Higher-order Büchi types

Martin Hofmann

Institute for Informatics, LMU Munich

Workshop on Higher-Order Model Checking (HOMC)
Singapore, 21 September 2016

Motivation: Programming guidelines for secure web-programming

- Invoke the right sanitization function according to context (FAST 2011)
- Request appropriate authorization prior to data access (HOFM 2015)
- Check passwords
- Avoid certain features in high-security context
- Write log information at regular intervals
- Eventually close open connections

Enforcing such guidelines with type systems

- Formulate guideline as finite automaton to be run on all possible program traces of annotated program
- Use aspect-orientation to define instrumentation (with ghost variables and special event-throwing instructions).
- Formulate type system capable of predicting outcome *without* actually instrumenting the program and *without* actually running the automaton.

This talk: focus on Büchi automata for infinitary behaviours.

State-of-the-art

- Aderhold & Mantel enforce guidelines with LTL model checking,
- Annamaa & Vene devise string analysis against SQL injection. Many other applications of static analysis to counter particular threats.
- Grabowski, MH, Li use types based on finite automata to get fine-grained string analysis to enforce sanitization policy
- Beringer, Grabowski, MH describe region-based type-and-effect system for Java subsuming and certifying various analysis notably points-to, alias, context-sensitive
- Jeffrey translates LTL into dependent types using first-order semantics
- Vast body of work on model checking of pushdown systems
- Ong and Kobayashi give intersection type system equivalent to model checking of mu-calculus against HO-pushdown systems.

Type system vs. model checking (very contentious)

- Type system naturally extends existing practice in Java / ML, etc.
- Deals well with higher-order functions, modules, objects, etc. without having to go through low-level encodings.
- Offers some feedback to users.

Conundrum with typing rules for infinite behaviours

Suppose $a()$ issues an a event, $b()$ issues a b event and that $GFb = (a^* b)^\omega$ is a (type-and-effect) type asserting "infinitely many" b-events.

Consider the procedure

$$f() = a(); f()$$

Assuming $f() : GFb$ we obtain $a(); f() : aGFb$ (formally: $f() : GFb \vdash a(); f() : aGFb$ hence $f() : GFb \vdash a(); f() : GFb$ since $aGFb$ and GFb describe the same language.

Thus, assuming a purported type (GFb) for a recursive call, we were able to ascribe that very type for the procedure's body.

Under the usual (Java, ML) typing rules, this suffices to establish $\vdash f() : GFb$ (unconditionally) which is clearly wrong.

Our solution

If $f() = e$ is a recursive procedure (with body e) derive a typing

$$f() : X_f \vdash e : T(X_f)$$

where $T(X_f)$ is a type *expression* containing the *type variable* X_f.

Then conclude (unconditionally) $\vdash f() : \text{gfp } X_f.T(X_f)$. In particular, if $T(X_f) = U \cup VX_f$ then $\text{gfp } X_f.T(X_f) = V^*U \cup V^\omega$.

Traces

Let Σ be a finite set of events. We denote $\Sigma^{\leq\omega}$ the set of finite and infinite sequences over Σ.

For expression e (assuming an ambient program) we write $L_*(e)$ for the set of (finite!) traces of terminating executions of e, thus $w \in L_*(e)$ if the execution of e may terminate with finite trace w. ("may" = for some initial store, input, non-deterministic choice, etc.)

We write $L_{\leq\omega}(e)$ for the set of finite and infinite traces of **nonterminating** executions of e.

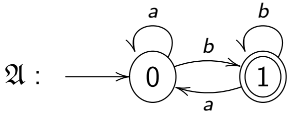
These languages can be defined by a structured operational semantics. We omit the details.

Policy automaton

We assume a Büchi automaton $\mathfrak{A}$ and want to ascertain (using a type system) that whenever $w \in L_*(e)$ or $w \in L_{\leq\omega}(e)$ then w is accepted by $\mathfrak{A}$.

Finite words are accepted by the Büchi automaton by running it as an NFA.

Example:



Finite words must end in b; infinite words must contain infinitely many b.

