Nominal logic
for reasoning about terms
with variable bindings

(Logika dziedzinowa do wnioskowania
o termach z wiązaniem zmiennych)

Dominik Gulczyński

Praca magisterska

Promotor: dr Piotr Polesiuk

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

5 sierpnia 2023

Abstract

We describe logic for reasoning about terms with variable bindings.

Streszczenie

Przedstawiamy logikę dziedzinową do wnioskowania o termach z wiązaniem zmiennych.

Contents

Chapter 1

Introduction

1.1 Problem statement

...

1.2 Motivation

...

1.3 Related work

1.3.1 Nominal logics & permutations

1.4 Contributions

...

Chapter 2

Terms and constraints

In classical first-order logic, terms are constructed from variables and applications of functional symbols to other terms. This work introduces an extension to terms with expressions closely resembling the syntax of lambda calculus. The aim is to create a flexible framework for reasoning about the lambda calculus and its derivations.

To achieve this goal, we introduce an infinite set of *atoms* (represented by lowercase letters) which correspond to the bound variables in terms, analogous to the variables in lambda calculus. This set is disjoint from the set of variables commonly used in first-order logic, which we will refer to as *variables* (denoted by uppercase letters).

Terms are defined by the following grammar:

$$
\begin{array}{ll}
\pi & ::= \mathsf{id} \mid (\alpha\ \alpha)\pi \\
\alpha & ::= \pi\ a \\
t & ::= \alpha \mid \pi\ X \mid \alpha.t \mid t\ t \mid s
\end{array}
$$

It's important to note that terms do not inherently incorporate notions of computation, reduction, or binding. These expressions closely resemble lambda calculus syntax but lack its operational semantics. However, the intuitions associated with these expressions are not baseless. Their practical application is observed in the sublogic of constraints defined on top of terms, used to reason about concepts such as *freshness*, *variable binding*, and *structural* order, as well as their logical model.

Constraints are given by the following grammar:

$$c \quad ::= \quad \alpha \# t \mid t = t \mid t \sim t \mid t \prec t \quad \text{(constraints)}$$

with following semantics:

$\alpha \# t$ — atom α is Fresh in term t, i.e. does not occur in t as a free variable

$t_1 = t_2$ — terms t_1 and t_2 are alpha-equivalent

$t_1 \sim t_2$ — terms t_1 and t_2 possess an identical shape,
i.e. after erasing all atoms, terms t_1 and t_2 would be equal

$t_1 \prec t_2$ — shape of term t_1 is structurally smaller than the shape of term t_2,
i.e. after erasing all atoms t_1 would be equal to some subterm of t_2

We use metavariable Γ to represent finite sets of constraints.

$$\begin{array}{lcll} T & ::= & A \mid n \mid \$T \mid T@T \mid s & \text{(semantic terms)} \\ S & ::= & \_ \mid \_.S \mid S@S \mid s & \text{(semantic shapes)} \end{array}$$

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

$$\begin{array}{rcl} |A| & = & \_ \\ |n| & = & \_ \\ |\$T| & = & \_.|T| \\ 1|T_1@T_2| & = & |T_1|@|T_2| \end{array}$$

$$\begin{array}{rll} \rho \vDash t_1 = t_2 & \text{iff} & [\![t_1]\!]_\rho = [\![t_2]\!]_\rho \\ \rho \vDash \alpha \# t & \text{iff} & [\![\alpha]\!]_\rho \notin \mathsf{FreeAtoms}([\![t]\!]_\rho) \\ \rho \vDash t_1 \sim t_2 & \text{iff} & |[\![t_1]\!]_\rho| = |[\![t_2]\!]_\rho| \\ \rho \vDash t_1 \prec t_2 & \text{iff} & |[\![t_1]\!]_\rho| \text{ is a strict subshape of } |[\![t_2]\!]_\rho| \end{array}$$

We write $\rho \vDash \Gamma$ iff for all $c \in \Gamma$, we have $\rho \vDash c$. We write $\Gamma \vDash c$ iff for every ρ such that $\rho \vDash \Gamma$, we have $\rho \vDash c$.

Within this model, we establish the existence of a decidible algorithm for determining whether $C_1, \ldots, C_n \models C_0$, meaning there is a deterministic way to check whether constraints $C_1, \ldots, C_n$ imply C_0. This algorithm is presented in the following chapter.

Chapter 3

Constraint solver

At the heart of our work lies the Solver, an algorithm designed to resolve constraints. A high level perspective of the Solver is that it dissects constraints on both sides of the turnstile into irreducible components that are solved easily.

Given a set of assumptions $c_1, \ldots, c_n$, it verifies whether a given goal c_0 holds. Technically, the Solver determines whether, every possible substitution of variables into closed terms in $c_0, c_1, \ldots, c_n$, such that $c_1, \ldots, c_n$ are satisfied, will also satisfy c_0.

For the sake of convenience and implementation efficiency, the Solver operates on slightly different constraints compared to those found in formulas and kinds. The key distinction lies in the use of *shapes* in shape constraints rather than terms.

Solver constraints and shapes are defined by the following grammar:

$$\begin{array}{rcll} \mathcal{C} & ::= & \alpha \mathbin{\#} t \mid t = t \mid S \sim S \mid S \prec S & \text{(solver constraints)} \\ S & ::= & \_ \mid X \mid \_.S \mid S\ S \mid s & \text{(shapes)} \end{array}$$

Solver erases atoms from terms in shape constraints, effectively transforming them from *constraints* to *solver constraints.*

We add another environment Δ to distinguish between the potentially-reducible assumptions in Γ. For convenience, we will write $a \neq \alpha$ instead of $a \mathbin{\#} \alpha$ as it gives a clear intuition of atom freshness implying inequality. Additionally, when $\alpha = \pi a$, we will denote $\alpha \mathbin{\#} t$ to mean $a \mathbin{\#} \pi^{-1} t$.

Irreducible constraints are:

$$\begin{array}{rcl} a_1 \neq a_2 & — & \text{atoms } a_1 \text{ and } a_2 \text{ are different} \\ a \mathbin{\#} X & — & \text{atom } a \text{ is Fresh in variable } X \\ X_1 \sim X_2 & — & \text{variables } X_1 \text{ and } X_2 \text{ posses the same shape} \\ X \sim t & — & \text{variable } X \text{ has a shape of term } t \\ t \prec X & — & \text{term } t \text{ strictly subshapes variable } X \end{array}$$

After all the constraints are reduced to such simple constraints we reduce the goal-constraint and repeat the reduction procedure on new assumptions and goal. We either arrive at a contradictory environment or all the assumptions and goal itself are reduced to irreducible constraints, which is as simple as checking if the goal occurs on the left side of the turnstile:

$$\dfrac{\dfrac{\dfrac{\mathcal{C}'' \in \Delta''}{\cdots}}{\Gamma'; \Delta' \vDash \mathcal{C}' \qquad \cdots}}{\Gamma; \Delta \vDash \mathcal{C}}$$

