Formal Metatheory of Programming Languages in the Matita Interactive Theorem Prover

Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen and Enrico Tassi
Department of Computer Science, University of Bologna
Mura Anteo Zamboni, 7 — 40127 Bologna, ITALY
({asperti,ricciott,sacerdot,tassi}@cs.unibo.it*)*

Abstract. This paper is a report about the use of Matita, an interactive theorem prover under development at the University of Bologna, for the solution of the POPLmark Challenge, part 1a. We provide three different formalizations, including two direct solutions using pure de Bruijn and locally nameless encodings of bound variables, and a formalization using named variables, obtained by means of a sound translation to the locally nameless encoding. According to this experience, we also discuss some of the proof principles used in our solutions, which have led to the development of a generalized inversion tactic for Matita.

1. Introduction

The POPLmark challenge(Aydemir et al., 2005) is a set of "benchmarks" proposed by an international group of researchers in order to assess the advances of theorem proving for the verification of properties of programming languages and to promote the use and enhancement of proof assistant technology.

The set of problems has been chosen to capture many of the most critical issues in formalizing the metatheory of programming languages, comprising scope, binding, typing, and reduction. In particular, the challenge focuses on some theoretical aspects of System $F_{<:}$(Cardelli et al., 1991), that is a language joining a simple and tractable syntax with a sufficiently rich and complex metatheory.

Due to its intended goals, it is natural, for any new tool aimed at the mechanization of formal reasoning, to confront such a challenge, both to stress the tool against a nontrivial set of problems, and to test its expressiveness and actual usability. This paper is a report about a new solution for part 1a of the POPLmark challenge developed using Matita, a new interactive theorem prover under development at the Computer Science Department of the University of Bologna. Matita and its library, including the solution to the challenge discussed here, are available for download at `http://matita.cs.unibo.it` .

The structure of the paper is the following: section 2 introduces Matita; section 3 discusses the three representations of bound and

free variables which we used in our solutions; in section 4 we describe the proof principles and the main proofs of our solutions; finally in section 5 we briefly analyse the work on a quantitative basis and draw conclusions.

2. Matita

Matita is a new interactive prover developed at the University of Bologna by a research team coordinated by the first author. The architecture is discussed in (Asperti et al., 2006).

Matita is based on the Calculus of Inductive Constructions and is partially compatible, at proof object level, with the Coq System (Coq, 2004) developed at INRIA. It is written in Ocaml and its features were purposely developed to be similar to those of Coq, in order to provide a more modular, more maintainable and thus, in many ways, "lighter" alternative. Since the two systems share the same foundational framework, their kernel is also similar: the main differences are that modules are not implemented in Matita, that on the other side provide explicit substitutions as a primitive notion.

Some key differences lie in the refiner, that is the type inference shell surrounding the kernel. In particular, Matita takes advantage from a strong notion of existential variable that has no counterpart in Coq and has an extensive beneficial impact in many architectural aspects of the system (from tactics, to i/o). At user interaction level, Matita offers both a procedural and a declarative editing style, providing several innovative features(Asperti et al., 2007). In particular

- the sequent-window is based on a MathML-compliant GTK-widget supporting hyperlinks, a sophisticated bidimensional rendering, "content based" cut and paste, in place reduction and other direct form of interaction;

- editing is improved by means of "tinycals" (Sacerdoti Coen et al., 2006), featuring a sort of "semantic directed" editing mode, that allows to structure the text following the structure of the dynamically generated proof, and maintaining at the same time the possibility of a step by step execution (impossible in any other system);

- Matita offers a complex mechanism to solve notational ambiguities and overloading of operators, allowing the user to work with a (configurable) notation as close as possible to the standard mathematical practice;

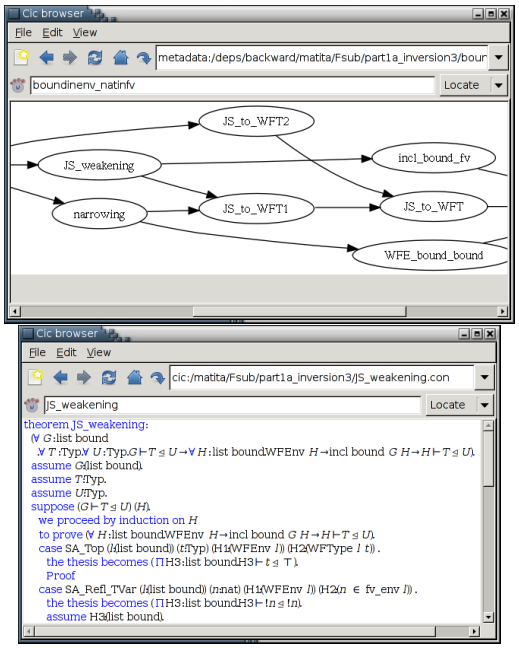

Figure 1. The Cic Browser showing dependencies between proofs and a proof in declarative style.

– the system has been conceived since the very beginning as an interface between the user and the library of already proved results; all new theorems are indexed using a metadata system explicitly conceived for this purpose, and several functionalities to search and browse the repository are implemented. These comprise a particularly useful "hint" operation, suggesting to the user a set of theorems applicable to the current goal, and a browser (see Fig. 2) to show graphs of dependencies among proofs and definitions and translations of procedural proofs in declarative style.

3. Concrete encodings of variable bindings

System $F_{<:}$ is a second order lambda calculus enriched with a subtyping relation. Since the focus of this paper is on the formalization of proofs concerning the type sublanguage of $F_{<:}$, we will assume knowledge of the full syntax of $F_{<:}$, and only report here the details needed to discuss the formalized syntax of types and typing environments here. For the full syntax of $F_{<:}$, see (Cardelli et al., 1991).

$S, T, \ldots$	::=		Types
	|	$X, Y, \ldots$	type variables
	|	`Top`	the supertype of any type
	|	$S \to T$	functions from S to T
	|	$\forall X <: S.T$	bounded universal quantifier

Figure 2. Syntax of the type sublanguage of $F_{<:}$

The type sublanguage of $F_{<:}$ (fig. 2) consists of type variables, the type `Top` (which is supertype of any type), arrow types (functions from one type to another) and universal types (polymorphic expressions); environments may carry both typing constraints (on term variables) and subtyping constraints (on type variables). As for the subtyping relation, it is formalized by means of an algorithmic subtyping judgement, whose rules are directed by the syntax. Part 1a of the POPLmark challenge asks to prove that algorithmic subtyping is reflexive and transitive. We provide the well-formedness and subtyping rules of $F_{<:}$ in fig. 3 as a reference for the following sections.

