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

19 października 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[2] 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.

> **Andrew M. Pitts**, *"Nominal logic, a first order theory of names and binding"*[2]:
>
> 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 chose to found his theory around the notion of swapping names as opposed to the classical renaming. In the author's previous work[6], written together with Murdoch J. Gabbay, it was shown that a theory based on this operation allows for all necessary concepts, including alpha-equivalence, freshness, and variable-binding, to be defined solely in terms of swapping pairs of names.

Additionaly, swapping has one other useful logical properties - it is involutive (i.e. a swap gets nullified by applying the same swap again, while substitutions cannot always be reversed), which, as Pitts argues, means that *equivariant* predicates (i.e. those whose validity is invariant under atom-swapping) have excellent logical properties. This class of equivariant predicates includes equality, alpha-equivalence and is closed under standard logical connectives, universal and existential quantification, and formation of least and greatest fixpoint.

As an example of Nominal Logic at work, consider the abstract syntax tree of untyped lambda calculus, given by the grammar below, where a ranges over an infinite set of names — or rather *atoms*.

$$t ::= a \mid \lambda a.t \mid t\ t \qquad \text{(lambda terms)}$$

Figure 1.1: Terms of untyped lambda calculus

The definition of swapping atoms a and b in some tree t, written $(a\ b)\ t$, is rather straightforward. It naturally follows the tree structure, touching only the affected

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

Figure 1.2: Swapping procedure

atoms, and doesn't need to distinct betwen free and bound names (like substitutions do), but simply changes them all the same exact way.

Relation of *freshness* of atom a in tree t, written $a \# t$, is similarly simple to

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

Figure 1.3: Freshness relation

define.[1] Note that it only assumes the comparability of atoms and is an *equivariant* relation, which can be shown by simplest induction.

With *swapping* and *freshness* already established, we define the alpha-equivalence of terms, written $t_1 =_\alpha t_2$.

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

Figure 1.4: Alpha-equivalence relation

We built this definition of alpha-equivalence using only induction, swapping, and freshness then, as Pitts argues, it is equivariant as well.

Andrew M. Pitts, *"Nominal logic, a first order theory of names and binding"*[2]:

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, such as alpha equivalence and freshness, as *constraints*. We introduce *the Solver*, an algorithm designed to automatically resolve new constraints based on the pre-established ones. It serves as the logical core of the constraints sublogic, that together with the embedding of constraints into propositional formulas constructs a higher-order logic capable of seamlessly expressing these properties. This approach we've taken, liberates users from the painstaking task of manually proving the seemingly trivial but crucial details, through automated resolution of constraints, while ensuring the completeness and correctness of written proofs.

For the user interface, we have developed a proof checker and proof assistant,

[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.

tying all the parts together in a cohesive framework. The proof assistant draws inspiration from the HOL family of theorem provers, initially introduced by Michael J. C. Gordon[7]. Similar to HOL, it utilizes the OCaml programming language as the interface to writing proofs and encoding theorems. While currently somewhat low-level, with further automation efforts, it should achieve intuitiveness and user-friendliness akin to other, more mature and powerful proof assistants.

1.3 Related work

Of course, there's other works that focus on reasoning about syntactical properties of binders, as they are essential in formalizing properites of programming languages.

- **Higher-Order Abstract Syntax** (HOAS) introduced by Frank Pfenning and Conal Elliott[9] is a uniform and generic representation of terms, formulas, programs, and other syntactic objects used in formal reasoning systems that focus on substitution and unification under the presence of binders. Authors utilize the binding construct of the implementation language to represent the binding in the language being formalized.

- **Beluga** is a programming framework designed for reasoning about formal systems. Based on the LF logical framework, it encodes HOAS approach using dependent types and provides support for reasoning with context and contextual objects. It's developed at the Complogic group at McGill University, led by Professor Brigitte Pientka[11].

- **Twelf** is a framework used to specify, implement, and prove properties of deductive systems and logics. It's based on the LF logical framework, and uses Elf constraint logic programming langauge. The principal authors of Twelf are Frank Pfenning, and Carsten Schürmann[10]. Multiple reasearch projects were developed using Twelf, including a type safety proof for Standard ML[8].

- **Parametric Higher-Order Abstract Syntax** (PHOAS) improves on the idea of HOAS, by utilizing dependently-typed abstract syntax trees to formalize it in general-purpose type theories, like Coq's Calculus of Inductive Constructions. Introduced by Adam Chlipala[4], it has been used to develop certified, executable program transformations over several formalizations of statically-typed functional programming languages.

- **Locally Nameless Representation** is an approach to representation of syntax with variable binders, introduced by Arthur Charguéraud[3]. It represents the bound variables through de Bruijn indices, while retaining names of the free variables, achieving strong induction principles. Utilizing the Coq library TLC developed by Charguéraud, the approach has successfully formalized diverse type systems and semantics.

- **Autosubst**[12] is a Coq library that automates some crucial parts of formalizing syntactic theories with variable binders, developed by Steven Schäfer, Tobias Tebbi, and Gert Smolka. Authors employ de Bruijn representation of terms with additional binding annotations to automatically derive the substitution operation and proofs of substitution lemmas. They introduce an automation tactic that solves equations involving terms and substitutions, based on their work on the decision procedure of equational theory of an extension of the sigma-calculus by Abadi et al[1].

Chapter 2

Terms and constraints

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

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.

The constraints are precise descriptions of syntactical properties, describing the relationship between their arguments — atoms and terms.

π	$::=$	$\mathsf{id} \mid (\alpha\ \alpha)\pi$	(permutations)
α	$::=$	$\pi\ a$	(atom expressions)
t	$::=$	$\alpha \mid \pi\ X \mid \alpha.t \mid t\ t \mid s$	(terms)
c	$::=$	$\alpha \# t \mid t = t \mid t \sim t \mid t \prec t \mid \text{symbol } t$	(constraints)

Figure 2.1: Syntax of constraint sublogic

Construction $\alpha.t$ represents a *binder* — informally, we think of it as binding the occurences of α in t, similarly to a lambda abstraction — yet it *isn't* a binder, but a simple syntactical construction glueing together an atom with another term. The semantics of binding will apply only after we interpret this syntactical term in the model.

Also note that we do not restrict this construction to the form of $a.t$, but allow permuted atoms to appear under 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 atom expressions *pure*. The same rules apply to permuted variables.

$\alpha \# t$	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.

Figure 2.2: Informal semantics of constraints

It's important to note that these terms and constraints are merely a data structure and do not incorporate notions of computation, reduction, or binding by themselves. These properties only appear in the sublogic of constraints after we interpret constraints within the logical model, which allows us to then reason about concepts such as *freshness*, *variable binding*, and *structural* order.

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 infinite set of names, representing the free variables.

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

Figure 2.3: Semantic representation of terms and shapes

Binders in semantic terms are achieved by De Bruijn indices[5] and consequently the bound names are represented by natural numbers, denoted by n, and the binding construction has no explicit argument, denoted by \$.
The term interpretation function, denoted $[\![\cdot]\!]_\rho$, maps syntactic terms to semantic terms, utilizing the standard shifting of De Bruijn indices (denoted by $\uparrow$). It is parametrized by function ρ that maps atoms and variables to semantic shapes.
The shape interpretation function, denoted $|\cdot|$, maps semantic terms to semantic shapes by erasing names.