Decidability of atom equality plays an important role in the reduce procedure:

$$\dfrac{\Gamma; \Delta \vDash a = \pi^{-1}\alpha}{\Gamma; \Delta \vDash \pi a = \alpha} \qquad \dfrac{\begin{array}{c} a \neq \alpha_1, a \neq \alpha_2, \Gamma; \Delta \vDash a = \alpha \\ a = \alpha_1, a \neq \alpha_2, \Gamma; \Delta \vDash \alpha_2 = \alpha \qquad a = \alpha_2, \Gamma; \Delta \vDash \alpha_1 = \alpha \end{array}}{\Gamma; \Delta \vDash a = (\alpha_1\ \alpha_2)\alpha}$$

$$\dfrac{\Gamma; \Delta \vDash \pi \text{ idempotent on } X}{\Gamma; \Delta \vDash X = \pi X} \qquad \dfrac{\Gamma; \Delta \vDash X_1 = \pi_1^{-1}\pi_2 X_2}{\Gamma; \Delta \vDash \pi_1 X_1 = \pi_2 X_2}$$

$$\dfrac{\Gamma; \Delta \vDash \alpha_1 \# t_2 \qquad \Gamma; \Delta \vDash t_1 = (\alpha_1\ \alpha_2)t_2}{\Gamma; \Delta \vDash \alpha_1.t_1 = \alpha_2.t_2} \qquad \dfrac{\Gamma; \Delta \vDash t_1 = t_2 \qquad \Gamma; \Delta \vDash t_1' = t_2'}{\Gamma; \Delta \vDash t_1 t_1' = t_2 t_2'}$$

$$\dfrac{}{\Gamma; \Delta \vDash a = a} \qquad \dfrac{}{\Gamma; \Delta \vDash X = X} \qquad \dfrac{}{\Gamma; \Delta \vDash s = s}$$

$$\dfrac{\forall a \in \pi.\ \Gamma; \Delta \vDash a = \pi a\ \vee\ \Gamma; \Delta \vDash a \# X}{\Gamma; \Delta \vDash \pi \text{ idempotent on } X}$$

$$\dfrac{a_1 \neq a_2 \in \Delta}{\Gamma; \Delta \vDash a_1 \# a_2} \qquad \dfrac{\begin{array}{c} a \neq \alpha_1, a \neq \alpha_2, \Gamma; \Delta \vDash a \# \alpha \\ a = \alpha_1, a \neq \alpha_2, \Gamma; \Delta \vDash \alpha_1 \# \alpha \qquad a = \alpha_2, \Gamma; \Delta \vDash \alpha_2 \# \alpha \end{array}}{\Gamma; \Delta \vDash a \# (\alpha_1\ \alpha_2)\alpha}$$

$$\dfrac{a \# X \in \Delta}{\Gamma; \Delta \vDash a \# X} \qquad \dfrac{\begin{array}{c} a \neq \alpha_1, a \neq \alpha_2, \Gamma; \Delta \vDash a \# \pi X \\ a = \alpha_1, a \neq \alpha_2, \Gamma; \Delta \vDash \alpha_1 \# \pi X \qquad a = \alpha_2, \Gamma; \Delta \vDash \alpha_2 \# \pi X \end{array}}{\Gamma; \Delta \vDash a \# (\alpha_1\ \alpha_2)\pi X}$$

$$\dfrac{a \neq \alpha, \Gamma; \Delta \vDash a \# t}{\Gamma; \Delta \vDash a \# \alpha.t} \qquad \dfrac{\Gamma; \Delta \vDash a \# t_1 \qquad \Gamma; \Delta \vDash a \# t_2}{\Gamma; \Delta \vDash a \# t_1 t_2} \qquad \dfrac{}{\Gamma; \Delta \vDash a \# s}$$

$$\dfrac{X_1 \sim X_2 \in \Delta}{\Gamma; \Delta \vDash X_1 \sim X_2} \qquad \dfrac{X \sim S' \in \Delta \qquad \Gamma; \Delta \vDash S' \sim S}{\Gamma; \Delta \vDash X \sim S}$$

$$\dfrac{\Gamma; \Delta \vDash S_1 \sim S_2}{\Gamma; \Delta \vDash \_.S_1 \sim \_.S_2} \qquad \dfrac{\Gamma; \Delta \vDash S_1 \sim S_2 \qquad \Gamma; \Delta \vDash S_1' \sim S_2'}{\Gamma; \Delta \vDash S_1 S_1' \sim S_2 S_2'} \qquad \dfrac{}{\Gamma; \Delta \vDash s \sim s}$$

$$\dfrac{S_2 \prec X \in \Delta \qquad \Gamma; \Delta \vDash S_2 \sim X}{\Gamma; \Delta \vDash S_1 \prec X} \qquad \dfrac{S_2 \prec X \in \Delta \qquad \Gamma; \Delta \vDash S_2 \prec X}{\Gamma; \Delta \vDash S_1 \prec X}$$

$$\frac{\Gamma; \Delta \vDash S_1 \sim S_2}{\Gamma; \Delta \vDash S_1 \prec \_.S_2} \qquad \frac{\Gamma; \Delta \vDash S_1 \prec S_2}{\Gamma; \Delta \vDash S_1 \prec \_.S_2}$$

$$\frac{\Gamma; \Delta \vDash S_1 \sim S_2}{\Gamma; \Delta \vDash S_1 \prec S_2S_2'} \qquad \frac{\Gamma; \Delta \vDash S_1 \sim S_2'}{\Gamma; \Delta \vDash S_1 \prec S_2S_2'} \qquad \frac{\Gamma; \Delta \vDash S_1 \prec S_2}{\Gamma; \Delta \vDash S_1 \prec S_2S_2'} \qquad \frac{\Gamma; \Delta \vDash S_1 \prec S_2'}{\Gamma; \Delta \vDash S_1 \prec S_2S_2'}$$

$$\frac{a_1 \neq a_2 \in \Delta}{a_1 = a_2, \Gamma; \Delta \vDash \mathcal{C}} \qquad \frac{\Gamma\{a_1 \mapsto a_2\}; \Delta\{a_1 \mapsto a_2\} \vDash \mathcal{C}\{a_1 \mapsto a_2\}}{a_1 = a_2, \Gamma; \Delta \vDash \mathcal{C}}$$

$$\frac{\begin{array}{c} a \neq \alpha_1, a \neq \alpha_2, a = \alpha, \Gamma; \Delta \vDash \mathcal{C} \\ a = \alpha_1, a \neq \alpha_2, \alpha_2 = \alpha, \Gamma; \Delta \vDash \mathcal{C} \qquad a = \alpha_2, \alpha_1 = \alpha, \Gamma; \Delta \vDash \mathcal{C} \end{array}}{a = (\alpha_1\ \alpha_1)\alpha, \Gamma; \Delta \vDash \mathcal{C}}$$

$$\frac{a = \pi^{-1}\alpha, \Gamma; \Delta \vDash \mathcal{C}}{\pi a = \alpha, \Gamma; \Delta \vDash \mathcal{C}} \qquad \frac{}{a = t_1t_2, \Gamma; \Delta \vDash \mathcal{C}} \qquad \frac{}{a = \alpha.t, \Gamma; \Delta \vDash \mathcal{C}} \qquad \frac{}{a = s, \Gamma; \Delta \vDash \mathcal{C}}$$

$$\frac{\vDash \text{ idempotent on } X}{X = \pi X, \Gamma; \Delta \vDash \mathcal{C}} \qquad \frac{\pi \text{ idempotent on } X, \Gamma; \Delta \vDash \mathcal{C}}{X = \pi X, \Gamma; \Delta \vDash \mathcal{C}}$$

$$\frac{\Gamma\{X \mapsto t\}; \Delta\{X \mapsto t\} \vDash \mathcal{C}\{X \mapsto t\}}{X = t, \Gamma; \Delta \vDash \mathcal{C}} \qquad \frac{X = \pi^{-1}t, \Gamma; \Delta \vDash \mathcal{C}}{\pi X = t, \Gamma; \Delta \vDash \mathcal{C}}$$

$$\frac{\alpha_1 \# \alpha_2.t_2,\ t_1 = (\alpha_1\ \alpha_2)t_2,\ \Gamma; \Delta \vDash \mathcal{C}}{\alpha_1.t_1 = \alpha_2.t_2, \Gamma; \Delta \vDash \mathcal{C}} \qquad \text{Other term constructors trivial}$$

$$\frac{t_1 = t_2,\ t_1' = t_2',\ \Gamma; \Delta \vDash \mathcal{C}}{t_1t_1' = t_2t_2', \Gamma; \Delta \vDash \mathcal{C}} \qquad \text{Other term constructors trivial}$$

$$\frac{s_1 \neq s_2}{s_1 = s_2, \Gamma; \Delta \vDash \mathcal{C}} \qquad \frac{}{s = s, \Gamma; \Delta \vDash \mathcal{C}} \qquad \text{Other term constructors trivial}$$