Equivalence relation

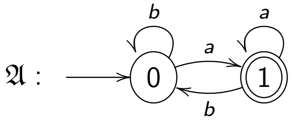
On finite, nonempty, words one defines an equivalence relation $u \sim v$ by

$$\begin{aligned} u \sim v \iff \forall q, q'.(u : q \to q' \Leftrightarrow v : q \to q')\wedge \\ (u : q \to_F q' \Leftrightarrow v : q \to_F q') \end{aligned}$$

where $u : q \to q'$ indicates the existence of a path labelled u in $\mathfrak{A}$ from q to q' and where $u : q \to_F q'$ indicates the existence of such a path which in addition contains a final state.

One formally adds to the equivalence classes of $\sim$ a class $[\epsilon]$ for the empty word.

Example



Equivalence classes: $[\epsilon]$ and $[a] = (a+b)^*a$ and $[b] = b^+$ and $[ab] = (a+b)^*b - [b]$.

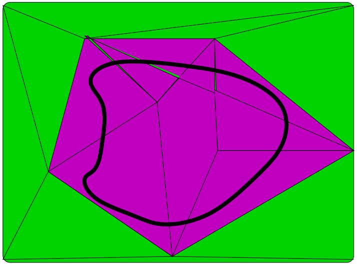
We have $[a][a] = [b][a] = [ab][a] = [a]$ and $[b][b] = [b]$ and $[a][b] = [a][ab] = [b][ab] = [ab][b] = [ab][ab] = [ab]$.

Now $(ab)^\omega \in [ab][ab]^\omega \cap [a][a]^\omega$, but $[ab][ab]^\omega \neq [a][a]^\omega$ because $a^\omega \in [a][a]^\omega \setminus [ab][ab]^\omega$.

Abstraction of languages of finite words

Define $\mathcal{M}_* = \mathcal{P}(\Sigma^*/\sim)$. For language $L \subseteq \Sigma^*$ define the ("Nerode") abstraction by

$$\alpha(L) = \{U \mid U \cap L \neq \emptyset\}$$



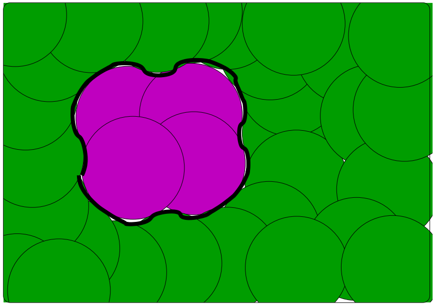
We have a concretization function $\gamma(\mathcal{U}) = \bigcup_{U \in \mathcal{U}} U$ so that α, γ form a Galois connection.

The abstraction is faithful w.r.t. to the policy automaton because $L \subseteq L(\mathfrak{A}) \iff \gamma(\alpha(L)) \subseteq L(\mathfrak{A})$.

Thus, we can analyse finite traces on the level of the abstraction without losing anything regarding acceptance or rejection by the policy automaton.

This is the essence of our automata-based type system for finite traces and strings (FAST2011).

The Büchi-abstraction



Let's do the same thing for infinite words. Classes $\leftrightarrow$ "Patches"
($=$ languages of the form UV^ω with U, V classes.)
Patches are similar to equivalence classes, but in general not disjoint!

Theorem (Büchi)

If U, V are such classes then if $w \in L(\mathfrak{A})$ and $w \in UV^\omega$ then $UV^\omega \subseteq L(\mathfrak{A})$.
For every word $w \in \Sigma^{\leq\omega}$ there exist classes U, V with $UV = U$ and $VV = V$ such that $w \in UV^\omega$.

Patches

Write $\mathcal{C} := \{(C, D) \mid C, D \in \mathcal{Q} \wedge CD = C, DD = D\}$.

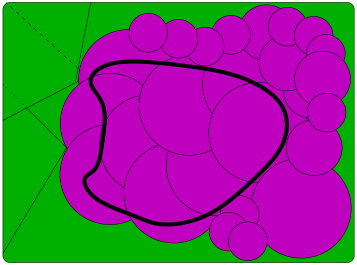
In the example the set $\mathcal{C}$ is
$\{([\epsilon], [\epsilon]), ([a], [\epsilon]), ([a], [a]), ([b], [\epsilon]), ([b], [b]), ([ba], [\epsilon]),$
$([ba], [a]), ([ba], [ba])\}$.

