# 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

3 września 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:

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

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 $\alpha$ 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 $\Gamma$ 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 $\rho$ 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 $c_1, \dots, c_n$, it verifies whether a given goal $c_0$ holds. Technically, the Solver determines whether, every possible substitution of variables into closed terms in $c_0, c_1, \dots, c_n$, such that $c_1, \dots, c_n$ 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:

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

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

We add another environment $\Delta$ to distinguish between the potentially-reducible assumptions in $\Gamma$. 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:

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

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:

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

And now for the solving procedure we start with the most simple equality check:

$$\dfrac{}{\Gamma; \Delta \vDash a = a} \qquad \dfrac{}{\Gamma; \Delta \vDash X = X} \qquad \dfrac{}{\Gamma; \Delta \vDash s = s} \qquad \dfrac{\Gamma; \Delta \vDash t_1 = t_2 \qquad \Gamma; \Delta \vDash t_1' = t_2'}{\Gamma; \Delta \vDash t_1 t_1' = t_2 t_2'}$$

Checking equality of abstraction terms requires that the left side's argument is fresh in the whole right side's term (either arguments are the same or left's argument doesn't occur in right's body) and that left body is equal to the right body with right argument swapped for the left one:

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

To compare a *pure* atom with permuted one, we employ the decidability of atom equality to strip the right hand-side permutation through applying the outermost swap on the left side, while adding to assumption. There's three possible ways:

1. $a$ is different from both $\alpha_1$ and $\alpha_2$, so the swap doesn't change the goal,
2. $a$ is equal to $\alpha_1$ but different from $\alpha_2$, so the swap substitutes it for $\alpha_2$,
3. $a$ is equal to $\alpha_2$, so the swap substitutes it for $\alpha_1$.

Notice that it is impossible for any two of these assumption to be valid at the same time — the contradictory branches will resolve through absurd environment.

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

If the left-hand side's term is permuted we simply move the permutation to the right-hand side:

$$\dfrac{\Gamma; \Delta \vDash a = \pi^{-1}\alpha}{\Gamma; \Delta \vDash \pi a = \alpha} \qquad \dfrac{\Gamma; \Delta \vDash X_1 = \pi_1^{-1}\pi_2 X_2}{\Gamma; \Delta \vDash \pi_1 X_1 = \pi_2 X_2}$$

Variables can be equal to their permuted selves if that permutation is idempotent:

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

Freshness is checked through the $\Delta$ environment and freshness in symbols is trivial:

$$\frac{a_1 \neq a_2 \in \Delta}{\Gamma; \Delta \vDash a_1 \# a_2} \qquad \frac{a \# X \in \Delta}{\Gamma; \Delta \vDash a \# X} \qquad \frac{}{\Gamma; \Delta \vDash a \# s}$$

Similarly we recurse on the term structure, assuming checked atom is different than abstraction argument — otherwise it would be trivially true:

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

Again when faced with swap on the right side, we apply it on the left side:

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

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

All atoms have the same shape, while only equal symbols have equal shape:

$$\frac{}{\Gamma; \Delta \vDash \_ \sim \_} \qquad \frac{}{\Gamma; \Delta \vDash s \sim s}$$

Variables can share shape and be shape-substituted through $\Delta$:

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

Shape equality is naturally structural:

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

Solving subshape recurses through right-hand side shape's structure to find a shape-equal sub-shape:

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

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

Environment $\Delta$ keeps track of all shapes that given variable subshapes:

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

And that finishes solving rules that recurse on the goal. In mathematical jargon, the Solver must first reduce all assumptions in the $\Gamma$ environment before it starts reducing the goal. Luckily, most of the assumption reducing rules are similar to the goal reducing analogues.

For variables equal to some term, we first deal with permutation by moving it to the right-hand side.

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

Once again, we consider the special case where a variable is equal to itself when permuted. While the assumption of the permutation being idempotent might appear to multiply the number of assumptions exponentially based on the number of atoms in the given permutation, it's worth noting that this number is unlikely to be very high, as permutations rarely consist of more than a few swaps.

In practice, the solver implementation will initially check whether the permutation is idempotent with an empty set of assumptions. Only if this initial check fails, will it proceed to examine the permutation atom by atom.

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

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

Otherwise we just substitute the variable for the equal term, and while substitution over the environment $\Gamma$ and goal $\mathcal{C}$ is indeed a simple term substitution, substituting in $\Delta$ is a more involved process that we will describe in the section on implementation.

$$\frac{\Gamma\{X \mapsto t\}; \Delta\{X \mapsto t\} \vDash \mathcal{C}\{X \mapsto t\}}{X = t, \Gamma; \Delta \vDash \mathcal{C}}$$

With atom equality, we either arrive at a contradiction with $\Delta$ or update the environment accordingly — merging the now equal atoms into one through substitution:

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

Just like in reduction on the goal, we deal with permutations through moving it to the right-hand side and then reducing it swap by swap through the left-hand side:

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

If the constructors of the term don't match, then we arrive at a contradiction and consider the judgement solved:

$$\frac{}{a = t_1 t_2, \Gamma; \Delta \vDash \mathcal{C}} \qquad \frac{}{a = \alpha.t, \Gamma; \Delta \vDash \mathcal{C}} \qquad \frac{}{a = s, \Gamma; \Delta \vDash \mathcal{C}}$$

To save some ink, from now on we will simply write that other constructors are trivial and not consider all the contradictory possibilities in writing. Other rules mirror the ones we defined for the goal reduction:

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

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

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

Atom inequality and freshness in variable simply contradict or extend the $\Delta$ enviroment:

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

Otherwise it's a recursion on the right-hand side with the already established rules for dealing with permutations:

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

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

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

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

Variable being the same shape as other term is added to the $\Delta$ environment:

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

Otherwise shape assumptions recurse on the shape structure:

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

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

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

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

Again, $\Delta$ keeps track of terms that subshape given variable:

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

Otherwise subshape assumptions recurse on the shape structure:

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

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

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

In the next section we will explaining the semantics of environment extension ($\{\mathcal{C}\} \cup \Delta$), which can fail by arriving at contradictory environment $\lightning$, which short-cuircuts the procedure:

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

And that finishes the Solver's rules description. Now the curious reader should feel obliged to ask themselves an important question: does that procedure always stop?