Since $F_{<:}$ makes use of binders not only in terms, but also in types, we must deal with the well-known problems of α-equivalence and avoidance of variable capture. The most common approaches to these difficulties require to rewrite the syntax in such a way that α-equivalent terms are syntactically equal. One way to do this is to drop names altogether: variables can be expressed by means of indices, whose value uniquely identifies the level at which the variable is bound; this is how de Bruijn's representation works.

One inconvenience with de Bruijn's representation is that when performing a substitution, indices representing free variables in the substituted term might need to be updated (*lifted*) in order to stay coherent; this can complicate both the statements and the proofs of many lemmata. The *locally nameless* representation (Pollack, 1993) is a variation on de Bruijn's representation, where bound variables are represented by indices (so that α-equivalence and equality are the same) and free variables are represented by names (eliminating the need to lift free indices in substituted terms). The syntax is the same as the de Bruijn representation, except for the addition of free type variables (fig. 5).

Typing environments in the locally nameless approach are similar to their informal counterparts. They are defined as lists of bounds, which are pairs (variable name, type), together with a boolean value to

$$\Gamma \vdash \texttt{Top} \qquad \text{(WFT-Top)}$$

$$\frac{X \in dom(\Gamma)}{\Gamma \vdash X} \qquad \text{(WFT-TVar)}$$

$$\frac{\Gamma \vdash S \qquad \Gamma \vdash T}{\Gamma \vdash S \to T} \qquad \text{(WFT-Arrow)}$$

$$\frac{\Gamma \vdash S \qquad X \notin dom(\Gamma) \qquad \Gamma, X <: T \vdash U}{\Gamma \vdash \forall X <: T.U} \qquad \text{(WFT-Forall)}$$

$$\emptyset \vdash \Diamond \qquad \text{(WFE-Empty)}$$

$$\frac{x \notin dom(\Gamma) \qquad \Gamma \vdash T}{\vdash \Gamma, x : T} \qquad \text{(WFE-Cons1)}$$

$$\frac{X \notin dom(\Gamma) \qquad \Gamma \vdash T}{\vdash \Gamma, X <: T} \qquad \text{(WFE-Cons2)}$$

$$\frac{\Gamma \vdash \Diamond \qquad \Gamma \vdash S}{\Gamma \vdash S <: \texttt{Top}} \qquad \text{(SA-Top)}$$

$$\frac{\Gamma \vdash \Diamond \qquad X \in dom(\Gamma)}{\Gamma \vdash X <: X} \qquad \text{(SA-Refl-TVar)}$$

$$\frac{X <: U \in \Gamma \qquad \Gamma \vdash U <: T}{\Gamma \vdash X <: T} \qquad \text{(SA-Trans-TVar)}$$

$$\frac{\Gamma \vdash T_1 <: S_1 \qquad \Gamma \vdash S_2 <: T_2}{\Gamma \vdash S_1 \to S_2 <: T_1 \to T_2} \qquad \text{(SA-Arrow)}$$

$$\frac{\Gamma \vdash T_1 <: S_1 \qquad \Gamma, X <: T_1 \vdash S_2 <: T_2}{\Gamma \vdash \forall X_{S_1}.S_2 <: \forall X_{T_1}.T_2} \qquad \text{(SA-All)}$$

Figure 3. Well-formedness and subtyping rules of $F_{<:}$

discriminate typing bounds on term variables from subtyping bounds on type variables.

In the de Bruijn approach, we don't have names and bounds are identified by their position inside the environment. The dangling indices inside a bound must be resolved in the part of the environment which precedes that bound. We will use the notation $\bullet <: T$ to refer to a subtyping bound in a de Bruijn typing environment.

The last concrete approach to binding we take into account is the named variables approach, in which names are used for both free and

$S, T, \ldots$	::=		Types
	|	$\#0, \#1, \ldots$	type indices
	|	`Top`	the supertype of any type
	|	$S \to T$	functions from S to T
	|	$\forall_S.T$	bounded universal quantifier

Figure 4. Syntax of $F_{<:}$ (de Bruijn): types

$S, T, \ldots$	::=		Types
	|	$X, Y, \ldots$	free type variables
	|	...	

Figure 5. Syntax of $F_{<:}$ (locally nameless): types

bound variables. Its syntax is the closest possible to the informal presentation of fig. 2: however we will see that the formalization of its type system requires some additional care.

$S, T, \ldots$	::=		Types
	|	$X, Y, \ldots$	type variables
	|	`Top`	the supertype of any type
	|	$S \to T$	functions from S to T
	|	$\forall X <: S.T$	bounded universal quantifier

Figure 6. Syntax of $F_{<:}$ (named variables): types

4. Formalization

We discuss now our formalizations of Part 1A of the POPLmark challenge. The first part deals with the formalization of the type system using the encodings mentioned in Section 3. In the second part, we present some of the proof principles used in the solutions. Finally, we describe the main proofs of each formalization.

4.1. Formalization of the type system

To restate the well-formedness and subtyping judgements in the de Bruijn encoding, it is sufficient to remember the key differences of this encoding with respect to the informal syntax:

- named variables are replaced by indices, with an explicit management of binding: the dangling index $\#n$ refers to the n-th entry of its environment (from right to left, 0 based);
- each environment entry lives in a different environment: in order to use the content of an environment entry in a judgement, we must relocate it to the environment of that judgement.

The first change happens to be more an advantage than an issue: it allows us not to worry at all about names, at the same time keeping the statement of rules concerning binding very natural, similar to informal practice. The second change, however, needs a more careful handling, since relocation must be treated explicitly. Fig. 7 shows the de Bruijn formalization of the less trivial rules of $F_{<:}$: the notations $|\Gamma|$ and $\Gamma(n)$ refer respectively to the length of environment Γ and to the n-th entry of Γ; $T \uparrow n$ is the variable lifting operation, defined as follows:

$$T \uparrow_k n \;=\; \begin{cases} \#m \uparrow_k n \;=\; m+n & \text{if } k \le m \\ \#m \uparrow_k n \;=\; m & \text{if } k > m \\ \mathtt{Top} \uparrow_k n \;=\; \mathtt{Top} & \\ (U \to V) \uparrow_k n \;=\; (U \uparrow_k n) \to (V \uparrow_k n) & \\ (\forall_U.V) \uparrow_k n \;=\; \forall_{U \uparrow_k n}.(V \uparrow_{k+1} n) & \end{cases}$$

$$T \uparrow n \;=\; T \uparrow_0 n$$

Lifting provides the notion of relocation we needed, since each environment entry lives in an initial segment of the full environment.

