Logika dziedzinowa do wnioskowania o termach z wiązaniem zmiennych

Domain-specific logic for terms with variable binding

Dominik Gulczyński
Promotor: dr Piotr Polesiuk

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

Obrona pracy magisterskiej, 19 Grudnia 2023

Dwa style prowadzenia rozumowań

1. Piszemy dla czytelnika

1. Piszemy dla komputera

Dwa style prowadzenia rozumowań

1. Piszemy dla czytelnika
2. Pomijamy detale

1. Piszemy dla komputera
2. Musimy obsłużyć wszystkie detale

Dwa style prowadzenia rozumowań

1. Piszemy dla czytelnika
2. Pomijamy detale
3. Korzystamy z niejawnych założeń lub niedowiedzonych twierdzeń

1. Piszemy dla komputera
2. Musimy obsłużyć wszystkie detale
3. Korzystamy tylko z jawnych założeń lub i dowiedzonych twierdzeń

Dwa style prowadzenia rozumowań

1. Piszemy dla czytelnika
2. Pomijamy detale
3. Korzystamy z niejawnych założeń lub niedowiedzonych twierdzeń
4. Prowadzimy rozumowanie w konwencji nazw zmiennych Barendregta

1. Piszemy dla komputera
2. Musimy obsłużyć wszystkie detale
3. Korzystamy tylko z jawnych założeń lub i dowiedzonych twierdzeń
4. ?

1 Simply Typed Lambda Calculus

$$
\begin{array}{rrcl}
\text{Variables} & x, y & & \ldots \\
\text{Types} & A, B & ::= & a \mid A \to B \\
\text{Base Types} & a, b & ::= & \mathsf{int} \\
\text{Variable Contexts} & \Gamma & ::= & \emptyset \mid \Gamma, x : A \\
\text{Source Terms} & E & ::= & x \mid \lambda x : A.\ E \mid E_1\ E_2 \mid E_1 + E_2 \mid \overline{n} \\
\text{Runtime Terms} & e & ::= & x \mid \lambda x.\ e \mid e_1\ e_2 \mid e_1 + e_2 \mid \overline{n} \\
\text{(Runtime) Values} & v & ::= & \lambda x.\ e \mid \overline{n}
\end{array}
$$

Variables and Substitution We use Barendregt's variable convention, which means we assume that all bound variables are distinct and maintain this invariant implicitly. Another way of saying this is: we will not worry about the formal details of variable names, alpha renaming, freshness, etc., and instead just assume that all variables bound in a variable context are distinct and that we keep it that way when we add a new variable to the context. Of course, getting such details right is very important when we mechanize our reasoning, but in this part of the course, we will not be using Coq, so we can avoid worrying about it.

źródło: Semantics lecture notes 2019/2020, Derek Dreyer, Universität des Saarlandes

To avoid confusion between the new binding and any bindings that may already appear in Γ, we require that the name $\mathtt{x}$ be chosen so that it is distinct from the variables bound by Γ. Since our convention is that variables bound by λ-abstractions may be renamed whenever convenient, this condition can always be satisfied by renaming the bound variable if necessary.

LEMMA [PRESERVATION OF TYPES UNDER SUBSTITUTION]: If $\Gamma, \mathtt{x}{:}\mathtt{S} \vdash \mathtt{t} : \mathtt{T}$ and $\Gamma \vdash \mathtt{s} : \mathtt{S}$, then $\Gamma \vdash [\mathtt{x} \mapsto \mathtt{s}]\mathtt{t} : \mathtt{T}$. □

Proof: By induction on a derivation of the statement $\Gamma, \mathtt{x}{:}\mathtt{S} \vdash \mathtt{t} : \mathtt{T}$. For a given derivation, we proceed by cases on the final typing rule used in the proof.- The most interesting cases are the ones for variables and abstractions.

Case T-ABS:

$$\mathtt{t} = \lambda\mathtt{y}{:}\mathtt{T_2}.\mathtt{t_1}$$
$$\mathtt{T} = \mathtt{T_2} \to \mathtt{T_1}$$
$$\Gamma, \mathtt{x}{:}\mathtt{S}, \mathtt{y}{:}\mathtt{T_2} \vdash \mathtt{t_1} : \mathtt{T_1}$$

By convention 5.3.4, we may assume $\mathtt{x} \neq \mathtt{y}$ and $\mathtt{y} \notin FV(\mathtt{s})$. Using permutation on the given subderivation, we obtain $\Gamma, \mathtt{y}{:}\mathtt{T_2}, \mathtt{x}{:}\mathtt{S} \vdash \mathtt{t_1} : \mathtt{T_1}$. Using weakening on the other given derivation ($\Gamma \vdash \mathtt{s} : \mathtt{S}$), we obtain $\Gamma, \mathtt{y}{:}\mathtt{T_2} \vdash \mathtt{s} : \mathtt{S}$. Now, by the induction hypothesis, $\Gamma, \mathtt{y}{:}\mathtt{T_2} \vdash [\mathtt{x} \mapsto \mathtt{s}]\mathtt{t_1} : \mathtt{T_1}$. By T-ABS, $\Gamma \vdash \lambda\mathtt{y}{:}\mathtt{T_2}.\ [\mathtt{x} \mapsto \mathtt{s}]\mathtt{t_1} : \mathtt{T_2} \to \mathtt{T_1}$. But this is precisely the needed result, since, by the definition of substitution, $[\mathtt{x} \mapsto \mathtt{s}]\mathtt{t} = \lambda\mathtt{y}{:}\mathtt{T_1}.\ [\mathtt{x} \mapsto \mathtt{s}]\mathtt{t_1}$.