With above machinery, we can establish relation $\rho \vDash c$ that interprets the constraints in our model, using some mapping ρ.
Note that the freshness can be expressed through membership check of $\mathsf{FreeAtoms}$

$$
\begin{aligned}
[\![\pi\ a]\!]_\rho &= [\![\pi]\!]_\rho(\rho(a)) \\
[\![\pi\ X]\!]_\rho &= [\![\pi]\!]_\rho(\rho(X)) \\
[\![\alpha.t]\!]_\rho &= \$([\![t]\!]_\rho\uparrow)\{[\![\alpha]\!]_\rho \mapsto 0\} \\
[\![t_1\ t_2]\!]_\rho &= [\![t_1]\!]_\rho @ [\![t_2]\!]_\rho \\
[\![s]\!]_\rho &= s
\end{aligned}
\qquad
\begin{aligned}
|A| &= \_ \\
|n| &= \_ \\
|\$T| &= \$|T| \\
|T_1 @ T_2| &= |T_1| @ |T_2|
\end{aligned}
$$

Figure 2.4: Interpretation of terms and shapes in the model

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

Figure 2.5: Constraint interpretation in the model

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 present the deterministic *Solver* algorithm that emulates this model by syntatically verifying statements of form $\Gamma \vDash c$.

Chapter 3

Constraint solver

At the heart of our work lies the Solver, an algorithm designed to resolve constraints. For any assumed constraints $c_1, \ldots, c_n$, and goal constraint c_0, the Solver determines whether judgment $c_1, \ldots, c_n \vDash c_0$ holds. Meaning that for every possible substitution of variables into closed terms in constraints $c_0, c_1, \ldots, c_n$, such that $c_1, \ldots, c_n$ are satisfied, would 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. It erases atoms in terms under shape constraints, effectively transforming them into *shapes*. We will write $a \neq \alpha$ instead of $a \# \alpha$ as it gives a clear intuition of atom freshness implying inequality.

$\mathcal{C}$	$::=$	$\alpha \# t \mid t = t \mid \mathcal{S} \sim \mathcal{S} \mid \mathcal{S} \prec \mathcal{S} \mid \text{symbol } t$	(solver constraints)
$\mathcal{S}$	$::=$	$\_ \mid X \mid \_.\mathcal{S} \mid \mathcal{S}\,\mathcal{S} \mid s$	(shapes)

Figure 3.1: Solver internal representation of terms and shapes

A high level perspective of the Solver is that it works on judgments of form $\Gamma; \Delta \vdash \mathcal{C}$, veryfying whether a given goal-constrint $\mathcal{C}$ holds in environments of assumed constraints (kept in Γ and Δ) through dissecting constraints on both sides of the turnstile into irreducible components that are straightforward to handle. En-

$a_1 \neq a_2$	Atoms a_1 and a_2 are different.
$a \# X$	Atom a is Fresh in variable X.
$X_1 \sim X_2$	Variables X_1 and X_2 posses the same shape.
$X \sim t$	Variable X has a shape of term t.
$t \prec X$	Term t strictly subshapes variable X.
symbol X	Variable X is some functional symbol.

Figure 3.2: Irreducible constraints

vironment Γ keeps the yet unprocessed assumptions, while another environment Δ keeps track of already analysed and irreducible assumptions. These assumptions usually flow from the former to the latter, but if we analyse a constraint that that affects other assumptions in Δ, they may flow back to Γ to be further disected by the Solver.

After all assumptions in Γ are reduced to irreducible constraints, we break down the goal-constraint $\mathcal{C}$ and repeat the reduction procedure on new assumptions and goal.

$$\frac{}{\Gamma; \lightning \vdash \mathcal{C}} \qquad \frac{\mathcal{C} \text{ is trivial}}{\Gamma; \Delta \vdash \mathcal{C}} \qquad \frac{\mathcal{C} \in \Delta}{\Gamma; \Delta \vdash \mathcal{C}}$$

Figure 3.3: Base cases of the Solver's judgement

This recursive procedure may stop at a contradictory environment $\lightning$, that short-cuircuts the procedure, or at a state in which all the assumptions and goal itself are reduced to irreducible components, which is then as simple as checking if the goal is trivial or if it occurs on the left side of the turnstile.

3.1 Goal-reducing rules

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

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

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

Figure 3.4: Equality-reduction rules

Checking equality of terms is rather straightforward and follows from the term structure if no permutations are involved. Only the case for abstraction terms is more involved: the left side's argument must be fresh in the whole right side's term (which informally means that either arguments are the same or the left's argument doesn't occur at all in the right's body) and that left body must be equal to the right body with if its argument was swapped for the left one.

To compare a *pure* atom a with permuted one, we employ the decidability of atom equality to reduce the right hand-side's permutation by applying it's outermost swap $(\alpha_1\ \alpha_2)$ on the left side's atom. There's three possible cases:

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.

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

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

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

$$\mathsf{id}^{-1}\ t := \mathsf{id}\ t \qquad ((\alpha_1\ \alpha_2)\pi)^{-1}\ t := \pi^{-1}((\alpha_1\ \alpha_2)\ t)$$

Figure 3.5: Permutation-reduction rules

If the left-hand side's term is permuted we move the permutation to the right-hand side by inverting it. There's also special check for variables equal to their permuteded selves — we check whether that permutation is idempotent on them.

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

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

Figure 3.6: Freshness-reduction rules

Freshness follows the term structure and is using assumptions from Δ environment. Unlike to how we defined freshness in abstraction in the introduction, we do not have two rules that differencing on whether $a = \alpha$. If they are indeed equal, then the assumption of inequality will immediately result in contradiction of environment, but if it wasn't yet established then we continue the solver procedure with an additional assumption.

Shape equality is naturally structural. All atoms and only equal symbols are considered to have the same shape. Variables can share shape and be have their

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

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

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

Figure 3.7: Shape rules

shape stored by Δ, which enables transitivity.

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

Figure 3.8: Subshape rules

Solving subshape recurses through right-hand side shape's structure to find a shape-equal sub-shape. Environment Δ keeps track of all shapes that given variable subshapes, enabling transitivity.

$$\frac{}{\Gamma; \Delta \vdash \text{symbol } s} \qquad \frac{\text{symbol } X \in \Delta}{\Gamma; \Delta \vdash \text{symbol } X}$$

Figure 3.9: Symbol rules

Symbol constraints are really simple to check, either the term is already a symbol, or it is a variable that we already assumed to be a symbol.

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.

$$\frac{X = \pi^{-1}t, \Gamma; \Delta \vdash \mathcal{C}}{\pi X = t, \Gamma; \Delta \vdash \mathcal{C}} \qquad \frac{a = \pi^{-1}\alpha, \Gamma; \Delta \vdash \mathcal{C}}{\pi a = \alpha, \Gamma; \Delta \vdash \mathcal{C}} \quad \textsc{Perm Reduce}$$

$$\frac{\pi \text{ idempotent on } X, \Gamma; \Delta \vdash \mathcal{C}}{X = \pi X, \Gamma; \Delta \vdash \mathcal{C}} \quad \textsc{Perm Idempotent}$$

$$\frac{\varnothing \vdash \text{ idempotent on } X \qquad \Gamma; \Delta \vdash \mathcal{C}}{\pi \text{ idempotent on } X, \Gamma; \Delta \vdash \mathcal{C}} \quad \textsc{Perm Shortcircuit}$$

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

Figure 3.10: Permutation-reducing rules

For variables equal to some term and atoms equal to some atom expressions, we first deal with permutation by inverting it and moving it to the right-hand side. Then 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. Otherwise both equality and freshness assumptions follow from the term structure.