In the locally nameless encoding, we get a more immediate treatment of environments, since relocation of environment entries is not needed. On the contrary, binding needs a more complex treatment, because of the use of explicit names for free variables. In particular, the rules whose conclusions involve binders cannot be fully structural on types: on one side, we want the type system to only deal with locally closed types (since locally-closedness is a necessary condition for a type to be well formed); on the other side, in a well formed type $\forall U.V$, V is in general not locally closed.

Of course the solution is to replace the dangling index $\#0$ of V with a proper free variable X. However this kind of reasoning hides more complexity than meets the eye. For example, we might translate the All rule to the locally nameless encoding, obtaining easily:

$$\frac{n < |\Gamma|}{\Gamma \vdash \#n} \quad \text{(WFT-TFree)}$$

$$\frac{\Gamma \vdash T \qquad \Gamma, \bullet <: T \vdash U}{\Gamma \vdash \forall_T.U} \quad \text{(WFT-All)}$$

$$\frac{\Gamma \vdash \Diamond \qquad n < |\Gamma|}{\Gamma \vdash \#n <: \#n} \quad \text{(SA-Refl-TVar)}$$

$$\frac{\Gamma(n) = \bullet <: U \qquad \Gamma \vdash (U \uparrow n+1) <: T}{\Gamma \vdash \#n <: T} \quad \text{(SA-Trans-TVar)}$$

$$\frac{\Gamma \vdash T_1 <: S_1 \qquad \Gamma, \bullet <: T_1 \vdash S_2 <: T_2}{\Gamma \vdash \forall_{S_1}.S_2 <: \forall_{T_1}.T_2} \quad \text{(SA-All)}$$

Figure 7. Some rules of the de Bruijn-style formalization of $F_{<:}$.

$$\frac{\Gamma \vdash T_1 <: S_1 \qquad \Gamma, X <: T_1 \vdash S_2\{^X/_{\#0}\} <: T_2\{^X/_{\#0}\}}{\Gamma \vdash \forall_{S_1}.S_2 <: \forall_{T_1}.T_2}$$

where $S_2\{^X/_{\#0}\}$ means "the type S_2 where every occurrence of the dangling index #0 has been replaced with a free type name X". Please notice the use of X: nowhere do we state if the right premise should hold for a specific X or for any X. Indeed, both alternatives are partially incorrect because, for reasons of well-formedness, we must require that X be fresh; assuming this condition of well-formedness is met, alternative solutions for quantification have been proposed in literature. Universal and existential quantification lead to formulations of the type system which we respectively call *strong* and *weak* (after Urban and Pollack(Urban and Pollack, 2007)). However, these names are somewhat misleading since it can be proved that the two formulations are logically equivalent: this comes from the fact that the subtyping judgement is an *equivariant* predicate, i.e. one whose validity is invariant under swapping of variable names.

The concept of equivariance, which is a key point of nominal logics (Pitts, 2003), was exploited in the solution proposed by Leroy (Leroy, 2007), as well as in a previous version of our locally nameless solution. However, upon discovering that the proofs related to equivariance accounted for about one third of our code, we decided to go for a more standard approach.

It can be noted that, in informal logical practice, it is convenient to use the weak (existential) variant when we want to construct a proof

of $\Gamma \vdash \forall_{S_1}.S_2 <: \forall_{T_1}.T_2$ (we only need to show that the premises hold for one suitable X); on the other hand, when reasoning backwards, the strong (universal) variant is more useful, as it provides stronger induction principles. A more complex co-finite quantification (Aydemir et al., 2008), providing the benefits of both the strong and the weak versions of the type system, has been used by Charguéraud for his locally nameless solution in Coq. In our locally nameless solution, we chose to use the strong formulation of the type system, which is sufficient to obtain very compact proofs. In fact, up to minor syntactical differences between Coq and Matita's tactic languages, it turns out that our solution is the most compact among those based on the locally nameless encoding (see Sect. 5).

Figure 8 describes the rules for well-formedness and subtyping of universal types, as formalized in the locally nameless encoding.

$$\frac{\Gamma \vdash T \qquad \text{for all } X : \left(\begin{array}{l} X \notin dom(\Gamma) \wedge X \notin \mathrm{FV}(U) \Rightarrow \\ \Gamma, X <: T \vdash U\{^X/_{\#0}\} \end{array} \right)}{\Gamma \vdash \forall_T.U} \quad \text{(WFT-All)}$$

$$\frac{\Gamma \vdash T_1 <: S_1 \qquad \text{for all } X : \left(\begin{array}{l} X \notin dom(\Gamma) \Rightarrow \\ \Gamma, X <: T_1 \vdash S_2\{^X/_{\#0}\} <: T_2\{^X/_{\#0}\} \end{array} \right)}{\Gamma \vdash \forall_{S_1}.S_2 <: \forall_{T_1}.T_2} \quad \text{(SA-All)}$$

Figure 8. Some rules of the locally nameless formalization of $F_{<:}$.

Our last formalization uses the named variables approach. Ideally, the formalization of the type system should be very close to the informal presentation of fig. 3. However, at some point, α-conversion must be taken into account, otherwise one will never be able to prove a subtyping relation between two universal types binding different variables.

There are basically two ways to deal with α-conversion:

– α-conversion can be formalized separately from the subtyping judgement (either algorithmically or as an inductive predicate); then an additional rule for the subtyping judgement will be provided:

$$\frac{\Gamma \vdash S <: T \qquad S =_\alpha S' \qquad T =_\alpha T'}{\Gamma \vdash S' <: T'} \quad \text{(SA-Alpha)}$$

– the rules WFT-All and SA-All can be rephrased in such a way that the subtyping judgement is directly derivable even if their bound variables are different:

$$\frac{\text{for all } Z \notin \mathrm{dom}(\Gamma):\ \left(\begin{array}{l}\Gamma \vdash T_1 <: S_1 \\ (Z \in \mathrm{FV}(S_2) \Rightarrow Z = X) \Rightarrow \\ (Z \in \mathrm{FV}(T_2) \Rightarrow Z = Y) \Rightarrow \\ \Gamma, Z <: T_1 \vdash (Z\ X) \cdot S_2 <: (Z\ Y) \cdot T_2\end{array}\right)}{\Gamma \vdash \forall X <: S_1.S.2 <: \forall Y <: T_1.T_2} \text{(SA-All)}$$

where $(X\ Y) \cdot -$ is the name swapping operator, replacing every occurrence of X with Y and vice-versa, not caring for binders.

