Domain-specific logic for terms with variable binding

(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

27 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

One of the fundamental distinctions between conducting proofs manually with pen and paper and using a computer lies in the flexibility and liberties one can take in the first case. Human provers and reviewers often agree upon unexplained or unproven assumptions and may skip some unimportant boilerplate. Computers, on the other hand, are less forgiving and demand transparency and justification down to the smallest details.

A common assumption we commonly make when writing pen-and-paper proofs pertains to working with abstract syntax trees, where we assume that the variables we choose are fresh enough or that substitutions avoid issues like variable capture. For instance, when dealing with lambda calculus, we often construct inductive proofs over the structure of expression, where in the case for an abstracion we will implicitly only show the case where the variable bound in that abstraction is *sufficiently fresh*. Addressing the general case could introduce unnecessary complexities unrelated to the theorem at hand. Justifiably, we skip over this detail — however, the induction principle obliges us to prove the case for arbitrary variable names.

Addressing this gap in formal reasoning requires careful considerations to come up with a resolution. Fortunately, there exist some solutions to that problem — and one particular approach, coined *nominal logic* and introduced by Andrew M. Pitts[3] is of most interest to this work.

1.1 Nominal approach

Pitts' work introduces *nominal logic*, a first-order theory of names, swapping, and freshness, that amongst other novelties, introduces the precise mathematical definition describing the concept of "sufficiently fresh names", which, as Pitts argues, bridges the gap between formal mathematical reasoning and the informal practices mentioned earlier.

Pitts, 2003[3]

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

Pitts, 2003[3]

Why the emphasis on the operation of swapping two names, rather than on the apparently more primitive notion of renaming one name by another? The answer to this question lies in the combination of the following two facts.

1. First, even though swapping seems less general than renaming (since after all, the act of swapping a and b can be expressed as the simultaneous renaming of b by a and a by b), it is possible to found a theory of syntax modulo α-equivalence, free and bound variables, substitution, etc., upon this notion— this is the import of the work in [1].
2. Secondly, swapping is an involutive operation: a swap followed by the same swap is equivalent to doing nothing. This means that the class of equivariant predicates, i.e., those whose validity is invariant under atom-swapping, has excellent logical properties. It contains the equality predicate and is closed under negation, conjunction, disjunction, existential and universal quantification, formation of least and greatest fixed points of monotone operators, etc., etc. The same is not true for renaming. For example, the validity of a negated equality between atoms is not necessarily preserved under renaming.

In other words, we can found a theory of variable-binding upon swapping, and it is convenient to do so because of its good logical properties.

A crucial takeaway from Pitts' work is that switching from substitutions to permutations of names allows for all necessary concepts, including alpha-equivalence, freshness, and variable-binding, to be defined solely in terms of the operation of swapping pairs of names. As an example, consider the abstract syntax tree of untyped lambda calculus, given by the grammar:

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

Where a ranges over an infinite set of names — or rather *atoms*. Now, let's define what it means to swap atoms a and b in some tree t, written $(a\ b)t$:

$$\begin{aligned}
(a\ b)c &= \begin{cases} a & \text{if } c = b \\ b & \text{if } c = a \\ c & \text{otherwise} \end{cases} \\
(a\ b)(\lambda c.t) &= \lambda((a\ b)c).((a\ b)t) \\
(a\ b)(t_1\ t_2) &= ((a\ b)t_1)\ ((a\ b)t_2)
\end{aligned}$$

Notice how straightforward is operation of swapping — simpler than the substitution, as it doesn't distinct betwen free and bound names, but simply changes them all the same exact way.

Freshness of atom a in tree t, written $a \# t$, can be similarly simple to define.[1]

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

Note that *freshness* assumes only the comparability of atoms and is an *equivariant* relation, meaning that it's validity is invariant under swapping atoms — which can be shown by simplest induction. Then with *swapping* and *freshness*, we can define the alpha-equivalence of terms, written $t_1 =_\alpha t_2$.

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

As we built this definition of alpa-equivalence using only induction, swapping, and freshness, then as Pitts argues, it's also equivariant.

> **Pitts, 2003**
>
> The fundamental assumption underlying Nominal Logic is that *the only predicates we ever deal with* (when describing properties of syntax) *are equivariant ones, in the sense that their validity is invariant under swapping* (i.e., transposing, or interchanging) *names.*

1.2 Contributions

We categorize the fundamental properties of terms with variable binding, including alpha equivalence and freshness, as *constraints* and construct *the Solver*, an algorithm tasked with automatically resolving new constraints based off the already established constraints. We use it as a logical core of the constraints sublogic that together with embeddedment of constraints into propositional formulas builds the logical framework that effortlessly expresses these properties. Through handling the constraints

[1]Pitts defines it as a not being a member of the *support set* of t — but for our purposes, the simple inductive definition will suffice.

automatically, it liberates its users from the painstaking task of manually proving the seemingly trivial, yet crucial details, while ensuring the completeness and correctness of the written proofs.

TODO: write about proof assistant.

TODO: add related work

Chapter 2

Terms and constraints

To properly describe our framework we must start with the simplest elements: *names* and *terms*.

The names are drawn from an infinite set of *atoms* (represented by lowercase letters) and correspond to the bound variables in terms, analogous to the variables in the 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).