$$\frac{(\forall a \in \pi.\ \Gamma; \Delta \vDash a = \pi a \ \vee\ \Gamma; \Delta \vDash a \# X), \Gamma; \Delta \vDash \mathcal{C}}{\pi \text{ idempotent on } X, \Gamma; \Delta \vDash \mathcal{C}}$$

$$\frac{}{a \neq a, \Gamma; \Delta \vDash \mathcal{C}} \qquad \frac{\Gamma; \{a_1 \neq a_2\} \cup \Delta \vDash \mathcal{C}}{a_1 \neq a_2,\ \Gamma; \Delta \vDash \mathcal{C}} \qquad \frac{\Gamma; \{a \# X\} \cup \Delta \vDash \mathcal{C}}{a \# X,\ \Gamma; \Delta \vDash \mathcal{C}}$$

$$\frac{\begin{array}{c} a \neq \alpha_1, a \neq \alpha_2, a \# \alpha, \Gamma; \Delta \vDash \mathcal{C} \\ a = \alpha_1, a \neq \alpha_2, \alpha_2 \# \alpha, \Gamma; \Delta \vDash \mathcal{C} \qquad a = \alpha_2, \alpha_1 \# \alpha, \Gamma; \Delta \vDash \mathcal{C} \end{array}}{a \# (\alpha_1\ \alpha_1)\alpha, \Gamma; \Delta \vDash \mathcal{C}}$$

$$\frac{\begin{array}{c} a \neq \alpha_1, a \neq \alpha_2, a \# X, \Gamma; \Delta \vDash \mathcal{C} \\ a = \alpha_1, a \neq \alpha_2, \alpha_2 \# X, \Gamma; \Delta \vDash \mathcal{C} \qquad a = \alpha_2, \alpha_1 \# X, \Gamma; \Delta \vDash \mathcal{C} \end{array}}{a \# (\alpha_1\ \alpha_1)X, \Gamma; \Delta \vDash \mathcal{C}}$$

$$\frac{a \# \alpha,\ \Gamma; \Delta \vDash \mathcal{C} \qquad a \# \alpha,\ a \# t,\ \Gamma; \Delta \vDash \mathcal{C}}{a \# \alpha.t, \Gamma; \Delta \vDash \mathcal{C}}$$

$$\frac{a \# t_1, \Gamma; \Delta \vDash \mathcal{C} \qquad a \# t_2, \Gamma; \Delta \vDash \mathcal{C}}{a \# t_1 t_2, \Gamma; \Delta \vDash \mathcal{C}}$$

$$\frac{\Gamma; \Delta \vDash \mathcal{C}}{a \# s, \Gamma; \Delta \vDash \mathcal{C}}$$

$$\frac{\Gamma; \{X_1 \sim X_2\} \cup \Delta \vDash \mathcal{C}}{X_1 \sim X_2, \Gamma; \Delta \vDash \mathcal{C}} \qquad \frac{\Gamma; \{X \sim S\} \cup \Delta \vDash \mathcal{C}}{X \sim S,\ \Gamma; \Delta \vDash \mathcal{C}}$$

$$\frac{\Gamma; \Delta \vDash \mathcal{C}}{a_1 \sim a_2, \Gamma; \Delta \vDash \mathcal{C}} \qquad \text{Other term constructors trivial}$$

$$\frac{t_1 \sim t_2, \Gamma; \Delta \vDash \mathcal{C}}{\_.t_1 \sim \_.t_2, \Gamma; \Delta \vDash \mathcal{C}} \qquad \text{Other term constructors trivial}$$

$$\frac{t_1 \sim t_2, \Gamma; \Delta \vDash \mathcal{C} \qquad t_1' \sim t_2', \Gamma; \Delta \vDash \mathcal{C}}{t_1 t_1' \sim t_2 t_2', \Gamma; \Delta \vDash \mathcal{C}} \qquad \text{Other term constructors trivial}$$

$$\frac{s_1 \neq s_2}{s_1 \sim s_2, \Gamma; \Delta \vDash \mathcal{C}} \qquad \frac{}{s \sim s, \Gamma; \Delta \vDash \mathcal{C}} \qquad \text{Other term constructors trivial}$$

$$\frac{\Gamma; \{t \prec X\} \cup \Delta \vDash \mathcal{C}}{t \prec X, \Gamma; \Delta \vDash \mathcal{C}}$$

$$\frac{t_1 \sim t_2, \Gamma; \Delta \vDash \mathcal{C} \qquad t_1 \prec t_2, \Gamma; \Delta \vDash \mathcal{C}}{t_1 \prec \_.t_2, \Gamma; \Delta \vDash \mathcal{C}}$$

$$\frac{t_1 \sim t_2, \Gamma; \Delta \vDash \mathcal{C} \qquad t_1 \sim t_2', \Gamma; \Delta \vDash \mathcal{C} \qquad t_1 \prec t_2, \Gamma; \Delta \vDash \mathcal{C} \qquad t_1 \prec t_2', \Gamma; \Delta \vDash \mathcal{C}}{t_1 \prec t_2 t_2', \Gamma; \Delta \vDash \mathcal{C}}$$

$$\frac{}{t \prec \alpha, \Gamma; \Delta \vDash \mathcal{C}} \qquad \frac{}{t \prec s, \Gamma; \Delta \vDash \mathcal{C}}$$

TODO: explain what is $\{\mathcal{C}\} \cup \Delta$ Additional rule for ariving in contradictory Δ:

$$\frac{}{\Gamma; \lightning \vDash \mathcal{C}}$$

Define state of the solver by triple $(\Gamma, \Delta, \mathcal{C}_0)$ and such ordering of the states:

1. Number of distinct variables in Γ, Δ,$\mathcal{C}_0$.

2. Depth of $\mathcal{C}_0$.

3. Number of assumptions of given depth in Γ and Δ.

4. Number of assumptions of given depth in Γ.

Then by analysing each rule we can see the reductions always arrive in a smaller state.

3.1 Implementation

Environment Δ is a quintuple $(NeqAtoms_\Delta, Fresh_\Delta, VarShape_\Delta, Shape_\Delta, Subshape_\Delta)$ where:
NeqAtoms is a set of pairs of atoms that we know are different,
Fresh is a mapping from atoms to variables that we know the atom is Fresh in,
VarShape is a mapping from variables to shape-representative variables (i.e. all variables that are mapped in *VarShape* to the same variable are of the same shape),
Shape is a mapping from shape-representative variables to the shape that we know this variable must have,
SubShape is a mapping from shape-representative variables to sets of shapes that we know this variable must supershape.

We can now define a way to compute the shape-representative variable:

$$X_\Delta := \begin{cases} X & \text{if } VarShape_\Delta(X) = \emptyset \\ X'_\Delta & \text{if } VarShape_\Delta(X) = X' \end{cases}$$

And shape-reconstruction:

$$\begin{array}{rcl} |X|_\Delta & := & \begin{cases} |X'|_\Delta & \text{if } VarShape_\Delta(X) = X' \\ S & \text{if } Shape_\Delta(X) = S \\ X & \text{otherwise} \end{cases} \\ |\_|_\Delta & := & \_ \\ |\_.S|_\Delta & := & \_.|S|_\Delta \\ |S_1 S_2|_\Delta & := & |S_1|_\Delta |S_2|_\Delta \\ |s|_\Delta & := & s \\ |t|_\Delta & := & ||t||_\Delta \end{array}$$

Now we can easily check for irreducible constraints in Δ:

$$\begin{array}{rcl} (a_1 \neq a_2) \in \Delta & := & (a_1 \neq a_2) \in NeqAtoms_\Delta \\ (a \# X) \in \Delta & := & X \in Fresh_\Delta(a) \\ (X_1 \sim X_2) \in \Delta & := & |X_1|_\Delta = |X_2|_\Delta \\ (X \sim S) \in \Delta & := & S = Shape_\Delta(X_\Delta) \\ (S \prec X) \in \Delta & := & S \in SubShape_\Delta(X_\Delta) \end{array}$$