źródło: Types and Programming Languages, Benjamin C. Pierce, MIT Press.

5.3.4 **Convention**: Terms that differ only in the names of bound variables are interchangeable in all contexts. □

What this means in practice is that the name of any λ-bound variable can be changed to another name (consistently making the same change in the body of the λ), at any point where this is convenient. For example, if we want to calculate $[x \mapsto y\ z](\lambda y.\ x\ y)$, we first rewrite $(\lambda y.\ x\ y)$ as, say, $(\lambda w.\ x\ w)$. We then calculate $[x \mapsto y\ z](\lambda w.\ x\ w)$, giving $(\lambda w.\ y\ z\ w)$.

This convention renders the substitution operation "as good as total," since whenever we find ourselves about to apply it to arguments for which it is undefined, we can rename as necessary, so that the side conditions are satisfied. Indeed, having adopted this convention, we can formulate the definition of substitution a little more tersely. The first clause for abstractions can be dropped, since we can always assume (renaming if necessary) that the bound variable y is different from both x and the free variables of s. This yields the final form of the definition.

5.3.5 **Definition [Substitution]**:

$$
\begin{array}{lcll}
[x \mapsto s]x & = & s & \\
[x \mapsto s]y & = & y & \text{if } y \neq x \\
[x \mapsto s](\lambda y.t_1) & = & \lambda y.\ [x \mapsto s]t_1 & \text{if } y \neq x \text{ and } y \notin FV(s) \\
[x \mapsto s](t_1\ t_2) & = & [x \mapsto s]t_1\ [x \mapsto s]t_2 &
\end{array}
$$
□

źródło: Types and Programming Languages, Benjamin C. Pierce, MIT Press.

Rozwiązania problemu wiązania zmiennych w systemach formalnych

- Rachunek kombinatorów

Rozwiązania problemu wiązania zmiennych w systemach formalnych

- Rachunek kombinatorów
- Reprezentacja De Bruijna

Rozwiązania problemu wiązania zmiennych w systemach formalnych

- Rachunek kombinatorów
- Reprezentacja De Bruijna
- Higher-Order Abstract Syntax

Rozwiązania problemu wiązania zmiennych w systemach formalnych

- Rachunek kombinatorów
- Reprezentacja De Bruijna
- Higher-Order Abstract Syntax
- Techniki nominalne

Logika nominalna

- Rozszerzenie logiki pierwszego rzędu o narzędzia do formalizacji i rozumowania na temat struktur syntaktycznych z wiązaniem nazw
- Matematyzacja pojęcia “wystarczająco świeżych nazw” zmiennych
- Opiera się o zamianę nazw i relację świeżości nazwy w termie.

Andrew M. Pitts, *“Nominal logic, a first order theory of names and binding”*:

Names of what? Names of entities that may be subject to binding by some of the syntactical constructions under consideration. In Nominal Logic these sorts of names, the ones that may be bound and hence that may be subjected to swapping without changing the validity of predicates involving them, will be called atoms.

$$t \ ::= \ a \mid \lambda a.t \mid t\ t$$

$$\begin{array}{rcl} (a\ b)(\lambda c.t) & := & \lambda((a\ b)c).((a\ b)t) \\ (a\ b)(t_1\ t_2) & := & ((a\ b)t_1)\ ((a\ b)t_2) \end{array} \qquad (a\ b)c \ := \ \begin{cases} a & \text{if } c = b \\ b & \text{if } c = a \\ c & \text{wpp.} \end{cases}$$

$$\frac{a \neq b}{a \# b} \qquad \frac{a \# t_1 \quad a \# t_2}{a \# t_1\ t_2} \qquad \frac{}{a \# \lambda a.t} \qquad \frac{a \# t}{a \# \lambda b.t}$$

$$\frac{}{a =_\alpha a} \qquad \frac{t_1 =_\alpha t_1' \quad t_2 =_\alpha t_2'}{t_1\ t_2 =_\alpha t_1'\ t_2'} \qquad \frac{(a\ b)t =_\alpha (a'\ b)t' \quad b \# t \quad b \# t'}{\lambda a.t =_\alpha \lambda a'.t'}$$

Andrew M. Pitts, *"Nominal logic, a first order theory of names and binding"***:**

The fundamental assumption underlying Nominal Logic is that *the only predicates we ever deal with* (when describing properties of syntax) *are equivariant ones, in the sense that their validity is invariant under swapping* (i.e., transposing, or interchanging) *names*.

Pomiędzy logiką nominalną a konwencją Barendgreta

- Logika nominalna umożliwia eleganckie wyrażanie alfa-równoważności, świeżości i innych podstawowych właściwości syntaktycznych, dzięki czemu może być używana jako baza do prowadzenia rozumowań o językach programowania.

- Ale najlepiej byłoby nie przejmować się w ogóle takimi sprawami, tak jak w konwencji Barendgreta, powiedzieć jedynie że zajmujemy się tylko "wystarczająco świeżymi nazwami" i nie martwić się o technikalia.

