, p) \xrightarrow{a} \left((q_1, p_1), \ldots, (q_{ar(a)}, p_{ar(a)})\right) \in \delta$$
$$\text{if and only if}$$
$$q \xrightarrow{a} \left(q_1, \ldots, q_{ar(a)}\right) \in \delta_1 \ \wedge\ p \xrightarrow{a} \left(p_1, \ldots, p_{ar(a)}\right) \in \delta_2$$

and α is the Büchi condition with $B_\alpha = B_{\alpha_1} \times B_{\alpha_2}$.

Let's show that if node v is accepted by $\mathcal{A}_1$ and $\mathcal{A}_2$ in some environment *env*, then v is also accepted by $\mathcal{A}_1 \cap \mathcal{A}_2$ in *env*. For this we take a run r_1 of $\mathcal{A}_1$ and r_2 of $\mathcal{A}_2$ that are accepting for v in *env*, and we set a run that is a witness of v being accepted by $\mathcal{A}_1 \cap \mathcal{A}_2$ in *env* as $r = \langle \mathcal{P}ath\mathcal{E}nv(\langle env, v \rangle) + \imath, \varepsilon \rangle$ where $\imath$ is defined as $\imath(w) = (r_{1_w}, r_{2_w})$. The symbol function is correctly defined as $\mathcal{A}_1$, $\mathcal{A}_2$ do not have variables and so $paths(\langle env, v \rangle) = paths(r_1) = paths(r_2)$. To show r is accepting, let's note that $\mathcal{A}_1$ accepts a branch in a run only if from some point nodes on the branch are labeled only with accepting states. Hence, for each branch ϖ in r there is a point from which nodes on it are labeled with pairs of accepting states of $\mathcal{A}_1$ and states coming from the branch ϖ in r_2. As there must be infinitely many nodes labeled with accepting states on ϖ in r_2, we have also infinitely many nodes labeled by accepting states of $\mathcal{A}_1 \cap \mathcal{A}_2$ on ϖ in r and so ϖ is accepted.

Proving the oppose implication is even simpler and we omit it. □

Remark A.2. The size of $\mathcal{A}_1 \cap \mathcal{A}_2$:

- *states* — $|Q_1||Q_2|$
- *transitions through* $a \in \mathcal{F}$ — $|\delta_a| = |\delta_{1_a}||\delta_{2_a}|$
- *accepting states* — $|B_\alpha| = |B_{\alpha_1}||B_{\alpha_2}|$

A.2 Complementation

Complementation is an operation that for a given automaton produces another automaton accepting exactly these APGs that are not accepted by the former automaton. While there are no practical advantages, known to the author, that would flow from restricting ourselves here, we will give a very general concept of this construction for alternating automata.

Theorem A.3. *For a closed alternating automaton $\mathcal{A}$, there exists a closed alternating automaton $\overline{\mathcal{A}}$ that accepts exactly these APGs that are not accepted by $\mathcal{A}$.*

Proof. Let $\mathcal{A} = \langle \mathcal{F}, Q, \varnothing, q_0, \varnothing, \delta, \alpha \rangle$. Then we set $\overline{\mathcal{A}}$

as

$$\langle \mathcal{F}, Q, \varnothing, q_0, \varnothing, \delta', \neg\alpha \rangle$$

where $\neg\alpha$ is negated function α and $q \xrightarrow{a} \left(P_1, \dots, P_{ar(a)}\right) \in \delta'$ if and only if

- there exists a choice function

$$\sigma : \left\{ q \xrightarrow{a} \left(P'_1, \dots, P'_{ar(a)}\right) \in \delta \right\} \to \{1, \dots, ar(a)\} \times Q$$

  such that if $\sigma\left(q \xrightarrow{a} \left(P'_1, \dots, P'_{ar(a)}\right)\right) = (i, p)$, then $p \in P'_i$, and
- $P_i = \{ q \mid \exists_{\rho \in \delta}\, \sigma(\rho) = (i, q) \}$ for all $i = 1, \dots, ar(a)$

Intuitively, the choice function σ says for each transition $q \xrightarrow{a} \left(P'_1, \dots, P'_{ar(a)}\right)$ which branch of an APG is not accepted (the one with the number i) is not accepted and which member of P'_i fails to accept.

The proof of correctness can be found in the work of Muller and Schupp [MuSch95, appendix C]. It is done for automata on n-fully-branched trees with equivalence classes of $(\vee, \wedge)$-formulas over (k, q) literals on right sides of transitions, where $k \in [n]$ and q is a state. However, the proof does not depend on the full-branching and our transitions can be easily translated to the formulas. Precisely, for our automaton $\langle \langle \Sigma, ar \rangle, Q, \varnothing, q_0, \varnothing, \delta, \alpha \rangle$ a corresponding Muller-Schupp's automaton would be $\langle Q, \Sigma, \delta', q_0, \{P \mid \alpha(P) = \mathit{true}\} \rangle$ where

$$\delta'(a, q) = \left[\bigvee_{q \xrightarrow{a} (P_1, \dots, P_{ar(a)}) \in \delta} \ \bigwedge_{i \in \{1, \dots, ar(a)\}} \ \bigwedge_{p \in P_i} (i, p) \right]_{\equiv}$$

□

Lemma A.4. *For a weak Büchi AA $\mathcal{A}$ the complementation is also a weak Büchi AA with the set of accepting states $B_{\neg\alpha} = Q \setminus B_\alpha$.*

