General Binding Structures in Nominal Isabelle 2

Christian Urban

joint work with **Cezary Kaliszyk**

Binding in Old Nominal

- the old Nominal Isabelle provided a reasoning infrastructure for single binders

Lam [a].(Var a)

for example

a # Lam [a]. t
Lam [a]. (Var a) = Lam [b]. (Var b)
Barendregt-style reasoning about bound variables
(variable convention can lead to faulty reasoning)

Bob Harper
(CMU)

Frank Pfenning
(CMU)

published a proof in **ACM Transactions on Computational Logic**, 2005, ∼31pp

Bob Harper
(CMU)

Frank Pfenning
(CMU)

published a proof in **ACM Transactions on Computational Logic**, 2005, ∼31pp

Andrew Appel
(Princeton)

relied on their proof in a **security** critical application


Bob Harper
(CMU)


Frank Pfenning
(CMU)

published a proof in **ACM Transactions on Computational Logic**, 2005, ~31pp


Andrew Appel
(Princeton)

relied on their proof in a **security** critical application

(I also found an **error** in my Ph.D.-thesis about cut-elimination examined by Henk Barendregt and Andy Pitts.)

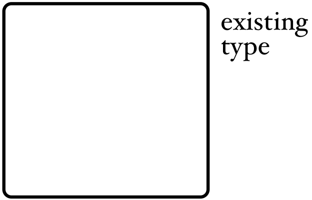
Binding in Old Nominal

- but representing

$$\forall\{a_1, \ldots, a_n\}.\ T$$

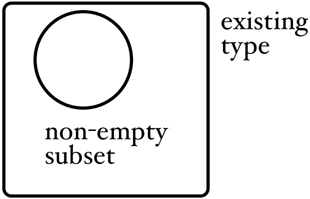
with single binders and reasoning about it was a **major** pain; take my word for it!

New Types in HOL



existing
type

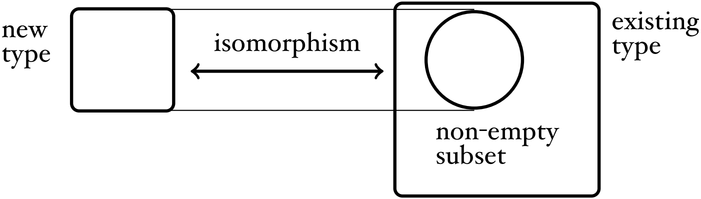
New Types in HOL



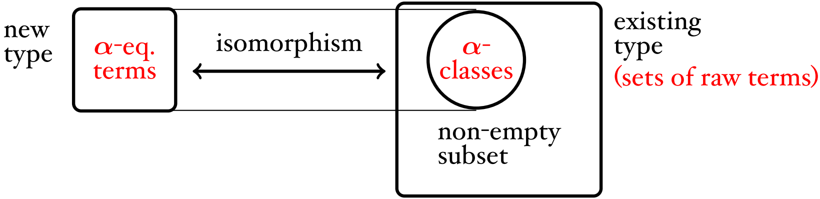
existing type

non-empty subset

New Types in HOL



New Types in HOL



new type

α-eq. terms

isomorphism

α-classes

non-empty subset

existing type

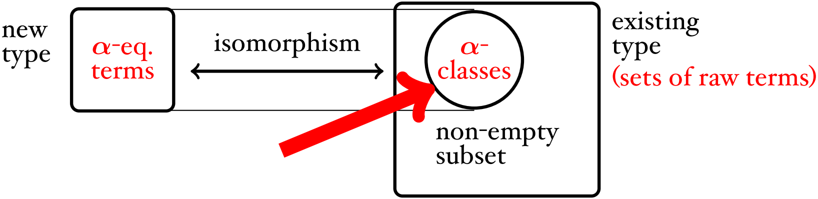
(sets of raw terms)

New Types in HOL

new
type

α-eq.
terms

New Types in HOL


new type

α-eq. terms

isomorphism

α-classes

non-empty subset

existing type

(sets of raw terms)

define α-equivalence

Binding Sets of Names

- binding sets of names has some interesting properties:

$$\forall\{x, y\}.\, x \rightarrow y \quad \approx_\alpha \quad \forall\{y, x\}.\, y \rightarrow x$$

* x, y, z are assumed to be distinct

Binding Sets of Names

- binding sets of names has some interesting properties:

$$\forall\{x, y\}.\, x \rightarrow y \quad \approx_\alpha \quad \forall\{y, x\}.\, y \rightarrow x$$

$$\forall\{x, y\}.\, x \rightarrow y \quad \not\approx_\alpha \quad \forall\{z\}.\, z \rightarrow z$$