Pomiędzy logiką nominalną a konwencją Barendgreta

- Logika nominalna umożliwia eleganckie wyrażanie alfa-równoważności, świeżości i innych podstawowych właściwości syntaktycznych, dzięki czemu może być używana jako baza do prowadzenia rozumowań o językach programowania.
- Stajemy po środku i przedstawiamy wariant logiki nominalnej która oparta jest o półautomatyczne narzędzie do zajmowania się wybranymi własności syntaktycznymi, które nazywamy więzami.
- Ale najlepiej byłoby nie przejmować się w ogóle takimi sprawami, tak jak w konwencji Barendgreta, powiedzieć jedynie że zajmujemy się tylko "wystarczająco świeżymi nazwami" i nie martwić się o technikalia.

Więzy

$\alpha \# t$	Atom α jest *świeży* w termie t, czyli nie występuje w t jako wolna nazwa.
$t_1 = t_2$	Termy t_1 i t_2 są alfa-równoważne.
$t_1 \sim t_2$	Termy t_1 i t_2 mają taki sam kształt, czyli po wymazaniu atomów byłyby sobie równe.
$t_1 \prec t_2$	Kształt termu t_1 jest strukturalnie mniejszy od kształtu termu t_2, czyli po wymazaniu atomów t_1 byłby równy jakiemuś podtermowi t_2.
symbol t	Term t jest jakimś symbolowem funkcyjnym.

Logika więzów

$$\begin{array}{rcll}
a & \in & \mathit{Atom} & \text{(atomy)} \\
X & \in & \mathit{Var} & \text{(zmienne)} \\
f & \in & \mathit{Symb} & \text{(symbole funkcyjne)} \\
\\
\alpha & ::= & \pi\ a & \text{(wyrażenia atomowe)} \\
\pi & ::= & \mathsf{id} \mid (\alpha\ \alpha)\pi & \text{(permutacje)} \\
t & ::= & \alpha \mid \pi\ X \mid \alpha.t \mid t\ t \mid f & \text{(termy)} \\
s & ::= & _ \mid X \mid _.s \mid s\ s \mid f & \text{(kształty)} \\
c & ::= & \alpha \# t \mid t = t \mid t \sim t \mid t \prec t \mid \mathsf{symbol}\ t & \text{(więzy)}
\end{array}$$

$$\begin{array}{rcl}
\pi\ (\pi'\ a) & := & (\pi +\!\!+\ \pi')\ a \\
\pi\ (\pi'\ X) & := & (\pi +\!\!+\ \pi')\ X \\
\pi\ (\alpha.t) & := & (\pi\ \alpha).(\pi\ t) \\
\pi\ (t_1\ t_2) & := & (\pi\ t_1)\ (\pi\ t_2) \\
\pi\ f & := & f
\end{array}
\qquad
\begin{array}{rcl}
|\pi\ a| & := & _ \\
|\pi\ X| & := & X \\
|\alpha.t| & := & _.|t| \\
|t_1\ t_2| & := & |t_1|\ |t_2| \\
|f| & := & f
\end{array}$$

Semantyczne termy

$$
\begin{array}{lcll}
A & \in & SemAtom & \text{(wolne atomy)} \\
n & \in & Nat & \text{(związane atomy)} \\
T & ::= & A \mid n \mid \$T \mid T@T \mid f & \text{(semantyczne termy)} \\
S & ::= & \_ \mid \$S \mid S@S \mid f & \text{(semantyczne kształty)} \\
\rho & \in & (Atom \to SemAtom) \times (Var \to SemTerm) & \text{(interpretacje)}
\end{array}
$$

$$
\begin{array}{rcl}
[\![\pi\ a]\!]_\rho & := & [\![\pi]\!]_\rho(\rho(a)) \\
[\![\pi\ X]\!]_\rho & := & [\![\pi]\!]_\rho(\rho(X)) \\
[\![\alpha.t]\!]_\rho & := & \$([\![t]\!]_\rho\uparrow)\{[\![\alpha]\!]_\rho \mapsto 0\} \\
[\![t_1\ t_2]\!]_\rho & := & [\![t_1]\!]_\rho @ [\![t_2]\!]_\rho \\
[\![f]\!]_\rho & := & f
\end{array}
$$

$$
\begin{array}{rcl}
|A| & := & \_ \\
|n| & := & \_ \\
|\$T| & := & \$|T| \\
|T_1@T_2| & := & |T_1|@|T_2| \\
|f| & := & f
\end{array}
$$

$$
\begin{array}{rcl}
[\![\mathsf{id}]\!]_\rho(A) & := & A \\
\\
[\![\pi +\!\!+ (\alpha_1\ \alpha_2)]\!]_\rho(A) & := & [\![\pi]\!]_\rho(A') \\
\text{where}\ \ A_1 & := & [\![\alpha_1]\!]_\rho \\
\text{and}\ \ A_2 & := & [\![\alpha_2]\!]_\rho \\
\text{and}\ \ A' & := & \begin{cases} A_2 & \text{if } A = A_1 \\ A_1 & \text{if } A = A_2 \\ A & \text{wpp.} \end{cases}
\end{array}
$$