$$\frac{\alpha_1 \,\#\, \alpha_2.t_2\ ,\ t_1 = (\alpha_1\ \alpha_2)t_2\ ,\ \Gamma; \Delta \vdash \mathcal{C}}{\alpha_1.t_1 = \alpha_2.t_2, \Gamma; \Delta \vdash \mathcal{C}} \qquad \frac{\begin{array}{c} a = \alpha,\ \Gamma; \Delta \vdash \mathcal{C} \\ a \neq \alpha,\ a \,\#\, t,\ \Gamma; \Delta \vdash \mathcal{C} \end{array}}{a \,\#\, \alpha.t, \Gamma; \Delta \vdash \mathcal{C}}$$

Figure 3.11: Binding assumption rules

Again, the binding term constructor is of most interest to use: equality behaves the same as on the goal side, we simply split up the assumption into two assumptions the same way we would split the goal. For freshness of an atom in an abstraction, we consider two cases: either the atom is equal to the argument, or different from the argument but fresh in the body. In constrast to the goal-reducing rules where we would be satisifed with just one branch successing, here we expect both possibilities to be satisfiable.

$$\frac{\Gamma\{X \mapsto t\}; \Delta\{X \mapsto t\} \vdash \mathcal{C}\{X \mapsto t\}}{X = t, \Gamma; \Delta \vdash \mathcal{C}}\ \textsc{Subst Term}$$

$$\frac{\Gamma\{a_1 \mapsto a_2\}; \Delta\{a_1 \mapsto a_2\} \vdash \mathcal{C}\{a_1 \mapsto a_2\}}{a_1 = a_2, \Gamma; \Delta \vdash \mathcal{C}}\ \textsc{Subst Atom}$$

Figure 3.12: Substitution rules

In the end, all assumptions reach the irreducible components that are handled through the special environment Δ enviroment. Equality assumptions reduce to substitution of the name for the expression, and while substitution over the environment Γ and goal $\mathcal{C}$ is indeed a simple substitution, substituting in Δ environment is a more involved process that can can arrive at a contradiction. Otherwise assumption are

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

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

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

Figure 3.13: Moving irreducible assumptions inside Δ

simply moved to the environment of irreducible constraints via procedure that we describe in the next section.

3.3 Irreducible constraints

Environment Δ that containts all the irreducible assumptions is given by a sextuple $(\texttt{neq\_atoms}_\Delta, \texttt{fresh}_\Delta, \texttt{var\_shape}_\Delta, \texttt{shape}_\Delta, \texttt{subshape}_\Delta, \texttt{symbols}\Delta)$.

`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`	Set of pairs of shape-representative variables and shapes that subshape the variable.
`symbols`	Set of shape-representative variables that are known to be some unknown functional symbols.

Figure 3.14: Description of environment Δ

$X_\Delta :=$
  | $\texttt{if}\ Y \leftarrow \texttt{var\_shape}_\Delta\ X\ \texttt{then}\ Y_\Delta$
  | $\texttt{otherwise}\ X$

$|X|_\Delta :=$
  | $\texttt{if}\ Y \leftarrow \texttt{var\_shape}_\Delta\ X\ \texttt{then}\ |Y|_\Delta$
  | $\texttt{if}\ \mathcal{S} \leftarrow \texttt{shape}_\Delta\ X\ \texttt{then}\ \mathcal{S}$
  | $\texttt{otherwise}\ X$

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

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

Figure 3.15: Shape and assumptions interpretation in Δ

We can now establish a method to compute the shape-representative variable and outline the procedure for reconstructing the shape within the environment Δ, denoted $|\mathcal{S}|_\Delta$. With such environment structure, verifying whether a constraint is included in Δ can be accomplished straightforwardly.

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

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

Figure 3.16: Occurs check rules

Additionally, we establish rules for a special occurs check procedure, which safeguards against handling circular references, and does so while considering all occurences in the assumptions of Δ. This is needed because of the shape assumptions we introduced, we must go with the occurence check through the "shape-similar" variables and shapes.

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')


{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

{X ~ S} ∪ Δ :=
  | if X_Δ occurs in |S|_Δ then ↯
  | otherwise Δ |> symbols  {X_Δ ⇝ |S|_Δ}
                |> subshape {X_Δ ⇝ |S|_Δ}
                |> shape    {X_Δ ⇝ |S|_Δ}
```

Figure 3.17: Adding constraints to Δ

We are using OCaml's pipelining notation of `x |> f1 |> ... |> fn` for `fn (... (f1 x))` and treat expressions like `fresh += x` as functions, meaning `fun Δ -> { Δ with fresh = x :: Δ.fresh }` and alike.

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.

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.

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*.

```
Δ {X ↦ t} :=
  Δ |> fresh -= X
    |> assumptions += (X ∼ |t|_Δ)
    |> assumptions += ⋃_{(a # X)∈Δ} (a # t)

Δ {a ↦ a′} :=
  Δ |> fresh -= a
    |> fresh += (a′ # fresh_Δ a)
    |> clear neq_atoms
    |> assumptions += ⋃_{(a₁≠a₂)∈Δ} (a₁{a ↦ a′} ≠ a₂{a ↦ a′})
```

Figure 3.18: Substitution in Δ

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

```
symbols {X ⇝ S} Δ :=
  | if X_Δ ∉ symbols_Δ then Δ
  | otherwise Δ |> symbols -= X
                |> assumptions += (symbol S)

shape {X ⇝ S} Δ :=
  | if S′ ← shape_Δ X then Δ |> assumptions += (S ∼ S′)
  | otherwise Δ |> shapes += (X ↦ S)

subshape {X ⇝ S} Δ :=
  Δ |> assumptions += (subshapes_Δ X ≺ S)