Now we can define rules for the special occurs check:

$$\frac{X_\Delta \text{ occurs syntactically in } |S|_\Delta}{\Delta \vDash X \text{ occurs in } S}$$

$$\frac{X'_\Delta \text{ occurs syntactically in } |S|_\Delta \qquad (S' \prec X') \in \Delta \qquad \Delta \vDash X \text{ occurs in } S'}{\Delta \vDash X \text{ occurs in } S}$$

And finally, the rules for $\mathcal{C} \cup \Delta$. Note that we are using the meta-field of *Assumptions* to indicate that some of the assumptions in Δ are no longer "simple" and escape from Δ back to Γ to be broken up by the *Solver*.

$$\{a \# X\} \cup \Delta := \Delta[Fresh(a) \mathrel{+}= X]$$

$$\{a \neq a'\} \cup \Delta := \begin{cases} \lightning & \text{if } a = a' \\ \Delta[NeqAtoms \mathrel{+}= (a \neq a')] & \text{otherwise.} \end{cases}$$

$$\begin{aligned} \{X \sim S\} \cup \Delta \quad &:= \begin{cases} \lightning & \text{if } \Delta \vDash X \text{ occurs in } S \\ \Delta' & \text{otherwise.} \end{cases} \\ \text{where } \Delta' \quad &= \quad \Delta.Symbols\{X_\Delta \rightsquigarrow |S|_\Delta\} \\ &\qquad .Subshapes\{X_\Delta \rightsquigarrow |S|_\Delta\} \\ &\qquad .Shape\{X_\Delta \rightsquigarrow |S|_\Delta\} \end{aligned}$$

$$\begin{aligned} \{X \sim X'\} \cup \Delta \quad &:= \begin{cases} \Delta & \text{if } X_\Delta = X'_\Delta \\ \Delta & \text{if } |X|_\Delta = |X'|_\Delta \\ \lightning & \text{if } X_\Delta \text{ occurs in } |X'|_\Delta \\ \lightning & \text{if } X'_\Delta \text{ occurs in } |X|_\Delta \\ \Delta' & \text{otherwise.} \end{cases} \\ \text{where } \Delta' \quad &= \quad \Delta.Symbols\{X_\Delta \rightsquigarrow X'_\Delta\} \\ &\qquad .Subshapes\{X_\Delta \rightsquigarrow X'_\Delta\} \\ &\qquad .TransferShape\{X_\Delta \rightsquigarrow X'_\Delta\} \\ &\qquad [\, Shape \mathrel{-}= (X_\Delta) \\ &\qquad , SubShape \mathrel{-}= (X_\Delta) \\ &\qquad , VarShape \mathrel{+}= (X_\Delta \mapsto X'_\Delta) \\ &\qquad] \end{aligned}$$

$$\Delta.Symbols\{X \rightsquigarrow S\} := \begin{cases} \Delta[Symbols \mathrel{-}= X, Assumptions \mathrel{+}= \text{symbol } S] & \text{if } X_\Delta \in \Delta.Symbols \\ \Delta & \text{otherwise.} \end{cases}$$

$$\Delta.Shape\{X \rightsquigarrow S\} := \begin{cases} \Delta[Assumptions \mathrel{+}= (S \sim S')] & \text{if } Shape_\Delta(X_\Delta) = S' \\ \Delta[Shapes \mathrel{+}= (X \mapsto S)] & \text{otherwise.} \end{cases}$$

$$\Delta.SubShapes\{X \rightsquigarrow S\} := \Delta[Assumptions \mathrel{+}= Subshapes_\Delta(X) \prec S]$$

$$\Delta.TransferShape\{X \rightsquigarrow X'\} := \begin{cases} \Delta.Shape\{termv' \rightsquigarrow S'\} & \text{if } Shape_\Delta(X_\Delta) = S \\ \Delta & \text{otherwise.} \end{cases}$$

$$\Delta\{X \mapsto t\} := \{X \sim |t|_\Delta\} \cup \Delta.Fresh\{X \mapsto t\}$$

$$\Delta.Fresh\{X \mapsto t\} := \Delta[Fresh.map(\text{fun } (a \,\#\, \mathbb{X}) \mapsto a \,\#\, (\mathbb{X} \backslash \{X\})] \cup \bigcup_{\substack{(a \,\#\, \mathbb{X}) \in Fresh_\Delta \\ X \in \mathbb{X}}} \{a \,\#\, t\}$$

$$\Delta\{a \mapsto a'\} := \Delta.Fresh\{a \mapsto a'\}.NeqAtoms\{a \mapsto a'\}]$$

$$\Delta.Fresh\{a \mapsto a'\} := \Delta[Fresh \mathrel{-}= a][Fresh \mathrel{+}= \{a' \,\#\, \Delta.Fresh(a)\}]$$

$$\Delta.NeqAtoms\{a \mapsto a'\} := \Delta[NeqAtoms = \emptyset] \cup \bigcup_{(a_1 \neq a_2) \in NeqAtoms_\Delta} \{a_1\{a \mapsto a'\} \neq a_2\{a \mapsto a'\}\}$$

Chapter 4

Higher Order Logic

On top of the sublogic of constraints, we build a higher-order logic. Due to the involvement of atoms, terms, binders, and constraints, we introduce kinds to ensure that the formulas we deal with *make sense.*

4.1 Kinds

$$\kappa \quad ::= \quad \star \mid \kappa \to \kappa \mid \forall_A a.\, \kappa \mid \forall_T X.\, \kappa \mid [c]\kappa \quad \text{(kinds)}$$

$\varphi :: \star$ — φ is a propositional formula.

$\varphi :: \kappa_1 \to \kappa_2$ — φ is function that takes a formula of kind κ_1,
and produces a formula of kind κ_2.

$\varphi :: \forall_A a.\, \kappa$ — φ is function that takes an an atom expression,
binds it to a and produces a formula of kind κ.

$\varphi :: \forall_T X.\, \kappa$ — φ is function that takes a term,
binds it to X and produces a formula of kind κ.

$\varphi :: [c]\kappa$ — φ is a formula of kind κ as long as c is satisfied.

Notice that as constraints occur in kinds, we cannot simply give functions from atoms some kind *Atom* $\to \kappa$, but we must know *which* atom is bound there, to substitute for it in κ the same way we substitute that atom for an atom expression in the function body when applying it to the formula. The *guarded kind* $[c]\kappa$ is most importantly used in kinding of the fixpoint formulas, which we will explain in later sections.

4.2 Subkinding

Kinding relation is relaxed through the *subkinding*, a relation that is naturally reflexive and transitive:

$$\frac{}{\Gamma \vdash \kappa <: \kappa} \qquad \frac{\Gamma \vdash \kappa_1 <: \kappa_2 \quad \Gamma \vdash \kappa_2 <: \kappa_3}{\Gamma \vdash \kappa_1 <: \kappa_3}$$

Universally quantified kinds only subkind if they are quantified over the same name:

$$\frac{\Gamma \vdash \kappa_1 <: \kappa_2}{\Gamma \vdash \forall_A a.\, \kappa_1 <: \forall_A a.\, \kappa_2} \qquad \frac{\Gamma \vdash \kappa_1 <: \kappa_2}{\Gamma \vdash \forall_T X.\, \kappa_1 <: \forall_T X.\, \kappa_2}$$

Function kind is contravariant to the subkinding relation on the left argument:

$$\frac{\Gamma \vdash \kappa_1' <: \kappa_1 \quad \Gamma \vdash \kappa_2 <: \kappa_2'}{\Gamma \vdash \kappa_1 \to \kappa_2 <: \kappa_1' \to \kappa_2'}$$

Constraints that are solved through $\vDash$ relation can be dropped:

$$\frac{\Gamma \vDash c}{\Gamma \vdash [c]\kappa <: \kappa}$$

And constraints can be moved to the enviroment from the right-hand side:

$$\frac{\Gamma, c \vdash \kappa_1 <: \kappa_2}{\Gamma \vdash \kappa_1 <: [c]\kappa_2}$$

Note that there is no structural subkinding rule for guarded kinds like

$$\frac{\Gamma \vdash \kappa_1 <: \kappa_2}{\Gamma \vdash [c]\kappa_1 <: [c]\kappa_2} \times$$

Such a rule can be derived from both subkinding rules for guarded kind, transitivity, and weakening.