Proof. The lemma follows from the observation, that the construction of $\overline{\mathcal{A}}$ keeps layers of $\mathcal{A}$ if it is weak. That is, for each transition $q \xrightarrow{a} \left(P_1, \dots, P_{ar(a)}\right)$ in $\overline{\mathcal{A}}$ and a state $p \in P_1 \cup \dots \cup P_{ar(a)}$ we have also a transition $q \xrightarrow{a} \left(P'_1, \dots, P'_{ar(a)}\right)$ in $\mathcal{A}$, such that $p \in P'_1 \cup \dots \cup P'_{ar(a)}$, and so the layers of states of $\mathcal{A}$ are also layers of $\overline{\mathcal{A}}$. In such case each branch in a run of $\overline{\mathcal{A}}$ is labeled from some point only with states from one layer of $\mathcal{A}$. But this says, that such a branch has finitely many labels form B_α if and only if there is infinitely many labels from $Q \setminus B_\alpha$ on it. As the left side of this equivalence is exactly what $\neg\alpha$ says and the right side is exactly what Büchi condition with the set of accepting states $Q \setminus B_\alpha$ says, we get the lemma. □

Remark A.5. The size of $\overline{\mathcal{A}}$:
- *states* — $|Q|$
- *transitions from $q \in Q$ through $a \in \mathcal{F}$* — $\left|\delta'_{q,a}\right| = \prod_{q \xrightarrow{a} (P_1, \dots, P_{ar(a)}) \in \delta} |P_1| + \dots + \left|P_{ar(a)}\right|$
- in the case $\mathcal{A}$ is weak Büchi: *accepting states* — $|B_{\neg\alpha}| = 2^{|Q|}$

A.3 Alternation removing

One more thing that we need about alternating automata is a translation of them to nondeterministic automata. We give one construction for such translation here. The idea for it is generally widely spread, however it is mostly given for automata on infinite words and the author is not aware of a paper, where it would be given for infinite trees (equivalently APGs). While constructions for automata on words are known to be non-applicable to automata on trees, we describe this construction once more to show that this one does not follow this pattern and it can be used for trees/APGs. We give it also for completeness of the inclusion checking procedure for what this appendix stands.

Remark A.6. This is generally hard. For the Rabin acceptance condition, which would fit our needs the best, the size of resulting automata from such translation reaches $\mathcal{O}\left(2^{n \log n}\right)$ or more [MuSch95, Lö11], where n corresponds to the size of a translated automaton. Moreover, the result is another automaton with the Rabin condition, which may lead us to testing emptiness for quite big Rabin NDAs, that is known to be NP-complete [Tho90, Lö11]. That is why we limit ourselves to Büchi and weak Büchi automata in this paper.

Definition A.7. We say a run r of some AA is ***normalized*** if for each $w_1 = (k_1, q_1) \dots (k_n, q_n)$, $w_2 = (k_1, p_1) \dots (k_n, p_n)$ in *paths*(r) such that $q_n = p_n$ sub-APGs $r_{\triangle w_1}$, $r_{\triangle w_2}$ are equal.

Lemma A.8. *For each accepting run of an AA on an APG there exists a normalized accepting run of this AA on this APG.*

Proof. Let's take any accepting run r of some

AA on some APG. First we retrieve its path tree, so let $r' = \mathcal{L}ab\mathcal{P}ath\mathcal{T}ree\,(r)$. This is superfluous, but it may simplify the presentation of the construction. Now we join nodes of r' into equivalence classes be the relation: $(k_1, q_1) \ldots (k_n, q_n) \sim (l_1, p_1) \ldots (l_m, p_m)$ if and only if $n = m$, $k_1 = l_1, \ldots, k_n = l_n$ and $q_n = p_n$ (recall that nodes of r' are paths of r and each node w of r' describes the path from the root of r' to this node, that is $w = r'_{\downarrow w}$). So equivalence classes upon $\sim$ join all nodes of r' that should be equal in a normalized run. Now split nodes of r' into layers $L_0, L_1, \ldots$ such that $(k_1, q_1) \ldots (k_n, q_n)$ is in L_i if and only if the number of positions j such that $q_j \in B_\alpha$ is equal to i (or in other words, the node has exactly i ancestors labeled with accepting states in r'). Of course each node belongs to exactly one layer. Then we fix any choice function $\sigma : carr\,(r') / \sim \to carr\,(r')$ such that $\sigma\,([w]_\sim)$ is a member of $[w]_\sim$ that belongs of the layer with the least index — precisely, $\sigma\,([w]_\sim) \in [w]_\sim$, $\sigma\,([w]_\sim) \in L_i$ and for each $w' \in [w]_\sim$ if $w' \in L_j$, then $i \le j$. The normalized run can be fixed as $r'' = \langle\langle carr\,(r') / \sim, \downarrow, \imath\rangle, \varepsilon\rangle$ such that $\imath\,([w]_\sim) = r'_w$ and $[w]_\sim$ has an (i, p)'th child $[w']_\sim$ if and only if $\sigma\,([w]_\sim)$ has an (i, p)'th child in $[w']_\sim$.

The APG r'' is surely a run for $\mathcal{A}$, as for each node in r'' its children are copied from the corresponding node from r'. To show r'' is accepting note that each layer is finite — if not, there would be an non-accepting branch in r'. So for any branch $(k_1, q_1)(k_2, q_2) \ldots$ in r'' there is only finitely many positions n such that $\sigma\left(r''_{\downarrow(k_1,q_1)\ldots(k_n,q_n)}\right)$ belongs to one layer. This makes there must be infinitely many n's such that $\sigma\left(r''_{\downarrow(k_1,q_1)\ldots(k_n,q_n)}\right) \in L_i$, $\sigma\left(r''_{\downarrow(k_1,q_1)\ldots(k_{n+1},q_{n+1})}\right) \in L_j$ and $i < j$. But

$$\left(\sigma\left(r''_{\downarrow(k_1,q_1)\ldots(k_n,q_n)}\right)\right)(k_{n+1}, q_{n+1}) \sim \sigma\left(r''_{\downarrow(k_1,q_1)\ldots(k_{n+1},q_{n+1})}\right)$$

so the path $\left(\sigma\left(r''_{\downarrow(k_1,q_1)\ldots(k_n,q_n)}\right)\right)(k_{n+1}, q_{n+1})$ has at least j positions with states from B_α and $\sigma\left(r''_{\downarrow(k_1,q_1)\ldots(k_n,q_n)}\right)$ has only i such positions. Then q_{n+1} must be accepting and as there is infinitely many such n's, the whole branch is accepted. □