Model logiki więzów

$$\begin{array}{lcll} \rho & \in & (Atom \to SemAtom) \times (Var \to SemTerm) & \text{(interpretacje)} \\ \Gamma & ::= & \varnothing \mid c,\ \Gamma & \text{(środowisko więzów)} \end{array}$$

$$\begin{array}{lll}
\rho \vDash t_1 = t_2 & \text{iff} & [\![t_1]\!]_\rho = [\![t_2]\!]_\rho \\
\rho \vDash \alpha \,\#\, t & \text{iff} & [\![\alpha]\!]_\rho \notin \mathsf{FreeAtoms}([\![t]\!]_\rho) \\
\rho \vDash t_1 \sim t_2 & \text{iff} & |[\![t_1]\!]_\rho| = |[\![t_2]\!]_\rho| \\
\rho \vDash t_1 \prec t_2 & \text{iff} & |[\![t_1]\!]_\rho| \text{ jest ścisłym podkształtem } |[\![t_2]\!]_\rho| \\
\rho \vDash \mathsf{symbol}\ t & \text{iff} & |[\![t]\!]_\rho| \text{ jest symbolem funkcyjnym}
\end{array}$$

$$\rho \vDash \Gamma \quad \text{iff} \quad \forall c \in \Gamma.\ \rho \vDash c \qquad\qquad \Gamma \vDash c \quad \text{iff} \quad \forall \rho.\ \rho \vDash \Gamma \Longrightarrow \rho \vDash c$$

Solver

$$\mathcal{C} ::= \alpha \# t \mid t = t \mid s \sim s \mid s \prec s \mid \textsf{symbol}\ t \qquad\qquad \Gamma \vDash \mathcal{C} \;\equiv\; \Gamma; \varnothing \vdash \mathcal{C}$$

$$\begin{tikzcd}
\boxed{\cdots} \arrow[d] \\
\boxed{\dfrac{}{\Gamma; \lightning \vdash \mathcal{C}}}
\end{tikzcd}
\qquad
\begin{tikzcd}
 & \boxed{\mathcal{C}_0,\ \Gamma; \Delta \vdash \mathcal{C}} \arrow[dl] \arrow[d] \arrow[dr] & \\
\boxed{\mathcal{C}_1, \ldots,\ \Gamma; \Delta \vdash \mathcal{C}} & \boxed{\cdots} & \boxed{\mathcal{C}_N, \ldots,\ \Gamma; \Delta \vdash \mathcal{C}}
\end{tikzcd}
\qquad
\begin{tikzcd}
\boxed{\mathcal{C}_0,\ \Gamma; \Delta \vdash \mathcal{C}} \arrow[d] \\
\boxed{\Gamma'; \Delta' \vdash \mathcal{C}'}
\end{tikzcd}$$

$$\begin{tikzcd}
 & \boxed{\varnothing; \Delta \vdash \mathcal{C}_0} \arrow[dl] \arrow[d] \arrow[dr] & \\
\boxed{\Gamma_1; \Delta \vdash \mathcal{C}_1} & \boxed{\cdots} & \boxed{\Gamma_N; \Delta \vdash \mathcal{C}_N}
\end{tikzcd}
\qquad
\begin{tikzcd}
\boxed{\cdots} \arrow[d] \\
\boxed{\dfrac{\mathcal{C} \text{ jest trywialne}}{\varnothing; \Delta \vdash \mathcal{C}}}
\end{tikzcd}
\qquad
\begin{tikzcd}
\boxed{\cdots} \arrow[d] \\
\boxed{\dfrac{\mathcal{C} \in \Delta}{\varnothing; \Delta \vdash \mathcal{C}}}
\end{tikzcd}$$

$$
\begin{tikzcd}[column sep=small]
 & & \boxed{a \neq b,\ a \neq c,\ b \# a.(b\ c)E\ ;\ \varnothing \vdash c \# E} \arrow[d] & & \\
 & & \boxed{b \# a.(b\ c)E\ ;\ \{a \neq b,\ a \neq c\} \vdash c \# E} \arrow[dll] \arrow[dr] & & \\
\boxed{b = a\ ;\ \{a \neq b,\ a \neq c\} \vdash c \# E} \arrow[d] & & & \boxed{b \neq a,\ b \# (b\ c)E\ ;\ \{a \neq b,\ a \neq c\} \vdash c \# E} \arrow[d] & \\
\boxed{\{b = a\} \cup \{a \neq b,\ a \neq c\} = \lightning} & & & \boxed{b \# (b\ c)E\ ;\ \{a \neq b,\ a \neq c\} \vdash c \# E} \arrow[dl] \arrow[d] \arrow[dr] & \\
 & & \boxed{b \neq b,\ b \neq c,\ b \# E\ ;\ \{a \neq b,\ a \neq c\} \vdash c \# E} \arrow[d] & \boxed{b = b,\ b \neq c,\ c \# E\ ;\ \{a \neq b,\ a \neq c\} \vdash c \# E} \arrow[d] & \boxed{b = c,\ b \# E\ ;\ \{a \neq b,\ a \neq c\} \vdash c \# E} \arrow[d] \\
 & & \boxed{\{b \neq b\} \cup \{a \neq b,\ a \neq c\} = \lightning} & \boxed{b \neq c,\ c \# E\ ;\ \{a \neq b,\ a \neq c\} \vdash c \# E} \arrow[d] & \boxed{c \# E\ ;\ \{a \neq c\} \vdash c \# E} \arrow[d] \\
 & & & \boxed{c \# E\ ;\ \{a \neq b,\ a \neq c,\ b \neq c\} \vdash c \# E} \arrow[d] & \boxed{\varnothing\ ;\ \{a \neq c,\ c \# E\} \vdash c \# E} \arrow[d] \\
 & & & \boxed{\varnothing\ ;\ \{a \neq b,\ a \neq c,\ b \neq c,\ c \# E\} \vdash c \# E} \arrow[d] & \boxed{(c \# E) \in \{a \neq c,\ c \# E\}} \\
 & & & \boxed{(c \# E) \in \{a \neq b,\ a \neq c,\ b \neq c,\ c \# E\}} &
\end{tikzcd}
$$