We will avoid the first solution, since rules like SA-Alpha make the subtyping judgement less algorithmic, which would contrast with the spirit of the POPLmark challenge. However the second solution can seem a little puzzling at first. The swap-based statement of α-conversion was originally due to Gabbay and Pitts (Gabbay and Pitts, 1999) and is very well-suited to formalization, since it simplifies the handling of name-capture. For what concerns quantification over free variables, again we follow the schema of universal quantification over all acceptable names Z. Z is acceptable if:

– it's not in the domain of Γ;

– it does not cause variable capture inside S_2 or T_2: for this condition to hold, one must verify that if $Z \in \mathrm{FV}(S_2)$, then $Z = X$, and that if $Z \in \mathrm{FV}(T_2)$, then $Z = Y$.

4.2. Proof principles

Most proofs given in the specification of the POPLmark challenge are by structural induction on some type. However it is often the case, particularly in the locally-nameless representation, that structural induction on types does not yield a strong enough induction hypothesis to reason on sub-typing in the case of bounded quantification: for example, to prove $\forall_S.T$, we obtain an induction hypothesis on T, whereas we now need an induction hypothesis on $T[^X/_{\#0}]$ for all X.

Instead of using induction on types, a very natural proof technique consists in doing structural induction on (proof trees for) the well-formed type judgement. For instance, induction over a proof of $\Gamma \vdash T$ yields exactly the four cases of a proof by induction over T (i.e. $T =$ `Top`,

$T = X$, $T = T_1 \to T_2$ and $T = \forall_{T_1}.T_2$); the second induction hypothesis in the latter case is the strong one we usually need, i.e. that the binary property P (on pairs typing context-type) we are proving holds for $\Gamma, X <: T_1$ and $T_2[^X/_{\#0}]$ for any type variable X free in both Γ and T_2.

In our opinion, and as already noticed by others (as (Pollack, 1993)) proofs by structural induction on the well-formed judgement are more than a technical trick due to an unnatural representation: they are the natural way to reason on types (and terms) of a language. Indeed, note that structural induction on types and structural induction on well-formed type judgements yield exactly the same hypotheses when types are considered up to α-equivalence. Thus we may think of the proofs in the specification of the POPLmark challenge as proofs by structural induction on well-formed judgements.

Once decided that informal proofs by structural induction on types are to be formalized with structural induction on the well-formed judgement for types, the informal proof still presents a suspicious proof step. In (Aydemir et al., 2005), Lemma A.3 (transitivity and narrowing), the proof is done "*by induction on the structure of* Q*. ... We proceed by an inner induction on the structure of* $\Gamma \vdash S <: Q$*, with a case analysis on the final rule of this derivation and of* $\Gamma \vdash Q <: T$*. ... by the inner induction hypothesis ...*". The question is how to formalize an "inner induction on the structure, with a case analysis on the final rule". Indeed, as we will see in a few moments, at least in the Calculus of (Co)Inductive Constructions, structural induction does not allow to perform at once case analysis on the final rule, unless we give up on using the "inner induction hypotheses". The proof may probably still be understood as a proof by induction on the size of the derivation, followed by case analysis on the last rule used. However, such a proof is more involved and more difficult to carry out in systems that favour structural induction (such as Coq and Matita). In the rest of this paragraph we will explain that the previous informal proof rule can be justified by the unusual technique of *induction/inversion*, explaining in what cases induction/inversion is logically justified. Although this proof principle has been "implicitly" exploited in several solutions in Coq to the POPLmark challenge, none of them make it explicit, resulting in an obfuscated proof whose key point is unclear and which is difficult to port to variations of the calculus. Instead, we have formalized the proof principle as a lemma, and we claim that an interactive theorem prover should be able to automatically derive it from the definition of the subtyping judgement, exactly as it is already done for the induction and inversion principles.

4.2.1. *Induction/inversion principles*
To informally explain induction/inversion and its use in our proof, we start recalling the rule for induction over inductive families (which generalise abstract data types in the Haskell community). For the sake of conciseness, in the following pages we will write $\vec{t_n}$ for a sequence of terms $t_1, \ldots, t_n$, and $\vec{t}$ for a sequence of terms of unspecified length. We also allow us to bind several variables at once, writing $B\vec{x_n} : \vec{T_n}.t$ for $Bx_1 : T_1 \ldots Bx_n : T_n.t$ (where $B \in \{\lambda, \forall\}$); the meaning of the notation $B\vec{x} : \vec{T}.t$ is similar, with $\vec{x}$ and $\vec{T}$ having the same, unspecified length.

An inductive family is similar to an inductive type, but it defines at once a set of mutually-recursive inductive types differing from the values of some parameters. Syntactically, the declaration of an inductive family is isomorphic to the declaration of a judgement by giving its introduction rules. The family parameters are the arguments of the judgement. The derivation rules are the constructors of the inductive family. Positivity conditions must be satisfied by the derivation rules to ensure existence of the least fixpoint solution of the set of recursive rules. When the judgement is 0-ary (i.e. it has no parameters), we obtain a simple inductive definition. In this case, the conclusion of all derivation rules is simply the name of the inductive type being defined, providing no information.

Once an inductive family $\mathcal{I}$ is declared by giving its derivation rules (its constructors), we obtain for free recursion over the inductive family as the elimination principle corresponding to the introduction rules. We briefly recall the typing judgement of induction principles for arbitrary inductive families: our syntax is similar to the one given by (Werner, 1994) up to some minor differences.

Given an inductive type $\mathcal{I}$ of arity $\forall\vec{x_n} : \vec{T_n}.\sigma$, where σ is a sort. Suppose that P is a predicate of type $\forall\vec{x_n} : \vec{T_n}.\mathcal{I}\ \vec{x_n} \Rightarrow \tau$, where τ is a sort, and t has type $\mathcal{I}\ \vec{u_n}$ for some (properly typed) terms $\vec{u_n}$. The application of the proper induction principle on t to prove $P\ \vec{u_n}\ t$ is written

$$\mathcal{E}^{\tau}_{\mathcal{I}}(\vec{u_n}, t, P)\{\vec{f_m}\}$$

where $\vec{f_m}$ are the proof terms for each of the m sub-cases of the induction (one for each of the constructors of $\mathcal{I}$). The expected type for the $\vec{f_m}$ is computed by the following definition:

DEFINITION 1. *Let Γ be a CIC context, c, T, Q CIC terms. The operators $\Delta^Q\{\Gamma; c : T\}$ and $\Theta^Q\{\Gamma; T\}$ are defined as follows:*

$$\begin{array}{ll} \Delta^Q\{\Gamma; c : \mathcal{I}\ \vec{t}\} & \equiv\ \Theta^Q\{\Gamma; Q\ \vec{t}\ c\} \\ \Delta^Q\{\Gamma; c : \forall x : T.U\} & \equiv\ \forall x : T.\Delta^Q\{\Gamma, x : T; c\ x : U\} \\ \textit{otherwise undefined} & \end{array}$$

$$
\begin{array}{lcl}
\Theta^Q\{\emptyset; T\} & \equiv & T \\
\Theta^Q\{\Gamma, x : \forall \vec{y} : \vec{V}.\mathcal{I}\ \vec{t}; T\} & \equiv & \Theta^Q\{\Gamma; \forall \vec{y} : \vec{V}.Q\ \vec{t}\ (x\ \vec{y}) \Rightarrow T\} \\
\Delta^Q\{\Gamma, x : U; T\} & \equiv & \Delta^Q\{\Gamma; T\} \textit{ if the head of } U \textit{ is not } \mathcal{I} \\
\textit{otherwise undefined} & &
\end{array}
$$

Let $k_i^{\mathcal{I}}$ of type $K_i^{\mathcal{I}}$ $(i = 1, \ldots, m)$ be the constructors of type $\mathcal{I}$. Then we can write the typing rule for the induction principle as follows:

$$
\frac{\begin{array}{c}
\Gamma \vdash \mathcal{I} : \forall \vec{x_n} : \vec{T_n}.\sigma \\
\text{for all } i = 1, \ldots, n\text{: } \Gamma \vdash u_i : T_i\{{}^{u_1,\ldots,u_{i-1}}/_{x_1,\ldots,u_{i-1}}\} \\
\Gamma \vdash t : \mathcal{I}\ \vec{u_n} \qquad \Gamma \vdash P : \forall \vec{x_n} : \vec{T_n}.\mathcal{I}\ \vec{x_n} \Rightarrow \tau \\
\text{for all } j = 1, \ldots, m\text{: } \Gamma \vdash f_j : \Delta^P\{\emptyset; k_j : K_j\} \\
\text{elimination of } \mathcal{I} \text{ towards sort } \tau \text{ is allowed}^1
\end{array}}{\Gamma \vdash \mathcal{E}_{\mathcal{I}}^{\tau}(\vec{u_n}, t, P)\{\vec{f_m}\} : P\ \vec{u_n}\ t} \qquad \text{(Elim)}
$$

A well-known fact about induction principles for inductive families is that they are not well-suited for immediate applications to hypothesis in which the family parameters are instantiated with anything different from a variable. For example, applying the induction principle to the premise $\Gamma \vdash \texttt{Top} <: T$ we are left with five cases to prove, disregarding the fact that only the first derivation rule could have been applied in this case. Moreover, they are exactly the same five cases we would obtain by changing Γ, `Top` or T to any other well-typed expression. Inversion (Cornes and Terrasse, 1996; McBride, 2002) is the (derived) proof principle we need in these cases. For the previous reasons, the "inner induction with case analysis" in the informal proof suggested in the POPLmark challenge does not correspond to an induction principle.

Inversion allows to invert derivation rules by replacing in a hypothesis a judgement with a disjunction of all the ways in which it can be obtained. Operationally, it is sufficient to perform first order unification of the hypothesis with the conclusion of every derivation rule and, in case of success, augment the conjunction of the premises of the derivation rules with the equalities imposed by the unifier. For instance, inverting an hypothesis $\Gamma \vdash X <: T_1 \to T_2$ yields

$$
\begin{array}{c}
(X <: U \in \Gamma \wedge \Gamma \vdash U <: T_1 \to T_2) \vee \\
(\Gamma \vdash T_1 <: S_1 \wedge \Gamma \vdash S_2 <: T_2 \wedge X = S_1 \to S_2)
\end{array}
$$

since unification fails for all rules but Trans and Arrow.

[1] The condition on allowed sort eliminations is not relevant to the subject of this paper; the interested reader can find more information in (Werner, 1994) (for a general account of elimination in CIC) and (Asperti et al., 2009) (for the actual type system implemented in Matita).

Usually, in pen&paper proofs, it is inversion, and not induction, that is used in presence of judgements. The problem with inversion is that it does not provide inductive hypotheses over the new premises. Thus, most of the time, inversion on a judgement follows induction on the arguments of the judgement. For instance, the specification of POPLmark proves transitivity for $F_{<:}$ by induction over T followed by "induction with case analysis" (apparently similar to inversion) for $\Gamma \vdash S <: T$. Note, however, that the similarity may not be correct since inversion does not provide access to an "inner inductive hypothesis".

The natural question is then whether an inversion rule that also provides induction hypotheses is admissible. We call such a rule induction/inversion.

To answer the question, we will now consider how inversion is proved in terms of induction.

Given a predicate $P : \forall \vec{z_n} : \vec{T_n}.\mathcal{I}\ \vec{z_n} \Rightarrow \sigma$ and a vector of properly typed variables $\vec{x_n}$, we define the augmented predicate

$$\widehat{P}[\vec{x_n}] \triangleq \lambda \vec{z_n} : \vec{T_n}.\lambda z : \mathcal{I}\ \vec{z_n}.x_1 = z_1 \Rightarrow \ldots \Rightarrow x_n = z_n \Rightarrow P\ \vec{z_n}\ z$$

Depending on the actual arity of $\mathcal{I}$, the well typedness of $\widehat{P}$ might depend on the definition of $=$. In the general discussion of inversion and induction/inversion principles, we will assume that $=$ is John Major's equality: under this assumption, $\widehat{P}$ is always well typed

It is possible to prove the inversion principle applying the regular induction principle for $\mathcal{I}$ to the augmented predicate, as follows

$$\begin{array}{rcl} \widehat{\mathcal{E}}^{\tau}_{\mathcal{I}} & \triangleq & \lambda P, \vec{x_n}, x, \vec{H_m}.\mathcal{E}^{\tau}_{\mathcal{I}}(\vec{x_n}, x, \widehat{P}[\vec{x_n}])\{\vec{f_m}\}\ \mathcal{R}_{x_1} \cdots \mathcal{R}_{x_n} \\ & : & \forall P : (\forall \vec{z_n} : T_n[\vec{z_n}].\mathcal{I}\ \vec{z_n} \Rightarrow \sigma). \\ & & \forall \vec{x_n} : T_n[\vec{x_n}].\forall x : \mathcal{I}\ \vec{x_n}.\forall \vec{f_m} : \vec{H_m}.P\ \vec{x_n}\ x \end{array}$$

where the actual shape of the H_i is the one required by the induction principle and $\mathcal{R}_t$ is the trivial reflexivity proof of $t = t$. We will justify this definition later in the more general case of induction/inversion, however it is interesting to see the result in the case of the subtyping judgement of $F_{<:}$. Suppose for example we want to invert a hypothesis H stating $\Gamma \vdash T_1 \to T_2 <: T'_1 \to T'_2$ to prove P. Since induction over H ignores the actual family parameters Γ, $T_1 \to T_2$ and $T'_1 \to T'_2$, we are forced to generalise our goal to proving $(\Delta = \Gamma) \Rightarrow (T = T_1 \to T_2) \Rightarrow (T' = T'_1 \to T'_2) \Rightarrow P$ under the hypothesis $\Delta \vdash T <: T'$, and to perform induction thereafter. We only show two cases.