The terms are constructed to mimic the structure of abstract syntax trees of the lambda calculus, extending it with notion of permutations (of atoms) and functional symbols, denoted by metavariable s, that are drawn from yet another set disjoint with atoms and variables. Terms are defined by the following grammar:

$$
\begin{array}{rcll}
\pi & ::= & \mathsf{id} \mid (\alpha\ \alpha)\pi & \text{(permutations)} \\
\alpha & ::= & \pi\ a & \text{(atom expressions)} \\
t & ::= & \alpha \mid \pi\ X \mid \alpha.t \mid t\ t \mid s & \text{(terms)}
\end{array}
$$

Construction $\alpha.t$ represents a *binder* — binding the occurences of α in t, similarly to a lambda abstraction. Note that we do not restrict the binding construction to the form of $a.t$, allowing permuted atoms to appear in binders. Additionaly, when dealing with atom expressions with identity permutation $\mathsf{id}a$ we will skip the permutation and simply write a and sometimes call such atoms *pure*. The same rule applies to permuted variabless

It's important to note that these terms are merely a data structure and do not incorporate notions of computation, reduction, or binding by themselves. Their practical application is observed in the sublogic of constraints defined on top of them, 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 and informal semantics:

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

$\alpha \,\#\, t$	Atom α is fresh in term t, meaning it 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.
$t_1 \prec t_2$	The 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.
symbol t	term t is equal to some functional symbol.

2.1 Model

To build the mathematical model of terms and constraints, we introduce *semantic terms* and *semantic shapes* that will inhabit it. We will use metavariable A for *semantic names* drawn from an arbitrary infinite set of names, representing the free variables. Binders in semantic terms will be achieved by De Bruijn indices[2] and consequently the bound names will be represented by natural numbers, denoted by n, and the binding construction will have no explicit argument, denoted by \$.

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

The term interpretation function, mapping syntactic terms to semantic terms utilizes the standard shifting of De Bruijn indices (denoted by $\uparrow$). It is parametrized by function ρ from atoms and variables into semantic shapes.

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

The shape interpretation function is a mapping from semantic terms to semantic shapes by erasing names.

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

With above machinery, we can establish how to *interpret* the constraints in our model:

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

Note that the freshness can be expressed through membership check of $\mathsf{FreeAtoms}$ set, which is trivial to compute as a consequence of using of De Bruijn indices. Note that it's possible for terms of form $a.X$ and $b.Y$ to be equal in this model.

We will use metavariable Γ to represent finite sets of constraints, and write $\rho \vDash \Gamma$ if for all $c \in \Gamma$, we have $\rho \vDash c$, as well as write $\Gamma \vDash c$ if for every ρ such that $\rho \vDash \Gamma$, we have $\rho \vDash c$.

In the next chapter, we will present the deterministic *Solver* algorithm that works within this model to check whether assumed constraints $c_1, \ldots, c_n$, would imply the constraint-goal c_0.

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 when given judgement $c_1, \ldots, c_n \vDash c_0$ it dissects constraints on both sides of the turnstile into irreducible components that are solved easily, veryfying 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 its own internal representation of constraints, that slightly differs from constraints described in the previous section, as it uses *shapes* instead of terms to to construct shape constraints.

TODO: add symbol constraints and its rules

Solver constraints and shapes are defined by the following grammar:

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

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 $a \mathrel{\#} \alpha$ as it gives a clear intuition of atom freshness implying inequality. Additionally, when $\alpha = \pi a$, we will denote $\alpha \mathrel{\#} t$ to mean $a \mathrel{\#} \pi^{-1} t$.

Irreducible constraints are:

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

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:

$$
\cfrac{\cfrac{\cfrac{\mathcal{C}'' \in \Delta''}{\cdots}}{\Gamma'; \Delta' \vdash \mathcal{C}' \qquad \cdots}}{\Gamma; \Delta \vdash \mathcal{C}}
$$

3.1 Goal-reducing rules

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

$$
\frac{}{\Gamma; \Delta \vdash a = a} \qquad \frac{}{\Gamma; \Delta \vdash X = X} \qquad \frac{}{\Gamma; \Delta \vdash s = s} \qquad \frac{\Gamma; \Delta \vdash t_1 = t_2 \qquad \Gamma; \Delta \vdash t_1' = t_2'}{\Gamma; \Delta \vdash t_1 t_1' = t_2 t_2'}
$$

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:

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

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

1. a is different from both α_1 and α_2, so the swap doesn't change the goal,
2. a is equal to α_1 but different from α_2, so the swap substitutes it for α_2,
3. a is equal to α_2, so the swap substitutes it for α_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.

$$\frac{a \neq \alpha_1, a \neq \alpha_2, \Gamma; \Delta \vdash a = \alpha \qquad a = \alpha_1, a \neq \alpha_2, \Gamma; \Delta \vdash \alpha_2 = \alpha \qquad a = \alpha_2, \Gamma; \Delta \vdash \alpha_1 = \alpha}{\Gamma; \Delta \vdash a = (\alpha_1\ \alpha_2)\alpha}$$

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

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

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

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

Freshness is checked through the Δ 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:

$$\frac{a \neq \alpha_1, a \neq \alpha_2, \Gamma; \Delta \vdash a \,\#\, \alpha \qquad a = \alpha_1, a \neq \alpha_2, \Gamma; \Delta \vdash \alpha_1 \,\#\, \alpha \qquad a = \alpha_2, \Gamma; \Delta \vdash \alpha_2 \,\#\, \alpha}{\Gamma; \Delta \vdash a \,\#\, (\alpha_1\ \alpha_2)\alpha}$$

$$\frac{a \neq \alpha_1, a \neq \alpha_2, \Gamma; \Delta \vdash a \,\#\, \pi X \qquad a = \alpha_1, a \neq \alpha_2, \Gamma; \Delta \vdash \alpha_1 \,\#\, \pi X \qquad a = \alpha_2, \Gamma; \Delta \vdash \alpha_2 \,\#\, \pi X}{\Gamma; \Delta \vdash a \,\#\, (\alpha_1\ \alpha_2)\pi X}$$