Definition A.9. For a closed Büchi alternating automaton $\mathcal{A} = \langle \mathcal{F}, Q, \varnothing, q_0, \varnothing, \delta, \alpha\rangle$ we define a Büchi non-deterministic automaton $ND\,(\mathcal{A})$ as

$$ND\,(\mathcal{A}) = \langle \mathcal{F}, Q \hookrightarrow \{\circ, \bullet\}, \varnothing, \{q_0 \mapsto y\}, \varnothing, \delta', \alpha'\rangle$$

where $\hookrightarrow$ means a set of partial functions (cf. Notation) and the symbols are given as

$$y = \begin{cases} \bullet, & \text{if } q_0 \in B_\alpha \\ \circ, & \text{otherwise} \end{cases} \qquad B_{\alpha'} = Q \hookrightarrow \{\bullet\}$$

$f \xrightarrow{a} \left(f_1, \ldots, f_{ar(a)}\right) \in \delta'$ if and only if

a) there exists a choice that for each $q \in Dom\,(f)$ assigns one transition $q \xrightarrow{a} \left(P_{q,1}, \ldots, P_{q,\,ar(a)}\right)$ from δ,

b) $Dom\,(f_i) = \bigcup_{q \in Dom(f)} P_{q,i}$ for all $i = 1, \ldots, ar\,(a)$, and

c) for $i = 1, \ldots, ar\,(a)$ the values of f_i are set as follows

- if $f \notin B_{\alpha'}$, then

$$f_i\,(q) = \begin{cases} \circ, & \text{if } q \notin B_\alpha \wedge \\ & \quad \exists_p\, f\,(p) = \circ \wedge q \in P_{p,i} \\ \bullet, & \text{otherwise} \end{cases}$$

- if $f \in B_{\alpha'}$, then

$$f_i\,(q) = \begin{cases} \bullet, & \text{if } q \in B_\alpha \\ \circ, & \text{otherwise} \end{cases}$$

Note that values of $f_1, \ldots, f_{ar(a)}$ are fully determined by f, a and the first conditions. In other words we have the following observation.

Observation A.10. *For each state f of $ND\,(\mathcal{A})$, symbol $a \in \mathcal{F}$ and choice $Dom\,(f) \to \delta_{q,a}$ there exists exactly one transition $f \xrightarrow{a} \left(f_1, \ldots, f_{ar(a)}\right) \in \delta'$ that satisfies the conditions b), c) for this choice.*

One may also note that for every $a \in \mathcal{F}$ we have $\varnothing \xrightarrow{a} (\varnothing, \ldots, \varnothing)$ in δ' and $\varnothing \in B_{\alpha'}$. So the empty super-state in $ND\,(\mathcal{A})$ accepts everything, same as the empty set standing on the right side of a transition in $\mathcal{A}$.

Theorem A.11. *A closed alternating Büchi automaton $\mathcal{A}$ accepts an APG if and only if $ND\,(\mathcal{A})$ accepts this APG.*

Proof. First we should note, the given automaton does not simulate each run of the original. So for showing the implication *if $\mathcal{A}$ accepts, then $ND\,(\mathcal{A})$ accepts* we consider only normalized runs. That is, we take an APG t accepted by $\mathcal{A}$ with a witness normalized run r and we settle an accepting run r' of $ND\,(\mathcal{A})$ on t.

To give some intuition, let's start from fixing a 2^Q-APG d that can be seen as a foundation of r'.

This is superfluous, but we give it to clarify the construction of r'. The letter d comes from "domain", as this APG establishes the domains of states in r'. We define d as $\langle \mathcal{P}aths\mathcal{E}nv\,(t) + \imath,\, \varepsilon \rangle$ where

$$\imath\,(k_1 \dots k_n) = \{ r_{(k_1,q_1)\dots(k_n,q_n)} \mid (k_1,q_1)\dots(k_n,q_n) \in paths\,(r) \}$$

By Observation A.10 we may note that there exists r' such that $Dom\,(r'_w) = d_w$ for each $w \in paths\,(t)$. But let's show it more precisely to guarantee its acceptance.

Let ρ_w be a transition from $\mathcal{A}$ used in r at node $r_{\downarrow w}$ for any $w = (k_1,q_1)\dots(k_n,q_n) \in paths(r)$. Precisely, $\rho_w = q \xrightarrow{a} (P_1,\dots,P_{ar(a)})$ such that $q = r_w$, $a = t_{k_1\dots k_n}$, and the node $r_{\downarrow w}$ has a child $r_{\downarrow w(i,p)}$ if and only if $i \in \{1,\dots,ar\,(a)\}$ and $p \in P_i$. Then note that for each $k_1 \dots k_n \in paths\,(d)$ and $q \in d_{k_1\dots k_n}$ we have exactly one transition in δ, let's name it $\rho_{k_1\dots k_n,q}$, such that there exists $w = (k_1,q_1)\dots(k_n,q_n) \in paths\,(r)$, $r_w = q$ and $\rho_{k_1\dots,k_n,q} = \rho_w$ — the existence can be shown with a simple induction on n and the uniqueness is guaranteed by r being normalized.

Now we set r' as $\langle \mathcal{P}aths\mathcal{E}nv\,(t) + \imath',\, \varepsilon \rangle$ where $\imath'$ is given recurrently as follows. Note that for each defined below $\imath'\,(w)$ we have $Dom\,(\imath'\,(w)) = d_w$.