In the Top case we need to prove $(\Delta = \Delta) \Rightarrow (S = T_1 \to T_2) \Rightarrow (\texttt{Top} = T'_1 \to T'_2) \Rightarrow P$ (under two additional hypotheses), which is trivially done since the third premise is false. This corresponds to the case where first order unification (of `Top` and $T'_1 \to T'_2$) fails.

In the ARROW case, we need to prove $(\Delta = \Delta) \Rightarrow (S_1 \to S_2 = T_1 \to T_2) \Rightarrow (S_1' \to S_2' = T_1' \to T_2') \Rightarrow P$ under the additional hypotheses $\Delta \vdash S_1' <: S_1$, $\Delta \vdash S_2 <: S_2'$ and the relative induction hypotheses $(\Delta = \Delta) \Rightarrow (S_1' = T_1 \to T_2) \Rightarrow (S_1 = T_1' \to T_2') \Rightarrow P$ and $(\Delta = \Delta) \Rightarrow (S_2 = T_1 \to T_2) \Rightarrow (S_2' = T_1' \to T_2') \Rightarrow P$. The induction hypotheses are clearly inaccessible, since their second and third premises are false. Thus we can simply ignore them and propagate (i.e. apply) the equalities (the first order unifier) to reduce the thesis to P under the hypotheses $\Delta \vdash T_1' <: T_1$ and $\Delta \vdash T_2 <: T_2'$. Exactly what we expected from the inversion principle.

It is now clear why induction hypotheses are not provided by inversion rules: in the general case, they are dropped because inaccessible. This is what actually happens in the implementation of the Coq and Matita proof assistants, where inversion principles are automatically generated following the idea just described. However, there are situations where the induction hypothesis remains accessible. Indeed, in the previous example the inductive hypotheses became inaccessible since the second and third actual parameters of the inductive family in the conclusion $\Gamma \vdash S_1 \to S_2 <: T_1 \to T_2$ and in each premise ($\Gamma \vdash T_1 <: S_1$ and $\Gamma \vdash S_2 <: T_2$) of rule ARROW are different. On the other hand, since Γ remains Γ in the premises of the rule, the inductive hypotheses were guarded by a satisfiable premise $\Delta = \Delta$ which is trivially satisfied.

We can generalise the previous observation to obtain the following improved induction rule: if a family parameter is *globally constant*, i.e. it remains the same in each recursive occurrence of the inductive family in its derivation rules, then the family parameter is not quantified in each premise of the induction principle, but it occurs instantiated with the value of the actual parameter in the hypothesis the principle is applied to. This is indeed the case for the induction principles of the Coq and Matita theorem provers.[2]

Moreover, we can also derive the following induction/inversion principle we are interested in:

Theorem 1. (Induction/inversion principle) Given an inductive family, we say that a formal family parameter is *locally constant* to one premise of one derivation rule if its actual value does not differ from that in the conclusion of the rule. We get a different induction/inversion principle for each subset $\mathcal{S}$ of the family parameters subject to the restriction that, if the type of a family parameter in $\mathcal{S}$ depends on another

[2] This observation is actually internalised in the meta-theory of the Calculus of (Co)Inductive Constructions, that allows global universal quantifications for inductive families to simplify the implementation and to have more liberal type-checking rules.

family parameter, the latter must also be in $\mathcal{S}$. The principle has the shape of an inversion principle with additional, accessible induction hypotheses provided for all those recursive arguments whose locally constant parameters are a superset of $\mathcal{S}$. Moreover, as in inversion principles, in each case we get a unifier as a set of additional hypotheses $F_i = A_i$ where F_i is a family parameter in $\mathcal{S}$ and A_i is the corresponding actual parameter in the conclusion of the inference rule.

Proof. Let $\mathcal{I}$ be an inductive family with arity $\forall \vec{z_n} : \vec{T_n}.\sigma$ and constructors $k_i : K_i$, $i = 1, \ldots, m$. Let P be a predicate of type $\forall \vec{z_n} : \vec{T_n}.\mathcal{I}\ vecz_n \Rightarrow \tau$. Assume that $\mathcal{S} = \{s_1, \ldots, s_{|\mathcal{S}|}\}$ and that $x_1, \ldots, x_n$ are properly typed variables. We define $P_{\mathcal{S}}[\vec{x_n}]$ as the predicate P partially augmented over the set $\mathcal{S}$:

$$P_{\mathcal{S}}[\vec{x_n}] \triangleq \lambda \vec{z_n} : \vec{T_n}.\lambda z : \mathcal{I}\ \vec{z_n}.x_{s_1} = z_{s_1} \Rightarrow \ldots \Rightarrow x_{s_{|\mathcal{S}|}} = z_{s_{|\mathcal{S}|}} \Rightarrow P\ \vec{z_n}\ z$$

Then we can prove the $\mathcal{S}$-induction/inversion principle as follows

$$\begin{array}{rcl}
\mathcal{E}^{\tau}_{\mathcal{I},\mathcal{S}} & \triangleq & \lambda P : (\forall \vec{x_n} : \vec{T_n}.\mathcal{I}\ \vec{x_n} \Rightarrow \tau). \\
& & \lambda \vec{x_n} : \vec{T_n}.\lambda x : \mathcal{I}\ \vec{x_n}. \\
& & \lambda f_1 : \Delta^{P_{\mathcal{S}}[\vec{x_n}]}\{\emptyset; k_1 : K_1\}. \\
& & \cdots \\
& & \lambda f_m : \Delta^{P_{\mathcal{S}}[\vec{x_n}]}\{\emptyset; k_m : K_m\}. \\
& & \mathcal{E}^{\tau}_{\mathcal{I}}(\vec{x_n}, x, P_{\mathcal{S}}[\vec{x_n}])\{\vec{f_m}\}\ \mathcal{R}_{x_{s_1}} \cdots \mathcal{R}_{x_{s_{|\mathcal{S}|}}} \\
& : & \forall P : (\forall \vec{x_n} : \vec{T_n}.\mathcal{I}\ \vec{x_n} \Rightarrow \tau). \\
& & \forall \vec{x_n} : \vec{T_n}.\forall x : \mathcal{I}\ \vec{x_n}. \\
& & \Delta^{P_{\mathcal{S}}[\vec{x_n}]}\{\emptyset; k_1 : K_1\} \Rightarrow \\
& & \cdots \\
& & \Delta^{P_{\mathcal{S}}[\vec{x_n}]}\{\emptyset; k_m : K_m\} \Rightarrow \\
& & P\ \vec{x_n}\ x
\end{array}$$