transfer_shape {X ⇝ X′} Δ :=
  | if S ← shape_Δ X then Δ |> shape {X′ ⇝ S}
  | otherwise Δ
```

Figure 3.19: Auxiliary functions in Δ

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 $\Gamma; \emptyset \vdash \mathcal{C}$, as by the construction of $\vdash$ we consider it equivalent to $\vDash$ 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

We introduce kinds to ensure that the formulas we will deal with are *making sense*, due to the multiple ways atoms, terms, binders, and constraints can occur in them.

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

Figure 4.1: Kinds grammar

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.

$\varphi :: \star$	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.

Figure 4.2: Kinds semantics

4.2 Subkinding

We relax kinding rules are through the *subkinding* relation. Function kind is con-

$$\frac{}{\Gamma \vdash \kappa <: \kappa}\ \textsc{Subkind Refl} \qquad \frac{\Gamma \vdash \kappa_1 <: \kappa_2 \quad \Gamma \vdash \kappa_2 <: \kappa_3}{\Gamma \vdash \kappa_1 <: \kappa_3}\ \textsc{Subkind Trans}$$

$$\frac{\Gamma \vdash \kappa_1 <: \kappa_2}{\Gamma \vdash \forall_A a.\, \kappa_1 <: \forall_A a.\, \kappa_2}\ \textsc{Subkind ForallAtom}$$

$$\frac{\Gamma \vdash \kappa_1 <: \kappa_2}{\Gamma \vdash \forall_T X.\, \kappa_1 <: \forall_T X.\, \kappa_2}\ \textsc{Subkind ForallTerm}$$

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

$$\frac{\Gamma \vDash c}{\Gamma \vdash [c]\kappa <: \kappa}\ \textsc{Subkind Reduce} \qquad \frac{\Gamma, c \vdash \kappa_1 <: \kappa_2}{\Gamma \vdash \kappa_1 <: [c]\kappa_2}\ \textsc{Subkind Guard}$$

Figure 4.3: Subkinding Rules

travariant to the subkinding relation on the left argument: Universally quantified kinds only subkind if they are quantified over the same name. Constraints from the left side that are solved through $\vDash$ relation can be dropped, and constraints from the right-hand side can be moved inside of the enviroment.

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

Note that there is no structural subkinding rule for guarded kinds like the one above, but such a rule can be derived from SubkindReduce, SubkindGuard, transitivity, and weakening.

4.3 Formulas

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

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

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

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

Propositional variables, functions and applications:

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

Constraints and guards:

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

$$\frac{(P :: \kappa) \in \Sigma}{\Gamma; \Sigma \vdash P :: \kappa} \ \textsc{Kind Var} \qquad \frac{}{\Gamma; \Sigma \vdash c :: \star} \ \textsc{Kind Constr}$$

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

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

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

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

Figure 4.4: Kinding rules

Naturally, constraints can act as propositions, as we can reason about their validity, and thus they are of kind $\star$. Constructions $[c] \Longrightarrow \varphi$ and $[c] \wedge \varphi$ are called *guards* and make assumptions about the environment in which one shall interpret the guarded formula. The former states that the formula φ holds if the constraint c is valid, analogously to a propositional implication. The latter additionaly requires that c already holds. We can see how guards interact with kinding rules after we define fixpoint.

4.4 Fixpoint

We finish the definition of formulas with *fixpoint* function that allows us to write recursive predicates over terms:

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

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

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

Figure 4.5: Fixpoint kinding rule

By the kinding rules, the fixpoint can only be recursively applied on structurally smaller terms, which is expressed 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. This way we enable the kind-checker to verify the soundness of fixpoint formulas and enforce usage of special guard formulas resembling implications and conjunctions. Because the applied term is finite and we always recurse on structurally smaller terms, the final formula after all substitutions must also be finite and safe —– 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 ($Nat\ N$) that a term N models some natural number, and ($PlusEq\ N\ M\ K$) that two terms N and M added together are equal to K.

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

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

Figure 4.6: Peano arithmetic predicates expressed with fixpoint

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 ($Nat :: \forall_T N.\, \star$), analogously in $PlusEq$. This is exactly the reason of introducing kinds — to allow us to use recursive predicates in a safe and sound fashion.

See more interesting examples of fixpoints usage in the chapter on STLC.

4.5 Natural deduction

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

$$\frac{\Gamma \vDash c}{\Gamma; \Theta \vdash c}\ \textsc{ConstrI} \qquad \frac{\Gamma \vDash \bot_c}{\Gamma; \Theta \vdash \varphi}\ \textsc{ConstrE}$$

$$\frac{\Gamma; \Theta \vdash \varphi_1}{\Gamma; \Theta \vdash \varphi_1 \vee \varphi_2}\ \textsc{OrI1} \qquad \frac{\Gamma; \Theta \vdash \varphi_2}{\Gamma; \Theta \vdash \varphi_1 \vee \varphi_2}\ \textsc{OrI2}$$

$$\frac{\begin{array}{c}\Gamma; \Theta \vdash \varphi_1 \vee \varphi_2 \\ \Gamma; \Theta, \varphi_1 \vdash \psi \qquad \Gamma; \Theta, \varphi_2 \vdash \psi\end{array}}{\Gamma; \Theta \vdash \psi}\ \textsc{OrE}$$

Figure 4.7: 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 and equivalence to our description from the chapter about the Solver.

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.

$$\frac{\Gamma; \Theta, \varphi_1 \vdash \varphi_2}{\Gamma; \Theta \vdash \varphi_1 \Longrightarrow \varphi_2}\ \textsc{ImpI} \qquad \frac{\Gamma_1; \Theta_1 \vdash \varphi_1 \quad \Gamma_2; \Theta_2 \vdash \varphi_1 \Longrightarrow \varphi_2}{\Gamma_1 \cup \Gamma_2; \Theta_2 \cup \Theta_2 \vdash \varphi_2}\ \textsc{ImpE}$$

$$\frac{\Gamma, c; \Theta \vdash \varphi}{\Gamma; \Theta \vdash [c] \Longrightarrow \varphi}\ \begin{array}{l}\textsc{Constr}\\ \textsc{ImpI}\end{array} \qquad \frac{\Gamma_1; \Theta_1 \vdash c \quad \Gamma_2; \Theta_2 \vdash [c] \Longrightarrow \varphi}{\Gamma_1 \cup \Gamma_2; \Theta_2 \cup \Theta_2 \vdash \varphi}\ \begin{array}{l}\textsc{Constr}\\ \textsc{ImpE}\end{array}$$

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

$$\frac{\Gamma; \Theta \vdash \varphi_1 \wedge \varphi_2}{\Gamma; \Theta \vdash \varphi_1}\ \textsc{AndE1} \qquad \frac{\Gamma; \Theta \vdash \varphi_1 \wedge \varphi_2}{\Gamma; \Theta \vdash \varphi_2}\ \textsc{AndE2}$$

$$\frac{\Gamma_1; \Theta_1 \vdash c \quad \Gamma_2; \Theta_2 \vdash \varphi}{\Gamma_1 \cup \Gamma_2; \Theta_2 \cup \Theta_2 \vdash [c] \wedge \varphi}\ \begin{array}{l}\textsc{Constr}\\ \textsc{AndI}\end{array}$$

$$\frac{\Gamma; \Theta \vdash [c] \wedge \varphi}{\Gamma; \Theta \vdash c}\ \begin{array}{l}\textsc{Constr}\\ \textsc{AndE1}\end{array} \qquad \frac{\Gamma \vdash [c] \wedge \varphi \quad \Gamma; \Theta \vdash \varphi : \star}{\Gamma; \Theta \vdash \varphi}\ \begin{array}{l}\textsc{Constr}\\ \textsc{AndE2}\end{array}$$

Figure 4.8: Natural deduction for guard formulas

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. 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.

$$\frac{a \notin \mathrm{FV}(\Gamma;\Theta) \qquad \Gamma;\Theta \vdash \varphi}{\Gamma;\Theta \vdash \forall_A a.\,\varphi}\ \textsc{Forall AtomI} \qquad \frac{\Gamma;\Theta \vdash \forall_A a.\,\varphi}{\Gamma;\Theta \vdash \varphi\{a \mapsto a'\}}\ \textsc{Forall AtomE}$$

$$\frac{X \notin \mathrm{FV}(\Gamma;\Theta) \qquad \Gamma;\Theta \vdash \varphi}{\Gamma;\Theta \vdash \forall_T X.\,\varphi}\ \textsc{Forall TermI} \qquad \frac{\Gamma;\Theta \vdash \forall_T X.\,\varphi}{\Gamma;\Theta \vdash \varphi\{X \mapsto X'\}}\ \textsc{Forall TermE}$$

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

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

Figure 4.9: Natural deduction for quantifiers

Inference rules for quantifiers are rather straightforward, with the only novelty being that we differtiate between atom and term quantification, and restrict the quantified name to be *fresh* in the environment (it may not occur in any of the assumptions).

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

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

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

Figure 4.10: Axioms

The only axioms of our logic are strictly related to constraints:

1. We can deterministically compare any two atoms,
2. There always exists a fresh atom,
3. We can always deduce the structure of a term.

$$\frac{\Gamma \vDash a = \alpha \qquad \Gamma; \Theta \vdash \varphi}{\Gamma\{a \mapsto \alpha\}; \Theta\{a \mapsto \alpha\} \vdash \varphi\{a \mapsto \alpha\}} \ \textsc{Sub Atom}$$

$$\frac{\Gamma \vDash X = t \qquad \Gamma; \Theta \vdash \varphi}{\Gamma\{X \mapsto t\}; \Theta\{X \mapsto t\} \vdash \varphi\{X \mapsto t\}} \ \textsc{Sub Term}$$

$$\frac{\Gamma; \Theta \vdash \psi \qquad \Gamma; \Theta \vdash \psi \equiv \varphi}{\Gamma; \Theta \vdash \varphi} \ \textsc{Equiv}$$

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

Figure 4.11: Flexibility rules

To make the framework more flexible we introduce a way for using equivalent formulas: And a way to substitute atoms for atomic expression and variables for terms, if the solver can prove their equality: 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.

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.

$$\begin{array}{rl}
\texttt{compute}\ \Sigma\ n\ P \ \rightsquigarrow & \texttt{compute}\ \Sigma\ n\ \varphi \\
\text{when} & \Sigma(P) = \varphi \\
\\
\texttt{compute}\ \Sigma\ n\ (\varphi\, \alpha) \ \rightsquigarrow & \texttt{compute}\ \Sigma\ (n'-1)\ \varphi'\{a \mapsto \alpha\} \\
\text{when} & \texttt{compute}\ \Sigma\ n\ \varphi \ \rightsquigarrow^*\ (n', \lambda_A a.\, \varphi') \\
\\
\texttt{compute}\ \Sigma\ n\ (\varphi\, t) \ \rightsquigarrow & \texttt{compute}\ \Sigma\ (n'-1)\ \varphi'\{X \mapsto t\} \\
\text{when} & \texttt{compute}\ \Sigma\ n\ \varphi \ \rightsquigarrow^*\ (n', \lambda_T X.\, \varphi') \\
\\
\texttt{compute}\ \Sigma\ n\ (\varphi\, t) \ \rightsquigarrow & \texttt{compute}\ \Sigma\{P \mapsto \phi'\}\ (n'-1)\ \varphi'\{X \mapsto t\} \\
\text{when} & \texttt{compute}\ \Sigma\ n\ \varphi \ \rightsquigarrow^*\ (n', \text{fix}\ P(X) :: \kappa = \varphi') \\
\\
\texttt{compute}\ \Sigma\ n\ (\varphi_1\, \varphi_2) \ \rightsquigarrow & \texttt{compute}\ \Sigma\ (n_2-1)\ \psi_1\{P \mapsto \psi_2\} \\
\text{when} & \texttt{compute}\ \Sigma\ n\ \varphi_1 \ \rightsquigarrow^*\ (n_1, \lambda P :: \kappa.\, \psi_1) \\
\text{and} & \texttt{compute}\ \Sigma\ n_1\ \varphi_2 \ \rightsquigarrow^*\ (n_2, \psi_2)
\end{array}$$

Figure 4.12: Computing weak head normal form

Equivalence checking procedure starts by computing weak head normal form (up to some *depth* denoted by n). If we have a WHNF computed or if we've reached the limit of computation (when $n \leqslant 0$) then we try to progress with equivelnce by recursing on the structure of formulas:

$$\frac{\Gamma; \Sigma \vdash \varphi_1 \equiv \varphi_2 \qquad \Gamma; \Sigma \vdash \psi_1 \equiv \psi_2}{\Gamma; \Sigma \vdash \varphi_1 \Longrightarrow \psi_1 \equiv \varphi_2 \Longrightarrow \psi_2}$$

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

$$\frac{\begin{array}{c} X \notin \mathrm{FV}(\Gamma; \Sigma) \\ \Gamma; \Sigma \vdash \varphi_1[X_1 \mapsto X] \equiv \varphi_2[X_2 \mapsto X] \end{array}}{\Gamma; \Sigma \vdash \lambda_T X_1.\, \varphi_1 \equiv \lambda_T X_2.\, \varphi_2} \qquad \frac{\Gamma \vDash t_1 = t_2 \qquad \Gamma; \Sigma \vdash \varphi_1 \equiv \varphi_2}{\Gamma; \Sigma \vdash \varphi_1\ t_1 \equiv \varphi_2\ t_2}$$

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

Figure 4.13: Selected equivalence rules

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.

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.

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 stored in binders themselves (quantifiers and functions). Terms, constraints, kinds, and formulas are defined in `Types` module, mirroring their previously 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 string 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 natural deduction from previous chapter is distributed over modules `Proof`, `ProofEnv`, `ProofEquiv` and is a direct implementation of the described rules.