* x, y, z are assumed to be distinct

Binding Sets of Names

- binding sets of names has some interesting properties:

$$\forall\{x,y\}.\, x \rightarrow y \quad \approx_\alpha \quad \forall\{y,x\}.\, y \rightarrow x$$

$$\forall\{x,y\}.\, x \rightarrow y \quad \not\approx_\alpha \quad \forall\{z\}.\, z \rightarrow z$$

$$\forall\{x\}.\, x \rightarrow y \quad \approx_\alpha \quad \forall\{x,z\}.\, x \rightarrow y$$

provided z is fresh for the type

* x, y, z are assumed to be distinct

Binding Sets of Names

- bind
  pro

$\forall$

For type-schemes the order of bound names does not matter, and α-equivalence is preserved under vacuous binders.

$$\forall\{x, y\}.\, x \rightarrow y \quad \not\approx_\alpha \quad \forall\{z\}.\, z \rightarrow z$$

$$\forall\{x\}.\, x \rightarrow y \quad \approx_\alpha \quad \forall\{x, z\}.\, x \rightarrow y$$

provided z is fresh for the type

* x, y, z are assumed to be distinct

Other Binding Modes

- alpha-equivalence being preserved under vacuous binders is <u>not</u> always wanted:

let $x = 3$ and $y = 2$ in $x - y$ end

Other Binding Modes

- alpha-equivalence being preserved under vacuous binders is not always wanted:

$$\mathrm{let}\ x = 3\ \mathrm{and}\ y = 2\ \mathrm{in}\ x - y\ \mathrm{end}$$

$$\approx_\alpha \mathrm{let}\ y = 2\ \mathrm{and}\ x = 3\ \mathrm{in}\ x - y\ \mathrm{end}$$

Other Binding Modes

- alpha-equivalence being preserved under vacuous binders is <u>not</u> always wanted:

$$\mathrm{let}\ x = 3\ \mathrm{and}\ y = 2\ \mathrm{in}\ x - y\ \mathrm{end}$$

$$\not\approx_\alpha \mathrm{let}\ y = 2\ \mathrm{and}\ x = 3\ \mathrm{and}\ z = \mathrm{loop}\ \mathrm{in}\ x - y\ \mathrm{end}$$

Even Another Binding Mode

- sometimes one wants to abstract more than one name, but the order does matter

$$\begin{aligned}&\mathrm{let}\ (x, y) = (3, 2)\ \mathrm{in}\ x - y\ \mathrm{end}\\ &\quad\not\approx_\alpha \mathrm{let}\ (y, x) = (3, 2)\ \mathrm{in}\ x - y\ \mathrm{end}\end{aligned}$$

Three Binding Modes

- the order does not matter and alpha-equivelence is preserved under vacuous binders (restriction)
- the order does not matter, but the cardinality of the binders must be the same (abstraction)
- the order does matter (iterated single binders)

Three Binding Modes

- the order does not matter and alpha-equivelence is preserved under vacuous binders (restriction)
- the order does not matter, but the cardinality of the binders must be the same (abstraction)
- the order does matter (iterated single binders)

bind (set+)   **bind (set)**   **bind**

Specification of Binding

```
nominal_datatype trm =
    Var name
  | App trm trm
  | Lam name trm
  | Let assns trm
and assns =
    ANil
  | ACons name trm assns
```

Specification of Binding

```
nominal_datatype trm =
    Var name
  | App trm trm
  | Lam x::name t::trm     bind x in t
  | Let as::assns t::trm   bind bn(as) in t
and assns =
    ANil
  | ACons name trm assns
```

Specification of Binding

```
nominal_datatype trm =
    Var name
  | App trm trm
  | Lam x::name t::trm     bind x in t
  | Let as::assns t::trm   bind bn(as) in t
and assns =
    ANil
  | ACons name trm assns
binder bn where
    bn(ANil) = []
  | bn(ACons a t as) = [a] @ bn(as)
```

Alpha-Equivalence

- lets first look at pairs

$$(as, x)$$

as is a set of names…the binders
x is the body (might be a tuple)
$\approx_{\text{set}}$ is where the cardinality of the binders has to be the same

Alpha-Equivalence

- lets first look at pairs

$$(as, x) \approx_{\text{set}} (bs, y)$$

Alpha-Equivalence

- lets first look at pairs

$$(as, x) \approx_{\text{set}} (bs, y)$$

$$\overset{\text{def}}{=} \qquad \text{fv}(x) - as = \text{fv}(y) - bs$$

Alpha-Equivalence

- lets first look at pairs

$$(as, x) \approx_{\text{set}} (bs, y)$$

$$\begin{aligned} \overset{\text{def}}{=} \quad & \exists \pi.\ \text{fv}(x) - as = \text{fv}(y) - bs \\ & \wedge\ \text{fv}(x) - as \ \#^{*}\ \pi \\ & \wedge\ (\pi \bullet x) = y \end{aligned}$$