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 Δ:

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

Shape equality is naturally structural:

$$\frac{\Gamma; \Delta \vdash \mathcal{S}_1 \sim \mathcal{S}_2}{\Gamma; \Delta \vdash \_.\mathcal{S}_1 \sim \_.\mathcal{S}_2} \qquad \frac{\Gamma; \Delta \vdash \mathcal{S}_1 \sim \mathcal{S}_2 \qquad \Gamma; \Delta \vdash \mathcal{S}_1' \sim \mathcal{S}_2'}{\Gamma; \Delta \vdash \mathcal{S}_1\mathcal{S}_1' \sim \mathcal{S}_2\mathcal{S}_2'}$$

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

$$\frac{\Gamma; \Delta \vdash \mathcal{S}_1 \sim \mathcal{S}_2}{\Gamma; \Delta \vdash \mathcal{S}_1 \prec \_.\mathcal{S}_2} \qquad \frac{\Gamma; \Delta \vdash \mathcal{S}_1 \prec \mathcal{S}_2}{\Gamma; \Delta \vdash \mathcal{S}_1 \prec \_.\mathcal{S}_2}$$

$$\frac{\Gamma; \Delta \vdash \mathcal{S}_1 \sim \mathcal{S}_2}{\Gamma; \Delta \vdash \mathcal{S}_1 \prec \mathcal{S}_2\mathcal{S}_2'} \qquad \frac{\Gamma; \Delta \vdash \mathcal{S}_1 \sim \mathcal{S}_2'}{\Gamma; \Delta \vdash \mathcal{S}_1 \prec \mathcal{S}_2\mathcal{S}_2'} \qquad \frac{\Gamma; \Delta \vdash \mathcal{S}_1 \prec \mathcal{S}_2}{\Gamma; \Delta \vdash \mathcal{S}_1 \prec \mathcal{S}_2\mathcal{S}_2'} \qquad \frac{\Gamma; \Delta \vdash \mathcal{S}_1 \prec \mathcal{S}_2'}{\Gamma; \Delta \vdash \mathcal{S}_1 \prec \mathcal{S}_2\mathcal{S}_2'}$$

Environment Δ keeps track of all shapes that given variable subshapes:

$$\frac{\mathcal{S}_2 \prec X \in \Delta \qquad \Gamma; \Delta \vdash \mathcal{S}_2 \sim X}{\Gamma; \Delta \vdash \mathcal{S}_1 \prec X} \qquad \frac{\mathcal{S}_2 \prec X \in \Delta \qquad \Gamma; \Delta \vdash \mathcal{S}_2 \prec X}{\Gamma; \Delta \vdash \mathcal{S}_1 \prec X}$$

3.2 Assumptions-reducing rules

But before the Solver can reduce the goal-constraint, it must first reduce all assumptions in the Γ environment. We will now present the rules for reducing the left side of the turnstile, but fortunately 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 inverting it and moving it to the right-hand side. TODO: describe notation of pseudocode.

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

TODO: rewrite this

$$\mathsf{id}^{-1}\ t = \mathsf{id}\ t$$

$$((\alpha_1\ \alpha_2)\pi)^{-1}\ t = \pi^{-1}(\alpha_1\ \alpha_2)\ t$$

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 Γ and goal $\mathcal{C}$ is indeed a simple term substitution, substituting in Δ environment 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 Δ 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.

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

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

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

Atom inequality and freshness simply added to the Δ enviroment:

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

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

$$\frac{a \neq \alpha_1, a \neq \alpha_2, a \,\#\, \alpha, \Gamma; \Delta \vdash \mathcal{C} \qquad a = \alpha_1, a \neq \alpha_2, \alpha_2 \,\#\, \alpha, \Gamma; \Delta \vdash \mathcal{C} \qquad a = \alpha_2, \alpha_1 \,\#\, \alpha, \Gamma; \Delta \vdash \mathcal{C}}{a \,\#\, (\alpha_1\ \alpha_1)\alpha, \Gamma; \Delta \vdash \mathcal{C}}$$

$$\frac{a \neq \alpha_1, a \neq \alpha_2, a \,\#\, \pi X, \Gamma; \Delta \vdash \mathcal{C} \qquad a = \alpha_1, a \neq \alpha_2, \alpha_2 \,\#\, \pi X, \Gamma; \Delta \vdash \mathcal{C} \qquad a = \alpha_2, \alpha_1 \,\#\, \pi X, \Gamma; \Delta \vdash \mathcal{C}}{a \,\#\, (\alpha_1\ \alpha_1)\pi X, \Gamma; \Delta \vdash \mathcal{C}}$$

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

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

Variable being the same shape as other term is added to the Δ environment:

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

Otherwise shape assumptions recurse on the shape structure:

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

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

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

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

Again, Δ 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:

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

3.3 Irreducible constraints

Environment Δ that containts all the irreducible assumptions is given by a sextuple (`neq_atoms`$_\Delta$, `fresh`$_\Delta$, `var_shape`$_\Delta$, `shape`$_\Delta$, `subshape`$_\Delta$, `symbols`Δ) with following semantics:

`neq_atoms`	Set of pairs of atoms that are known to be different.
`fresh`	Set of pairs of atom and variable, indicating that the atom is *fresh* in the variable.
`var_shape`	Mapping from variables to shape-representative variables. All variables mapped to the same representative are considered to inhabit the same shape.
`shape`	Mapping from shape-representative variables to the actual shape it must inhabit.
`subshape`	Mapping from shape-representative variables to sets of shapes that the variable must supershape.
`symbols`	Set of shape-representative variables that are known to be some unknown functional symbols.