```
(* Module: Proof *)

(* ----------- *)
(*  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. Smart constructors of `proof` data type ensure correctness by implementing the descripted natural deduction and serve as the logical core for writing proofs.

Human provers, working within intuitionistic logic, generally prefer to conduct proofs not in this *bottom-up* fasion, but through simplifying the goal to be proven until we reach the trivial matters. To accomodate for that, we included the *top-down* proof structure in `incproof` data type. As such proofs have incomplete parts by nature, they must have *holes*, and live within some *proof context*, as defined in modules `IncProof`, which serves the role of being a convenient facade for writing proofs, while responsibility of keeping proofs correct is delegated to the `Proof` module.

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.

```
type prover_state = S_Unfinished of (goal * proof_context)
                  | S_Finished of proof

type tactic = prover_state → prover_state
```

Figure 5.1: Basic interface of the Prover.

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

$$
\begin{array}{rcl}
\texttt{proof}\ (\Gamma;\Theta;\Sigma)\ \varphi & \rightsquigarrow & \Gamma;\Theta;\Sigma \vdash \bullet :: \varphi \\
\\
\texttt{intro} & & \\
\Gamma;\Theta;\Sigma \vdash \bullet :: [c] \Longrightarrow \varphi & \rightsquigarrow & \Gamma, c;\Theta;\Sigma \vdash \bullet :: \varphi \\
\\
\texttt{intro' x} & & \\
\Gamma;\Theta;\Sigma \vdash \bullet :: \psi \Longrightarrow \varphi & \rightsquigarrow & \Gamma;\Theta, \texttt{x} :: \psi;\Sigma \vdash \bullet :: \varphi \\
\Gamma;\Theta;\Sigma \vdash \bullet :: \ \forall_A a.\, \varphi & \rightsquigarrow & \Gamma;\Theta;\Sigma, \texttt{x} :: a \vdash \bullet :: \varphi \\
\Gamma;\Theta;\Sigma \vdash \bullet :: \forall_T X.\, \varphi & \rightsquigarrow & \Gamma;\Theta;\Sigma, \texttt{x} :: X \vdash \bullet :: \varphi \\
\\
\texttt{apply}\ (\psi \Longrightarrow \varphi) & & \\
\Gamma;\Theta;\Sigma \vdash \bullet :: \varphi & \rightsquigarrow & \Gamma;\Theta;\Sigma \vdash \bullet :: \psi \\
 & \text{and} & \Gamma;\Theta;\Sigma \vdash \bullet :: \psi \Longrightarrow \varphi \\
\\
\texttt{apply\_assm H} & & \\
\Gamma;\Theta;\Sigma \vdash \bullet :: \varphi & \rightsquigarrow & \Gamma;\Theta;\Sigma \vdash \varphi \\
 & \text{when} & (\texttt{H} :: \varphi) \in \Theta \\
\Gamma;\Theta;\Sigma \vdash \bullet :: \varphi & \rightsquigarrow & \Gamma;\Theta;\Sigma \vdash \bullet :: \psi \\
 & \text{when} & (\texttt{H} :: \psi \Longrightarrow \varphi) \in \Theta \\
\\
\texttt{solve} & & \\
\Gamma;\Theta;\Sigma \vdash \bullet :: c & \rightsquigarrow & \Gamma;\Theta;\Sigma \vdash c \\
 & \text{when} & \Gamma \vDash c
\end{array}
$$