Alpha-Equivalence

- lets first look at pairs

$$(as, x) \approx_{\text{set}} (bs, y)$$

$$\begin{aligned} \stackrel{\text{def}}{=} \quad \exists \pi.\ & \text{fv}(x) - as = \text{fv}(y) - bs \\ \wedge\ & \text{fv}(x) - as \ \#^{*}\ \pi \\ \wedge\ & (\pi \bullet x) = y \\ \wedge\ & \pi \bullet as = bs \end{aligned}$$

Alpha-Equivalence

- lets first look at pairs

$$(as, x) \approx_{\text{list}} (bs, y)$$

$$\begin{aligned} \stackrel{\text{def}}{=}\quad & \exists \pi.\ \text{fv}(x) - as = \text{fv}(y) - bs \\ & \wedge\ \text{fv}(x) - as \#^{*}\ \pi \\ & \wedge\ (\pi \bullet x) = y \\ & \wedge\ \pi \bullet as = bs \end{aligned}$$

* as and bs are lists of names

Alpha-Equivalence

- lets first look at pairs

$$(as, x) \approx_{\text{set+}} (bs, y)$$

$$\begin{aligned} &\overset{\text{def}}{=} \quad \exists \pi.\ \text{fv}(x) - as = \text{fv}(y) - bs \\ &\qquad \wedge\ \text{fv}(x) - as \ \#^{*}\ \pi \\ &\qquad \wedge\ (\pi \bullet x) = y \\ &\qquad \cancel{\wedge\ \pi \bullet as = bs} \end{aligned}$$

Examples

- lets look at type-schemes:

$$(as, x) \approx_{\text{set}} (bs, y)$$

Examples

- lets look at type-schemes:

$$(\boldsymbol{as}, \boldsymbol{x}) \approx_{\text{set}} (\boldsymbol{bs}, \boldsymbol{y})$$

$$\begin{aligned}\text{fv}(\boldsymbol{x}) &= \{\boldsymbol{x}\}\\ \text{fv}(\boldsymbol{T}_1 \rightarrow \boldsymbol{T}_2) &= \text{fv}(\boldsymbol{T}_1) \cup \text{fv}(\boldsymbol{T}_2)\end{aligned}$$

Examples

- lets look at type-schemes:

$$(as, x) \approx_{\text{set}} (bs, y)$$

$$\begin{aligned} \mathrm{fv}(x) &= \{x\} \\ \mathrm{fv}(T_1 \to T_2) &= \mathrm{fv}(T_1) \cup \mathrm{fv}(T_2) \end{aligned}$$

set+:

$$\begin{aligned} &\exists \pi.\ \mathrm{fv}(x) - as = \mathrm{fv}(y) - bs \\ &\quad \wedge\ \mathrm{fv}(x) - as \ \#^*\ \pi \\ &\quad \wedge\ \pi \cdot x = y \end{aligned}$$

set:

$$\begin{aligned} &\exists \pi.\ \mathrm{fv}(x) - as = \mathrm{fv}(y) - bs \\ &\quad \wedge\ \mathrm{fv}(x) - as \ \#^*\ \pi \\ &\quad \wedge\ \pi \cdot x = y \\ &\quad \wedge\ \pi \cdot as = bs \end{aligned}$$

list:

$$\begin{aligned} &\exists \pi.\ \mathrm{fv}(x) - as = \mathrm{fv}(y) - bs \\ &\quad \wedge\ \mathrm{fv}(x) - as \ \#^*\ \pi \\ &\quad \wedge\ \pi \cdot x = y \\ &\quad \wedge\ \pi \cdot as = bs \end{aligned}$$

Examples

$$(\{x, y\}, x \rightarrow y) \approx_{?} (\{x, y\}, y \rightarrow x)$$

- $\approx_{\mathrm{set+}}$, $\approx_{\mathrm{set}}$

set+:

$$\begin{aligned} &\exists \pi.\ \mathrm{fv}(x) - as = \mathrm{fv}(y) - bs \\ &\quad \wedge\ \mathrm{fv}(x) - as \#^{*} \pi \\ &\quad \wedge\ \pi \cdot x = y \end{aligned}$$

set:

$$\begin{aligned} &\exists \pi.\ \mathrm{fv}(x) - as = \mathrm{fv}(y) - bs \\ &\quad \wedge\ \mathrm{fv}(x) - as \#^{*} \pi \\ &\quad \wedge\ \pi \cdot x = y \\ &\quad \wedge\ \pi \cdot as = bs \end{aligned}$$

list:

$$\begin{aligned} &\exists \pi.\ \mathrm{fv}(x) - as = \mathrm{fv}(y) - bs \\ &\quad \wedge\ \mathrm{fv}(x) - as \#^{*} \pi \\ &\quad \wedge\ \pi \cdot x = y \\ &\quad \wedge\ \pi \cdot as = bs \end{aligned}$$