To show that this definition satisfies the specification of an $\mathcal{S}$-induction/inversion principle, we must prove that:

1. The type of each subcase f_i is provided with an equational hypothesis for each index j in the set $\mathcal{S}$, stating that the family parameter of index j of the inverted term is equal to the corresponding paremeter in the target type of k_i. But this is, by definition, the role of the partially augmented predicate $P_{\mathcal{S}}$. If $K_i : \forall \vec{y} : \vec{U}.V\ \vec{u}$, then the type of f_i is of the form

$$\begin{array}{l}
\forall \vec{y} : \vec{U}.IH_1 \Rightarrow \cdots \Rightarrow \cdots IH_r \\
\quad \Rightarrow x_{s_1} = u_{s_1} \Rightarrow \cdots \Rightarrow x_{s_{|\mathcal{S}|}} = u_{s_{|\mathcal{S}|}} \\
\quad \Rightarrow P\ \vec{u}\ (k_i\ \vec{y})
\end{array}$$

thus satisfying the request.

2. Induction hypotheses such that all the parameters with index in $\mathcal{S}$ are locally constant are accessible. For this to happen, induction hypotheses must be in the form

$$IH \triangleq \forall \vec{a}.x_{s_1} = u_{s_1} \Rightarrow \cdots \Rightarrow x_{s_{|\mathcal{S}|}} = u_{s_{|\mathcal{S}|}} \Rightarrow P\ \vec{u_n}\ v$$

such that $FV(u_{s_i}) \cap \vec{a} = \emptyset$ for all i, and the goal must be

$$x_{s_1} = u'_{s_1} \Rightarrow \cdots \Rightarrow x_{s_{|\mathcal{S}|}} = u'_{s_{|\mathcal{S}|}} \Rightarrow P\ \vec{u'_n}\ v'.$$

such that for all $i \in \mathcal{S}$, $u_i = u'_i$. We can then introduce from the goal the new hypotheses $e_{|\mathcal{S}|}^{\rightarrow}$ and feed them to IH, obtaining

$$IH' \triangleq \lambda \vec{a}.IH\ \vec{a}\ e_{|\mathcal{S}|}^{\rightarrow} : \forall \vec{a}.P\ \vec{u_n}\ v$$

whose shape is the same of a regular, accessible induction hypothesis.

As an example, that we have used in our solution to the POPLmark challenge, we show the induction/inversion principle for the sub-typing judgement of Fig. 8 where we choose $\mathcal{S} = \{\Gamma, T\}$ (i.e. the typing context and the second type). The choice is driven by the TRANS rule where Γ and T are locally constant parameters, whereas the second argument is not (being U in the premise, and X in the conclusion). Indeed, note that we get an almost-standard inversion principle were we have traded hypotheses on L with the induction hypothesis in the **Trans** case.

Theorem 2. ($\{\Gamma, T\}$-Induction/inversion for $\Gamma \vdash S <: T$) Let P be a ternary predicate over triples (Δ, L, R). For all Δ, L, R we have $\Delta \vdash L <: R$ implies $P(\Delta, L, R)$ provided that

Top $\forall \Gamma, S.\ \vdash \Gamma \Rightarrow \Gamma \vdash S \Rightarrow (\Delta = \Gamma) \Rightarrow (R = \mathtt{Top}) \Rightarrow P(\Gamma, S, \mathtt{Top})$

Refl $\forall \Gamma, X.\ \vdash \Gamma \Rightarrow X \in dom\Gamma \Rightarrow (\Delta = \Gamma) \Rightarrow (R = X) \Rightarrow P(\Gamma, X, X)$

Trans $\forall \Gamma, X, U, T,\ X <: U \in \Gamma \Rightarrow \Gamma \vdash U <: T \Rightarrow P(\Gamma, U, T) \Rightarrow (\Delta = \Gamma) \Rightarrow (R = T) \Rightarrow P(\Gamma, X, T)$

Arrow $\forall \Gamma, S_1, S_2, T_1, T_2.\ \Gamma \vdash T_1 <: S_1 \Rightarrow \Gamma \vdash S_2 <: T_2 \Rightarrow (\Delta = \Gamma) \Rightarrow (R = T_1 \to T_2) \Rightarrow P(\Gamma, S_1 \to S_2, T_1 \to T_2)$

All $\forall \Gamma, S_1, S_2, T_1, T_2.\ \Gamma \vdash T_1 <: S_1 \Rightarrow (\forall X, X \notin dom(\Gamma) \Rightarrow \Gamma, X <: T_1 \vdash S_2[^X/_{\#0}] <: T_2[^X/_{\#0}]) \Rightarrow (\Delta = \Gamma) \Rightarrow (R = \forall_{T_1}.T_2) \Rightarrow P(\Gamma, \forall_{S_1}.S_2, \forall_{T_1}.T_2)$

It is now clear that this induction/inversion lemma is exactly what we need to justify the informal proof, since it allows to use the "inner hypothesis" (in the **Trans** case), but also to (partially) perform "case analysis on the final rule of the derivation". What is surprising at first is that such a proof principle, that seems quite ad-hoc in the informal proof, is actually a general proof principle. Indeed, we want to remark two additional facts.

The first one is that these induction/inversion rules can be automatically generated from the derivation rules of the judgement and, as well as the standard induction and inversion rules, are fully determined once the judgement is inductively defined. On the other hand, when the judgement has n family parameters, we can generate 2^n different induction/inversion principles. Indeed, standard induction and standard inversion corresponds respectively to the empty and full sets of family parameters. A first observation to reduce the number of principles to generate is that a set of induction/inversion principles makes sense only if its elements provide different induction hypotheses. In turn, this depends on the variety of locally constant parameters in the rules. For instance, in the case of our sub-typing judgement, only three induction/inversion principles have to be considered: standard induction, standard inversion and $\{\Gamma, T\}$-induction/inversion. Even if a large number of principles are worth generating, we can expect the proof assistant to dynamically generate them when needed.

The second fact is that, as far as we know, the conditions for induction/inversion principles have never been characterised before. However, we have detected them in other proofs about the meta-theory of programming languages, such as the ones on LambdaDelta by F. Guidi(Guidi, 2006). We claim that better knowledge on them could easily result in shorter and deeper proofs, as the ones we present here.

The generation of induction/inversion principles is performed by Matita on demand, using the inverter command.