We can now establish a method to compute the shape-representative variable and outline the procedure for reconstructing the shape within the environment Δ:

$X_\Delta :=$
  | `if` $Y \leftarrow$ `var_shape`$_\Delta$ X `then` Y_Δ
  | `otherwise` X

$|X|_\Delta :=$
  | `if` $Y \leftarrow$ `var_shape`$_\Delta$ X `then` $|Y|_\Delta$
  | `if` $\mathcal{S} \leftarrow$ `shape`$_\Delta$ X `then` $\mathcal{S}$
  | `otherwise` X

$$\begin{aligned} |\_|_\Delta &:= \_ \\ |\_.\mathcal{S}|_\Delta &:= \_.|\mathcal{S}|_\Delta \\ |\mathcal{S}_1\mathcal{S}_2|_\Delta &:= |\mathcal{S}_1|_\Delta|\mathcal{S}_2|_\Delta \\ |s|_\Delta &:= s \\ |t|_\Delta &:= ||t||_\Delta \end{aligned}$$

Then, verifying whether a constraint is included in Δ can be accomplished straightforwardly:

$$
\begin{aligned}
(a_1 \neq a_2) \in \Delta &:= (a_1 \neq a_2) \in \texttt{neq\_atoms}_\Delta \\
(a \# X) \in \Delta &:= X \in \texttt{fresh}_\Delta(a) \\
(X_1 \sim X_2) \in \Delta &:= |X_1|_\Delta = |X_2|_\Delta \\
(X \sim \mathcal{S}) \in \Delta &:= \mathcal{S} = \texttt{shape}_\Delta(X_\Delta) \\
(\mathcal{S} \prec X) \in \Delta &:= \mathcal{S} \in \texttt{subshape}_\Delta(X_\Delta)
\end{aligned}
$$

And establish rules for a special 'occurs check' mechanism, which safeguards against handling circular references.

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

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

Incorporating constraints into Δ proceeds as follows: freshness of an atom in a in a variables is simply acknowledged in the `fresh` mapping. Inequality of two atoms simply adds to the set `neq_atoms`, unless invoked with identical atoms, in which case we report a contradiction.

```
{a # X} ∪ Δ :=
  Δ  |> fresh += (a # X)
```

```
{a ≠ a'} ∪ Δ :=
  | if a = a' then ↯
  | otherwise Δ |> neq_atoms += (a ≠ a')
```

To meld together two shape-variables, we first check whether they have already been merged. If they have, we return contradiction.
Next, we conduct an occurs check to ensure that merging them won't create a circular reference. If this check fails, we again report a contradiction.
Finally, we merge all the information pertaining to X into X' and remove any traces of X from within Δ environment.
To maintain a high-level description, we delegate the detailed implementation aspects to auxiliary functions responsible for substituting shape-variables within the given field of Δ.

```
{X ~ X'} ∪ Δ :=
  | if X_Δ = X'_Δ then Δ
  | if |X|_Δ = |X'|_Δ then Δ
  | if X_Δ occurs in |X'|_Δ then ↯
  | if X'_Δ occurs in |X|_Δ then ↯
  | otherwise Δ |> symbols        {X_Δ ⇝ X'_Δ}
                |> subshape       {X_Δ ⇝ X'_Δ}
                |> transfer_shape {X_Δ ⇝ X'_Δ}
                |> var_shape += (X_Δ ↦ X'_Δ)
                |> shape     -= X_Δ
                |> subshape  -= X_Δ
```

To set variable shape, we first make sure to perform occurs check on the proposed shape and then substitute the shape-variable in all affected fields.

$\{X \sim \mathcal{S}\} \cup \Delta :=$
  | `if` $\Delta \vdash X$ occurs in $\mathcal{S}$ `then` $\lightning$
  | `otherwise` Δ `|> symbols` $\{X_\Delta \rightsquigarrow |\mathcal{S}|_\Delta\}$
        `|> subshape` $\{X_\Delta \rightsquigarrow |\mathcal{S}|_\Delta\}$
        `|> shape` $\{X_\Delta \rightsquigarrow |\mathcal{S}|_\Delta\}$

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*.

`symbols` $\{X \rightsquigarrow \mathcal{S}\}$ Δ $:=$
  | `if` $X_\Delta \notin$ `symbols`$_\Delta$ `then` Δ
  | `otherwise` Δ `|> symbols -=` X
        `|> assumptions +=` (symbol $\mathcal{S}$)

`shape` $\{X \rightsquigarrow \mathcal{S}\}$ Δ $:=$
  | `if` $\mathcal{S}' \leftarrow$ `shape`$_\Delta$ X `then` Δ `|> assumptions +=` $(\mathcal{S} \sim \mathcal{S}')$
  | `otherwise` Δ `|> shapes +=` $(X \mapsto \mathcal{S})$

`subshape` $\{X \rightsquigarrow \mathcal{S}\}$ Δ $:=$
  Δ `|> assumptions +=` (`subshapes`$_\Delta X \prec \mathcal{S}$)

`transfer_shape` $\{X \rightsquigarrow X'\}$ Δ $:=$
  | `if` $\mathcal{S} \leftarrow$ `shape`$_\Delta$ X `then` Δ `|> shape` $\{X' \rightsquigarrow \mathcal{S}\}$
  | `otherwise` Δ

Finally, we demonstrate how the substitution of variables and atoms is accomplished, thereby concluding the description of the *Solver* and its environment.