To answer that question, we define the Solver's state by triple $(\Gamma, \Delta, \mathcal{C})$ and introduce an ordering of the states by:

1. Number of distinct variables in $\Gamma$, $\Delta$, and $\mathcal{C}$.
2. Depth of $\mathcal{C}$.
3. Number of assumptions of given depth in both $\Gamma$ and $\Delta$.
4. Number of assumptions of given depth in $\Gamma$.

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

## 3.1 Implementation

Environment $\Delta$ 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:

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

Now we can easily check for irreducible constraints in $\Delta$:

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

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 $\Delta$ are no longer "simple" and escape from $\Delta$ back to $\Gamma$ to be broken up by the *Solver*.

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

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

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

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

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

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

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

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

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

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

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

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

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

# 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

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

| | | |
|---|---|---|
| $\varphi ::$ | $\star$ | — $\varphi$ is a propositional formula. |
| $\varphi ::$ | $\kappa_1 \to \kappa_2$ | — $\varphi$ is function that takes a formula of kind $\kappa_1$,<br>and produces a formula of kind $\kappa_2$. |
| $\varphi ::$ | $\forall_A a.\, \kappa$ | — $\varphi$ is function that takes an an atom expression,<br>binds it to $a$ and produces a formula of kind $\kappa$. |
| $\varphi ::$ | $\forall_T X.\, \kappa$ | — $\varphi$ is function that takes a term,<br>binds it to $X$ and produces a formula of kind $\kappa$. |
| $\varphi ::$ | $[c]\kappa$ | — $\varphi$ is a formula of kind $\kappa$ 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 $\kappa$ 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 $\star$):

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

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

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

Constraints, guards, and propositional variables:

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

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

Propositional variables, functions and applications:

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

$$\frac{\Gamma; \Sigma \vdash \varphi :: \kappa}{\Gamma; \Sigma \vdash \lambda_A a.\, \varphi :: \forall_A a.\, \kappa} \qquad \frac{\Gamma; \Sigma \vdash \varphi :: \forall_A a.\, \kappa}{\Gamma; \Sigma \vdash \varphi\ \alpha :: \kappa\{a \mapsto \alpha\}}$$

$$\frac{\Gamma; \Sigma \vdash \varphi :: \kappa}{\Gamma; \Sigma \vdash \lambda_T X.\, \varphi :: \forall_T X.\, \kappa} \qquad \frac{\Gamma; \Sigma \vdash \varphi :: \forall_T X.\, \kappa}{\Gamma; \Sigma \vdash \varphi\ t :: \kappa\{X \mapsto t\}}$$

$$\frac{\Gamma; \Sigma, P :: \kappa_1 \vdash \varphi :: \kappa_2}{\Gamma; \Sigma \vdash \lambda P :: \kappa_1.\, \varphi :: \kappa_1 \to \kappa_2} \qquad \frac{\Gamma; \Sigma \vdash \varphi_1 :: \kappa' \to \kappa \qquad \Gamma; \Sigma \vdash \varphi_2 :: \kappa'}{\Gamma; \Sigma \vdash \varphi_1\ \varphi_2 :: \kappa}$$

## 4.4 Fixpoint

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

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

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

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:

fix $PlusEq(N) :: \forall_T M.\, \forall_T K.\, \star = \lambda_T M.\, \lambda_T K.$
$([N = 0] \wedge (M = K)) \vee (\exists_T N', K'.[N = S\ N'] \wedge [K = S\ K'] \wedge (PlusEq\ N'\ M\ K'))$
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.

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

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.

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

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.

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

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

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

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

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

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:

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

Finally, we define induction over term structure, and thanks to the constraints sublogic we can easily define the notion of *smaller terms* needed for the 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 always exists a *fresh* atom,

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

3. We can always 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 subkinding, existence of formulas with fixpoints, functions, applications, and presence of an environment with variable mapping. Nonetheless, it's simply that - *an equivalence relation* - and it behaves as expected. We will only highlight the interesting parts.