For $\mathcal{V} \subseteq \mathcal{C}$ define $\gamma(\mathcal{V}) = \bigcup_{(U,V) \in \mathcal{V}} UV^{\omega}$.

Closed sets

Call $\mathcal{V}$ closed if $UV^\omega \cap \gamma(\mathcal{V}) \neq \emptyset$ and $UV = U, VV = V$ implies $(U, V) \in \mathcal{V}$. "If a patch meets the thing throw it in."

We define $\mathcal{M}_{\leq\omega}$ (an infinitary cousin of $\mathcal{M}_*$) as the complete lattice of closed sets.



In the example, the set $\{([ba], [b])\}$ is not closed; we must add $([ba], [b]), ([b], [b])$, $([b], [ba]), ([\epsilon], [b])$, $([\epsilon], [ba])$.

In fact, $\mathcal{M}_*$ can be identified with a sublattice of $\mathcal{M}_{\leq\omega}$.

The Büchi abstraction

For $L \subseteq \Sigma^{\leq\omega}$ define $\alpha(L)$ as the least closed set containing $\{(C,D) \mid CD = C, DD = D, CD^\omega \cap L \neq \emptyset\}$.

Lemma (Galois connection)

$$\alpha(L) \subseteq \mathcal{V} \iff L \subseteq \gamma(\mathcal{V})$$

Corollary (of Büchi's result)

$L \subseteq L(\mathfrak{A}) \iff \alpha(L) \subseteq L(\mathfrak{A})$.

Language operations on the level of the abstractions

The idea is to build types from the abstractions.

Essentially, the abstractions play the role of effects; the usual machinery consisting of function types, regions, polymorphism, . . . , can be built around them.

It remains, however, to match the language operations arising from the typing rules on the level of the abstractions.

For concatenation, union, least fixpoints this is either obvious or follows abstractly from the fact that we have a Galois connection.

Since, however, Galois connections do *not* in general commute with greatest fixpoints, the case of ω-iteration required a special and nontrivial treatment:

Infinite iteration

We seek an operation $(-)^{(\omega)}$ so that for $L \subseteq \sigma^*$ we have

$$\alpha(L^\omega) = \alpha(L)^{(\omega)} \qquad (**)$$

It is tempting to define $\mathcal{U}^{(\omega)}$ (for $\mathcal{U} \in \mathcal{M}_*$) as the greatest solution of $X = \mathcal{U} \cdot X$. Then, however, the desired property above is not true!

We rather put

$$\mathcal{U}^{(\omega)} = \alpha(\gamma(\mathcal{U})^\omega)$$

Note that this can be effectively computed from $\mathcal{U}$.

The proof that (**) holds is still nontrivial and requires

Lemma

Let $(L_i)_{i\geq 1}$ be a family of classes and put $P = \prod_{i\geq 1} L_i \subseteq \Sigma^{\leq\omega}$, i.e., P comprises finite or infinite words of the form $w_1 w_2 w_3 \ldots$ where $w_i \in L_i$ for $i \geq 1$. There exist classes $U, V \in \mathcal{Q}$ where $UV = U, VV = V$ such that $P \subseteq UV^\omega$.

Typechecking & inference

We can use this to devise a type system with judgements of the form $\Gamma \vdash_{\mathfrak{A}} e : \tau \& (\mathcal{U}, \mathcal{V})$ with $\mathcal{U} \in \mathcal{M}_*$ and $\mathcal{V} \in \mathcal{M}_{\leq\omega}$.