4.3 Formulas

Formulas include standard connectives (of kind $\star$):

$$\varphi \quad ::= \quad \bot \mid \top \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \varphi \to \varphi \mid \ldots \quad \text{(formulas)}$$

Quantification over atoms and terms (on formulas of kind $\star$):

$$\varphi \quad ::= \quad \ldots \mid \forall_A a.\, \varphi \mid \forall_T X.\, \varphi \mid \exists_A a.\, \varphi \mid \exists_T X.\, \varphi \mid \ldots \quad \text{(formulas)}$$

Constraints, guards, and propositional variables:

$$\varphi \quad ::= \quad \ldots \mid c \mid [c] \wedge \varphi \mid [c] \to \varphi \mid P \mid \ldots \text{(formulas)}$$

$$\frac{}{\Gamma; \Sigma \vdash c :: \star} \qquad \frac{\Gamma, c; \Sigma \vdash \varphi :: \star}{\Gamma; \Sigma \vdash [c] \wedge \varphi :: \star} \qquad \frac{\Gamma, c; \Sigma \vdash \varphi :: \star}{\Gamma; \Sigma \vdash [c] \to \varphi :: \star} \qquad \frac{(P :: \kappa) \in \Sigma}{\Gamma; \Sigma \vdash P :: \kappa}$$

Propositional variables, functions and applications:

$$\varphi \quad ::= \quad \dots \mid \lambda_A a.\, \varphi \mid \lambda_T X.\, \varphi \mid \lambda P :: \kappa.\, \varphi \mid \varphi\ \alpha \mid \varphi\ t \mid \varphi\ \varphi \mid \dots \qquad \text{(formulas)}$$

$$\frac{\Gamma; \Sigma \vdash \varphi :: \kappa}{\Gamma; \Sigma \vdash \lambda_A a.\, \varphi :: \forall_A a.\, \kappa} \qquad \frac{\Gamma; \Sigma \vdash \varphi :: \kappa}{\Gamma; \Sigma \vdash \lambda_T X.\, \varphi :: \forall_T X.\, \kappa} \qquad \frac{\Gamma; \Sigma, P :: \kappa_1 \vdash \varphi :: \kappa_2}{\Gamma; \Sigma \vdash \lambda P :: \kappa_1.\, \varphi :: \kappa_1 \to \kappa_2}$$

$$\frac{\Gamma; \Sigma \vdash \varphi :: \forall_A a.\, \kappa}{\Gamma; \Sigma \vdash \varphi\ \alpha :: \kappa\{a \mapsto \alpha\}} \qquad \frac{\Gamma; \Sigma \vdash \varphi :: \forall_T X.\, \kappa}{\Gamma; \Sigma \vdash \varphi\ t :: \kappa\{X \mapsto t\}} \qquad \frac{\Gamma; \Sigma \vdash \varphi_1 :: \kappa' \to \kappa \qquad \Gamma; \Sigma \vdash \varphi_2 :: \kappa'}{\Gamma; \Sigma \vdash \varphi_1\ \varphi_2 :: \kappa}$$

4.4 Fixpoint

And finish the definition of formulas with *fixpoint* function:

$$\varphi \quad ::= \quad \dots \mid \text{fix } P(X) :: \kappa = \varphi \qquad \text{(formulas)}$$

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

The fixpoint constructor allows us to express *recursive* predicates over terms, but only such that the recursive applications are on structurally smaller terms, which we express in the kinding rule through the kinding $(P :: \forall_T Y.\, [Y \prec X]\kappa\{X \mapsto Y\})$. To evaluate a fixpoint function applied to a term, simply substitute the bound variable with the given term and replace recursive calls inside the fixpoint's body with the fixpoint itself.

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

Because the applied term is finite and we always recurse on structurally smaller terms, the final formula after all substitutions must also be finite —– thanks to the semantics of constraints and kinds.

To familiarize the reader with the fixpoint formulas, we present how Peano arithmetic can be modeled in our logic. Given symbols 0 and S for natural number construction, one can write a predicate that a term models some natural number:

$$\text{fix } Nat(N) :: \star = (N = 0) \vee (\exists_T M.\, [N = S\ M] \wedge (Nat\ M))$$

Notice how the constraint $(N = S\ M)$ guards the recursive call to Nat, ensuring that constraint $(M \prec N)$ will be satisfied during kind checking of $(Nat\ M)$ in the kind derivation of the whole formula $(Nat :: \forall_T N.\, \star)$.

Similarly, we can define addition:

$$\begin{aligned}&\text{fix } PlusEq(N) :: \forall_T M.\, \forall_T K.\, \star = \lambda_T M.\, \lambda_T K.\\ &\qquad ([N = 0] \wedge (M = K)) \vee (\exists_T N', K'.[N = S\ N'] \wedge [K = S\ K'] \wedge (PlusEq\ N'\ M\ K'))\end{aligned}$$

TODO: Write how N is treated differently from M and K?
See more interesting examples of fixpoints usage in the chapter on STLC.

4.5 Proof theory

Finally, we can define proof-theoretic rules. Starting with inference rules for assumption, we can already define its constraint-sublogic analogues that employ the solver. And while the $\vdash$ relation we define is purely syntactic, we can still use semantic $\vDash$ because of its decidability.

$$\frac{\varphi \in \Theta}{\Gamma; \Theta \vdash \varphi}\ (\mathit{Assumption}) \qquad \frac{\Gamma \vDash c}{\Gamma; \Theta \vdash c}\ (\mathit{constr}^i)$$

Again, for *ex falso*, we define an analogous proof constructor for dealing with a contradictory constraint environment. Note that there are many constraints that can be used as $\bot_c$, i.e. constraints that are always false, and the solver will only *prove* them if we supply it with contradictory assumptions.

$$\frac{\Gamma; \Theta \vdash \bot}{\Gamma; \Theta \vdash \varphi}\ (\bot^e) \qquad \frac{\Gamma \vDash \bot_c}{\Gamma; \Theta \vdash \varphi}\ (\mathit{constr}^e)$$

Inference rules for implication are standard, and the reason we present them here is not to bore the reader, but to point out the similarities to their constraint analogues.

$$\frac{\Gamma; \Theta, \varphi_1 \vdash \varphi_2}{\Gamma; \Theta \vdash \varphi_1 \to \varphi_2}\ (\to^i) \qquad \frac{\Gamma_1; \Theta_1 \vdash \varphi_1 \quad \Gamma_2; \Theta_2 \vdash \varphi_1 \to \varphi_2}{\Gamma_1 \cup \Gamma_2; \Theta_2 \cup \Theta_2 \vdash \varphi_2}\ (\to^e)$$

$$\frac{\Gamma, c; \Theta \vdash \varphi}{\Gamma; \Theta \vdash [c] \to \varphi}\ ([\cdot] \to^i) \qquad \frac{\Gamma_1; \Theta_1 \vdash c \quad \Gamma_2; \Theta_2 \vdash [c] \to \varphi}{\Gamma_1 \cup \Gamma_2; \Theta_2 \cup \Theta_2 \vdash \varphi}\ ([\cdot] \to^e)$$

Notice that in the case of constraint-and-guard, the rule for elimination is restricted to only formulas of kind $\star$. This is due to the nature of the guard — if we want to eliminate it, we can only do so with formulas that *make sense* on their own, without that c guard.