Equivalence checking procedure starts by computing weak head normal form (up to some *depth* denoted by $n$):

$$\begin{aligned}
\texttt{compute}\ \Sigma\ n\ P \quad &\rightsquigarrow \quad \texttt{compute}\ \Sigma\ n\ \varphi \\
\text{when} \quad &\Sigma(P) = \varphi
\end{aligned}$$

$$\begin{aligned}
\texttt{compute}\ \Sigma\ n\ (\varphi\ \alpha) \quad &\rightsquigarrow \quad \texttt{compute}\ \Sigma\ (n'-1)\ \varphi'\{a \mapsto \alpha\} \\
\text{when} \quad &\texttt{compute}\ \Sigma\ n\ \varphi \rightsquigarrow^* (n', \lambda_A a.\, \varphi')
\end{aligned}$$

$$\begin{aligned}
\texttt{compute}\ \Sigma\ n\ (\varphi\ t) \quad &\rightsquigarrow \quad \texttt{compute}\ \Sigma\ (n'-1)\ \varphi'\{X \mapsto t\} \\
\text{when} \quad &\texttt{compute}\ \Sigma\ n\ \varphi \rightsquigarrow^* (n', \lambda_T X.\, \varphi')
\end{aligned}$$

$$\begin{aligned}
\texttt{compute}\ \Sigma\ n\ (\varphi\ t) \quad &\rightsquigarrow \quad \texttt{compute}\ \Sigma\{P \mapsto \phi'\}\ (n'-1)\ \varphi'\{X \mapsto t\} \\
\text{when} \quad &\texttt{compute}\ \Sigma\ n\ \varphi \rightsquigarrow^* (n', \text{fix}\ P(X) :: \kappa = \varphi')
\end{aligned}$$

$$\begin{aligned}
\texttt{compute}\ \Sigma\ n\ (\varphi_1\ \varphi_2) \quad &\rightsquigarrow \quad \texttt{compute}\ \Sigma\ (n_2-1)\ \psi_1\{P \mapsto \psi_2\} \\
\text{when} \quad &\texttt{compute}\ \Sigma\ n\ \varphi_1 \rightsquigarrow^* (n_1, \lambda P :: \kappa.\, \psi_1) \\
\text{and} \quad &\texttt{compute}\ \Sigma\ n_1\ \varphi_2 \rightsquigarrow^* (n_2, \psi_2)
\end{aligned}$$

After we've reached WHNF computation *depth* ($n \leqslant 0$) or cannot reduce the formula further, we can progress naively:

$$\frac{\Gamma;\Sigma \vdash \varphi_1 \equiv \varphi_2 \qquad \Gamma;\Sigma \vdash \psi_1 \equiv \psi_2}{\Gamma;\Sigma \vdash \varphi_1 \to \psi_1 \equiv \varphi_2 \to \psi_2} \qquad \frac{\Gamma;\Sigma \vdash \varphi_1 \equiv \varphi_2 \qquad \Gamma;\Sigma \vdash \psi_1 \equiv \psi_2}{\Gamma;\Sigma \vdash \varphi_1 \land \psi_1 \equiv \varphi_2 \land \psi_2} \qquad \ldots$$

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

Note that we allow *different terms* in equivalent formulas as long as constraints-enviroment $\Gamma$ ensures their equality is provable. For functions, we simply substitute the arguments of both left and right side to the same, fresh name.

$$\frac{X \notin \text{FV}(\Gamma;\Sigma) \qquad \Gamma;\Sigma \vdash \varphi_1[X_1 \mapsto X] \equiv \varphi_2[X_2 \mapsto X]}{\Gamma;\Sigma \vdash \lambda_T X_1.\, \varphi_1 \equiv \lambda_T X_2.\, \varphi_2}$$

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

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

Quantifiers are handled the same way as function above — as they all are a form of bind. To handle formulas with constraints we introduce *constraint equivalence* relation, which does nothing more than use the Solver to check that the constructors of constraint are the same and that arguments are equal to each other in the Solver's sense, analogusly as with terms above.

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

Chapter 5

# Implementation

All the concepts discussed in previous chapters have been implementation in OCaml. Atoms and variables are represented internally by integers (yet remain disjoint sets) — and their string *names* are kept within the environment and binders (quantifiers and functions). Terms, constraints, kinds, and formulas are defined in `Types` module, mirroring their previously described grammars. The only difference is that we accommodate conjunction and disjunction with more than two arguments, with the added feature of arguments being labeled by names. This naming approach empowers users to easily select desired branches while composing proofs. For the Solver,there's a dedicated `Solver` module along with `SolverEnv`, responsible for implementing the specialized environment denoted as $\Delta$ in section X. Analogously, the `KindChecker` and `KindCheckerEnv` modules serve similar roles. The proof theory described in previous chapter is distributed over modules `Proof`, `ProofEnv`, `ProofEquiv` and is a direct implementation of the proof-theoretic rules.

```
(* Module: Types *)
type name_internal = int

type atom = A of name_internal

type var = V of name_internal

type term =  T_Lam of permuted_atom * term | ...

type shape = S_Lam of shape | ...

type constr = C_Fresh of atom * term | ...

type kind = K_Prop | ...

type formula = F_Constr of constr | ...

(* Module: Solver *)
val ( ⊢: ) : constr list -> constr -> bool
(* env ⊢: c ⟺ env ⊨ c *)
```

```

(* Module: SolverEnv *)
type SolverEnv.t

val add_fresh : atom -> var -> SolverEnv.t -> SolverEnv.t

...

val occurs_check : SolverEnv.t -> var -> shape -> bool

(* Module: KindChecker *)
val ( -: ) : formula -> kind -> KindCheckerEnv.t -> bool
(* (f -: k) env ⟺ env ⊢ f :: k *)

val ( <=: ) : kind -> kind -> KindCheckerEnv.t -> bool
(* (k1 <=: k2) env ⟺ env ⊢ k1 <: k2 *)

(* Module: ProofEnv *)
type 'a env
(* Polymorphic in assumption type *)

(* Module: ProofEquiv *)
val computeWHNF : 'a ProofEnv.env
               -> int
               -> Types.formula
               -> 'a ProofEnv.env * int * Types.formula

val ( === ) : Types.formula -> Types.formula -> 'a ProofEnv.env -> bool
(* (f1 === f2) env ⟺ env ⊢ f1 ≡ f2 *)


(* Module: Proof *)
type proof_env = formula env

type judgement = proof_env * formula

type proof = P_Ax of judgement | ...

(* ----------- *)
(*  Γ; f ⊢ f    *)
val assumption : 'a env -> formula -> proof

(*    Γ; Θ, f1 ⊢ f2    *)
(* ------------------- *)
(*  Γ; Θ ⊢ f1  ⟹  f2   *)
val imp_i : formula -> proof -> proof

(*  Γ1; Θ1 ⊢ f1  ⟹  f2    Γ2; Θ2 ⊢ f2  *)
(* ------------------------------------- *)
(*         Γ1 ∪ Γ2; Θ1 ∪ Θ2 ⊢ f2          *)
val imp_e : proof -> proof -> proof

(*  Γ; Θ ⊢ ⊥  *)
```

```
(* ----------- *)
(*   Γ; Θ ⊢ f   *)
val bot_e : formula -> proof -> proof

(*    Γ ⊨ c     *)
(* ---------- *)
(*   Γ; Θ ⊢ c   *)
val constr_i : proof_env -> constr -> proof
```