It means that terminating executions of e in an environment respecting Γ yield results of type τ and their trace matches $\mathcal{U}$. Nonterminating executions (without result) match $\mathcal{V}$. Exemplary typing rule

$$(\textsc{T-Let})\ \frac{\begin{array}{c}\Gamma \vdash_{\mathfrak{A}} e_1 : \tau_1 \ \& \ (\mathcal{U}_1, \mathcal{V}_1) \\ \Gamma, x : \tau_1 \vdash_{\mathfrak{A}} e_2 : \tau_2 \ \& \ (\mathcal{U}_2, \mathcal{V}_2)\end{array}}{\Gamma \vdash_{\mathfrak{A}} \textbf{let } x = e_1 \textbf{ in } e_2 : \tau_2 \ \& \ (\mathcal{U}_1 \cdot \mathcal{U}_2, \mathcal{V}_1 \cup \mathcal{U}_1 \cdot \mathcal{V}_2)}$$

The typing rule for recursive procedures uses lfp on the finitary part and $(-)^\omega$ on the infinitary part.
The typing rules are sound and even complete assuming that all state dependencies of the control-flow are abstracted as non-determinism.

Inference by constraint solving (over a finite domain!)

Fairness

Fairness is often used as an abstraction of timing constraints:

```
0   #define TIMEOUT 65536
1   while (true) {
2     i = 0;
3     while (i++ < TIMEOUT && s != 0) {
4       unsigned int s = auth(); /* a(); */
5     }  /* c(); */
6     work(); /* b(); */
7   }
```

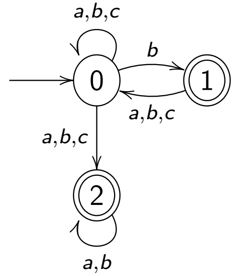
Show that line 6 is executed infinitely often assuming (fairness) that the loop 3 always terminates.

Abstracted program

$$\begin{aligned} f &= g\,;\, b()\,;\, f \\ g &= \text{if } (*)\ (a()\,;\, g)\ \ \text{else } c() \end{aligned}$$

We are then interested in the property "infinitely many *b*" assuming that "infinitely often *c*" (fairness) or equivalently: "infinitely many *b* or finitely many *c*."

Policy automaton

$\mathfrak{A}$:


Related work

- Higher-order model checking (Ong, Kobayashi et al): Type system for μ-calculus formulas on higher-order recursion schemes.
  - On traces Büchi is equally (actually more) powerful as μ-calculus.
  - Once equivalence classes and patches are tabulated analysis is inexpensive.
  - Relationship with abstract interpretation.
- Work on infinitary semigroups (Pin, Wilke, et al). Recast Büchi's result in terms of semigroups. Defines "Wilke algebra" as semigroup with ω-iteration. Our results generalise Wilke algebras to lattices and abstract interpretation.
- Model checking pushdown systems. (Esparza, Bouajjani, Schwoon)
- Type-based model checking (Skalka et al). Use type system to derive pushdown system from complex program. Then use "off-the-shelf" model checking to verify extracted pushdown system. We hope to achieve better integration & feedback

Higher-order functions

- So long as higher-order functions are used only as "extended macros" nothing essentially new happens:
- Use type-and-effect system with function types of the form $\tau_1 \overset{(\mathcal{U},\mathcal{V})}{\to} \tau_2$.
- In addition: effect variables and regions to accommodate interprocedural and pointer-sensitive analysis [Beringer, Grabowski, H.]
- The annotations $\mathcal{U}, \mathcal{V}$ are the same as before **provided** that recursive definitions are only at first order.
- If we allow the recursive definition of **higher-order** functions then new phenomena appear, e.g. trace languages beyond Type-2.
- Igor Walukiewicz: need to consider (ω) tree languages to handle this (private conversation in 2014).
- Our new results (with Jérémy Ledent): no trees needed, but some category theory.

Syntax for higher-order

- Simply-typed lambda calculus with one base type o representing “commands”
- basic commands $a : o$ for $a \in \Sigma$.
- conditionals: $+ : o \to o \to o$ (since we abstract guards anyway we don’t even have them for simplicity)
- sequential composition: $; : o \to o \to o$
- λ-abstraction and application,
- Recursion $\mathsf{fix} : (\tau \to \tau) \to \tau$.