Logika wyższego rzędu

$$\varphi \ ::= \quad \top \mid \bot \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \varphi \Longrightarrow \varphi$$

Logika wyższego rzędu

$$
\begin{aligned}
\varphi \ ::= \quad & \top \mid \bot \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \varphi \Longrightarrow \varphi \\
& \mid \forall_A a.\, \varphi \mid \forall_T X.\, \varphi \mid \forall_\kappa P.\varphi \\
& \mid \exists_A a.\, \varphi \mid \exists_T X.\, \varphi \mid \exists_\kappa P.\varphi
\end{aligned}
$$

Logika wyższego rzędu

$$\begin{aligned}\varphi \ ::= \quad & \top \mid \bot \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \varphi \Longrightarrow \varphi \\ & \mid \forall_A a.\, \varphi \mid \forall_T X.\, \varphi \mid \forall_\kappa P.\varphi \\ & \mid \exists_A a.\, \varphi \mid \exists_T X.\, \varphi \mid \exists_\kappa P.\varphi \\ & \mid \lambda_A a.\, \varphi \mid \lambda_T X.\, \varphi \mid \lambda P :: \kappa.\, \varphi \\ & \mid P \mid \varphi\, \alpha \mid \varphi\, t \mid \varphi\, \varphi\end{aligned}$$

Logika wyższego rzędu

$$
\begin{aligned}
\varphi \ ::= \quad & \top \mid \bot \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \varphi \Longrightarrow \varphi \\
& \mid \forall_A a.\, \varphi \mid \forall_T X.\, \varphi \mid \forall_\kappa P.\varphi \\
& \mid \exists_A a.\, \varphi \mid \exists_T X.\, \varphi \mid \exists_\kappa P.\varphi \\
& \mid \lambda_A a.\, \varphi \mid \lambda_T X.\, \varphi \mid \lambda P :: \kappa.\, \varphi \\
& \mid P \mid \varphi\ \alpha \mid \varphi\ t \mid \varphi\ \varphi \\
& \mid c \mid [c] \wedge \varphi \mid [c] \Longrightarrow \varphi
\end{aligned}
$$

Logika wyższego rzędu

$$
\begin{aligned}
\varphi \; ::= \quad & \top \mid \bot \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \varphi \Longrightarrow \varphi \\
& \mid \forall_A a.\, \varphi \mid \forall_T X.\, \varphi \mid \forall_\kappa P. \varphi \\
& \mid \exists_A a.\, \varphi \mid \exists_T X.\, \varphi \mid \exists_\kappa P. \varphi \\
& \mid \lambda_A a.\, \varphi \mid \lambda_T X.\, \varphi \mid \lambda P :: \kappa.\, \varphi \\
& \mid P \mid \varphi\, \alpha \mid \varphi\, t \mid \varphi\, \varphi \\
& \mid c \mid [c] \wedge \varphi \mid [c] \Longrightarrow \varphi \\
& \mid \textsf{fix}\ P(X) :: \kappa = \varphi
\end{aligned}
$$

Dedukcja naturalna

$$\frac{\varphi \in \Theta}{\Gamma; \Theta \vdash \varphi}\ \textsc{Assumption}$$

$$\frac{\Gamma \models c}{\Gamma; \Theta \vdash c}\ \textsc{ConstrI}$$

$$\frac{\Gamma; \Theta, \varphi_1 \vdash \varphi_2}{\Gamma; \Theta \vdash \varphi_1 \implies \varphi_2}\ \textsc{ImpI}$$

$$\frac{\Gamma, c; \Theta \vdash \varphi}{\Gamma; \Theta \vdash [c] \implies \varphi}\ \textsc{Constr ImpI}$$

$$\frac{\Gamma_1; \Theta_1 \vdash \varphi_1 \qquad \Gamma_2; \Theta_2 \vdash \varphi_2}{\Gamma_1 \cup \Gamma_2; \Theta_2 \cup \Theta_2 \vdash \varphi_1 \wedge \varphi_2}\ \textsc{AndI}$$

$$\frac{\Gamma_1; \Theta_1 \vdash c \qquad \Gamma_2; \Theta_2 \vdash \varphi}{\Gamma_1 \cup \Gamma_2; \Theta_2 \cup \Theta_2 \vdash [c] \wedge \varphi}\ \textsc{Constr AndI}$$