Note that the `Proof` modules provide methods for constructing forward proofs, i.e., those in which more complex conclusions are built from simpler, already proven facts. Unfortunately, this *bottom-up* way is not the most convenient method for conducting proofs in intuitionistic logic — it is significantly easier to construct proofs in *top-down*, backwards fashion through simplifying the goal to be proven until we reach trivial matters. As such proofs are incomplete by nature, they must have *holes*, and live within some *proof context*, as defined in modules `IncProof`.

Naturally that makes the implementation much more complex, so the appropriate level of confidence in proven propositions will be achieved through other means: we delegate the responsibility for the correctness of the proofs to the `Proof` module, and the `IncProof` module serves as a kind of facade for it.

## 5.1 Proof assistant

To facilitate user interaction with this framework, we provide a practical *proof assistant*. While simple, it is also powerful and easy to use. The interface defined in modules `Prover`, `ProverInternals`, and `Tactics` provides multiple *tactics* (functions that manipulate *prover state*) and ways to combine them — inspired by the HOL family of theorem provers.

```
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
```

$$\mathtt{proof}\ (\Gamma, \Theta, \Sigma)\ \varphi \quad \rightsquigarrow \quad \Gamma; \Theta; \Sigma \vdash \bullet :: \varphi$$

We begin description of the Prover interface with *empty* proof constructor, using $\bullet :: \varphi$ to describe incomplete proofs, called *holes* or *goals*.

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

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

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

$$\begin{array}{rcl}
\mathtt{apply_thm}\ \mathcal{T} & & \\
\Gamma; \Theta; \Sigma \vdash \bullet :: \varphi & \rightsquigarrow & \Gamma; \Theta; \Sigma \vdash \bullet :: \psi \\
 & \text{where} & \mathcal{T} \text{ is a proof of } \ \psi \to \varphi
\end{array}$$

$$\begin{array}{rcl}
\mathtt{apply_assm\ H} & & \\
\Gamma; \Theta; \Sigma \vdash \bullet :: \varphi & \rightsquigarrow & \Gamma; \Theta; \Sigma \vdash \bullet :: \psi \\
 & \text{when} & (\mathsf{H} :: \psi \to \varphi) \in \Theta
\end{array}$$

$$\begin{array}{rcl}
\mathtt{apply_assm_specialized\ H\ [e;\ a]} & & \\
\Gamma; \Theta; \Sigma \vdash \bullet :: \varphi(\mathsf{e}, \mathsf{a}) & \rightsquigarrow & \Gamma; \Theta; \Sigma \vdash \bullet :: \psi(\mathsf{e}, \mathsf{a}) \\
 & \text{when} & (\mathsf{H} :: \forall_T X.\, \forall_A a.\, \psi(X, a) \to \varphi(X, a)) \in \Theta
\end{array}$$

Now, some typical tactics: introduction of names and assumptions and applying of propositions and theorems. Note that propositions can be applied not only on the goal, but also on other assumptions via `apply_in_assumption` tactic. One can also add introduce assumptions to the proof context from theorems via `add_assumption_thm` (specialized if needed via `add_assumption_thm_specialized`) – or simply add any assumption to the current context together with a new goal (of proving that assumption) via `add_assumption`.

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

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

$$\begin{array}{rcl}
\texttt{discriminate} & & \\
\Gamma;\Theta;\Sigma \vdash \bullet :: \varphi & \rightsquigarrow & \Gamma;\Theta;\Sigma \vdash \varphi \\
& \text{when} & \Gamma \vDash \bot
\end{array}$$

Above tactics finish the proofs, either by finding the goal in assumptions (which can be made automatically via tactical`assumption`), or by running Solver on constraint-assumption and the goal. Technical detail is that all formulas in $\Theta$ that are actually constraints will also be included in calls to Solver.