Figure 5.2: Basic 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_spec`) – or simply add any assumption to the current context together with a new goal (of proving that assumption) via `add_assumption`.

$$\begin{array}{rcl}
\texttt{apply_thm}\ \mathcal{T} & & \\
\Gamma;\Theta;\Sigma \vdash \bullet :: \varphi & \rightsquigarrow & \Gamma;\Theta;\Sigma \vdash \bullet :: \psi \\
& \text{where} & \mathcal{T} \text{ is a proof of } \psi \Longrightarrow \varphi \\
& & \\
\texttt{apply_assm_spec H [e; a]} & & \\
\Gamma;\Theta;\Sigma \vdash \bullet :: \varphi(\mathsf{e},\mathsf{a}) & \rightsquigarrow & \Gamma;\Theta;\Sigma \vdash \bullet :: \psi(\mathsf{e},\mathsf{a}) \\
& \text{when} & (\mathsf{H} :: \forall_T X.\, \forall_A a.\, \psi(X,a) \Longrightarrow \varphi(X,a)) \in \Theta \\
& & \\
\texttt{apply_in_assm H1 H2} & & \\
\Gamma;\Theta;\Sigma \vdash \bullet :: \psi & \rightsquigarrow & \Gamma;\Theta;\Sigma' \vdash \bullet :: \psi \\
& \text{when} & \Sigma = \{\mathsf{H1} :: \psi_2 \Longrightarrow \psi_1, \mathsf{H2} :: \psi_2\} \cup \Sigma'' \\
& \text{and} & \Sigma' = \{\mathsf{H1} :: \psi_1, \mathsf{H2} :: \psi_2\} \cup \Sigma''
\end{array}$$

Figure 5.3: More ways to use `apply` tactic

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. Tactics above reduce the

$$\begin{array}{rcl}
\texttt{destr_goal} & & \\
\Gamma;\Theta;\Sigma \vdash \bullet :: [c] \wedge \varphi & \rightsquigarrow & \Gamma;\Theta;\Sigma \vdash \bullet :: c \\
& \text{and} & \Gamma, c;\Theta;\Sigma \vdash \bullet :: \varphi \\
\Gamma;\Theta;\Sigma \vdash \bullet :: \varphi_1 \wedge \varphi_2 & \rightsquigarrow & \Gamma;\Theta;\Sigma \vdash \bullet :: \varphi_1 \\
& \text{and} & \Gamma;\Theta;\Sigma \vdash \bullet :: \varphi_2 \\
& & \\
\texttt{left} & \equiv & \texttt{case l} \\
\Gamma;\Theta;\Sigma \vdash \bullet :: (\texttt{l:}\ \varphi_1) \vee (\texttt{r:}\ \varphi_2) & \rightsquigarrow & \Gamma;\Theta;\Sigma \vdash \bullet :: \varphi_1 \\
\texttt{right} & \equiv & \texttt{case r} \\
\Gamma;\Theta;\Sigma \vdash \bullet :: (\texttt{l:}\ \varphi_1) \vee (\texttt{r:}\ \varphi_2) & \rightsquigarrow & \Gamma;\Theta;\Sigma \vdash \bullet :: \varphi_2
\end{array}$$

Figure 5.4: Tactis that disect the goal

current goal. 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. Finally we can prove goals through generalization, induction on terms, and through reduction to absurd. We also provide shorthand formulas for using the axioms of our logic, described in previous chapter. Again argument `a` of `get_fresh_atom` is given by name and is bound by a fresh

`destr_assm H`		
$\Gamma; \Theta \cup \{\mathtt{H} :: [c] \wedge \varphi\}; \Sigma \vdash \bullet :: \varphi$	$\rightsquigarrow$	$\Gamma \cup \{c\}; \Theta \cup \{\mathtt{H} :: \varphi\}; \Sigma \vdash \bullet :: \varphi$
$\Gamma; \Theta \cup \{\mathtt{H} :: \varphi_1 \wedge \varphi_2\}; \Sigma \vdash \bullet :: \varphi$	$\rightsquigarrow$	$\Gamma; \Theta \cup \{\mathtt{H\_1} :: \varphi_1, \mathtt{H\_2} :: \varphi_2\}; \Sigma \vdash \bullet :: \varphi$
$\Gamma; \Theta \cup \{\mathtt{H} :: \varphi_1 \vee \varphi_2\}; \Sigma \vdash \bullet :: \varphi$	$\rightsquigarrow$	$\Gamma; \Theta \cup \{\mathtt{H} :: \varphi_1\}; \Sigma \vdash \bullet :: \varphi$
	and	$\Gamma; \Theta \cup \{\mathtt{H} :: \varphi_2\}; \Sigma \vdash \bullet :: \varphi$
`destr_assm' H x`		
$\Gamma; \Theta \cup \{\mathtt{H} :: \exists_A a.\, \varphi\}; \Sigma \vdash \bullet :: \varphi$	$\rightsquigarrow$	$\Gamma; \Theta \cup \{\mathtt{H} :: \varphi\{a \mapsto \mathtt{x}\}\}; \Sigma' \vdash \bullet :: \varphi$
	where	$\Sigma' = \Sigma \cup \{\mathtt{x} :: A\}$
$\Gamma; \Theta \cup \{\mathtt{H} :: \exists_T X.\, \varphi\}; \Sigma \vdash \bullet :: \varphi$	$\rightsquigarrow$	$\Gamma; \Theta \cup \{\mathtt{H} :: \varphi\{X \mapsto \mathtt{x}\}\}; \Sigma' \vdash \bullet :: \varphi$
	where	$\Sigma' = \Sigma \cup \{\mathtt{x} :: T\}$
	when	$\mathtt{x} \notin \mathrm{FV}(\Gamma; \Theta; \Sigma)$

Figure 5.5: Tactics that disect assumptions

`ex_falso`		
$\Gamma; \Theta; \Sigma \vdash \bullet :: \varphi$	$\rightsquigarrow$	$\Gamma; \Theta; \Sigma \vdash \bullet :: \bot$
`discriminate`		
$\Gamma; \Theta; \Sigma \vdash \bullet :: \varphi$	$\rightsquigarrow$	$\Gamma; \Theta; \Sigma \vdash \varphi$
	when	$\Gamma \vDash \bot_c$
`exists e`		
$\Gamma; \Theta; \Sigma \vdash \bullet :: \exists_A a.\, \varphi$	$\rightsquigarrow$	$\Gamma; \Theta; \Sigma \vdash \bullet :: \varphi\{a \mapsto \mathtt{e}\}$
$\Gamma; \Theta; \Sigma \vdash \bullet :: \exists_T X.\, \varphi$	$\rightsquigarrow$	$\Gamma; \Theta; \Sigma \vdash \bullet :: \varphi\{X \mapsto \mathtt{e}\}$
`generalize x`		
$\Gamma; \Theta; \Sigma \vdash \bullet :: \varphi$	$\rightsquigarrow$	$\Gamma; \Theta; \Sigma' \vdash \bullet :: \forall_T \mathtt{x}.\, \varphi$
	when	$\Sigma = \Sigma' \cup \{\mathtt{x}\}$ and $\mathtt{x} \notin \mathrm{FV}(\Gamma)$
`by_induction x IH`		
$\Gamma; \Theta; \Sigma \vdash \bullet :: (\forall_T X.\, \varphi(X))$	$\rightsquigarrow$	$\Gamma; \Theta \cup \{\mathtt{IH} :: \psi\}; \Sigma \cup \{\mathtt{x} :: T\} \vdash \bullet :: \varphi(X)$
	where	$\psi = \forall_T \mathtt{x}.\, [\mathtt{x} \prec X] \Longrightarrow \varphi(\mathtt{x})$

Figure 5.6: More tactics and operators.