- $\imath'\,(\varepsilon) = \begin{cases} \{q_0 \mapsto \bullet\}, & \text{if } q_0 \in B_\alpha \\ \{q_0 \mapsto \circ\}, & \text{otherwise} \end{cases}$

- let $\imath'\,(w) \xrightarrow{t_w} (f_1,\,\dots,\,f_{ar(t_w)})$ be the only transition in δ' for the state $\imath'\,(w)$ of $ND\,(\mathcal{A})$, the symbol t_w and the choice $\{\, p \mapsto \rho_{w,p} \mid p \in Dom\,(\imath'\,(w)) \,\}$, then

$$\imath'\,(wi) = f_i \qquad \text{for } i = 1,\dots,ar\,(t_w)$$

Assuming we believe that r' is a proper run of $ND\,(\mathcal{A})$, let's show r' is accepting. We will reason by reductio ad absurdum. First take in mind that for each path $k_1 \dots k_n \in paths\,(r')$ and $q \in r'_{k_1\dots k_n}$ there is a path $w = (k_1,q_1)\dots(k_n,q_n) \in paths\,(r)$ such that $r_w = q$. Now let's consider r' is non-accepting and so there exists a non-accepting infinite branch $\varpi = k_1k_2\dots$ in it. Let's collect all states q from $r'_{k_1\dots k_i}$ such that $(r'_{k_1\dots k_i})\,(q) = \circ$ in U_i for each $i \in \mathbb{N}$. Note that there must be such n that all U_i are not empty for $i \geq n$, as otherwise we would have infinitely many accepting states of $ND\,(\mathcal{A})$ on the branch. But for each $i \geq n$ and $p \in U_{i+1}$ there is a transition $\rho_{k_1\dots k_i,q} = q \xrightarrow{a} (P_{q,1},\dots,P_{q,\,ar(a)}) \in \delta$ such that $p \in P_{q,k_{i+1}}$ and $q \in U_i$. So for each $p \in U_{i+1}$ there is a node in $r_{\downarrow(k_1,q_1)\dots(k_{i+1},q_{i+1})}$ labeled with p for which $r_{\downarrow(k_1,q_1)\dots(k_i,q_n)}$ is labeled with $q \in U_i$. Hence, we can surely choose a branch r which from the position n is labeled with states form $\bigcup_{i\in\mathbb{N}} U_i$. As states in all U_i are not accepting for $\mathcal{A}$, the branch is non-accepting in r. ↯

The inverse is luckily easer. We take an accepting run r of $ND\,(\mathcal{A})$ for an APG t. Then for each path $w \in paths\,(t)$ and state $q \in Dom\,(r_w)$ there is the transition $q \xrightarrow{t_w} \left(P_1^{w,q},\dots,P_{ar(a)}^{w,q}\right)$ in δ chosen in the definition of δ'. So we can build a valid run r' of $\mathcal{A}$ on t as $\langle\langle (\mathbb{N}\times Q)^*, \downarrow, \imath\rangle, \varepsilon\rangle$, where $w_{\downarrow(i,p)} = w'$ if and only if $w' = w\,(i,p)$ and $p \in P_i^{w,q}$ for $q = r'_w$, and $\imath$ is defined in the only possible way (that is, $\imath\,(\varepsilon) = q_0$, $\imath\,(w\,(i,q)) = q$). The acceptance of r' can be argued as follows. For each infinite branch $(k_1,q_1)\,(k_2,q_2)\dots$ from r' there are infinitely many n that the state $r_{k_1\dots k_n} \in B_{\alpha'}$. This means that between each two such n_1 and n_2, there must be some m that $r'_{(k_1,q_1)\dots(k_i,q_m)} \in B_\alpha$, as in the other case for $i = n_1+1,\dots,n_n$ we would have $r_{k_1\dots k_i}\,(q_i) = \circ$ and $r_{k_1\dots k_{n_2}} \notin B_{\alpha'}$. Thus we have infinitely many accepting states on $(k_1,q_1)\,(k_2,q_2)\dots$. ☐

Remark A.12. The size of $ND\,(\mathcal{A})$:
- *states* — $3^{|Q|}$
- *transitions trough* $a \in \mathcal{F}$ — $|\delta'_a| = \prod_{q\in Q} 2\,|\delta_{q,a}|+1$
- *accepting states* — $|B_{\alpha'}| = 2^{|Q|}$

A.4 Checking emptiness

The ***emptiness*** property says if an automaton does not accept any APG. The complexity of testing this property depends on the class of considered automata and generally may by high — e.g., it is NP-complete for Rabin APG automata [EmJu88]. That is why we again limit ourselves and give a claim only for Büchi NDAs.

Theorem A.13. *For a closed Büchi NDA $\mathcal{A}$, there exists an algorithm checking if there exists any APG that is accepted by $\mathcal{A}$.*

This theorem is a generally known fact, nevertheless we give the algorithm here. The reason for this is, that we want to check emptiness for exponentially blown automata in this paper and so we quite do care about minimizing number of operations the algorithm performs. But this is not a matter of concern in descriptions of solutions of this problem known to the author. That is why we decide to give another description here, that evades

unneeded iterations on elements of a checked automaton — as the number of these elements may be exponential in size of some input data, it seems significant.

Definition. Let $\mathcal{A}$ be a closed NDA and $q \in Q_{\mathcal{A}}$. Then a ***q-run finite prefix*** is any $Q_{\mathcal{A}}$-APG r indexed with naturals that satisfies the conditions

- the height of r is finite,
- $r_\varepsilon = q$, and
- for each $w \in dom(r)$
  - no $r_{\triangle wi}$ is defined, or
  - there is $r_w \xrightarrow{a} (q_1, \dots, q_{ar(a)})$ in $\delta_{\mathcal{A}}$ for which $r_{w1} = q_1, \dots, r_{w\,ar(a)} = q_{ar(a)}$ and r_{wi} is undefined for $i \neq 1, \dots, ar(a)$

The set of nodes v of r which do not have children, but for which there is no transition $\imath_r(v) \xrightarrow{a} ()$ in δ, where $\imath_r$ is the symbol function of r, is called the ***cut*** of r. The states of $\imath_r(cut)$ are called ***cut states*** of r. Finally, if the root of r is in the cut, we say the prefix is ***trivial***.