Aksjomaty

$$\frac{\Gamma;\Theta,(\forall_T X'.\,[X' \prec X] \Longrightarrow \varphi(X')) \vdash \varphi(X)}{\Gamma;\Theta \vdash \forall_T X.\,\varphi(X)}\ \textsc{Induction}$$

$$\frac{}{\vdash \forall_A\ a,\ a'.\,(a = a') \vee (a \neq a')}\ \textsc{Axiom Compare}$$

$$\frac{}{\vdash \forall_T X.\ \exists_A a.\,(a \mathrel{\#} X)}\ \textsc{Axiom Fresh}$$

$$\frac{}{\begin{array}{c}\vdash \forall_T X.\,(\exists_A a.\ X = a) \vee (\exists_A a.\ \exists_T X'.\ X = a.X') \\ \vee\,(\exists_T X_1,\ X_2.\ X = X_1\ X_2) \vee (symbol\ X)\end{array}}\ \textsc{Axiom Inversion}$$

Rekursja — punkt stały

$$(\mathsf{fix}\ P(X) :: \kappa = \varphi)\ t \ \equiv\ \varphi\{X \mapsto t\}\{P \mapsto (\mathsf{fix}\ P(X) :: \kappa = \varphi)\} \qquad \textsc{Fixpoint Unwrap}$$

Rekursja — punkt stały

$$(\mathsf{fix}\ P(X) :: \kappa = \varphi)\ t \ \equiv\ \varphi\{X \mapsto t\}\{P \mapsto (\mathsf{fix}\ P(X) :: \kappa = \varphi)\} \qquad \textsc{Fixpoint Unwrap}$$

$$\frac{\Gamma; \Sigma, (P :: \forall_T Y.\, [Y \prec X]\kappa\{X \mapsto Y\}) \vdash \varphi :: \kappa}{\Gamma; \Sigma \vdash (\mathsf{fix}\ P(X) :: \kappa = \varphi) :: \forall_T X.\, \kappa}\ \textsc{Kind Fixpoint}$$

Rekursja — punkt stały

$$(\text{fix } P(X) :: \kappa = \varphi)\ t \ \equiv\ \varphi\{X \mapsto t\}\{P \mapsto (\text{fix } P(X) :: \kappa = \varphi)\} \qquad \textsc{Fixpoint Unwrap}$$

$$\frac{\Gamma; \Sigma, (P :: \forall_T Y.\, [Y \prec X]\kappa\{X \mapsto Y\}) \vdash \varphi :: \kappa}{\Gamma; \Sigma \vdash (\text{fix } P(X) :: \kappa = \varphi) :: \forall_T X.\, \kappa}\ \textsc{Kind Fixpoint}$$

$$\frac{}{\Gamma; \Sigma \vdash c :: \star}\ \textsc{Kind Constr} \qquad \frac{\Gamma, c; \Sigma \vdash \varphi :: \star}{\Gamma; \Sigma \vdash [c] \wedge \varphi :: \star}\ \textsc{Kind ConstrAnd}$$

$$\frac{\Gamma; \Sigma \vdash \varphi :: \forall_T X.\, \kappa}{\Gamma; \Sigma \vdash \varphi\ t :: \kappa\{X \mapsto t\}}\ \textsc{Kind AppTerm} \qquad \frac{\Gamma \vDash c}{\Gamma \vdash [c]\kappa <: \kappa}\ \textsc{Subkind Reduce}$$

Implementacja

`Proof`	"Interfejs dla sprawdzającego dowód": dedukcja naturalna zaimplementowana w postaci smart-konstruktorów abstrakcyjnego typu danych `proof`.
`IncProof`	"Interfejs dla prowadzącego dowód": drzewa niepełnych dowodów.
`Prover`	"Interfejs dla człowieka": asystent dowodzenia.