$$\frac{\Gamma_1; \Theta_1 \vdash \varphi_1 \quad \Gamma_2; \Theta_2 \vdash \varphi_2}{\Gamma_1 \cup \Gamma_2; \Theta_2 \cup \Theta_2 \vdash \varphi_1 \wedge \varphi_2}\ (\wedge^i) \qquad \frac{\Gamma; \Theta \vdash \varphi_1 \wedge \varphi_2}{\Gamma; \Theta \vdash \varphi_1}\ (\wedge^e_1) \qquad \frac{\Gamma; \Theta \vdash \varphi_1 \wedge \varphi_2}{\Gamma; \Theta \vdash \varphi_2}\ (\wedge^e_2)$$

$$\frac{\Gamma \vDash c \quad \Gamma, c; \Theta \vdash \varphi}{\Gamma; \Theta \vdash [c] \wedge \varphi}\ ([\cdot]\wedge^i) \qquad \frac{\Gamma; \Theta \vdash [c] \wedge \varphi}{\Gamma; \Theta \vdash c}\ ([\cdot]\wedge^e_1) \qquad \frac{\Gamma \vdash [c] \wedge \varphi \quad \Gamma; \Theta \vdash \varphi : \star}{\Gamma; \Theta \vdash \varphi}\ ([\cdot]\wedge^e_2)$$

Inference rules for disjunction and quantifiers are rather straightforward. As one would expect, we restrict the generalized name to be *fresh* in the environment (it may not occur in any of the assumptions), and the names given to witnesses of existential quantification must also be *fresh.* Rules for quantifiers always come in pairs — one for the atoms and one for the variables.

$$\frac{\Gamma; \Theta \vdash \varphi_1}{\Gamma; \Theta \vdash \varphi_1 \vee \varphi_2}\ (\vee^i_1) \qquad \frac{\Gamma; \Theta \vdash \varphi_2}{\Gamma; \Theta \vdash \varphi_1 \vee \varphi_2}\ (\vee^i_2) \qquad \frac{\begin{array}{c}\Gamma; \Theta \vdash \varphi_1 \vee \varphi_2 \\ \Gamma; \Theta, \varphi_1 \vdash \psi \quad \Gamma; \Theta, \varphi_2 \vdash \psi\end{array}}{\Gamma; \Theta \vdash \psi}\ (\vee^e)$$

$$\frac{a \notin \mathrm{FV}(\Gamma; \Theta) \quad \Gamma; \Theta \vdash \varphi}{\Gamma; \Theta \vdash \forall_A a.\ \varphi}\ (\forall_A.\,^i) \qquad \frac{\Gamma; \Theta \vdash \forall_A a.\ \varphi}{\Gamma; \Theta \vdash \varphi\{a \mapsto a'\}}\ (\forall_A.\,^e)$$

$$\frac{X \notin \mathrm{FV}(\Gamma; \Theta) \quad \Gamma; \Theta \vdash \varphi}{\Gamma; \Theta \vdash \forall_T X.\ \varphi}\ (\forall_T.\,^i) \qquad \frac{\Gamma; \Theta \vdash \forall_T X.\ \varphi}{\Gamma; \Theta \vdash \varphi\{X \mapsto X'\}}\ (\forall_T.\,^e)$$

$$\frac{\Gamma; \Theta \vdash \varphi\{a \mapsto a'\}}{\Gamma; \Theta \vdash \exists_A a.\, \varphi}\ (\exists_A.^{\,i}) \qquad \frac{\begin{array}{c}\Gamma_1; \Theta_1 \vdash \exists_A a.\, \varphi \\ \Gamma_2; \Theta_2, \varphi\{a \mapsto a'\} \vdash \psi \\ a' \notin \mathrm{FV}(\Gamma_1 \cup \Gamma_2; \Theta_2 \cup \Theta_2)\end{array}}{\Gamma_1 \cup \Gamma_2; \Theta_2 \cup \Theta_2 \vdash \psi}\ (\exists_A.^{\,e})$$

$$\frac{\Gamma; \Theta \vdash \varphi\{X \mapsto X'\}}{\Gamma; \Theta \vdash \exists_T X.\, \varphi}\ (\exists_T.^{\,i}) \qquad \frac{\begin{array}{c}\Gamma_1; \Theta_1 \vdash \exists_T X.\, \varphi \\ \Gamma_2; \Theta_2, \varphi\{X \mapsto X'\} \vdash \psi \\ X' \notin \mathrm{FV}(\Gamma_1 \cup \Gamma_2; \Theta_2 \cup \Theta_2)\end{array}}{\Gamma_1 \cup \Gamma_2; \Theta_2 \cup \Theta_2 \vdash \psi}\ (\exists_T.^{\,e})$$

To make the framework more flexible we introduce a way for using equivalent formulas:

$$\frac{\Gamma; \Theta \vdash \psi \qquad \Gamma; \Theta \vdash \psi \equiv \varphi}{\Gamma; \Theta \vdash \varphi}\ (\mathit{Equiv})$$

And a way to substitute atoms for atomic expression and variables for terms, if the solver can prove their equality:

$$\frac{\Gamma \vDash a = \alpha \qquad \Gamma; \Theta \vdash \varphi}{\Gamma\{a \mapsto \alpha\}; \Theta\{a \mapsto \alpha\} \vdash \varphi\{a \mapsto \alpha\}}\ (\mapsto_A) \qquad \frac{\Gamma \vDash X = t \qquad \Gamma; \Theta \vdash \varphi}{\Gamma\{X \mapsto t\}; \Theta\{X \mapsto t\} \vdash \varphi\{X \mapsto t\}}\ (\mapsto_T)$$

Finall we define induction over term structure, and thanks to the constraints sublogic we can easily define the notion of *smaller terms* for inductive hypothesis:

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

We also define some axioms about constraint sublogic:

1. Atoms can be compared in a deterministic fashion,

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

2. There are always exists a *fresh* atom,

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

3. We can deduce the structure of a term.

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

The equivalence relation ($\varphi_1 \equiv \varphi_2$) is a bit complicated due to the presence of an environment with variable mapping, subkinding, and formulas with fixpoints, functions, and applications. Nonetheless, it's simply that - *an equivalence relation* - and it behaves as expected. We will only highlight the interesting parts.

$$\frac{\Gamma; \Sigma \vdash \varphi_1[X_1 \mapsto t_1] \equiv \varphi_2[X_2 \mapsto t_2]}{\Gamma; \Sigma \vdash (\lambda_T X_1.\, \varphi_1)\ t_1 \equiv (\lambda_T X_2.\, \varphi_2)\ t_2}$$

Otherwise we compute weak head normal form (up to some *depth*) and recurse on subformulas:

$$\frac{\Gamma; \Sigma; Sn \vdash (\text{fix } P(X) :: \kappa = \varphi)\ t}{\Gamma; \Sigma, P \mapsto \varphi; n \vdash \varphi[X \mapsto t]}$$

Until we reach WHNF computation *depth* or cannot compute the formula further, we resort to *naive* checking:

$$\frac{\Gamma \vDash t_1 = t_2 \qquad \Gamma; \Sigma \vdash \varphi_1 \equiv \varphi_2}{\Gamma; \Sigma \vdash \varphi_1\ t_1 \equiv \varphi_2\ t_2}$$

$$\frac{\begin{array}{c} X \notin \mathrm{FV}(\Gamma; \Sigma) \\ \Gamma; \Sigma \vdash \varphi_1[X_1 \mapsto X] \equiv \varphi_2[X_2 \mapsto X] \end{array}}{\Gamma; \Sigma \vdash \lambda_T X_1.\, \varphi_1 \equiv \lambda_T X_2.\, \varphi_2}$$

$$\frac{\begin{array}{c} \kappa_1 <: \kappa_2 \\ \Gamma; \Sigma \vdash \varphi_1[P_1 \mapsto P] \equiv \varphi_2[P_2 \mapsto P] \end{array}}{\Gamma; \Sigma \vdash \lambda P_1 :: \kappa_1.\, \varphi_1 \equiv \lambda P_2 :: \kappa_2.\, \varphi_2}$$

$$\frac{\begin{array}{c} \kappa_1 <: \kappa_2 \qquad P \notin \mathrm{FV}(\Gamma; \Sigma) \qquad X \notin \mathrm{FV}(\Gamma; \Sigma) \\ \Gamma; \Sigma \vdash \varphi_1[P_1 \mapsto P, X_1 \mapsto X] \equiv \varphi_2[P_2 \mapsto P, X_2 \mapsto X] \end{array}}{\Gamma; \Sigma \vdash \text{fix } P_1(X_1) :: \kappa_1 = \varphi_1 \equiv \text{fix } P_2(X_2) :: \kappa_2 = \varphi_2}$$

Note that we allow *different terms* in equivalent formulas as long as constraints-enviroment Γ ensures their equality is provable. Quantifiers are handled the same way as function above — as they all a form of bind.

$$\frac{\Gamma \vdash c_1 \equiv c_2 \qquad \Gamma; \Sigma \vdash \varphi_1 \equiv \varphi_2}{\Gamma; \Sigma \vdash [c_1] \wedge \varphi_1 \equiv [c_2] \wedge \varphi_2} \qquad \frac{\Gamma \vDash a_1 = a_2 \qquad \Gamma \vDash t_1 = t_2}{\Gamma \vdash (a_1 \# t_1) \equiv (a_2 \# t_2)}$$

To handle formulas with constraints we introduce constraint equivalence relation, which does nothing more than use solver to check that arguments and constructors of constraint are equal in the solver sense.