Δ $\{X \mapsto t\}$ $:=$
  Δ `|> fresh -=` X
    `|> assumptions +=` $(X \sim |t|_\Delta)$
    `|> assumptions +=` $\bigcup_{(a \,\#\, X) \in \Delta} (a \,\#\, t)$

Δ $\{a \mapsto a'\}$ $:=$
  Δ `|> fresh -=` a
    `|> fresh +=` ($a' \,\#$ `fresh`$_\Delta a$)
    `|> clear neq_atoms`
    `|> assumptions +=` $\bigcup_{(a_1 \neq a_2) \in \Delta} (a_1\{a \mapsto a'\} \neq a_2\{a \mapsto a'\})$

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

To address this question, we define the state of the Solver as a triple $(\Gamma, \Delta, \mathcal{C})$. Upon analyzing the Solver rules, it becomes evident that each rule consistently leads to a lesser state by reducing it through one or more of the following actions:

1. Decreasing the number of distinct variables in Γ, Δ, and $\mathcal{C}$, or maintaining the same number while:
2. Decreasing the depth of $\mathcal{C}$, or preserving the current depth while:
3. Reducing assumptions with a given depth in either Γ or Δ into assumptions with lower depth, or maintaining the number and depth of assumptions, while:

4. Eliminating an assumption from Γ and introducing an assumption of the same depth into Δ.

In the following chapters, we will write $\Gamma \vDash c$ but mean $\Gamma; \emptyset \vdash \mathcal{C}$, which by the construction of $\vdash$ we consider equivalent to $\Gamma \vDash c$ as defined in the model.

Chapter 4

Higher Order Logic

On top of the sublogic of constraints, we build a higher-order logic.

4.1 Kinds

Due to the multiple ways atoms, terms, binders, and constraints can occur in formulas, we introduce kinds to ensure that the formulas we will deal with *make sense*, given by the following grammar and semantics:

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

$\varphi :: \star$	φ is a propositional formula.
$\varphi :: \kappa_1 \to \kappa_2$	φ is a function that takes a formula of kind κ_1, and produces a formula of kind κ_2.
$\varphi :: \forall_A a.\, \kappa$	φ is a function that takes an atom expression, binds it to a, and produces a formula of kind κ.
$\varphi :: \forall_T X.\, \kappa$	φ is a 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:

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

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

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

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

4.4 Fixpoint

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

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

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

The fixpoint constructor allows us to express *recursive* predicates over terms, but only such that the recursive applications of it are on structurally smaller terms, which we express in it's 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 N models some natural number and that two terms N and M added together are the equal to K:

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

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

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 $(Nat :: \forall_T N.\, \star)$.

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 Natural deduction

Finally, we come to the definition of proof-theoretic rules. Starting with inference rules for assumption, we can see first an analogue between the worlds of propositional logic and constraint sublogic. 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.

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

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

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

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

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

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

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

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

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

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):

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

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

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

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

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

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

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

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

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

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

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

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

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 previusly described grammars. The only difference is that we allow conjunction and disjunction to be used with more than two arguments, with the added feature of arguments being labeled by names. This naming approach lets the user to easily select desired branches while composing proofs or to give meaningful names within the definition of properties.