$$\begin{array}{rcl}
\texttt{exists e} & & \\
\Gamma;\Theta;\Sigma \vdash \bullet :: \exists_A a.\, \varphi & \rightsquigarrow & \Gamma;\Theta;\Sigma \vdash \bullet :: \varphi\{a \mapsto \texttt{e}\} \\
\Gamma;\Theta;\Sigma \vdash \bullet :: \exists_T X.\, \varphi & \rightsquigarrow & \Gamma;\Theta;\Sigma \vdash \bullet :: \varphi\{X \mapsto \texttt{e}\}
\end{array}$$

$$\begin{array}{rcl}
\texttt{destr_goal} & & \\
\Gamma;\Theta;\Sigma \vdash \bullet :: [c] \wedge \varphi & \rightsquigarrow & \Gamma;\Theta;\Sigma \vdash \bullet :: c \\
& \text{and} & \Gamma;\Theta;\Sigma \vdash \bullet :: \varphi \\
\Gamma;\Theta;\Sigma \vdash \bullet :: \varphi_1 \wedge \varphi_2 & \rightsquigarrow & \Gamma;\Theta;\Sigma \vdash \bullet :: \varphi_1 \\
& \text{and} & \Gamma;\Theta;\Sigma \vdash \bullet :: \varphi_2
\end{array}$$

$$\begin{array}{rcl}
\texttt{left} & \equiv & \texttt{case l} \\
\Gamma;\Theta;\Sigma \vdash \bullet :: (\texttt{l:}\ \varphi_1) \vee (\texttt{r:}\ \varphi_2) & \rightsquigarrow & \Gamma;\Theta;\Sigma \vdash \bullet :: \varphi_1 \\
\texttt{right} & \equiv & \texttt{case r} \\
\Gamma;\Theta;\Sigma \vdash \bullet :: (\texttt{l:}\ \varphi_1) \vee (\texttt{r:}\ \varphi_2) & \rightsquigarrow & \Gamma;\Theta;\Sigma \vdash \bullet :: \varphi_2
\end{array}$$

Tactics above reduce the current goal.

`destr_assm H`

$$\begin{array}{rcl}
\Gamma; \Theta \cup \{\mathsf{H} :: [c] \wedge \varphi\}; \Sigma \vdash \bullet :: \varphi & \rightsquigarrow & \Gamma \cup \{c\}; \Theta \cup \{\mathsf{H} :: \varphi\}; \Sigma \vdash \bullet :: \varphi \\
\Gamma; \Theta \cup \{\mathsf{H} :: \varphi_1 \wedge \varphi_2\}; \Sigma \vdash \bullet :: \varphi & \rightsquigarrow & \Gamma; \Theta \cup \{\mathsf{H_1} :: \varphi_1, \mathsf{H_2} :: \varphi_2\}; \Sigma \vdash \bullet :: \varphi \\
\Gamma; \Theta \cup \{\mathsf{H} :: \varphi_1 \vee \varphi_2\}; \Sigma \vdash \bullet :: \varphi & \rightsquigarrow & \Gamma; \Theta \cup \{\mathsf{H} :: \varphi_1\}; \Sigma \vdash \bullet :: \varphi \\
& \text{and} & \Gamma; \Theta \cup \{\mathsf{H} :: \varphi_2\}; \Sigma \vdash \bullet :: \varphi
\end{array}$$

`destr_assm' H x`

$$\begin{array}{rcl}
\Gamma; \Theta \cup \{\mathsf{H} :: \exists_A a.\, \varphi\}; \Sigma \vdash \bullet :: \varphi & \rightsquigarrow & \Gamma; \Theta \cup \{\mathsf{H} :: \varphi\{a \mapsto \mathsf{x}\}\}; \Sigma \cup \{\mathsf{x} :: A\} \vdash \bullet :: \varphi \\
\Gamma; \Theta \cup \{\mathsf{H} :: \exists_T X.\, \varphi\}; \Sigma \vdash \bullet :: \varphi & \rightsquigarrow & \Gamma; \Theta \cup \{\mathsf{H} :: \varphi\{X \mapsto \mathsf{x}\}\}; \Sigma \cup \{\mathsf{x} :: T\} \vdash \bullet :: \varphi \\
& \text{when} & \mathsf{x} \notin \mathrm{FV}(\Gamma; \Theta; \Sigma)
\end{array}$$

Tactics above reduce formulas in assumptions. Note that the user provides `destr_assm'` with a *name* that will be bound with existential variable, but the binding is done *behind the scenes* and actually any string can be given and an unique internal identifier is generated.

`ex_falso`

$$\Gamma; \Theta; \Sigma \vdash \bullet :: \varphi \quad \rightsquigarrow \quad \Gamma; \Theta; \Sigma \vdash \bullet :: \bot$$

`generalize x`

$$\begin{array}{rcl}
\Gamma; \Theta; \Sigma \vdash \bullet :: \varphi & \rightsquigarrow & \Gamma; \Theta; \Sigma' \vdash \bullet :: \forall_T \mathsf{x}.\, \varphi \\
& \text{when} & \Sigma = \Sigma' \cup \{\mathsf{x}\} \text{ and } \mathsf{x} \notin \mathrm{FV}(\Gamma)
\end{array}$$

`by_induction x IH`

$$\begin{array}{rcl}
\Gamma; \Theta; \Sigma \vdash \bullet :: (\forall_T X.\, \varphi(X)) & \rightsquigarrow & \Gamma; \Theta \cup \{\mathsf{IH} :: \psi\}; \Sigma \cup \{\mathsf{x} :: T\} \vdash \bullet :: \varphi(X) \\
& \text{where} & \psi := \forall_T \mathsf{x}.\, [\mathsf{x} \prec X] \to \varphi(\mathsf{x})
\end{array}$$

Finally we can prove goals through generalization, induction on terms, and through reduction to absurd.