Examples

$$([x, y], x \rightarrow y) \approx_{?} ([x, y], y \rightarrow x)$$

- $\approx_{\text{set+}}$, $\approx_{\text{set}}$, $\not\approx_{\text{list}}$

set+:

$$\begin{aligned} \exists \pi.\ & \text{fv}(x) - as = \text{fv}(y) - bs \\ \wedge\ & \text{fv}(x) - as \ \#^{*}\ \pi \\ \wedge\ & \pi \cdot x = y \end{aligned}$$

set:

$$\begin{aligned} \exists \pi.\ & \text{fv}(x) - as = \text{fv}(y) - bs \\ \wedge\ & \text{fv}(x) - as \ \#^{*}\ \pi \\ \wedge\ & \pi \cdot x = y \\ \wedge\ & \pi \cdot as = bs \end{aligned}$$

list:

$$\begin{aligned} \exists \pi.\ & \text{fv}(x) - as = \text{fv}(y) - bs \\ \wedge\ & \text{fv}(x) - as \ \#^{*}\ \pi \\ \wedge\ & \pi \cdot x = y \\ \wedge\ & \pi \cdot as = bs \end{aligned}$$

Examples

$$(\{x\}, x) \approx_{?} (\{x, y\}, x)$$

- $\approx_{\text{set+}}$, $\not\approx_{\text{set}}$, $\not\approx_{\text{list}}$

set+:

$$\begin{aligned}&\exists \pi.\ \mathrm{fv}(x) - as = \mathrm{fv}(y) - bs\\ &\quad\wedge\ \mathrm{fv}(x) - as \ \#^{*}\ \pi\\ &\quad\wedge\ \pi \cdot x = y\end{aligned}$$

set:

$$\begin{aligned}&\exists \pi.\ \mathrm{fv}(x) - as = \mathrm{fv}(y) - bs\\ &\quad\wedge\ \mathrm{fv}(x) - as \ \#^{*}\ \pi\\ &\quad\wedge\ \pi \cdot x = y\\ &\quad\wedge\ \pi \cdot as = bs\end{aligned}$$

list:

$$\begin{aligned}&\exists \pi.\ \mathrm{fv}(x) - as = \mathrm{fv}(y) - bs\\ &\quad\wedge\ \mathrm{fv}(x) - as \ \#^{*}\ \pi\\ &\quad\wedge\ \pi \cdot x = y\\ &\quad\wedge\ \pi \cdot as = bs\end{aligned}$$

Examples

- $\approx_{\text{set}}$

- α-equivalences coincide when a single name is abstracted
- in that case they are equivalent to "old-fashioned" definitions of α

set+:

$$\exists \pi.\ \text{fv}(x) - as = \text{fv}(y) - bs \\ \wedge\ \text{fv}(x) - as \mathrel{\#^*} \pi \\ \wedge\ \pi \cdot x = y$$

set:

$$\exists \pi.\ \text{fv}(x) - as = \text{fv}(y) - bs \\ \wedge\ \text{fv}(x) - as \mathrel{\#^*} \pi \\ \wedge\ \pi \cdot x = y \\ \wedge\ \pi \cdot as = bs$$

list:

$$\exists \pi.\ \text{fv}(x) - as = \text{fv}(y) - bs \\ \wedge\ \text{fv}(x) - as \mathrel{\#^*} \pi \\ \wedge\ \pi \cdot x = y \\ \wedge\ \pi \cdot as = bs$$

Our Specifications

```
nominal_datatype trm =
    Var name
  | App trm trm
  | Lam x::name t::trm    bind x in t
  | Let as::assns t::trm  bind bn(as) in t
and assns =
    ANil
  | ACons name trm assns
binder bn where
    bn(ANil) = []
  | bn(ACons a t as) = [a] @ bn(as)
```

Binder Clauses

- We need to have a ‘clear scope’ for a bound variable, and bound variables should not be free and bound at the same time.

shallow binders

Lam x::name t::trm **bind** x **in** t
All xs::name set T::ty **bind** xs **in** T
Foo x::name t_1::trm t_2::trm **bind** x **in** t_1, **bind** x **in** t_2
Bar x::name t_1::trm t_2::trm **bind** x **in** t_1 t_2

Binder Clauses

- We need to have a ‘clear scope’ for a bound variable, and bound variables should not be free and bound at the same time.