Studium przypadku: rachunek lambda z typami prostymi

```
let lambda_symbols = (* symbole używane w formalizacji STLC *)
  ["lam"; "app"; "base"; "arrow"; "nil"; "cons"]

let term_predicate = (* Term e *)
  "fix Term(e): * =
   var: (∃ a :atom. (e = a))
   ∨
   lam: (∃ a :atom. ∃ e' :term. [e = lam (a.e')] ∧ (Term e'))
   ∨
   app: (∃ e1 e2 :term. [e = app e1 e2] ∧ (Term e1) ∧ (Term e2))"

let type_predicate = (* Type t *)
  "fix Type(t): * =
   base: (t = base)
   ∨
   arrow: (∃ t1 t2 :term. [t = arrow t1 t2]
             ∧ (Type t1) ∧ (Type t2))"
```

```
let typing_relation = (* Typing e env t *)
  "fix Typing(e): ∀ env t :term. * =
   fun env t :term →
     var: (∃ a :atom. [e = a] ∧ (InEnv env a t))
     ∨
     lam: (∃ a :atom.∃ e' t1 t2 :term.
            [e = lam (a.e')] ∧ [t = arrow t1 t2]
              ∧ (Type t1) ∧ (Typing e' {cons a t1 env} t2))
     ∨
     app: (∃ e1 e2 t2 :term. [e = app e1 e2]
            ∧ (Typing e1 env {arrow t2 t}) ∧ (Typing e2 env t2))"

let inenv_relation = (* InEnv env a t *)
  "fix InEnv(env): ∀ a :atom. ∀ t :term. * =
   fun (a :atom) (t :term) →
     current: (∃ env': term. env = cons a t env')
     ∨
     next: (∃ b :atom. ∃ s env': term.
            [env = cons b s env'] ∧ [a ≠ b] ∧ (InEnv env' a t))"
```

```
let empty_contradiction_thm = lambda_thm
  "forall a :atom.
   forall t :term.
     (InEnv nil a t) => false"


  let empty_contradiction =
  proof' empty_contradiction_thm
```

```
Unfinished:

[ ]
[ ]
⊢ ∀ a : atom. ∀ t : term. (InEnv nil a t) => ⊥
```

```
let empty_contradiction_thm = lambda_thm
  "forall a :atom.
   forall t :term.
     (InEnv nil a t) => false"
```

```
  let empty_contradiction =
  proof' empty_contradiction_thm
  |> intros ["a"; "t"; "H"]
```

```
Unfinished:

[ ]
[ H : InEnv nil a t ]
⊢ ⊥
```

```
let empty_contradiction_thm = lambda_thm
  "forall a :atom.
   forall t :term.
     (InEnv nil a t) => false"


  let empty_contradiction =
  proof' empty_contradiction_thm
  |> intros ["a"; "t"; "H"] %> destruct_assm "H"
```

```
Unfinished:

[ ]
[ ]
⊢ (∃ env' : term. (nil = cons a t env')) => ⊥
```

```
let empty_contradiction_thm = lambda_thm
  "forall a :atom.
   forall t :term.
     (InEnv nil a t) => false"
```

```
  let empty_contradiction =
  proof' empty_contradiction_thm
  |> intros ["a"; "t"; "H"] %> destruct_assm "H"
  |> intros' ["contra"; "env'"]
```

```
Unfinished:

[ ]
[ contra : (nil = cons a t env') ]
```
$\vdash \bot$

```
let empty_contradiction_thm = lambda_thm
  "forall a :atom.
   forall t :term.
     (InEnv nil a t) => false"


  let empty_contradiction =
  proof' empty_contradiction_thm
  |> intros ["a"; "t"; "H"] %> destruct_assm "H"
  |> intros' ["contra"; "env'"] %> discriminate
```

```
Unfinished:

[ ]
[ ]
⊢ (∃ b : atom. ∃ s env' : term.
      [nil = cons b s env'] ∧ [a =/= b] ∧ InEnv env' a t)
       => ⊥
```

```
let empty_contradiction_thm = lambda_thm
  "forall a :atom.
   forall t :term.
     (InEnv nil a t) => false"
```

```
  let empty_contradiction =
  proof' empty_contradiction_thm
  |> intros ["a"; "t"; "H"] %> destruct_assm "H"
  |> intros' ["contra"; "env'"] %> discriminate
  |> intros' ["contra"; "b"; "s"; "env'"; ""]
```

```
Unfinished:

[ nil = cons b s env' ]
[ contra : [a =/= b] ∧ InEnv env' a t ]
⊢ ⊥
```

```
let empty_contradiction_thm = lambda_thm
  "forall a :atom.
   forall t :term.
     (InEnv nil a t) => false"


  let empty_contradiction =
  proof' empty_contradiction_thm
  |> intros ["a"; "t"; "H"] %> destruct_assm "H"
  |> intros' ["contra"; "env'"] %> discriminate
  |> intros' ["contra"; "b"; "s"; "env'"; ""] %> solve
```

```
Finished:

[ ]
[ ]
⊢ ∀ a : atom. ∀ t : term. (InEnv nil a t) => ⊥
```

```
let empty_contradiction_thm = lambda_thm
  "forall a :atom.
   forall t :term.
     (InEnv nil a t) => false"


  let empty_contradiction =
  proof' empty_contradiction_thm
  |> intros ["a"; "t"; "H"] %> destruct_assm "H"
  |> intros' ["contra"; "env'"] %> discriminate
  |> intros' ["contra"; "b"; "s"; "env'"; ""] %> solve
  |> qed
```

```
let value_predicate = (* Value v *)
  "fun e :term →
     var: (∃ a :atom. [e = a])
     ∨
     lam: (∃ a :atom. ∃ e' : term. [e = lam (a.e')] ∧ (Term e'))"

let steps_relation = (* Steps e e' *)