Chapter 5

Proof assistant

All the stuff mentioned above has their implementation in OCaml, in modules `Solver` and `SolverEnv`, `KindChecker` and `KindCheckerEnv`, `Proof` and `ProofEnv`, respectively.

Constraints, kinds, and formulas constructors mirror the grammars we defined in previous chapters. Atoms and variables are represented internally by integers (but still are disjoint sets) — and their string *names* are kept in the environment and binders (quantifiers and functions).

Additionally, we provide a *proof assistant* (in module `Prover`), that enables the user to conveniently work with *backwards* and incomplete proof — inspired by the HOL family of theorem provers. While simple, it is also powerful and easy to use.

The interface to the Prover provides multiple *tactics* (functions that manipulate *prover state*) and ways to combine them:

```
type goal_env = (string * formula) ProofEnv.env

type goal = goal_env * formula

type prover_state = S_Unfinished of {goal: goal; context: proof_context}
                  | S_Finished of proof

type tactic = prover_state -> prover_state

val proof : goal_env -> formula -> prover_state

val qed : prover_state -> proof

val (|>) : prover_state -> tactic -> prover_state

val (%>) : tactic -> tactic -> tactic

val repeat : tactic -> tactic

val try_tactic : tactic -> tactic
```

We will use $\bullet$ to indicate *holes* in the incomplete proofs:

$$\begin{array}{rcl}
\texttt{intro} & & \\
\Gamma;\Theta;\Sigma \vdash \bullet :: [c] \to \varphi & \rightsquigarrow & \Gamma, c;\Theta;\Sigma \vdash \bullet :: \varphi
\end{array}$$

$$\begin{array}{rcl}
\texttt{intro' "x"} & & \\
\Gamma;\Theta;\Sigma \vdash \bullet :: \psi \to \varphi & \rightsquigarrow & \Gamma;\Theta, \mathsf{x} :: \psi;\Sigma \vdash \bullet :: \varphi \\
\Gamma;\Theta;\Sigma \vdash \bullet :: \forall_A a.\, \varphi & \rightsquigarrow & \Gamma;\Theta;\Sigma, \mathsf{x} :: a \vdash \bullet :: \varphi \\
\Gamma;\Theta;\Sigma \vdash \bullet :: \forall_T X.\, \varphi & \rightsquigarrow & \Gamma;\Theta;\Sigma, \mathsf{x} :: X \vdash \bullet :: \varphi
\end{array}$$

$$\begin{array}{rcl}
\texttt{apply\_assm "H"} & & \\
\Gamma;\Theta;\Sigma \vdash \bullet :: \varphi & \rightsquigarrow & \Gamma;\Theta;\Sigma \vdash \bullet :: \psi \\
 & \text{when} & (\mathsf{H} :: \psi) \in \Theta
\end{array}$$

$$\begin{array}{rcl}
\texttt{apply } (\psi \to \varphi) & & \\
\Gamma;\Theta;\Sigma \vdash \bullet :: \varphi & \rightsquigarrow & \Gamma;\Theta;\Sigma \vdash \bullet :: \psi \\
 & & \Gamma;\Theta;\Sigma \vdash \bullet :: \psi \to \varphi
\end{array}$$

$$\begin{array}{rcl}
\texttt{ex\_falso} & & \\
\Gamma;\Theta;\Sigma \vdash \bullet :: \varphi & \rightsquigarrow & \Gamma;\Theta;\Sigma \vdash \bullet :: \bot
\end{array}$$

$$\begin{array}{rcl}
\texttt{by\_solver} & & \\
\Gamma;\Theta;\Sigma \vdash \bullet :: c & \rightsquigarrow & \Gamma;\Theta;\Sigma \vdash c \\
 & \text{when} & \Gamma \vDash c
\end{array}$$

$$\begin{array}{rcl}
\texttt{compare\_atoms "a" "b"} & & \\
\Gamma;\Theta;\Sigma \vdash \bullet :: \varphi & \rightsquigarrow & \Gamma;\Theta;\Sigma \vdash \bullet :: (a = a' \vee a \neq a') \to \varphi \\
 & \text{when} & (\mathsf{a} :: a) \in \Sigma \text{ and } (\mathsf{b} :: a') \in \Sigma
\end{array}$$

Chapter 6

Case study: Progress and Preservation of STLC

The ultimate goal of our work is to create a logic for dealing with variable binding, and there's no better way to do that than to prove some things about lambda calculus.

We will take a look at simply typed lambda calculus and examine proofs of its two major properties of *type soundness*: *progress* and *preservation*. Before we delve into the proofs, let's first establish the needed relations:

$$\begin{array}{rc}
\text{fix } \mathit{Type}(t) :: \star = & (t = \mathit{base}) \\
\vee & (\exists_T t_1, t_2.\, [X = \mathit{arrow}\ t_1\ t_2] \wedge (\mathit{Type}\ T_1) \wedge (\mathit{Type}\ t_2 \\
\text{fix } \mathit{InEnv}(\mathit{env}) :: \forall_A a.\, \forall_T t.\, \star = \lambda_A a.\, \lambda_T t. & (\exists_T \mathit{env}'.\, \mathit{env} = \mathit{cons}\ a\ t\ \mathit{env}') \\
\vee & (\exists_A b.\, \exists_T t', \mathit{env}'.\, [\mathit{env} = \mathit{cons}\ b\ t'\ \mathit{env}'] \wedge [a \neq b] \wedge (\mathit{InEnv} \\
\ldots & \ldots
\end{array}$$

As one would expect, we will need a lemma about *canonical forms*, which states that all values are of *arrow* type and can be *inversed* into an abstraction term (since we did not consider any true base types like `Bool` or `Int`). Other lemmas are unimportant boilerplate.