Definition A.14. Let $\mathcal{A}$ be a closed Büchi automaton, $Rc : 2^{Q_{\mathcal{A}}} \to 2^{Q_{\mathcal{A}}}$ be the function where $Rc(P)$ is the set of all states q for which there exists a q-run finite prefix r with $\imath_r(cut) \subseteq P$. Then $\{G_i\}_{i \in \mathbb{N}}$ is the sequence such that $G_0 = B_{\alpha_{\mathcal{A}}}$ and $G_{i+1} = Rc(G_i) \cap B_{\alpha_{\mathcal{A}}}$.

Observation A.15. *Note that the sequence $\{G_i\}_{i \in \mathbb{N}}$ is decreasing, so there must exist the smallest index n such that $G_n = G_{n+1}$.*

Lemma A.16. *A Büchi NDA $\mathcal{A}$ is not empty if and only if $q_{0\mathcal{A}} \in Rc(G_n)$, for n from Observation A.15.*

Proof. For the right-to-left implication things are simple. For each $q \in G_n$ a q-run f. prefix with all its cut states in G_n, thus also in $B_{\alpha_{\mathcal{A}}}$, is guaranteed. So we can create an accepting q-run of $\mathcal{A}$ for each of these states by composing the guaranteed prefixes. Now, if we take a $q_{0\mathcal{A}}$-run f. prefix which is a witness of $q_{0\mathcal{A}} \in Rc(G_n)$ and compose it with the constructed runs, we acquire an accepting $q_{0\mathcal{A}}$-run. An APG for which the run is accepting can be then retrieved from the run itself.

For showing the inverse, we take some accepting run r for $\mathcal{A}$, then we fix the set *Accept* of all states from $B_{\alpha_{\mathcal{A}}}$ which occur in r infinitely often and we decompose r to prefixes by "cutting" it at states of *Accept*. So for each $q \in Accept \cup \{q_{0\mathcal{A}}\}$ we have at least one q-run f. prefix with all its cut states in *Accept*. Then by an induction on i we can show $Accept \subseteq G_i$ — it is straightforward as the prefixes from decomposition of r are witnesses of states from *Accept* being in succeeding $Rc(G_i)$ and thus in G_{i+1}. That makes the right side of the lemma satisfied. □

We are now ready to give the algorithm for checking emptiness. This may be little unintuitive at at the first glance, but during the proof of its correctness we give some invariants that, hopefully, will help in understanding how it works.

Definition A.17. For a closed Büchi NDA $\mathcal{A} = \langle \mathcal{F}, Q, \varnothing, q_0, \varnothing, \delta, \alpha \rangle$ the algorithm is given as follows.

Preprocessing.

1. with each state q associate a set $producers_q$ of $\left\{ p \xrightarrow{a} (q_1, \dots, q_{ar(a)}) \in \delta \;\middle|\; \exists_i\, q_i = q \right\}$ and a counter $available_q$ set to 0
2. with each transition $\rho = p \xrightarrow{a} (q_1, \dots, q_{ar(a)})$ associate a counter $available_\rho$ set to 0 and a counter $deadProducts_\rho$ set to $\left|\{q_1, \dots, q_{ar(a)}\}\right|$
3. initialize a queue $\hat{G}_0$ and put all states from B_α into it

Sub-algorithm "Next Layer" that for a given G_i finds $Rc(G_i)$ and G_{i+1}.

Input: $\hat{G}_i$

1. rename $\hat{G}_i$ to *cameAlive*
2. set new $\hat{G}_{i+1}$ as an empty queue
3. for each q such that there exists a transition $q \xrightarrow{a} () \in \delta$, set $available_q$ to $i+1$ and enqueue q into *cameAlive*
4. repeat until *cameAlive* is empty:
   a) dequeue q from *cameAlive* and for each $\rho = p \xrightarrow{a} (q_1, \dots, q_{ar(a)})$ in $producers_q$:
      i. if $available_\rho \neq i$:
         A. remove ρ from $producers_q$
      ii. else:
         A. decrement $deadProducts_\rho$
         B. if $deadProducts_\rho$ became 0 in the last step, set both $available_\rho$, $available_p$ to $i+1$ and $deadProducts_\rho$ to $\left|\{q_1, \dots, q_{ar(a)}\}\right|$
         C. if $available_p$ changed in the last step, enqueue p into *cameAlive* and if p is also in B_α, enqueue it into $\hat{G}_{i+1}$

Main algorithm testing emptiness.

1. do the preprocessing
2. run Next Layer algorithm on succeeding $\hat{G}_i$ staring from $\hat{G}_0$ until the number of elements of $\hat{G}_i$ does not decrease
3. if $\hat{G}_n$ is the last processed $\hat{G}_i$, then return $\mathcal{A}$ is not empty if $available_{q_0} = n + 1$, otherwise return it is empty

Lemma A.18. *Any state occurs in $\hat{G}_i$ at most once.*

Proof. The preprocessing guarantees it for $i = 0$. Later, during a run of Next Layer, when a state is put into $\hat{G}_{i+1}$ its *available* counter is set to $i + 1$ and during this run cannot be changed to anything else. Thus the condition in 4(a)ii.C cannot be passed second time. □

Lemma A.19. *Between runs of Next Layer $G_i = \{ q \mid q \text{ is in } \hat{G}_i \}$ and $Rc(G_i) = \{ q \mid available_q = i + 1 \}$.*

Proof. For showing $\{ q \mid q \text{ is in } \hat{G}_i \} \subseteq G_i$ and $\{ q \mid available_q = i + 1 \} \subseteq Rc(G_i)$ let's note the following invariants — assuming that we enumerate runs of Next Layer from 0.