The *Solver* ihabits its own dedicated `Solver` module along with `SolverEnv` responsible for implementing the specialized environment Δ handling the irreducible assumptions. 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
```

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

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

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

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

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

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

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

`apply_assm_specialized H [e; a]`

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

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`.

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

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

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

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 Θ that are actually constraints will also be included in calls to Solver.

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

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

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

Tactics above reduce the current goal.

$$
\begin{array}{rcl}
\multicolumn{1}{c}{\texttt{destr\_assm H}} & & \\
\Gamma; \Theta \cup \{\mathsf{H} :: [c] \wedge \varphi\}; \Sigma \vdash \bullet :: \varphi & \rightsquigarrow & \Gamma \cup \{c\}; \Theta \cup \{\mathsf{H} :: \varphi\}; \Sigma \vdash \bullet :: \varphi \\
\Gamma; \Theta \cup \{\mathsf{H} :: \varphi_1 \wedge \varphi_2\}; \Sigma \vdash \bullet :: \varphi & \rightsquigarrow & \Gamma; \Theta \cup \{\mathsf{H\_1} :: \varphi_1, \mathsf{H\_2} :: \varphi_2\}; \Sigma \vdash \bullet :: \varphi \\
\Gamma; \Theta \cup \{\mathsf{H} :: \varphi_1 \vee \varphi_2\}; \Sigma \vdash \bullet :: \varphi & \rightsquigarrow & \Gamma; \Theta \cup \{\mathsf{H} :: \varphi_1\}; \Sigma \vdash \bullet :: \varphi \\
 & \text{and} & \Gamma; \Theta \cup \{\mathsf{H} :: \varphi_2\}; \Sigma \vdash \bullet :: \varphi \\
\multicolumn{1}{c}{\texttt{destr\_assm' H x}} & & \\
\Gamma; \Theta \cup \{\mathsf{H} :: \exists_A a.\, \varphi\}; \Sigma \vdash \bullet :: \varphi & \rightsquigarrow & \Gamma; \Theta \cup \{\mathsf{H} :: \varphi\{a \mapsto \mathsf{x}\}\}; \Sigma \cup \{\mathsf{x} :: A\} \vdash \bullet :: \varphi \\
\Gamma; \Theta \cup \{\mathsf{H} :: \exists_T X.\, \varphi\}; \Sigma \vdash \bullet :: \varphi & \rightsquigarrow & \Gamma; \Theta \cup \{\mathsf{H} :: \varphi\{X \mapsto \mathsf{x}\}\}; \Sigma \cup \{\mathsf{x} :: T\} \vdash \bullet :: \varphi \\
 & \text{when} & \mathsf{x} \notin \mathrm{FV}(\Gamma; \Theta; \Sigma)
\end{array}
$$

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.

$$
\begin{array}{rcl}
\multicolumn{1}{c}{\texttt{ex\_falso}} & & \\
\Gamma; \Theta; \Sigma \vdash \bullet :: \varphi & \rightsquigarrow & \Gamma; \Theta; \Sigma \vdash \bullet :: \bot
\end{array}
$$

$$
\begin{array}{rcl}
\multicolumn{1}{c}{\texttt{generalize x}} & & \\
\Gamma; \Theta; \Sigma \vdash \bullet :: \varphi & \rightsquigarrow & \Gamma; \Theta; \Sigma' \vdash \bullet :: \forall_T \mathsf{x}.\, \varphi \\
 & \text{when} & \Sigma = \Sigma' \cup \{\mathsf{x}\} \text{ and } \mathsf{x} \notin \mathrm{FV}(\Gamma)
\end{array}
$$

$$
\begin{array}{rcl}
\multicolumn{1}{c}{\texttt{by\_induction x IH}} & & \\
\Gamma; \Theta; \Sigma \vdash \bullet :: (\forall_T X.\, \varphi(X)) & \rightsquigarrow & \Gamma; \Theta \cup \{\mathsf{IH} :: \psi\}; \Sigma \cup \{\mathsf{x} :: T\} \vdash \bullet :: \varphi(X) \\
 & \text{where} & \psi := \forall_T \mathsf{x}.\, [\mathsf{x} \prec X] \to \varphi(\mathsf{x})
\end{array}
$$

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

$$
\begin{array}{rcl}
\multicolumn{1}{c}{\texttt{compare\_atoms a b}} & & \\
\Gamma; \Theta; \Sigma \vdash \bullet :: \varphi & \rightsquigarrow & \Gamma; \Theta; \Sigma \vdash \bullet :: (\mathsf{a} = \mathsf{b} \vee \mathsf{a} \neq \mathsf{b}) \to \varphi
\end{array}
$$

$$
\begin{array}{rcl}
\multicolumn{1}{c}{\texttt{get\_fresh\_atom a e}} & & \\
\Gamma; \Theta; \Sigma \vdash \bullet :: \varphi & \rightsquigarrow & \Gamma \cup \{\mathsf{a} \# \mathsf{e}\}; \Theta; \Sigma \cup \{\mathsf{a} :: A\} \vdash \bullet :: \varphi \\
 & \text{where} & \mathsf{a} \notin \mathrm{FV}(\Gamma; \Theta; \Sigma)
\end{array}
$$

We also provide shorthand formuals 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))
```

To state the theorem of *progress*, we will naturally need the predicate that a term is *progressive*:

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

We will also require 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`).

```
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))
```

As well as some boilerplate lemmas:

```
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')
```

Lets begin with the proof of *canonical forms*:

```
let canonical_form =
  proof' canonical_form_thm
  |> intros ["v"; "t"; "Hv"; "Ht"]
(* Proof state:
[ ]
[ Ht : Typing v nil t ;
  Hv : Value v
]
⊢ ∃ a :atom. ∃ e :term. [v = lam (a.e)] ∧ Term e
*)
```

The proof will follow from case analysis of `Typing` relation, so let's *destruct* assumption `Ht` and consider the first case, where `v` is some variable `a`. This case is impossible in empty environment, so we named the assumption `contra` and show it through the tactic `ex_falso`.

```
  |> destruct_assm "Ht"
  |> intros' ["contra"; "a"; ""]
     %> ex_falso
(* Proof state:
[ v = a ]
[ Hv : Value v ;
  contra : InEnv nil a t
]
⊢ ⊥
*)
     %> apply_thm_specialized empty_contradiction ["a"; "t"]
        (* InEnv nil a t  ⟹  ⊥ *)
     %> apply_assm "contra"
```

Next case is the only sensible one: that `v` is some `lam (a.e)` of type `arrow t1 t2`.

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

Now, obviously every term that *types* is indeed a proper *term*, so we simply use the `typing_terms` lemma and we're done here.

```
     %> apply_thm_specialized typing_terms ["e"; "cons a t1 nil"; "t2"]
        (* Typing e {cons a t1 nil} t2  ⟹  Term e *)
     %> assumption
```

Final case is that `e` is an application, but then it can't be a value, so we analyse the `Hv` assumption, arriving at contradiction in either case:

```
  |> intros' ["contra"; "e1"; "e2"; "t2"; ""]
     %> ex_falso
     %> destruct_assm "Hv"
(* Proof state:
[ v = app e1 e2 ]
[ contra : Typing e1 nil {arrow t2 t} ∧ Typing e2 nil t2 ]
⊢ (∃ a : atom. v = a) ⟹ ⊥
*)
     %> intros' ["contra_var"; "a"]
     %> discriminate
(* Proof state:
[ v = app e1 e2 ]
[ contra : Typing e1 nil {arrow t2 t} ∧ Typing e2 nil t2 ]
⊢ (∃ a : atom. ∃ e' : term. v = lam (a.e)) ⟹ ⊥
*)
     %> intros' ["contra_lam"; "a"; "e"; ""] %> discriminate
     %> discriminate
  |> qed
```