`compare_atoms a b`

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

`get_fresh_atom a e`

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

`inverse_term e`

$$\begin{array}{rcl} \Gamma;\Theta;\Sigma \vdash \bullet :: \varphi & \rightsquigarrow & \Gamma;\Theta;\Sigma \vdash \bullet :: (\exists_A \mathsf{a}.\ \ \mathsf{e} = \mathsf{a}) \implies \varphi \\ & \text{and} & \Gamma;\Theta;\Sigma \vdash \bullet :: (\exists_A \mathsf{a}.\, \exists_T \mathsf{e'}.\ \ \mathsf{e} = \mathsf{a.e'}) \implies \varphi \\ & \text{and} & \Gamma;\Theta;\Sigma \vdash \bullet :: (\exists_T \mathsf{e1}, \mathsf{e2}.\ \ \mathsf{e} = \mathsf{e1}\ \mathsf{e2}) \implies \varphi \\ & \text{and} & \Gamma;\Theta;\Sigma \vdash \bullet :: (\text{symbol}\ \mathsf{e}) \implies \varphi \end{array}$$

Figure 5.7: Axiomatic tactics.

internal identifier automatically.

Proof are written as OCaml programs, but can be similiarly easy to read as the ones written with dedicated domain-specific languages, as provide the users with some helper functions and tacticals.

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:

```
(* define symbols used in arithmetical theorems *)
let arith_symbols = symbols ["0"; "S"]

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

let plus_eq_relation = (* PlusEq n m k *)
  "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 =
```

operator (|>)	Applies a tactic on the prover state.
operator (%>)	Combines two tactics together.
`subst`	Substitutes atoms for atom expressions and variables for terms in goal and environment — as long as Solver proves their equality.
`compute`	Computes WHNF of the current goal.
`try_tactic`	Tries applying a tactic and returns unchanged state if it fails.
`repeat`	Applies given tactic mutliple times (until failure).
`trivial`	Tries applying some simple tactics to progress the proof.
`qed`	Turns prover state of a finished proof into a forwards proof. Correctness of proof transformations is guaranteed through the usage of `proof` smart constructors that implement the natural deduction rules.

Figure 5.8: More tactics and operators.

```
proof' nat_1_thm (* goal: Nat {S 0} *)
|> case "succ"   (* goal: ∃ m :term. [S 0 = S m] ∧ Nat m *)
|> exists "0"    (* goal: [S 0 = S 0] ∧ Nat 0 *)
|> solve         (* goal: Nat 0 *)
|> case "zero"   (* goal: 0 = 0 *)
|> solve         (* 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:

```
(* define symbols used in lambda calculus theorems *)
let lambda_symbols = ["lam"; "app"; "base"; "arrow"; "nil"; "cons"]

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

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

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

let typing_relation = (* Typing e env t *)
  "fix Typing(e): ∀ env t :term. * = fun env t :term →
```

```
    var: (∃ a :atom. [e = a] ∧ (InEnv env a t))
    ∨
    lam: (∃ a :atom.∃ e’ t1 t2 :term.
           [e = lam (a.e’)] ∧ [t = arrow t1 t2]
             ∧ (Type t1) ∧ (Typing e’ {cons a t1 env} t2))
    ∨
    app: (∃ e1 e2 t2 :term.
           [e = app e1 e2]
             ∧ (Typing e1 env {arrow t2 t}) ∧ (Typing e2 env t2))"

 let sub_relation = (* Sub e a v e’ *)
   "fix Sub(e): ∀ a :atom. ∀ v e’:term.* = fun (a :atom) (v e’ :term) →
     var_same: ([e = a] ∧ [e’ = v])
     ∨
     var_diff: (∃ 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’) )"
```

Notice that in the definition of `Sub`, in the case for abstracion we only consider the case where the substituted name is different than the abstraction's argument ($\mathtt{a} \neq \mathtt{b}$). If we wanted to substitute `a` for `v` in term `a.e`, then we could swap the argument's name for a different atom b that is fresh in `e`, as then know that $\mathtt{a.e} = \mathtt{b.(a\ b)e}$ and can substitute in that term. In the end, as `b` was fresh in `e`, then `a` must be fresh in (`a b`)`e`, so either way we arrive at identity — but have one less case to consider while writing proofs.

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

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

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

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

(* lambda_thm parses the theorem in an env that includes lambda_symbols
    and all lambda predicates and relations *)
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 and relations:

```
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’)"

let env_inclusion_relation = (* EnvInclusion e1 *)
  "fun env1 env2 : term →
     ∀ a : atom. ∀ t : term. (InEnv env1 a t)  ⟹  (InEnv env2 a t)"
```

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_spec 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"]
     %> solve
(* 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 indee a proper *term*, so we simply use the `typing_terms` lemma and we're done here.

```
     %> apply_thm_spec 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 `ntro'intro'` tactic to destruct the assumption into multiple branches.

```
  |> 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_spec empty_contradiction ["a"; "t"]
     %> assumption
```

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

```
  |> intros' ["Hlam"; "a"; "e_a"; "t1"; "t2"; ""] %> case "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"]
     %> solve
```

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"
  |> add_assumption_parse "He1" "Progressive e1"
     %> apply_assm_spec "IH" ["e1"; "arrow t2 t"] %> solve
  |> add_assumption_parse "He2" "Progressive e2"
     %> apply_assm_spec "IH" ["e2"; "t2"] %> solve
  |> 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_spec "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` substituted 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_spec "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"] %> solve
(* 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 solve
(* 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 solve
    %> apply_assm "Hs1"
  |> apply_assm "Happ_2" %> apply_assm "Happ_1"
  |> qed
```

Now, to prove *Preservation*, we will need some more lemmas:

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

```
let sub_lemma_thm = lambda_thm
  "∀ 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"
            %> solve
            %> assumption )
```

Second case is that `e` is some abstraction `lam (b.e_b)`. Because of the way we defined subisution, 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 solve
(* 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_spec
        "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  *)
    %> solve
    %> ( apply_thm_spec
          cons_fresh_typing ["v"; "env"; "ta"; "c"; "t1"]
          (* [c # v] ⟹ Typing v env ta ⟹
               Typing v {cons c t1 env} ta *)
        %> solve
        %> apply_assm "Hv" )
    %> ( apply_thm_spec
         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 *)
        %> solve
        %> apply_thm_spec 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 *)
        %> solve
        %> 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, so we omit this part here.