Operational Semantics

We assume that the alphabet Σ contains all the basic commands ("events") and a special symbol $\checkmark$; each unrolling of a recursive definition emits a $\checkmark$-symbol. In this way, traces of nonterminating computations are always infinite. Thus,

- with each closed term $e : o$ we associate $L_*(e) \subseteq \Sigma^*$ representing finite terminating computations, e.g. $L_*(a; a + a) = \{a, aa\}$.
- …and $L_\omega \subseteq \Sigma^\omega$ representing nonterminating computations. We have in particular $L_\omega(\mathrm{fix}\ e\ e_1 \ldots e_k) = \checkmark L_\omega(e(\mathrm{fix}\ e)\ e_1 \ldots e_k)$. E.g. $L_\omega(\mathrm{fix}(\lambda x.x) + \mathrm{fix}(\lambda x.a; x)) = \checkmark^\omega + (\checkmark a)^\omega$.
- As before, we can define these using smallstep reduction rules and show that L_* is given as a lfp and L_ω as a gfp.

Context-sensitive behaviours

$$e := \mathrm{fix}(\lambda f^{o \to o}.\lambda x^{o}.a; f(b; x; c) + x)$$

$$L_\omega(e\ d) \simeq a^\omega \quad ; \quad L_*(e) \simeq \{a^n b^n d c^n | n \geq 0\}$$

$\simeq$ means without ✓

Denotational semantics

- With each type τ associate complete lattices $[\![\tau]\!]_*$ and $[\![\tau]\!]_\omega$:
  - $[\![o]\!]_* = \mathcal{P}(\Sigma^*)$ and $[\![o]\!]_\omega = \mathcal{P}(\Sigma^\omega)$.
  - $[\![\tau \to \tau']\!]_* = [\![\tau]\!]_* \Rightarrow [\![\tau']\!]_*$ and $[\![\tau \to \tau']\!]_\omega = [\![\tau]\!]_* \times [\![\tau]\!]_\omega \Rightarrow [\![\tau']\!]_\omega$.
  - $[\![\Gamma]\!]_{*/\omega} = \prod_{x \in \mathrm{dom}(\Gamma)} [\![\Gamma(x)]\!]_{*/\omega}$ and
- With each term $\Gamma \vdash e : \tau$ we associate functions $[\![e]\!]_* : [\![\Gamma]\!]_* \Rightarrow [\![\tau]\!]_*$ and $[\![e]\!]_\omega : [\![\Gamma]\!]_* \times [\![\Gamma]\!]_\omega \Rightarrow [\![\tau]\!]_\omega$.

Herein, $\Rightarrow$ stands for monotone function space.

Denotational semantics, cont'd

- $[\![\mathsf{fix}]\!]_* = \lambda f.\mathsf{lfp}(f)$
- $[\![\mathsf{fix}]\!]_\omega = \lambda(f, F).\mathsf{gfp}(\lambda X.F(\mathsf{lfp}(f), X))$
- $\ldots$

Theorem

If e is a closed term of type o then $[\![e]\!]_* = L_*(e)$ and $[\![e]\!]_\omega = L_\omega(e)$.

Towards a finite abstraction

- As before, we have the finite lattices $\mathcal{M}_*$ and $\mathcal{M}_\omega$ with their "ok" subsets $\mathbf{ok}_{*/\omega} \subseteq \mathcal{M}_{*/\omega}$ and we would like to know whether $\alpha_{*/\omega}([\![e]\!]_{*/\omega}) \subseteq \mathbf{ok}_{*/\omega}$.
- To this end, we need to extend $\mathcal{M}_{*/\omega}$ and $\alpha_{*/\omega}$ to higher order.
- If we only had the finitary component this would not be too hard, since Galois connections lift to (monotone) function spaces and least fixpoints: $\mathcal{M}_*^o = \mathcal{M}_*$ and $\mathcal{M}_*^{\tau \to \tau'} = \mathcal{M}_*^\tau \Rightarrow \mathcal{M}_*^{\tau'} \ldots$
- But α does not commute with gfp :-(

Important observation at type $o \to o$

Let us abbreviate $[\![o]\!]_* = \mathcal{P}(\Sigma^*)$ by o and $[\![o]\!]_\omega = \mathcal{P}(\Sigma^\omega)$ by O.