Now we can proceed with the proof of *progress*, 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 again 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
```

Now, 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)
```

To maintain reader engagement and prevent excessive technicality, we will omit here the proofs of rather obvious lemmas 2 and 3 and instead focus on the more important lemma 1:

```
let sub_lemma =
  proof' sub_lemma_thm
  |> by_induction "e0" "IH"
     %> repeat intro %> intros ["Hv"; "He"; "Hsub"]
(* Proof state:
[ ]
[
  He : Typing e {cons a ta env} t ;
  Hsub : Sub e a v e' ;
  Hv : Typing v env ta ;
  IH : ∀ e0 : term. [e0 ≺ e] ⟹
        ∀ env'1 t'1 : term. ∀ a'1 : atom. ∀ ta'1 v'1 e''1 : term.
          Typing v'1 env'1 ta'1 ⟹
          Typing e0 {cons a'1 ta'1 env'1} t'1 ⟹
          Sub e0 a'1 v'1 e''1 ⟹
```

```
            Typing e''1 env'1 t'1
]
⊢ Typing e' env t
*)
%> destruct_assm "He"
```

First case is that `e` is some variable `b`, with first subcases that it is equal to `a` and substitutes to `v`:

```
  |> intros' ["Hb"; "b"; ""]
     %> destruct_assm "Hsub"
     %> ( intros' ["Heq"; ""; ""]
(* Proof state:
[ e = a ; e' = v ; e = b ]
[
  Hb : InEnv {cons a ta env} b t ;
  Hv : Typing v env ta ;
  ...
]
⊢ Typing e' env t
*)
```

Now because in the goal `e'` has type `t`, but in assumption `Hv` it has `ta`, then we again case-analyse the assumption `Hb` and get that either `t = ta` or arrive at contradiction:

```
        %> destruct_assm "Hb"
        %> ( intros' ["Heq"; "env'"; ""] (* t = ta *)
             %> apply_assm "Hv" )
        %> ( intros' ["Hdiff"; "b'"; "t'"; "env'"; ""; ""] (* a =/= b *)
             %> discriminate )
```

Second subcase is that `b` is be different than `a` and thus is not be affected by the subistution. We will again case-analyse `Hb` assumption to extract additional facts.

```
     %> ( intros' ["Hdiff"; "b'"; ""; ""; ""] (* a =/= b *)
        %> destruct_assm "Hb"
        %> ( intros' ["Heq"; "env'"; ""] (* a = b *)
             %> discriminate )
        %> ( intros' ["Hdiff"; "a'"; "ta'"; "env'"; ""; ""]
(* Proof state:
[  e = b ; e' = b ; a =/= b ; ... ]
[
  Hdiff : InEnv env' b t ;
  ...
]
⊢ Typing e' env t
*)
             %> case "var"
             %> exists "b"
             %> by_solver
             %> assumption )
```

Second case is that `e` is some abstraction `lam (b.e_b)`. Because of the way we defined subsitution, abstraction argument must be different than the substituted variable and not occur in the substitutee value — which is made possible by swapping atoms while maintaining alpha-equality. Consequence of that is when we destruct `Hsub` we get that `e = lam (c.e_c)` and `e' = lam (c.e_c')` — while `b.e_b` and `c.e_c` are equal, `b` and `c` don't have to be. Abstracting the mundane details to auxiliary lemmas allows us to present the derivation in a simple chain of applications and assumptions:

```
  |> intros' ["Hlam"; "b"; "e_b"; "t1"; "t2"; ""; ""; ""]
     %> destruct_assm "Hsub"
     %> intros' ["Hsub"; "c"; "e_c"; "e_c'"; ""; ""; ""; ""]
     %> case "lam"
     %> exists' ["c"; "e_c'"; "t1"; "t2"]
     %> repeat by_solver
(* Proof state:
[ e = lam (b.e_b) ; e = lam (c.e_c) ; e' = lam (c.e_c') ;
  a =/= c ; c # v ; t = arrow t1 t2 ]
[
  Hsub : Sub e_c a v e_c' ;
  Hlam_1 : Type t1 ;
  Hlam_2 : Typing e_b {cons b t1 (cons a ta env)} t2 ;
  Hv : Typing v env ta ;
  ...
]
⊢ Type t1 ∧ Typing e_c' {cons c t1 env} t2
*)
     %> destruct_goal
     %> assumption
     %> apply_assm_specialized
        "IH" ["e_c"; "cons c t1 env"; "t2"; "a"; "ta"; "v"; "e_c'"]
     (* [e_c ≺ e]  ⟹  Typing v {cons c t1 env} ta  ⟹
          Typing e_c {cons a ta (cons c t1 env)} t2  ⟹
            Sub e_c a v e_c'  ⟹  Typing e_c' {cons c t1 env} t2  *)
     %> by_solver
     %> ( apply_thm_specialized
            cons_fresh_typing ["v"; "env"; "ta"; "c"; "t1"]
            (* [c # v]  ⟹  Typing v env ta  ⟹
                 Typing v {cons c t1 env} ta *)
          %> by_solver
          %> apply_assm "Hv" )
     %> ( apply_thm_specialized
           typing_env_shuffle ["e_c"; "env"; "t2"; "c"; "t1"; "a"; "ta"]
           (* [c =/= a]  ⟹
                Typing e_c {cons c t1 (cons a ta env)} t2  ⟹
                  Typing e_c {cons a ta (cons c t1 env)} t2 *)
          %> by_solver
          %> apply_thm_specialized swap_lambda_typing
              ["b"; "e_b"; "c"; "e_c"; "cons a ta env"; "t1"; "t2"]
              (* [b.e_b = c.e_c]  ⟹
                   Typing e_b {cons b t1 (cons a ta env)} t2  ⟹
                     Typing e_c {cons c t1 (cons a ta env)} t2 *)
```

```
          %> by_solver
          %> apply_assm "Hlam_2" )
      %> apply_assm "Hsub"
```