1. at the beginning of an i'th run of Next Layer all counters $available_q$ and $available_\rho$ are less then or equal to i
2. when running Next Layer i'th time a state can be put into *cameAlive* at most once
3. at the beginning of an i'th run of Next Layer, for each transition $\rho = p \xrightarrow{a} (q_1, \ldots, q_{ar(a)})$ if $available_\rho = i$, then $deadProducts_\rho = |\{q_1, \ldots, q_{ar(a)}\}|$
4. when running Next Layer i'th time if q is in *cameAlive*, then there exists a q-run f. prefix (possibly trivial) with all its cut states included in $\{ q \mid q \text{ is in } \hat{G}_i \}$

All invariants can be shown by an induction on number of runs of Next Layer. The first two are independent from others. The third can be shown using the two previous ones and the last using the two in the middle. Now with the fourth invariant it is obvious that the desired inclusions hold, as we can do a simple induction on number of runs of Next Layer with the claim that states on left sides of the inclusions except these of $\hat{G}_0$ must be put into *cameAlive* — the induction is needed to settle that the cut states of prefixes from the invariant 4 are in G_i.

To show the inverse inclusions we prove by an induction on number of runs of Next Layer that after i'th run the inclusions $\{ q \mid q \text{ is in } \hat{G}_{i+1} \} \supseteq G_{i+1}$, $\{ q \mid available_q = i + 1 \} \supseteq Rc(G_i)$ and the following invariant holds.

5. for each transition $\rho = p \xrightarrow{a} (q_1, \ldots, q_{ar(a)})$ where $ar(a) > 0$ if all $q_j \in Rc(G_i)$, then $available_\rho = i$ at the beginning of the i'th run of Next Layer and through the whole run ρ is in $producers_{q_j}$ for all q_j

For the iteration 0 the invariant is trivial. For an iteration $i > 1$ if we take a mentioned transition $\rho = p \xrightarrow{a} (q_1, \ldots, q_{ar(a)})$ with all $q_j \in Rc(G_i)$, then all $q_j \in Rc(G_{i-1})$ by the monotonicity of the sequence $\{Rc(G_i)\}$. So from the induction assumption flows that $available_\rho = i - 1$ at the beginning of the run $i - 1$, the transition ρ is in $producers_{q_j}$ for all q_j through the run and $available_{q_j} = i$ at the end of the run — the last one is from the second inclusion for the run $i - 1$. As we also know that the counters $available_{q_j} < i$ at the beginning of the run $i - 1$, all q_j must pass through the step 4(a)ii.C during this run and so they must be put into *cameAlive*. Thus after dequeuing all of them $available_\rho$ is set to i and the transition is never considered any more during this run, as no q_j can be put again into *cameAlive*. This makes the invariant hold at the beginning of the run i. The only thing to do now about the invariant is showing that ρ cannot be removed from any $producers_{q_j}$ during this run, which is straightforward, as until $available_\rho = i$ it cannot be removed and after setting it to $i + 1$ the transition is not considered again during the run.

Now we need only to show the goal inclusions. This may be done by an induction on height of non-trivial run f. prefixes with cut states included in G_i. Precisely we show that if we have such a prefix r, then $available_{r_\varepsilon}$ is set to $i + 1$ during the i'th run. Such induction is enough to prove the goal inclusions as

- for each state q from $Rc(G_i)$ a prefix like this with $r_\varepsilon = q$ is guaranteed and once $available_q$ is set to $i + 1$ it cannot be undone during the run,
- for each state q from $\hat{G}_{i+1}$ the state is also in $Rc(G_i)$, so by the above $available_q$ is set to $i + 1$ and while $available_q \leq i$ at the beginning of the run the step 4(a)ii.C must even-

tually be positively passed.

So let's consider cases. If the height is 1, then $available_{r_\varepsilon}$ is set to $i+1$ by the step 3 of Next Layer, as r cannot be trivial and so there must exist some transition $r_\varepsilon \xrightarrow{a} () \in \delta$. In the case of the height being greater then 1 we are going to show that for each state r_j the counter $available_{r_j} = i$ at the beginning of the run and r_j is put into *cameAlive* during it. With these claims setting $available_{r_\varepsilon} = i+1$ is straightforward — some transition $\rho = r_\varepsilon \xrightarrow{a} (r_1, \dots, r_{ar(a)})$ must be in all $producers_{r_j}$ through the run and at the beginning of the run the counter $deadProducts_\rho$ is fixed as the number of r_j states (cf. inv. 5, 3), so dequeuing all r_j must cause a positive pass through the step 4(a)ii.B. Now let's take care about the $available_{r_j} = i$. For $i = 0$ it is trivial. For $i > 0$ note that all $r_j \in Rc(G_{i-1})$. So by the induction on number of runs of Next Layer we have $\{ q \mid available_{r_j} = i \} \supseteq Rc(G_{i-1})$ at the end of the run $i-1$ and then $available_{r_j} = i$ also at the beginning of the i'th iteration. Thus the last thing to do is to show the r_j states are enqueued into *cameAlive*. For each j such that r has an j'th child we consider two cases: $r_{\triangle j}$ being trivial or not. If $r_{\triangle j}$ is not trivial, then surely it is lower then r. So by the induction hypothesis we know that the $available_{r_j}$ is set to $i+1$ during the run and as $available_{r_j} = i$ at the beginning of the run the step 4(a)ii.C must be positively passed during it. Thus the state is put into *cameAlive*. In the case $r_{\triangle j}$ is trivial, we have to note that $\{ q \mid q \text{ is in } \hat{G}_i \} \supseteq G_i$ at the beginning of the current run. For $i > 0$ this flows from the assumption of the induction on number of runs and for $i = 0$ from looking at the preprocessing. Then, as the state r_j must be in G_i, it becomes a member of *cameAlive* at the step 1 of Next Layer. □