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

let typing_terms_thm = lambda_thm
    "forall e env t : term. (Typing e env t) => (Term e)"

let canonical_form_thm = lambda_thm $ concat
      [ "forall v t :term."
      ; " (Value v) =>"
      ; " (Typing v nil t) =>"
      ; " (exists a :atom. exists e :term. [v = lam (a.e)] /\ (Term e))"
      ; " /\ "
      ; " (exists t1 t2 :term. [t = arrow t1 t2])" ]

let subst_exists_thm = lambda_thm $ unwords
      [ "forall a :atom."
      ; "forall v :term. (Value v) =>"
      ; "forall e :term. (Term e) =>"
      ; "exists e’ :term. (Sub e a v e’)" ]

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

Otherwise the proof goes the same way as usual, simple induction over *Typing*.

```
let progress =
  proof’ progress_thm
  |> by_induction "e0" "IH" %> intro %> destr_intro
  |> intros’ ["Ha"; "a"; ""]
    (* e is a var in empty env - contradiction *)
     %> ex_falso
     %> apply_thm_specialized empty_contradiction ["a"; "t"]
     %> assumption
  |> intros’ ["Hlam"; "a"; "e_a"; "t1"; "t2"; ""; ""; ""]
    (* e is a lambda - value *)
     %> case "value"
     %> case "lam"
     %> exists’ ["a"; "e_a"]
     %> by_solver
     %> apply_thm_specialized typing_terms ["e_a"; "cons a t1 nil"; "t2"]
     %> assumption
  |> intros’ ["Happ"; "e1"; "e2"; "t2"; ""; ""]
  (* e is an application - steps *)
  |> add_assumption_parse "He1" "Progressive e1"
  |> add_assumption_parse "He2" "Progressive e2"
  |> destruct_assm "He1"
     %> intros ["Hv1"]
     %> destruct_assm "He2"
     %> intros ["Hv2"] (* Value e1, Value e2 *)
     %> ( add_assumption_thm_specialized "He1lam" canonical_form’ ["e1";
    "t2"; "t"]
        (* He1lam: [e1 = lam (a.e_a)] /\ (Term e_a) *)
        %> apply_in_assm "He1lam" "Hv1"
        %> apply_in_assm "He1lam" "Happ_1"
        %> destruct_assm’ "He1lam" ["a"; "e_a"; ""] )
     %> ( add_assumption_thm_specialized "He_a" subst_exists ["a"; "e2";
    "e_a"]
        %> apply_in_assm "He_a" "Hv2"
        %> apply_in_assm "He_a" "He1lam"
        %> destruct_assm’ "He_a" ["e_a’"] (* He_a: Sub e_a a e2 e_a’ *) )
     %> case "steps"
     %> exists "e_a’"
     %> case "app"
     %> exists’ ["a"; "e_a"; "e2"]
     %> by_solver
     %> destruct_goal
     %> apply_assm "Hv2"
     %> apply_assm "He_a"
  |> intros’ ["Hs2"; "e2’"] (* Value e1, Steps e2 e2’ *)
     %> case "steps"
     %> exists "app e1 e2’"
     %> case "app_r"
     %> exists’ ["e1"; "e2"; "e2’"]
     %> by_solver
     %> by_solver
     %> destruct_goal
     %> apply_assm "Hv1"
     %> apply_assm "Hs2"
  |> intros’ ["Hs1"; "e1’"] (* Steps e1 *)
     %> case "steps"
     %> exists "app e1’ e2"
     %> case "app_l"
     %> exists’ ["e1"; "e1’"; "e2"]
     %> by_solver
     %> by_solver
     %> apply_assm "Hs1"
  |> apply_assm_specialized "IH" ["e2"; "t2"] %> by_solver
    %> apply_assm "Happ_2" (* Progressive e2 *)
  |> apply_assm_specialized "IH" ["e1"; "arrow t2 t"] %> by_solver
    %> apply_assm "Happ_1" (* Progressive e1 *)
  |> qed
```

To prove *Preservation*, we will need more lemmas:

1. Substitution lemma: if term e has a type t in enviroment *cons a ta env*, then we can substitute a for any value v of type *ta* in e without breaking the typing.

```
let sub_lemma_thm = lambda_thm $ concat
      [ "forall e env t :term."
      ; "forall a : atom. forall ta :term."
      ; "forall v e' :term."
      ; " (Typing v env ta) =>"
      ; " (Typing e {cons a ta env} t) =>"
      ; " (Sub e a v e') =>"
      ; " (Typing e' env t)" ]
```

2. Swap lemma: If we have a typing of e in *env* then we can swap a with (*fresh enough*) b in both e and *env* without breaking the typing. This is particularly useful for mainipulating the abstraction terms — we can have any atom we want in the argument position while preserving typing.

```
let swap_lambda_typing_lemma_thm = lambda_thm $ unwords
      [ "forall e env t :term. "
      ; "forall a b :atom. forall t' :term. "
      ; " [b # a e] => "
      ; " (Typing {e} {cons a t' env} t) => "
      ; " (Typing {[a b]e} {cons b t' env} t)" ]
```

3. Weakening lemma: for any enviroment env_1, we can use larger enviroment env_2 without breaking the typing.

```
let weakening_lemma_thm = lambda_thm $ concat
      [ "forall e env1 t env2 : term."
      ; " (Typing e env1 t) =>"
      ; " (EnvInclusion env1 env2) =>"
      ; " (Typing e env2 t)" ]
```

Now to the proof:

```
let preservation =
  let contra_var = intros' ["contra"; "_"; ""] %> discriminate in
  let contra_app = intros' ["contra"; "_e1"; "_e2"; "_t2"; ""] %>
    discriminate in
  let deduce_app_typing =
    destruct_assm "Htyp"
    %> (intros' ["contra"; "_"; ""] %> discriminate)
    %> (intros' ["contra"; "_"; "e_"; "t1"; "t2"; ""] %> discriminate)
    %> intros' ["Happ"; "e_1"; "e_2"; "t2"; ""; ""]
  in
  proof' preservation_thm
  |> by_induction "e0" "IH"
  |> intro %> intro %> intro %> intros ["Htyp"; "Hstep"]
  |> destruct_assm "Hstep"
  |> intros' ["He1"; "e1"; "e1'"; "e2"; ""; ""]
    (* e = app e1 e2, Steps e1 e1' *)
     %> deduce_app_typing
```

```
   %> case "app"
   %> exists' ["e1'"; "e2"; "t2"]
   %> by_solver
   %> destruct_goal
   %> apply_assm_specialized "IH" ["e1"; "e1'"; "env"; "arrow t2 t"]
   (* Typing e1 env {arrow t2 t} =>
          Steps e1 e1' => Typing e1' env {arrow t2 t} *)
   %> by_solver
   %> apply_assm "Happ_1"
   %> apply_assm "He1"
   (* Typing e2 env t2 *)
   %> apply_assm "Happ_2"
|> intros' ["He2"; "v1"; "e2"; "e2'"; ""; ""; ""]
   (* e = app v1 e2, Value e1, Steps e2 e2' *)
   %> deduce_app_typing
   %> case "app"
   %> exists' ["v1"; "e2'"; "t2"]
   %> by_solver
   %> destruct_goal
   (* Typing e1 env {arrow t2 t} *)
   %> apply_assm "Happ_1"
   (* Typing e2 env t2 => Steps e2 e2' => Typing e2' env t2*)
   %> apply_assm_specialized "IH" ["e2"; "e2'"; "env"; "t2"]
   %> by_solver
   %> apply_assm "Happ_2"
   %> apply_assm "He2_2"
|> intros' ["Hbeta"; "a"; "e_a"; "v"; ""; ""]
   (* e = app (lam (a.e_a)) v, Value v *)
   %> deduce_app_typing
   %> destruct_assm "Happ_1"
   %> contra_var (* e_1 =/= var *)
   %> intros' ["Hlam"; "b"; "e_b"; "t1b"; "t2b"; ""; ""; ""] (* e_1 = b
 .e_b *)
   %> apply_thm_specialized sub_lemma ["e_a"; "env"; "t"; "a"; "t2"; "v
 "; "e'"]
   (* Typing v env t2 => Typing e_a {cons a t2 env} t =>
          Sub e_a a v e' => Typing e' env t *)
   %> apply_assm "Happ_2" (* Typing v env t2 *)
   %> compare_atoms "a" "b" (* Typing e_a cons a t2 env t *)
   %> destr_intro
   (* a = b *) %> apply_assm "Hlam_2"
   %> destr_intro (* a =/= b *)
   (* [a # b e_b] => Typing e_b {cons b t2 env} t =>
          Typing {[b a]e_b} {cons a t2 env} t *)
   %> apply_thm_specialized
      swap_lambda_typing ["e_b"; "env"; "t"; "b"; "a"; "t2"]
   %> by_solver
   %> apply_assm "Hlam_2"
   %> apply_assm "Hbeta_2" (* Sub e_a a v e' *)
   %> contra_app (* e_1 =/= app _ _ *)
|> qed
```

Chapter 7

Conclusion and future work

…

Bibliography

[1] . . .