Finally, we consider the case that `e` is an application `e1 e2`, which goes straightly from inductive hypothesis:

```
  |> intros' ["Happ"; "e1"; "e2"; "t2"; ""; ""]
      %> intros' ["Hsub"; "_e1"; "_e2"; "e1'"; "e2'"; ""; ""; ""]
      %> case "app"
      %> exists' ["e1'"; "e2'"; "t2"]
      %> by_solver
(* Proof state:
[ e = app e1 e2 ; e' = app e1' e2']
[
  Happ_1 : Typing e1 {cons a ta env} {arrow t2 t} ;
  Happ_2 : Typing e2 {cons a ta env} t2 ;
  Hsub_1 : Sub e1 a v e1' ;
  Hsub_2 : Sub e2 a v e2' ;
  ...
]
⊢ Typing e1' env {arrow t2 t} ∧ Typing e2' env t2
*)
      %> destruct_goal
      %> ( apply_assm_specialized
             "IH" ["e1"; "env"; "arrow t2 t"; "a"; "ta"; "v"; "e1'"]
             (* [e1 ≺ e] ⟹
                  Typing v env ta ⟹
                    Typing e1 {cons a ta env} {arrow t2 t} ⟹
                      Sub e1 a v e1' ⟹
                        Typing e1' env {arrow t2 t} *)
          %> by_solver
          %> apply_assm "Hv"
          %> apply_assm "Happ_1"
          %> apply_assm "Hsub_1" )
      %> ( apply_assm_specialized
             "IH" ["e2"; "env"; "t2"; "a"; "ta"; "v"; "e2'"]
             (* [e2 ≺ e] ⟹
                  Typing v env ta ⟹
                    Typing e2 {cons a ta env} t2 ⟹
                      Sub e2 a v e2' ⟹
                        Typing e2' env t2 *)
          %> by_solver
          %> apply_assm "Hv"
          %> apply_assm "Happ_2"
          %> apply_assm "Hsub_2" )
  |> qed
```

Now that we've shown the `sub_lemma`, we can go on with the final proof of *preservation*. The proof goes through induction on term `e` the case analysis on assumption `Steps e e'`.

```
let preservation = 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.

```
  |> intros' ["He1"; "e1"; "e1'"; "e2"; ""; ""]
     %> 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
       %> (apply_assm_specialized "IH" ["e1"; "e1'"; "env"; "arrow t2 t"]
            (* [e1 ≺ e] ⟹
                 Typing e1 env {arrow t2 t} ⟹
                   Steps e1 e1' ⟹
                     Typing e1' env {arrow t2 t} *)
            %> by_solver
            %> apply_assm "Happ_1"
            %> apply_assm "He1" )
       %> apply_assm "Happ_2"
  |> intros' ["He2"; "v1"; "e2"; "e2'"; ""; ""; ""]
     %> 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
      %> apply_assm "Happ_1"
      %> ( apply_assm_specialized "IH" ["e2"; "e2'"; "env"; "t2"]
           (* [e2 ≺ e] ⟹
                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"; ""; ""]
(* 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
```

And that's it.

Chapter 7

Conclusion

In summary, we've introduced and demonstrated a specialized variant of Nominal Logic, designed for reasoning about variable binding through the utilization of constraints solving. We've also successfully implemented this logic in OCaml, complemented by essential tools, including a proof assistant.

Through the proofs of classical properties of simply typed lambda calculus we have validated the logic's suitablility for reasoning about programming languages theory. However, the true potential of this framework is expected to shine when applied to specific theorems reliant on the notions of variable binding. One such area would be the theory of bisimulation, where we expect that ??? would provide signifact aid in proving ???.

We must also acknowledge that our framework is still in its infancy, requiring substantial refinement to ensure a user-friendly experience, as the awkardness and low-level nature of the current tooling obscures the benefits of underlying constraint-based sublogic. Consequently, it cannot be directly compared to other theorem-proving frameworks like Coq or Agda.

Nonetheless, we are confident that with enough refinement, our framework can prove to be a valuable resource for the specific use cases and remain enthusiastic about the framework's potential to contribute to the field of formal methods and logic-based reasoning.

Bibliografia

[1] "A New Approach to Abstract Syntax with Variable Binding". W: *Formal Aspects of Computing* 13.3 (2002), s. 341–363. DOI: 10.1007/S001650200016.

[2] N.G de Bruijn. "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem". W: *Indagationes Mathematicae (Proceedings)* 75.5 (1972), s. 381–392. ISSN: 1385-7258. DOI: 10.1016/1385-7258(72)90034-0.

[3] Andrew M. Pitts. "Nominal logic, a first order theory of names and binding". W: *Information and Computation* 186.2 (2003). Theoretical Aspects of Computer Software (TACS 2001), s. 165–193. ISSN: 0890-5401. DOI: 10.1016/S0890-5401(03)00138-X.