Theorem A.20. *An NDA $\mathcal{A}$ is non-empty if and only if the algorithm claims it is non-empty.*

Proof. For testing emptiness the main algorithm builds succeeding G_i and $Rc(G_i)$, where G_i is mentioned in the previous lemma. So, as $\{G_i\}$ is decreasing wrt to inclusion and there are no duplicates in $\hat{G}_i$ the condition in step 2 stops the iteration when two successively constructed G_i and G_{i+1} become equal. This makes the queue $\hat{G}_n$, indeed, stand for the G_n from Lemma A.16 and checking $available_{q_0} = n+1$ is equivalent to checking $q_0 \in Rc(G_n)$ by Lemma A.19. □

Remark A.21. The time complexity of the algorithm we gave is $\mathcal{O}(|Q| + size(\delta) + size(\delta)|B_\alpha|)$, where $size(\delta) = \sum_{q \xrightarrow{a} (\dots) \in \delta} ar(a) + 1$. The $|Q| + size(\delta)$ component comes from the preprocessing. Then the main algorithm can perform maximally $|B_\alpha| + 1$ runs of Next Layer and each such a run can be completed in $\mathcal{O}(size(\delta))$ plus the cost of dequeuing states from *cameAlive* that become members of it in step 1 of Next Layer. But in each i'th run of Next Layer, for $i > 0$ these members come from processing some transition in δ, so their number is bound by $size(\delta)$, and for $i = 0$ it is bound by the number of states in the automaton. Thus summarizing we get what we want.

The space complexity is $\mathcal{O}(|Q| + size(\delta))$. This is trivial.

A.5 Checking inclusion

Recalling — inclusion checking stands for the problem of testing if each APG accepted by one automaton is also accepted by another. In this paper we are interested in checking such inclusion for closed weak Büchi NDAs. The procedure for this is given in the following observation.

Observation A.22. *Let $\mathcal{A}_1$, $\mathcal{A}_2$ be closed weak Büchi NDAs over $\mathcal{F}$ alphabet. Then $\mathcal{A}_1$ is included in $\mathcal{A}_2$ if and only if $\mathcal{A}_1 \cap ND\left(\overline{Alt(\mathcal{A}_2)}\right)$ is empty.*

The thesis is straightforward, as by the previous theorems $ND\left(\overline{Alt(\mathcal{A}_2)}\right)$ accepts exactly these APGs that are not accepted by $\mathcal{A}_2$. The ND construction applies to $\overline{Alt(\mathcal{A}_2)}$ as by Remark A.4 the automaton $\overline{Alt(\mathcal{A}_2)}$ is Büchi.

As we fixed a proper setting for building $\mathcal{A}_1 \cap ND\left(\overline{Alt(\mathcal{A}_2)}\right)$ and testing its emptiness, we then have a decidable procedure for testing the inclusion. We estimate the cost of this procedure below.

Remark A.23. The time complexity of inclusion testing is $\mathcal{O}(K + L + LM)$ and the space complexity is $\mathcal{O}(K + L)$, where K, L, M correspond to the size of $\mathcal{A}_1 \cap ND\left(\overline{Alt(\mathcal{A}_2)}\right)$ as follows.

- $K = |Q_1| 3^{|Q_2|}$ is the number of states
- $L = \sum_{a \in \mathcal{F}} (ar(a)+1) |\delta_{1_a}| \prod_{q \in Q_2} \left(2ar(a)^{|\delta_{2\,q,a}|} + 1\right)$ stands for *size* of the transition relation stated in the complexity remark A.21 for testing emptiness
- $M = |B_{\alpha_1}| 2^{|Q_2|}$ is the number of accepting states

where indexes 1, 2 says from which automaton,$\mathcal{A}_1$

or $\mathcal{A}_2$, respectively, a symbol comes.

Note that L can be little bit more figuratively bound, as

$$\prod_{q\in Q_2}\left(2ar\left(a\right)^{\left|\delta_{2\,q,a}\right|}+1\right)\leq 3^{|Q_2|}\left(ar\left(a\right)+1\right)^{\left|\delta_{2\,a}\right|}$$

Thus the procedure is straightly single exponential, what is a good result in context of automata on infinite trees/APGs. Still processing such big automata even for small $\mathcal{A}_1$ and $\mathcal{A}_2$ may be out of capabilities of current personal computers (June 2014) which we consider our targets. Then for what sake have we done all this work? The answer is — because this estimation does not take in account that during construction of $\mathcal{A}_1 \cap ND\left(\overline{Alt\left(\mathcal{A}_2\right)}\right)$ we can omit non-reachable states, what may result in tremendous decrease of the size of the automaton. E.g. for deterministic $\mathcal{A}_2$ after removing non-reachable states this size is not even exponential. More precisely, it may be estimated as

- $K = |Q_1|\,|Q_2|$
- $L = \sum_{a\in\mathcal{F}}\left(ar\left(a\right)+1\right)\left|\delta_{1_a}\right|ar\left(a\right)|Q_2|$
- $M = |B_{\alpha_1}|\left(|Q_2| - |B_{\alpha_2}|\right)$

The reason for this is that the complement of a deterministic automaton is, in fact, a non-deterministic automaton and all super-states in the construction of the complement that simulate more then one state of the original automaton are non-reachable.

This, somehow, follows intuitions flowing from studying the equation for L, as the exponent $\left|\delta_{2\,q,a}\right|$ describes the level of non-determinism of $\mathcal{A}_2$. Hence, despite the tight bound for the complexity depends on the structure of $\mathcal{A}_2$, we can say with good portion of probability that the complexity of the given procedure (with non-reachable states omitted) decreases significantly with rise of determinism of $\mathcal{A}_2$. Summarizing, for many automata we can effectively compute inclusion checking.