- We have $[\![o \to o]\!]_* = o \Rightarrow o$ and
- and
  $$[\![o \to o]\!]_\omega = o \times O \Rightarrow O$$
  but all the functions F in this lattice that **actually occur** as denotations are **linear** in the sense that there exist $A \in O$ and $b : o \Rightarrow O$ such that
  $$F(x, X) = A \cup b(x)X$$

- More generally, if u, v, w are lattices built up from o by $\times$ and $\Rightarrow$ then a function $\Phi : u \times (v \Rightarrow O) \Rightarrow O$ arising as the $[\![-]\!]_\omega$-denotation of a term will have the form

$$\Phi(x, X) = \lambda a. U(x, a) \cup \bigcup_{b \in v} V(x, a, b).X(b)$$

Given $x \in u$, $X \in v \Rightarrow O$, and $a \in w$ in order to compute "an O", we can either not use X at all or apply it to v-arguments and prefix the results. NB: $V(x, a, b)$ may be $\emptyset$ ("switching off a b").

A cartesian-closed category of linear maps

- Objects are pairs of sets $A = (\mathrm{fin}(A), \mathrm{arg}(A))$,
- Morphisms $A \to B$ are pairs of functions $f = (\mathrm{fin}(f), \mathrm{arg}(f))$ where $\mathrm{fin}(f) : \mathrm{fin}(A) \to \mathrm{fin}(B)$ and

  $$\mathrm{arg}(f) : \mathrm{fin}(A) \to O^{\mathrm{arg}(B)} \times o^{\mathrm{arg}(B) \times \mathrm{arg}(A)}$$

  Here B^A is short for $A \Rightarrow B$.
- $\mathrm{arg}(f)$ induces $F : \mathrm{fin}(A) \times O^{\mathrm{arg}(A)} \to O^{\mathrm{arg}(B)}$ by

  $$F(x, X) = \lambda b.\mathrm{arg}(f)(x).1(b) \cup \bigcup_{x \in \mathrm{arg}(A)} \mathrm{arg}(f)(x).2(b, a)X(a)$$

  which is linear in X.
- Notice also that $\mathrm{arg}(f)$ is uniquely determined by an F admitting such a presentation.

Product and function space

- $\mathrm{fin}(A \times B) = \mathrm{fin}(A) \times \mathrm{fin}(B)$ and $\mathrm{arg}(A \times B) = \mathrm{arg}(A) + \mathrm{arg}(B)$.
- For function space notice that a morphism $f : A \times B \to C$ has components
  - $\mathrm{fin}(f) : \mathrm{fin}(A) \times \mathrm{fin}(B) \to \mathrm{fin}(C)$, suggesting $\mathrm{fin}(B \Rightarrow C) = \mathrm{fin}(C)^{\mathrm{fin}(B)})$ and (up to iso)
  - $\mathrm{arg}(f) : \mathrm{fin}(A) \times \mathrm{fin}(B) \times \mathrm{arg}(C) \to O \times o^{\mathrm{arg}(A)} \times o^{\mathrm{arg}(B)}$
- This suggests $\mathrm{arg}(B \Rightarrow C) = \mathrm{fin}(B) \times \mathrm{arg}(C)$ and in fact
- $\mathrm{fin}(B \Rightarrow C) = (\mathrm{fin}(C) \times o^{\mathrm{arg}(B) \times \mathrm{arg}(C)})^{\mathrm{fin}(B)}$
- It is then possible to define application and λ-abstraction and in fact to show cartesian closure!

Recall that cartesian closure means a 1-1 correspondence between $A \times B \to C$ and $A \to B \Rightarrow C$.