  "fix Steps(e): ∀ e' :term.* = fun e' :term →
     app_l: (∃ e1 e1' e2 :term. [e = app e1 e2]
              ∧ [e' = app e1' e2] ∧ (Steps e1 e1') )
     ∨
     app_r: (∃ v e2 e2' :term. [e = app v e2]
              ∧ [e' = app v e2'] ∧ (Value v) ∧ (Steps e2 e2') )
     ∨
     app: (∃ a :atom.∃ e_a v :term. [e = app (lam (a.e_a)) v]
            ∧ (Value v) ∧ (Sub e_a a v e') )"

let progressive_predicate = (* Progressive e *)
  "fun e:term →
     value: (Value e) ∨ steps: (∃ e' :term. Steps e e')"
```

```
let progress_thm = lambda_thm
  "∀ e t :term. (Typing e nil t) ⟹ (Progressive e)"


let canonical_form_thm = lambda_thm
  "∀ v :term. (Value v) ⟹
   ∀ t :term. (Typing v nil t) ⟹
     (∃ a :atom. ∃ e :term. [v = lam (a.e)] ∧ (Term e))"
```

```
let canonical_form =
  proof' canonical_form_thm
  |> intros ["v"; "t"; "Hv"]
```

```
Unfinished:

[ ]
[ Hv : Value v ]
⊢ (Typing v nil t) =>
    ∃ a : atom. ∃ e : term. [v = lam (a.e)] ∧ Term e
```

```
let canonical_form =
  proof' canonical_form_thm
  |> intros ["v"; "t"; "Hv"] %> intro'
```

```
Unfinished:

[ ]
[ Hv : Value v ]
⊢ (∃ a : atom. [v = a] ∧ InEnv nil a t)
    ∃ a : atom. ∃ e : term. [v = lam (a.e)] ∧ Term e
```

```
let canonical_form =
  proof' canonical_form_thm
  |> intros ["v"; "t"; "Hv"] %> intro'
  |> intros' ["contra"; "a"; ""]
```

```
Unfinished:

[ v = a ]
[ Hv : Value v ; contra : InEnv nil a t ]
⊢ ∃ a'1 : atom. ∃ e : term. [v = lam (a'1.e)] ∧ Term e
```

```
let canonical_form =
  proof' canonical_form_thm
  |> intros ["v"; "t"; "Hv"] %> intro'
  |> intros' ["contra"; "a"; ""]
     %> ex_falso
     %> apply_thm_spec empty_contradiction ["a"; "t"]
     %> assumption
```

```
Unfinished:

[ ]
[ Hv : Value v ]
⊢ (∃ a : atom. ∃ e' t1 t2 : term.
     [v = lam (a.e')] ∧ [t = arrow t1 t2]
     ∧ Type t1 ∧ Typing e' {cons a t1 nil} t2)
       => ∃ a : atom. ∃ e : term. [v = lam (a.e)] ∧ Term e
```

```
let canonical_form =
  proof' canonical_form_thm
  |> intros ["v"; "t"; "Hv"] %> intro'
  |> intros' ["contra"; "a"; ""] ...
  |> intros' ["Hlam"; "a"; "e"; "t1"; "t2"; ""; ""; ""]
```

```
Unfinished:

[ v = lam (a.e) ; t = arrow t1 t2 ]
[
  Hlam_2 : Typing e {cons a t1 nil} t2 ;
  Hlam_1 : Type t1 ;
  Hv : Value v
]
⊢ ∃ a'1 : atom. ∃ e'1 : term. [v = lam (a'1.e'1)] ∧ Term e'1
```

```
let canonical_form =
  proof' canonical_form_thm
  |> intros ["v"; "t"; "Hv"] %> intro'
  |> intros' ["contra"; "a"; ""] ...
  |> intros' ["Hlam"; "a"; "e"; "t1"; "t2"; ""; ""; ""]
     %> exists' ["a"; "e"] %> solve
```

```
Unfinished:

[ v = lam (a.e) ; t = arrow t1 t2 ]
[
  Hlam_2 : Typing e {cons a t1 nil} t2 ;
  Hlam_1 : Type t1 ;
  Hv : Value v
]
⊢ Term e
```

```
let canonical_form =
  proof' canonical_form_thm
  |> intros ["v"; "t"; "Hv"] %> intro'
  |> intros' ["contra"; "a"; ""] ...
  |> intros' ["Hlam"; "a"; "e"; "t1"; "t2"; ""; ""; ""]
     %> exists' ["a"; "e"] %> solve
     %> apply_thm_spec typing_terms ["e"; "cons a t1 nil"; "t2"]
     %> assumption
```

```
Unfinished:

[ ]
[ Hv : Value v ]
⊢ (∃ e1 e2 t2 : term.
     [v = app e1 e2] ∧ Typing e1 nil {arrow t2 t}
      ∧ Typing e2 nil t2)
      => ∃ a : atom. ∃ e : term. [v = lam (a.e)] ∧ Term e
```

```
let canonical_form =
  proof' canonical_form_thm
  |> intros ["v"; "t"; "Hv"] %> intro'
  |> intros' ["contra"; "a"; ""]
     %> ex_falso
     %> apply_thm_spec empty_contradiction ["a"; "t"]
     %> assumption
  |> intros' ["Hlam"; "a"; "e"; "t1"; "t2"; ""; ""; ""]
     %> exists' ["a"; "e"] %> solve
     %> apply_thm_spec typing_terms ["e"; "cons a t1 nil"; "t2"]
     %> assumption
  |> intros' ["contra"; "e1"; "e2"; "t2"; ""]
     %> ex_falso
     %> destruct_assm "Hv"
     %> (intros' ["contra_var"; "a"] %> discriminate)
     %> (intros' ["contra_lam"; "a"; "e"; ""] %> discriminate)
  |> qed
```

```
let preservation_thm = lambda_thm
  "forall e e' env t :term.
     (Typing e env t) =>
     (Steps e e') =>
       (Typing e' env t)"

let sub_lemma_thm = lambda_thm
  "∀ e env t :term.
   ∀ a : atom. ∀ ta :term.
   ∀ v e' :term.
     (Typing v env ta) ⟹
     (Typing e {cons a ta env} t) ⟹
     (Sub e a v e') ⟹
       (Typing e' env t)"

let weakening_lemma_thm = lambda_thm
  "∀ e env1 t env2 : term.
     (Typing e env1 t) ⟹
     (EnvInclusion env1 env2) ⟹
       (Typing e env2 t)"
```

Koniec