`compare_atoms a b`

$$\Gamma; \Theta; \Sigma \vdash \bullet :: \varphi \quad \rightsquigarrow \quad \Gamma; \Theta; \Sigma \vdash \bullet :: (\mathsf{a} = \mathsf{b} \vee \mathsf{a} \neq \mathsf{b}) \to \varphi$$

`get_fresh_atom a e`

$$\begin{array}{rcl}
\Gamma; \Theta; \Sigma \vdash \bullet :: \varphi & \rightsquigarrow & \Gamma \cup \{\mathsf{a} \# \mathsf{e}\}; \Theta; \Sigma \cup \{\mathsf{a} :: A\} \vdash \bullet :: \varphi \\
& \text{where} & \mathsf{a} \notin \mathrm{FV}(\Gamma; \Theta; \Sigma)
\end{array}$$

We also provide shorthand formulals for using the axioms of our logic, described in previous chapter. Again argument `a` to `get_fresh_atom` is given by name and is bound by a fresh internal identifier automatically.

Additional we provide the user with some auxiliary tactics: `trivial` th

- `subst` — substitutes atoms for atom expreesions and variables for terms in goal and environment — as long as Solver proves their equality,
- `compute` — computes WHNF of the current goal,
- `try` — applies a tactic and returns unchanged state if the tactic fails
- `repeat` — applies given tactic (until failure),
- `trivial` — tries applying some simple tactics

Finally, the function `qed` accepts a prover state and finalizes it. If the proof state is indeed finished, the function transforms it into a forward proof. This transformation guarantees correctness through the utilization of straightforward rules embedded within the `proof` smart constructors.

Naturally, we also provide a pretty-printer, created using the `EasyFormat` library, along with a parser developed using the `Angstrom` parser combinator library, designed to handle terms, constraints, kinds, and formulas. See how predicates such as $Nat$ and $PlusEq$ can be expressed using programmer-friendly syntax:

```
let fix Nat(n) : ⋆ =
 zero: (n = 0)
 ∨
 succ: (∃ m :term. [n = S m] ∧ Nat m)

let fix PlusEq(n) : ∀ m k : term. ⋆ = fun m k : term →
 zero: ([n = 0] ∧ [m = k])
 ∨
 succ: (∃ n' k' :term. [n = S n'] ∧ [k = S k'] ∧ PlusEq n' m k')
```

And a short proof that 1 is a natural number:

```
let nat_1_thm = arith_thm
    Nat {S 0}

let nat_1 =
  proof' nat_1thm (* goal: Nat {S 0} *)
   |> case "succ"  (* goal: ∃ m :term. [S 0 = S m] ∧ Nat m *)
   |> exists "0"   (* goal: [S 0 = S 0] ∧ Nat 0 *)
   |> by_solver    (* goal: Nat 0 *)
   |> case "zero"  (* goal: 0 = 0 *)
   |> by_solver    (* finished *)
   |> qed
```

Another example theorem could be the symmetry of addition:

```
let plus_symm_thm = arith_thm
  ∀ x y z :term. (IsNum x) ⟹ (IsNum y) ⟹
    (PlusEq x y z) ⟹ (PlusEq y x z)
```

The proof of which is included in the `examples` subdirectory of the project, together with the case study from the next chapter.

# 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 put it to work 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*. But before we delve into the proofs, let's first establish the needed relations:

```
let lambda_symbols = [lam; app; base; arrow; nil; cons]

let 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 fix Type(t): ⋆ =
 base: (t = base)
 ∨
 arrow: (∃ t1 t2 :term. [t = arrow t1 t2] ∧ (Type t1) ∧ (Type t2))

let 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 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))
```

For the proof of *progress*, we will naturally need the predicate that a term is *progressive*, and a lemma about *canonical forms*, which states that all values in the empty environment 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 Value :: ∀ e :term.* = fun e :term →
 var: (exists a :atom. [e = a])
 ∨
 lam: (exists a :atom. ∃ e' : term. [e = lam (a.e')] ∧ (Term e'))

let 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 :: ∀ e :term.* = fun e:term →
 value: (Value e)
 ∨
 steps: (exists e' :term. Steps e 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))
       ∧
     (∃ t1 t2 :term. [t = arrow t1 t2])

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

let empty_contradiction_thm = lambda_thm
  ∀ a :atom. ∀ t :term. (InEnv nil a t)  ⟹  false

let typing_terms_thm = lambda_thm
  ∀ e env t : term. (Typing e env t)  ⟹  (Term e)

let subst_exists_thm = lambda_thm
  ∀ a :atom.
  ∀ v :term. (Value v)  ⟹
  ∀ e :term. (Term e)  ⟹
    ∃ e' :term. (Sub e a v e')
```

Otherwise the proof is a simple induction over *Typing* derivation:

```
let progress =
  proof' progress_thm
  |> by_induction "e0" "IH" %> intro
(* Proof state:
[ ]
[ IH : ∀ e0 : term. [e0 ≺ e] ⟹ ∀ t'1 : term.
        (Typing e0 nil t'1) ⟹ Progressive e0 ]
⊢ (Typing e nil t) ⟹ Progressive e
*)
```

To analyze all the possible branches of the `Typing` predicate, we simply use `destr_intro` tactic to destruct the assumption into multiple branches.

```
  |> destr_intro
```

First one is that `e` is a variable - which contradicts with empty enviroment:

```
  |> intros' ["contra"; "a"; ""]
     %> ex_falso
(* Proof state:
[ e = a ]
[
  contra : InEnv nil a t ;
  ...
]
⊢ ⊥
*)
     %> apply_thm_specialized empty_contradiction ["a"; "t"]
     %> assumption
```

Next, `e` is a lambda abstraction - so a value.