deep binders

Let as::assns t::trm **bind** bn(as) **in** t

Foo as::assns t_1::trm t_2::trm

bind bn(as) **in** t_1, **bind** bn(as) **in** t_2

×Bar as::assns t_1::trm t_2::trm

bind bn_1(as) **in** t_1, **bind** bn_2(as) **in** t_2

Binder Clauses

- We need to have a ‘clear scope’ for a bound variable, and bound variables should not be free and bound at the same time.

deep recursive binders

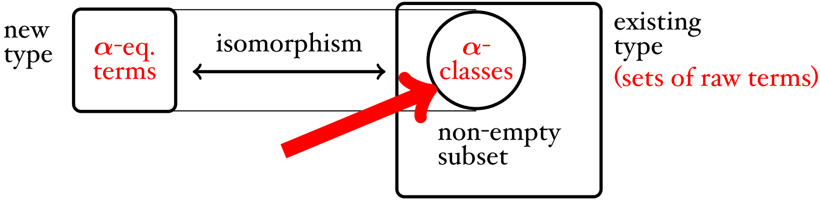
Let_rec as::assns t::trm **bind** bn(as) **in** t as

×Foo_rec as::assns t_1::trm t_2::trm

bind bn(as) **in** t_1 as, **bind** bn(as) **in** t_2

Our Work

- defined fv and α



new type

α-eq. terms

isomorphism

α-classes

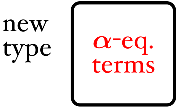
non-empty subset

existing type

(sets of raw terms)

Our Work

- defined fv and α
- built quotient / new type



new type

α-eq. terms

Our Work

new type

α-eq. terms

- defined fv and α
- built quotient / new type
- derived a reasoning infrastructure (#, distinctness, injectivity, cases,...)

Our Work

new type

α-eq. terms

- defined fv and α
- built quotient / new type
- derived a reasoning infrastructure (#, distinctness, injectivity, cases,...)
- derive a **stronger** cases lemma

Our Work

new
type

α-eq.
terms

- defined fv and α
- built quotient / new type
- derived a reasoning infrastructure (#, distinctness, injectivity, cases,...)
- derive a **stronger** cases lemma
- from this, a **stronger** induction principle (Barendregt variable convention built in)

Foo $(\boldsymbol{\lambda x.\lambda y.t})$ $(\boldsymbol{\lambda u.\lambda v.s})$

Part I: Conclusion

- the user does not see anything of the raw level

$$\text{Lam a (Var a)} = \text{Lam b (Var b)}$$

Part I: Conclusion

- the user does not see anything of the raw level
- http://isabelle.in.tum.de/nominal/

Part II: $\alpha\beta$-Equal Terms

- we have implemented a quotient package for Isabelle;
- can now introduce the type of $\alpha\beta$-equal terms (starting from α-equal terms).
- on paper this looks easy

Part II: $\alpha\beta$-Equal Terms

- we have implemented a quotient package for Isabelle;
- can now introduce the type of $\alpha\beta$-equal terms (starting from α-equal terms).
- on paper this looks easy

$$\begin{array}{rcl} x \approx_{\alpha\beta} y & \not\Rightarrow & \mathrm{supp}(x) = \mathrm{supp}(y) \\ & \not\Rightarrow & \mathrm{size}(x) = \mathrm{size}(y) \end{array}$$

Part II: $\alpha\beta$-Equal Terms

- we have implemented a quotient package for Isabelle;
- can now introduce the type of $\alpha\beta$-equal terms (starting from α-equal terms).
- on paper this looks easy

$$\begin{aligned} x \approx_{\alpha\beta} y \quad & \not\Rightarrow \quad \mathrm{supp}(x) = \mathrm{supp}(y) \\ & \not\Rightarrow \quad \mathrm{size}(x) = \mathrm{size}(y) \end{aligned}$$

Andy: $\mathrm{supp}[\![x]\!]_{\approx_{\alpha\beta}} = \bigcap\{\mathrm{supp}(y) \mid y \approx_{\alpha\beta} x\}$

$$\begin{aligned} x\,[y := s] &\overset{\text{def}}{=} \text{if } x = y \text{ then } s \text{ else } x \\ t_1 t_2\,[y := s] &\overset{\text{def}}{=} t_1[y := s]\; t_2[y := s] \\ \lambda x.t\,[y := s] &\overset{\text{def}}{=} \lambda x.\; t[y := s] \\ &\qquad\qquad \text{provided } x \# (y, s) \end{aligned}$$

Part III: Regular Languages

in Theorem Provers

e.g. Isabelle, Coq, HOL4, ...

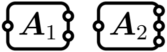
- automata ⇒ graphs, matrices, functions

Part III: Regular Languages

in Theorem Provers

e.g. Isabelle, Coq, HOL4, ...