Fixpoints

- Consider a fixpoint operator $\mathrm{fix} : A \Rightarrow A \to A$. (For it to exist we must move from sets to cpos or similar and monotone functions.
- Must add "op" in various places:

$$\mathrm{arg}(f) : \mathrm{fin}(A) \to O^{\mathrm{arg}(B)} \times o^{\mathrm{arg}(B)\times \mathrm{arg}(A)^{\mathrm{op}}}$$

$$\mathrm{fin}(A \Rightarrow B) = (\mathrm{fin}(B) \times o^{\mathrm{arg}(B)\times \mathrm{arg}(A)^{\mathrm{op}}})^{\mathrm{fin}(A)}$$

- The fixpoint equation $\mathrm{fix}(f) = f(\mathrm{fix}(f))$ becomes: for all $f \in \mathrm{fin}(A \Rightarrow A)$ and $q \in \mathrm{arg}(A)$:
  - $x_0 := \mathrm{fin}(\mathrm{fix})(f) = f(\mathrm{fin}(\mathrm{fix})(f)).1$,
  - $\mathrm{arg}(\mathrm{fix})(f) = (U, m)$ where
    - $U \in O^{\mathrm{arg}(A)}$ and $m \in o^{\mathrm{arg}(A)\times \mathrm{fin}(A)^{\mathrm{op}} \times \mathrm{arg}(A)^{\mathrm{op}}}$
    - $U(q) = \bigcup_{q'} f(x_0).2(q, q')U(q')$
    - $m(q, x', q'') = [x' \leq x \wedge q'' \leq q] \cup \bigcup_{q'} f(x_0).2(q, q')m(q', x', q'')$

    Where $[\phi] = \text{if } \phi \text{ then } \{\epsilon\} \text{ else} \emptyset$.
  - This dictates the definition of fix: Use lfp for fin-part and m, use ω-product for U. Both trackable by abstraction!

The fixpoint combinator

For any object $A = \mathrm{fin}(A), \mathrm{arg}(A)$ we have a fixpoint combinator $\mathrm{fix} : A{\Rightarrow}A \to A$ with

- $\mathrm{fin}(\mathrm{fix})(f \in \mathrm{fin}(A \Rightarrow A)) = \mathrm{lfp}(\lambda a.f(a).1) =: x_0$
- $\mathrm{arg}(\mathrm{fix})(f \in \mathrm{fin}(A \Rightarrow A)) = (U, m)$ where
  - $t := f(x_0).2 \in o^{\mathrm{arg}(A)\times \mathrm{arg}(A)^{\mathrm{op}}}$
  - $U \in O^{\mathrm{arg}(A)}$
  - $m \in o^{\mathrm{arg}(A)\times \mathrm{fin}(A)^{\mathrm{op}} \times \mathrm{arg}(A)^{\mathrm{op}}}$
  - $$U(q_0) = \bigcup_{q_1,q_2,q_3\dots} t(q_0,q_1)t(q_1,q_2)t(q_2,q_3)\dots$$
  - $$m(q_0,x',q') = \bigcup_{q_1,q_2,q_3\dots,q_n=q'} t(q_0,q_1)t(q_1,q_2)t(q_2,q_3)\dots t(q_{n-1},q_n)$$

    if $x' \leq x_0$ and $\emptyset$ otherwise.

Results

- Theorem: lfp/gfp semantics matches operational semantics.
- Theorem: the cartesian-closed category of linear maps based on O, o is sound and complete w.r.t. lfp/gfp semantics.
- Theorem: the ccc of linear maps based on the finite lattices $\mathcal{M}_*$ and $\mathcal{M}_\omega$ is sound w.r.t. linear maps and complete w.r.t. abstracted properties, in particular $L(\mathfrak{A})$.
- Corollary: Given a closed term $e : o$ we have that $L_*(e), L_\omega(e) \in L(\mathfrak{A})$ iff the interpretation of e in the finite ccc (computable!) is $\leq \alpha(L(\mathfrak{A}))$.
- $\rightsquigarrow$ **Type system** whose types are morphisms in the finite ccc (sound and complete) or portions thereof (sound).

Conclusion

- Extended automata-based type system for trace policies to infinite traces.
- Defined an abstract domain from a given Büchi automaton which may be of independent interest
- One of the first type systems for infinitary properties.
- Further extended to higher-order functions with general recursion
- Solves an question left open in Walukiewicz and Salvati's recent paper.