```
  |> intros' ["Hlam"; "a"; "e_a"; "t1"; "t2"; ""] %> case "value"
  (* e is a lambda - value *)
(* Proof state:
[ e = lam (a.e_a) ; t = arrow t1 t2 ]
[
  Hlam : Typing e_a {cons a t1 nil} t2 ∧ Type t1 ;
  ...
]
⊢ Value e
*)
     %> case "lam"
     %> case "lam"
     %> exists' ["a"; "e_a"]
     %> by_solver
```

Then `e` must be an application and thus must be reducing by taking steps, so we apply inductive hypothesis on its sub-expressions `e1` and `e2` and examine the possible cases.

```
  |> intros' ["Happ"; "e1"; "e2"; "t2"; ""; ""] %> case "steps"
```

```
    (* e is an application - steps *)
  |> add_assumption_parse "He1" "Progressive e1"
     %> apply_assm_specialized "IH" ["e1"; "arrow t2 t"] %> by_solver
  |> add_assumption_parse "He2" "Progressive e2"
     %> apply_assm_specialized "IH" ["e2"; "t2"] %> by_solver;;
  |> subst "e" "app e1 e2"
(* Proof state:
[ e = app e1 e2 ]
[
  Happ1 : Typing e1 nil {arrow t2 t} ;
  Happ2 : Typing e2 nil t2 ;
  He1 : Progressive e1 ;
  He2 : Progressive e2 ;
]
⊢ ∃ e' : term. Steps {app e1 e2} e'
*)
```

First we consider the case of both `e1` and `e2` being a value. From `canonical_form` theorem we know then `e1` must be an abstraction — we just need to ensure the Prover that all preconditions are met.

```
  |> destruct_assm "He1" %> intros ["Hv1"]
    %> destruct_assm "He2" %> intros ["Hv2"]  (* Value e1, Value e2 *)
    %> add_assumption_thm_specialized "He1lam"
         canonical_form ["e1"; "arrow t2 t"]
(* Proof state:
[ e = app e1 e2 ]
[
  He1lam : (Value e1)  ⟹  (Typing e1 nil {arrow t2 t})
           ⟹  ∃ a : atom. ∃ e'1 : term. [e1 = lam (a.e'1)] ∧ Term e'1 ;
  Hv1 : Value e1 ;
  Hv2 : Value e2 ;
  ...
]
⊢ ∃ e' : term. Steps {app e1 e2} e'
*)
    %> apply_in_assm "He1lam" "Hv1"
    %> apply_in_assm "He1lam" "Happ_1"
    %> destruct_assm' "He1lam" ["a"; "e_a"; ""]
    %> subst "e1" "lam (a.e_a)"
(* Proof state:
[ e = app e1 e2 ; e1 = lam (a.e_a) ]
[
  He1lam : Term e_a ;
  ...
]
⊢ ∃ e' : term. Steps {app (lam (a.e_a)) e2} e'
*)
```

Then we need to find the `e'` that `app e1 e2` reduces to, and now that we know `e1` is an abstraction, then we can use beta-reduction rule and find the term of abstracion body `e_a` with argument `a` subistuted with `e2`. Again, we ensure the Prover that preconditions are met and destruct on the final assumption to extract the term that we searched for: `e_a’`.

```
    %> add_assumption_thm_specialized "He_a"
         subst_exists ["a"; "e2"; "e_a"]
(* Proof state:
[ ... ]
[
  He_a : (Value e2) ⟹ (Term e_a) ⟹ ∃ e’ : term. Sub e_a a e2 e’ ;
  ...
]
⊢ ∃ e’ : term. Steps e e’
*)
    %> apply_in_assm "He_a" "Hv2"
    %> apply_in_assm "He_a" "He1lam"
    %> destruct_assm’ "He_a" ["e_a’"]
    %> exists "e_a’"
(* Proof state:
[ ... ]
[
  He_a : Sub e_a a e2 e_a’ ;
  ...
]
⊢ Steps {app (lam (a.e_a)) e2} e_a’
*)
    %> case "app" %> exists’ ["a"; "e_a"; "e2"] %> by_solver
(* Proof state:
[ ... ]
[ ... ]
⊢ Value e2 ∧ Sub e_a a e2 e_a’
*)
    %> destruct_goal %> apply_assm "Hv2" %> apply_assm "He_a"
```

Now what’s left is to examine straightforward cases where either `e1` or `e2` steps.

```
  |> intros’ ["Hs2"; "e2’"] (* Value e1, Steps e2 e2’ *)
     %> exists "app e1 e2’"
(* Proof state:
[ ... ]
[
  Hv1 : Value e1 ;
  Hs2 : Steps e2 e2’ ;
  ...
]
⊢ Steps {app e1 e2} {app e1 e2’}
*)
    %> case "app_r"
    %> exists’ ["e1"; "e2"; "e2’"]
    %> repeat by_solver
(* Proof state:
[ ... ]
[ ... ]
⊢ Value e1 ∧ Steps e2 e2’
*)
```

```
    %> destruct_goal
    %> apply_assm "Hv1"
    %> apply_assm "Hs2"
  |> intros' ["Hs1"; "e1'"] (* Steps e1 *)
(* Proof state:
[ ... ]
[
  Hs1 : Steps e1 e1' ;
  ...
]
⊢ Steps {app e1 e2} {app e1' e2}
*)
    %> exists "app e1' e2"
    %> case "app_l"
    %> exists' ["e1"; "e1'"; "e2"]
    %> repeat by_solver
    %> apply_assm "Hs1"
  |> apply_assm "Happ_2" %> apply_assm "Happ_1"
  |> qed
```

To prove *Preservation*, we will need some more relations and lemmas:

```
let fix Sub(e): ∀ a :atom. ∀ v e':term.* = fun (a :atom) (v e' :term) →
 var_same: ([e = a] ∧ [e' = v])
 ∨
 var_diff: (exists b :atom. [e = b] ∧ [e' = b] ∧ [a =/= b])
 ∨
 lam: (∃ b :atom. ∃ e_b e_b' :term. [e = lam (b.e_b)] ∧
        [e' = lam (b.e_b')] ∧ [b # v] ∧ [a =/= b] ∧ (Sub e_b a v e_b') )
 ∨
 app: (∃ e1 e2 e1' e2' :term.
        [e = app e1 e2] ∧ [e' = app e1' e2']
          ∧ (Sub e1 a v e1') ∧ (Sub e2 a v e2') )

let EnvInclusion :: ∀ env1 env2 :term.* = fun env1 env2 : term →
  ∀ a : atom. ∀ t : term. (InEnv env1 a t)  ⟹  (InEnv env2 a t)
```

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
  ∀ 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)
```

2. Weakening lemma: for any enviroment `env1`, we can use larger enviroment `env2` without breaking the typing.

```
let weakening_lemma_thm = lambda_thm
  ∀ e env1 t env2 : term.
```

```
    (Typing e env1 t) ⟹
    (EnvInclusion env1 env2) ⟹
      (Typing e env2 t)
```

3. Lambda abstraction typing inversion: If term `lam (a.e)` has a type `{arrow t1 t2}` in environment `env`, then it must be that the body `e` has a type `t2` in enviroment extended with the argument `{cons a t1 env}`.

```
let lambda_typing_inversion_thm = lambda_thm
  ∀ a :atom. ∀ e env t1 t2 :term.
    (Typing {lam (a.e)} env {arrow t1 t2}) ⟹
      (Typing e {cons a t1 env} t2)
```

We proof preservation through induction on term `e` the case analysis on assumption `Steps e e'`.

```
let preservation =
  let deduce_app_typing = ...
  in proof' preservation_thm
  |> by_induction "e0" "IH"
  |> intro %> intro %> intro %> intros ["Htyp"; "Hstep"]
(* Proof state:
[ ]
[
  Hstep : Steps e e' ;
  Htyp : Typing e env t ;
  IH : ∀ e0 : term. [e0 ≺ e]
          ⟹ ∀ e'1 env'1 t'1 : term. (Typing e0 env'1 t'1)
            ⟹ (Steps e0 e'1)
              ⟹ Typing e'1 env'1 t'1
]
⊢ Typing e' env t
*)
  |> destruct_assm "Hstep"
```

First two cases are rather simple: `e` is `app e1 e2` and either `e1` or `e2` take a step. We use local tactics `deduce_app_typing`, `contra_var` and `contra_app` not to repeat some boilerplate operations and omit the definition here to encourae the reader to look into the source code themselves.

```
  |> intros' ["He1"; "e1"; "e1'"; "e2"; ""; ""]
     %> deduce_app_typing
     %> case "app"
     %> exists' ["e1'"; "e2"; "t2"]
     %> by_solver
(* Proof state:
[ e = app e1 e2 ; e' = app e1' e2 ]
[
  Happ_2 : Typing e2 env t2 ;
  Happ_1 : Typing e1 env {arrow t2 t} ;
  He1 : Steps e1 e1 ;
  ...
]
```

```
⊢ Typing e1' env {arrow t2 t} ∧ Typing e2 env t2
*)
      %> destruct_goal
      (* Typing e1 env t1 *)
        %> 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'"; ""; ""; ""]
      %> deduce_app_typing
      %> case "app"
      %> exists' ["v1"; "e2'"; "t2"]
      %> by_solver
(* Proof state:
[ e = app e1 e2 ; e' = app e1' e2 ]
[
  He2 : Value v1 ∧ Steps e2 e2' ;
  ...
]
⊢ Typing e1 env {arrow t2 t} ∧ Typing e2' env t2
*)
      %> 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"]
        (* Typing e2 env t2 ⟹
               Steps e2 e2' ⟹ Typing e2' env t2 *)
          %> by_solver
          %> apply_assm "Happ_2"
          %> apply_assm "He2_2"
```

The next, final case is where we will need the established lemmas: application `app e1 e2` beta-reduces into some term `e'` and we use the `sub_lemma` to show that `e'` still types.

```
  |> intros' ["Hbeta"; "a"; "e_a"; "v"; ""; ""]
    %> deduce_app_typing
(* Proof state:
[ e = app (lam (a.e_a)) v ]
[
  Happ_2 : Typing v env t2 ;
  Happ_1 : Typing (lam (a.e_a)) env {arrow t2 t} ;
  Hbeta_1 : Value v ;
  Hbeta_2 : Sub e_a a v e' ;
  ...
]
⊢ Typing e' env t
*)
```

```
  %> 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"
  %> apply_thm_specialized
       lambda_typing_inversion ["a"; "e_a"; "env"; "t2"; "t"]
  (* Typing {lam (a.e_a)} env {arrow t2 t}
        ⟹  Typing e_a {cons a t2 env} t *)
  %> apply_assm "Happ_1"
  %> apply_assm "Hbeta_2"
|> qed
```

# Chapter 7

# Conclusion and future work

...

# Bibliography


[1] ...