- automata ⇒ graphs, matrices, functions
- combining automata/graphs



Part III: Regular Languages

in Theorem Provers

e.g. Isabelle, Coq, HOL4, ...

- automata ⇒ graphs, matrices, functions
- combining automata/graphs

Part III: Regular Languages

in Theorem Provers

e.g. Isabelle, Coq, HOL4, ...

- automata ⇒ graphs, matrices, functions
- combining automata/graphs

disjoint union:

$$A_1 \uplus A_2 \stackrel{\text{def}}{=} \{(1, x) \,|\, x \in A_1\} \,\cup\, \{(2, y) \,|\, y \in A_2\}$$

Part III: Regular Languages

in Theorem Provers

e.g. Isabelle, Coq, HOL4, ...

- automata ⇒ graphs, matrices, functions

Problems with definition for regularity:

$$\text{is\_regular}(A) \overset{\text{def}}{=} \exists M.\ \text{is\_dfa}(M) \wedge \mathcal{L}(M) = A$$

$$A_1 \uplus A_2 \overset{\text{def}}{=} \{(1, x) \mid x \in A_1\} \cup \{(2, y) \mid y \in A_2\}$$

Part III: Regular Languages

in Theorem Provers

e.g. Isabelle, Coq, HOL4, ...

- automata ⇒ graphs, matrices, functions
- combining automata/graphs

A solution: use nats ⇒ state nodes

Part III: Regular Languages

in Theorem Provers

e.g. Isabelle, Coq, HOL4, ...

- automata ⇒ graphs, matrices, functions
- combining automata/graphs

A solution: use nats ⇒ state nodes

You have to rename states!

Formal language theory...

in Theorem Provers

e.g. Isabelle, Coq, HOL4, ...

- Kozen's "paper" proof of Myhill-Nerode:
  requires absence of inaccessible states

$$\text{is\_regular}(A) \overset{\text{def}}{=} \exists M.\ \text{is\_dfa}(M) \wedge \mathcal{L}(M) = A$$

Definition:

A language A is regular, provided there exists a regular expression that matches all strings of A.

Definition:

A language A is regular, provided there exists a regular expression that matches all strings of A.

...and forget about automata

Definition:

A language A is regular, provided there exists a regular expression that matches all strings of A.

...and forget about automata

Infrastructure for free. But do we lose anything?

Definition:

A language A is regular, provided there exists a regular expression that matches all strings of A.

...and forget about automata

Infrastructure for free. But do we lose anything?

- pumping lemma

Definition:

A language A is regular, provided there exists a regular expression that matches all strings of A.

...and forget about automata

Infrastructure for free. But do we lose anything?

- pumping lemma
- closure under complementation

Definition:

A language A is regular, provided there exists a regular expression that matches all strings of A.

...and forget about automata

Infrastructure for free. But do we lose anything?

- pumping lemma
- closure under complementation
- regular expression matching

Definition:

A language A is regular, provided there exists a regular expression that matches all strings of A.

...and forget about automata

Infrastructure for free. But do we lose anything?

- pumping lemma
- closure under complementation
- ~~regular expression matching~~ (⇒Brozowski'64, Owens et al '09)

Definition:

A language A is regular, provided there exists a regular expression that matches all strings of A.

...and forget about automata

Infrastructure for free. But do we lose anything?

- pumping lemma
- closure under complementation
- ~~regular expression matching~~ (⇒Brozowski'64, Owens et al '09)
- most textbooks are about automata

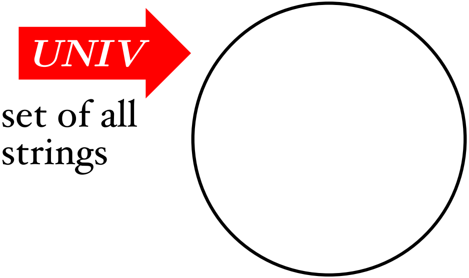
The Myhill-Nerode Theorem

- provides necessary and sufficient conditions for a language being regular (pumping lemma only necessary)

- key is the equivalence relation:

$$x \approx_A y \stackrel{\text{def}}{=} \forall z.\ x@z \in A \Leftrightarrow y@z \in A$$

The Myhill-Nerode Theorem

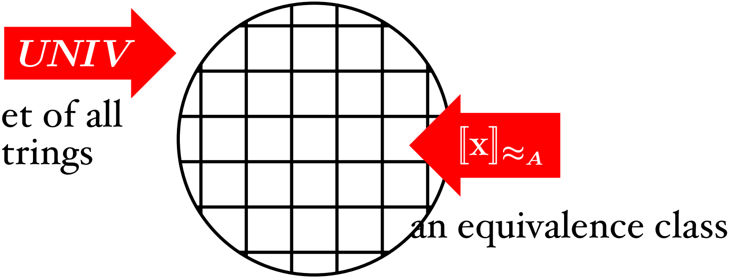


UNIV

set of all strings

- finite $(\boldsymbol{UNIV} /\!/ \approx_A)$ $\Leftrightarrow$ $\boldsymbol{A}$ is regular