One thing that we still need to do is strengthening the above theorem a little bit to meet our theorem from the main part of the paper. This goes as follows.

Theorem A.24. *Let $\mathcal{A}_1$, $\mathcal{A}_2$ be closed weak Büchi NDAs over alphabet $\mathcal{F}$ and env be an $\mathcal{F}$-APG environment closed over $\mathcal{A}_1$. Then the inclusion $\mathcal{A}_1^{env} \subseteq \mathcal{A}_2^{env}$ holds if and only if $\mathcal{A}_1$ is included in $\mathcal{A}_2$.*

Proof. One implication is obvious and the second is obvious by Theorem 2.17. □

Finally, the last thing we are left with is remarking that the above procedure works also for weak Büchi alternating automata, what is obvious as we need only to remove *Alt* in Theorem A.22, and saying that K, M parameters look the same as in the NDAs' case and L is upper bounded by

$$\sum_{a\in\mathcal{F}}\left(ar\left(a\right)+1\right)\left|\delta_{1_a}\right|\prod_{q\in Q_2}\left(2\left(ar\left(a\right)|Q_2|\right)^{\left|\delta_{2\,q,a}\right|}+1\right)$$

where we may again bound the factor

$$\prod_{q\in Q_2}\left(2\left(ar\left(a\right)|Q_2|\right)^{\left|\delta_{2\,q,a}\right|}+1\right)$$

with

$$3^{|Q_2|}\left(ar\left(a\right)|Q_2|+1\right)^{\left|\delta_{2\,a}\right|}$$

Appendix B

What is wrong with the automata for fixed points in the paper [Niw97]

It seems there is a missed case in the construction of equivalent automata for expressions that starts from the least and the greatest fixed-point operators in [Niw97] — specifically, in proofs of lemmas 3.4 and 3.5. In that paper automata $\mathcal{A}'$ and $\mathcal{A}''$, which correspond respectively to our $\mu x_{aut}(\mathcal{A})$, $\nu x_{aut}(\mathcal{A})$, use simple $\delta_{\mathcal{A}}$ in the place of our $\delta'_{\mathcal{A},x}$ (cf. def. 3.25), what appears to be not sufficient.

A counterexample may be here the automaton $\mathcal{A}'$ for $\mathcal{A} = \langle\{a/1, b/0\}, \{q_0\}, \{x, y\}, q_0, \{y\}, \left\{q_0 \xrightarrow{a} (x)\right\}, \bot\rangle$. The least fixed point of the function described by this automaton is $\mu x.y \cup a(x)$, where the variables range through sets of values in some arbitrary algebra of the signature $\{a/1, b/0\}$. So if we set the valuation of the variable y to $\{b\}$, then $a(b)$ should be in the fixed point for any interpretation of the symbols a and b. Now by the definition

$$\begin{aligned}\mathcal{A}' = \langle &\mathcal{F}, Q_{\mathcal{A}} \cup \{x\}, V_{\mathcal{A}} \setminus \{x\}, x, V_{0_{\mathcal{A}}} \setminus \{x\},\\ &\delta_{\mathcal{A}} \cup \left\{x \xrightarrow{a} \overrightarrow{q} \mid q_{0_{\mathcal{A}}} \xrightarrow{a} \overrightarrow{q} \in \delta_{\mathcal{A}}\right\},\\ &\{\langle L_1 \cup \{x\}, U_1\rangle, ..., \langle L_n \cup \{x\}, U_n\rangle\}_{\alpha}\rangle\end{aligned}$$

where $\{\langle L_1, U_1\rangle, ..., \langle L_n, U_n\rangle\}_{\alpha}$ is the Rabin condition of $\mathcal{A}$. In our case this means that $\mathcal{A}' = \left\langle \{a/1, b/0\}, \{q_0, x\}, \{y\}, x, \{y\}, \left\{q_0 \xrightarrow{a} (x), x \xrightarrow{a} (x)\right\}, \bot\right\rangle$. But this automaton clearly do not accept $a(b)$ wrt the valuation $y \mapsto \{b\}$ and any interpretation of a, b, as it accepts only y. $\lightning$

The issue may be easily identified if we take a closer look at the proof of the lemma 3.4. Precisely, in the third case of Ad (ii) we cannot do the requested substitution when the accepting run of A' on $r \uparrow_1 (w)$ is lazy. This is where $\delta'_{\mathcal{A},x}$ comes in handy, as it covers exactly this case of the proof and the rest seems to be valid.

The problem in the case of $\mathcal{A}''$ is analogous, however the proof of the lemma 3.5, which is responsible for it, is put in terms of a reference to another proof, so we do not go through it.

Bibliography

[TATA] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, M. Tommasi. *Tree Automata Techniques and Applications.* Draft.

[EmJu88] E. Allen Emerson, Charanjit S. Jutla. *The Complexity of Tree Automata and Logics of Programs.* In Proceedings of the 29th Annual Symposium on Foundations of Computer Science (FOCS'88), pages 328-337. IEEE Comp. Soc. Press, 1988.

[Lö11] Christof Löding. 2011. *Automata on Infinite Trees.* Draft.

[MuSch95] D. E. Muller, P. E Schupp. 1995. *Simulating alternating tree automata by nondeterministic automata: New results and new proofs of the theorems of Rabin, McNaughton and Safra.* Theoretical Computer Science, vol.141, pages 69 - 107.

[Niw97] Damian Niwiński. 1997. *Fixed Point Characterization of Infinite Behavior of Finite-State Systems.* Theor. Comput. Sci. 189(1-2): 1-69.

[Pc] Benjamin C. Pierce. *Types and programming languages.* MIT Press 2002, ISBN 978-0-262-16209-8.

[Tho90] Wolfgang Thomas. 1990. *Automata on infinite objects* in *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics.* Elsevier and MIT.