```
  |> intros' ["Happ"; "e1"; "e2"; "t2"; ""; ""]
    %> intros' ["Hsub"; "_e1"; "_e2"; "e1'"; "e2'"; ""; ""; ""]
    %> case "app"
    %> exists' ["e1'"; "e2'"; "t2"]
    %> solve
(* 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
*)
     ...
  |> 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"]
     %> solve
(* 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_spec "IH" ["e1"; "e1'"; "env"; "arrow t2 t"]
            (* [e1 ≺ e] ⟹
                 Typing e1 env {arrow t2 t} ⟹
                   Steps e1 e1' ⟹
                     Typing e1' env {arrow t2 t} *)
            %> solve
            %> apply_assm "Happ_1"
            %> apply_assm "He1" )
       %> apply_assm "Happ_2"
  |> intros' ["He2"; "v1"; "e2"; "e2'"; ""; ""; ""]
     %> case "app"
     %> exists' ["v1"; "e2'"; "t2"]
     %> solve
(* 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_spec "IH" ["e2"; "e2'"; "env"; "t2"]
```

```
            (* [e2 ≺ e] ⟹
                 Typing e2 env t2 ⟹
                   Steps e2 e2' ⟹
                     Typing e2' env t2 *)
            %> solve
            %> 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_spec
         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_spec
           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. However, the true potential of this framework is expected to shine when applied to specific theorems reliant on the notions of variable binding.

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 Twelf.

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 reasoning based on Nominal Logic.

Bibliografia

[1] Martín Abadi i in. “Explicit Substitutions”. W: *Journal of Functional Programming* 1 (1991), s. 375 –416. DOI: 10.1017/S0956796800000186.

[2] 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. DOI: 10.1016/S0890-5401(03)00138-X.

[3] Arthur Charguéraud. “The Locally Nameless Representation”. W: *Journal of Automated Reasoning - JAR* 49 (2012), s. 1–46. DOI: 10.1007/s10817-011-9225-2.

[4] Adam Chlipala. “Parametric Higher-Order Abstract Syntax for Mechanized Semantics”. W: *SIGPLAN Not.* 43.9 (2008), 143–156. DOI: 10.1145/1411203.1411226.

[5] 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. DOI: 10.1016/1385-7258(72)90034-0.

[6] Murdoch J. Gabbay i Andrew M. Pitts. “A New Approach to Abstract Syntax with Variable Binding”. W: *Formal Aspects of Computing* 13.3 (2002), s. 341–363. DOI: 10.1007/S001650200016.

[7] Michael J. C. Gordon. “HOL: A Proof Generating System for Higher-Order Logic”. W: *VLSI Specification, Verification and Synthesis*. Red. Graham Birtwistle i P. A. Subrahmanyam. Boston, MA: Springer US, 1988, s. 73–128. DOI: 10.1007/978-1-4613-2007-4_3.

[8] Daniel Lee, Karl Crary i Robert Harper. “Towards a mechanized metatheory of standard ML”. W: t. 42. Sty. 2007, s. 173–184. ISBN: 1595935754. DOI: 10.1145/1190216.1190245.

[9] Frank Pfenning i Conal Elliott. “Higher-Order Abstract Syntax”. W: t. 23. Lip. 1988, s. 199–208. DOI: 10.1145/960116.54010.

[10] Frank Pfenning i Carsten Schürmann. "System Description: Twelf — A Meta-Logical Framework for Deductive Systems". W: *Automated Deduction — CADE-16*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, s. 202–206. DOI: 10.1007/3-540-48660-7_14.

[11] Brigitte Pientka. "Beluga: Programming with Dependent Types, Contextual Data, and Contexts". W: *Functional and Logic Programming*. Red. Matthias Blume, Naoki Kobayashi i Germán Vidal. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, s. 1–12. DOI: 10.1007/978-3-642-12251-4_1.

[12] Steven Schäfer, Tobias Tebbi i Gert Smolka. "Autosubst: Reasoning with de Bruijn Terms and Parallel Substitutions". W: *Interactive Theorem Proving*. Red. Christian Urban i Xingyuan Zhang. Cham: Springer International Publishing, 2015, s. 359–374. DOI: 10.1007/978-3-319-22102-1_24.

Appendices

Appendix A

Solver rules

Goal-reducing equality rules:

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

$$\frac{\Gamma; \Delta \vDash t_1 = t_2 \qquad \Gamma; \Delta \vDash t_1' = t_2'}{\Gamma; \Delta \vDash t_1 t_1' = t_2 t_2'}$$

$$\frac{\begin{array}{c}\Gamma; \Delta \vDash \alpha_1 \# \alpha_2.t_2 \\ \Gamma; \Delta \vDash t_1 = (\alpha_1\ \alpha_2)t_2\end{array}}{\Gamma; \Delta \vDash \alpha_1.t_1 = \alpha_2.t_2} \qquad \frac{\begin{array}{c}a \neq \alpha_1, a \neq \alpha_2, \Gamma; \Delta \vDash a = \alpha \\ a = \alpha_1, a \neq \alpha_2, \Gamma; \Delta \vDash \alpha_2 = \alpha \\ a = \alpha_2, \Gamma; \Delta \vDash \alpha_1 = \alpha\end{array}}{\Gamma; \Delta \vDash a = (\alpha_1\ \alpha_2)\alpha}$$

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

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

Goal-reducing freshness rules:

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

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

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

Goal reducing shape rules:

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

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

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

Goal-reducing subshape rules:

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

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

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

Assumption-reducing equality rules:

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

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

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

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

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

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

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

$$\frac{\alpha_1 \,\#\, \alpha_2.t_2,\ t_1 = (\alpha_1\ \alpha_2)t_2,\ \Gamma;\Delta \vDash \mathcal{C}}{\alpha_1.t_1 = \alpha_2.t_2, \Gamma;\Delta \vDash \mathcal{C}}$$

Other term constructors trivial

$$\frac{t_1 = t_2,\ t_1' = t_2',\ \Gamma;\Delta \vDash \mathcal{C}}{t_1t_1' = t_2t_2', \Gamma;\Delta \vDash \mathcal{C}}$$

Other term constructors trivial

$$\frac{s_1 \neq s_2}{s_1 = s_2, \Gamma;\Delta \vDash \mathcal{C}} \qquad \frac{\Gamma;\Delta \vDash \mathcal{C}}{s = s, \Gamma;\Delta \vDash \mathcal{C}}$$

Other term constructors trivial

Assumption-reducing freshness rules:

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

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

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

$$\frac{\begin{array}{c} a \mathrel{\#} \alpha, \ \Gamma; \Delta \vDash \mathcal{C} \\ a \mathrel{\#} \alpha, \ a \mathrel{\#} t, \ \Gamma; \Delta \vDash \mathcal{C} \end{array}}{a \mathrel{\#} \alpha.t, \Gamma; \Delta \vDash \mathcal{C}}$$

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

Assumption-reducing shape rules:

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

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

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

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

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

Assumption-reducing subshape rules:

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

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

$$\frac{\begin{array}{cc} t_1 \sim t_2, \Gamma; \Delta \vDash \mathcal{C} & t_1 \sim t_2', \Gamma; \Delta \vDash \mathcal{C} \\ t_1 \prec t_2, \Gamma; \Delta \vDash \mathcal{C} & t_1 \prec t_2', \Gamma; \Delta \vDash \mathcal{C} \end{array}}{t_1 \prec t_2 t_2', \Gamma; \Delta \vDash \mathcal{C}}$$

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