The Myhill-Nerode Theorem



- finite $(UNIV \,/\!/ \approx_A) \;\Leftrightarrow\; A$ is regular

The Myhill-Nerode Theorem

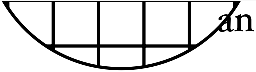
Two directions:

1.) finite ⇒ regular

$\text{finite}\,(UNIV\,//\approx_A) \Rightarrow \exists r.\ A = \mathcal{L}(r)$

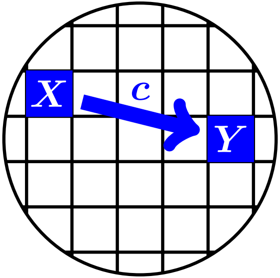
2.) regular ⇒ finite

$\text{finite}\,(UNIV\,//\approx_{\mathcal{L}(r)})$



- $\text{finite}\,(UNIV\,//\approx_A) \;\Leftrightarrow\; A$ is regular

Transitions between Eq-Clas



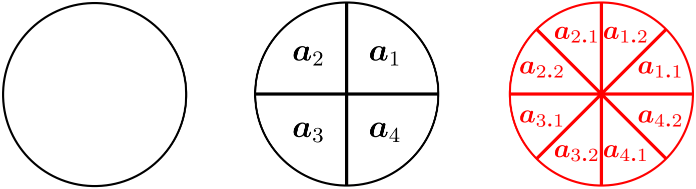
$$X \xrightarrow{c} Y \overset{\text{def}}{=} X\,;c \subseteq Y$$

The Other Direction

One has to prove

$$\mathrm{finite}(\boldsymbol{UNIV} \,/\!/ \approx_{\mathcal{L}(\boldsymbol{r})})$$

by induction on $\boldsymbol{r}$. Not trivial, but after a bit of thinking, one can find a refined relation:



$\boldsymbol{UNIV}$ $\qquad$ $\boldsymbol{UNIV} \,/\!/ \approx_{\mathcal{L}(\boldsymbol{r})}$ $\qquad$ $\boldsymbol{UNIV} \,/\!/ \boldsymbol{R}$

Derivatives of RExps

- introduced by Brozowski ’64
- a regular expressions after a character has been parsed

$$
\begin{array}{lcl}
\text{der c } \varnothing & \stackrel{\text{def}}{=} & \varnothing \\
\text{der c } [] & \stackrel{\text{def}}{=} & \varnothing \\
\text{der c d} & \stackrel{\text{def}}{=} & \text{if c} = \text{d then } [] \text{ else } \varnothing \\
\text{der c } (\boldsymbol{r}_1 + \boldsymbol{r}_2) & \stackrel{\text{def}}{=} & (\text{der c } \boldsymbol{r}_1) + (\text{der c } \boldsymbol{r}_2) \\
\text{der c } (\boldsymbol{r}^\star) & \stackrel{\text{def}}{=} & (\text{der c } \boldsymbol{r}) \cdot \boldsymbol{r}^\star \\
\text{der c } (\boldsymbol{r}_1 \cdot \boldsymbol{r}_2) & \stackrel{\text{def}}{=} & \text{if nullable } \boldsymbol{r}_1 \\
 & & \text{then } (\text{der c } \boldsymbol{r}_1) \cdot \boldsymbol{r}_2 + (\text{der c } \boldsymbol{r}_2) \\
 & & \text{else } (\text{der c } \boldsymbol{r}_1) \cdot \boldsymbol{r}_2
\end{array}
$$

Derivatives of RExps

- introduced by Brozowski '64
- a regular expressions after a character has been parsed

- *partial derivatives*
- *by Antimirov '95*

$$
\begin{array}{lcl}
\text{pder c } \varnothing & \stackrel{\text{def}}{=} & \{\} \\
\text{pder c } [] & \stackrel{\text{def}}{=} & \{\} \\
\text{pder c d} & \stackrel{\text{def}}{=} & \text{if c} = \text{d then } \{[]\} \text{ else } \{\} \\
\text{pder c } (r_1 + r_2) & \stackrel{\text{def}}{=} & (\text{pder c } r_1) \cup (\text{der c } r_2) \\
\text{pder c } (r^\star) & \stackrel{\text{def}}{=} & (\text{pder c } r) \cdot r^\star \\
\text{pder c } (r_1 \cdot r_2) & \stackrel{\text{def}}{=} & \text{if nullable } r_1 \\
 & & \text{then } (\text{pder c } r_1) \cdot r_2 \cup (\text{pder c } r_2) \\
 & & \text{else } (\text{pder c } r_1) \cdot r_2
\end{array}
$$

Partial Derivatives

- pders x r = pders y r refines $x \approx_{\mathcal{L}(r)} y$

Partial Derivatives

- $\underbrace{\text{pders } x\ r = \text{pders } y\ r}_{R}$ refines $x \approx_{\mathcal{L}(r)} y$

⬇ Antimirov ’95

- $\text{finite}(UNIV\,//\,R)$

Partial Derivatives

- $\underbrace{\text{pders}\ \boldsymbol{x}\ \boldsymbol{r} = \text{pders}\ \boldsymbol{y}\ \boldsymbol{r}}_{\boldsymbol{R}}$ refines $\boldsymbol{x} \approx_{\mathcal{L}(\boldsymbol{r})} \boldsymbol{y}$

⬇ Antimirov '95

- $\text{finite}(\boldsymbol{UNIV} /\!/ \boldsymbol{R})$
- Therefore $\text{finite}(\boldsymbol{UNIV} /\!/ \approx_{\mathcal{L}(\boldsymbol{r})})$. Qed.

What Have We Achieved?

- finite $(\mathit{UNIV} /\!/ \approx_A) \;\Leftrightarrow\; A$ is regular

What Have We Achieved?

- finite $(\mathit{UNIV}\,/\!/ \approx_A)$ $\;\Leftrightarrow\;$ A is regular
- regular languages are closed under complementation; this is now easy

$$\mathit{UNIV}\,/\!/ \approx_A \;=\; \mathit{UNIV}\,/\!/ \approx_{\overline{A}}$$

$$x \approx_A y \overset{\text{def}}{=} \forall z.\; x@z \in A \Leftrightarrow y@z \in A$$

What Have We Achieved?

- finite $(\mathit{UNIV} /\!/ \approx_A) \;\Leftrightarrow\; A$ is regular
- regular languages are closed under complementation; this is now easy
$$\mathit{UNIV} /\!/ \approx_A \;=\; \mathit{UNIV} /\!/ \approx_{\overline{A}}$$
- non-regularity $(a^n b^n)$

What Have We Achieved?

- finite $(\mathit{UNIV} /\!/ \approx_A) \;\Leftrightarrow\; A$ is regular
- regular languages are closed under complementation; this is now easy

$$\mathit{UNIV} /\!/ \approx_A \;=\; \mathit{UNIV} /\!/ \approx_{\overline{A}}$$

- non-regularity $(a^n b^n)$

If there exists a sufficiently large set B (for example infinitely large), such that

$$\forall x, y \in B.\; x \neq y \;\Rightarrow\; x \not\approx_A y.$$

then A is not regular. $(B \overset{\text{def}}{=} \bigcup_n a^n)$

What Have We Achieved?

- finite $(\mathit{UNIV} /\!/ \approx_A) \;\Leftrightarrow\; A$ is regular
- regular languages are closed under complementation; this is now easy

$$\mathit{UNIV} /\!/ \approx_A \;=\; \mathit{UNIV} /\!/ \approx_{\overline{A}}$$

- non-regularity $(a^n b^n)$
- take **any** language; build the language of substrings

What Have We Achieved?

- finite $(UNIV // \approx_A)$ $\Leftrightarrow$ A is regular
- regular languages are closed under complementation; this is now easy
  $$UNIV // \approx_A \;=\; UNIV // \approx_{\overline{A}}$$
- non-regularity $(a^n b^n)$
- take **any** language; build the language of substrings
  then this language **is** regular $(a^n b^n \Rightarrow a^\star b^\star)$

Thank you!

Questions?

Examples

$$(\{a,b\}, a \to b) \approx_\alpha (\{a,b\}, a \to b)$$
$$(\{a,b\}, a \to b) \approx_\alpha (\{a,b\}, b \to a)$$

$$\begin{aligned}&(\{a,b\}, (a \to b, a \to b))\\ &\quad \not\approx_\alpha (\{a,b\}, (a \to b, b \to a))\end{aligned}$$

Examples

$$(\{a,b\}, a \rightarrow b) \approx_\alpha (\{a,b\}, a \rightarrow b)$$
$$(\{a,b\}, a \rightarrow b) \approx_\alpha (\{a,b\}, b \rightarrow a)$$

$$(\{a,b\}, (a \rightarrow b, a \rightarrow b)) \not\approx_\alpha (\{a,b\}, (a \rightarrow b, b \rightarrow a))$$

1.) **bind (set)** as **in** τ_1, **bind (set)** as **in** τ_2

2.) **bind (set)** as **in** τ_1 τ_2