4.3. Proofs

In this section we will discuss briefly the main proofs of the solutions to POPLmark part 1a that we have formalized in Matita, based on the proof techniques of Section 4.2. The formal proofs in Matita can be retrieved from the Web pages of the Matita proof assistant. We will begin with the locally nameless solution, which is in some sense more basic than the other two.

4.3.1. *Locally nameless*
The first property we must show is the reflexivity of subtyping.

Theorem 3. (reflexivity (locally nameless encoding)) Let Γ be a typing environment, and T a type; if Γ is well-formed and T is well-formed in Γ, then $\Gamma \vdash T <: T$.

Proof. Once it has been proved that, for all Γ and T, $\Gamma \vdash T$ implies $\mathrm{FV}(T) \subseteq \mathrm{dom}(\Gamma)$, the proof is trivial by induction on the derivation of $\Gamma \vdash T$. Matita is able to prove almost every case of the induction by means of standard automation.

The following theorem asserts a stronger weakening property than the one described in the specifications: here weakening on well formed environments is defined as set inclusion, instead of concatenation of two disjoint environments. In this way we are exempted from proving the less tractable lemma on permutations.

Theorem 4. (weakening (locally nameless encoding))

1. Let Γ be a typing environment, T a type. If $\Gamma \vdash T$, then for all environments Δ such that $\Gamma \subseteq \Delta$, we get $\Delta \vdash T$.
2. Let Γ be a typing environment, T, U types. If $\Gamma \vdash T <: U$, for all well-formed environments Δ such that $\Gamma \subseteq \Delta$, we get $\Delta \vdash T <: U$.

Proof. The first point follows easily from a straightforward induction on the derivation of $\Gamma \vdash T$. The second point follows from an induction on the derivation of $\Gamma \vdash T <: U$ (the proposition proved in part (i) is used in the Top case). Once again standard automation turns out to be very useful.

Unlike the specifications, we decided to prove narrowing and transitivity separately. Our statements are also slightly stronger than the ones provided in the specifications. This is ultimately due to the locally nameless encoding: in fact, since the encoding of the All rule is not fully structural with respect to the types mentioned in it, the induction on the structure of a type, used in the informal proof, is not sufficient to prove narrowing and transitivity in our setting. Instead, we will use an induction on the derivation of some judgements.

Theorem 5. (narrowing (locally nameless encoding)) For all typing environments Γ, Γ', for all types U, P, M, N and for all variables X, if

1. $\Gamma' \vdash P <: U$
2. $\Gamma \vdash M <: N$
3. for all Γ'', T if $\Gamma', \Gamma'' \vdash U <: T$ then $\Gamma', \Gamma'' \vdash P <: T$

then forall Δ s.t. $\Gamma = \Gamma', X <: U, \Delta$, the judgement $\Gamma', X <: P, \Delta \vdash M <: N$ holds.

Proof. We proceed by induction on the derivation of $\Gamma \vdash M <: N$. The interesting case is SA-Trans-TVar: in this case, $M = Y$, where Y is a type variable. If $X = Y$, we prove the statement by means of rule SA-Trans-TVar. Since $X <: P \in (\Gamma', X <: P, \Delta)$ (trivially), we only need to prove $\Gamma', X <: P, \Delta \vdash P <: N$: this is obtained by means of hypothesis 3 ($\Gamma', X <: P, \Delta \vdash U <: N$ holds by induction hypothesis). If $X \neq Y$, the goal is obtained trivially by induction hypothesis.

Last, we turn to proving the main property, i.e. transitivity of subtyping. Again, we use a slightly different statement from the specifications, but the proof follows closely the suggested structure.

Theorem 6. (transitivity (locally nameless encoding)) Let T a type, Γ' a typing environment such that $\Gamma' \vdash T$. For any typing environment Γ such that $dom(\Gamma') \subseteq dom(\Gamma)$, and for all types R, U, if $\Gamma \vdash R <: T$ and $\Gamma \vdash T <: U$ then $\Gamma \vdash R <: U$.

Proof. We proceed by induction on the derivation of $\Gamma' \vdash T$, followed by $\{\Gamma, T\}$-induction/inversion on $\Gamma \vdash R <: T$. The interesting case is WFT-All: in this case, $T = \forall_{T'}.T''$ and, by applying the unifier provided by the principle, only two cases are possible:

- $R = X$ (where X is a type variable) and $X <: V \in \Gamma$ (for some type V). The thesis follows from the induction/inversion hypothesis, by means of rule Trans.
- $R = \forall_{R'}.R''$. In this case, by inversion on $\Gamma \vdash \forall_{T'}.T'' <: U$ we show U is either `Top` or $\forall_{U'}.U''$. In the first case, showing that $\Gamma \vdash \forall_{R'}.R'' <:$ `Top` is trivial. In the second case, the difficult part is to show that, for all $X \notin dom(\Gamma)$, $\Gamma, X <: U' \vdash R''\{^X/_{\#0}\} <: U''\{^X/_{\#0}\}$. By the induction hypothesis, we only need to prove $\Gamma, X <: U' \vdash R''\{^X/_{\#0}\} <: T''\{^X/_{\#0}\}$ and $\Gamma, X <: U' \vdash T''\{^X/_{\#0}\} <: U''\{^X/_{\#0}\}$: these follow from the induction/inversion hypothesis, together with narrowing. Please notice that the hypothesis $dom(\Gamma') \subseteq dom(\Gamma)$, here, is essential, since otherwise the typing environments in the induction hypothesis and in the goal would not match.

When proving reflexivity and transitivity, our formalization of the ALL rule requires to prove that some judgement holds for any fresh variable X. As we pointed out in section 4.1, since the subtyping judgement is equivariant, it is sufficient that it hold for one fresh X: following this intuition, Leroy, in his solution to the challenge, decided to prove this alternate "for one" rule. Apparently, this should have simplified the proofs of reflexivity and transitivity, thus in a previous version of our solution we decided to follow closely his approach; however, proving the "for one" rule required a great effort (approximately 500 lines of code out of 1500). Moreover, proofs can be completed quite easily even without the "for one" rule. The most difficult case is probably in the reflexivity: we must prove that $\Gamma \vdash \forall_T.U <: \forall_T.U$, knowing (by hypothesis) that $\forall_T.U$ is well-formed in Γ, and (by induction hypothesis) that for any $X \notin dom(\Gamma) \cup \mathrm{FV}(U)$, $\Gamma, X <: T \vdash U[{}^{X}/_{\#0}] <: U[{}^{X}/_{\#0}]$ holds; now if we apply the "for one" version of the rule, it's sufficient to prove that for some $Y \notin dom(\Gamma)$, the judgement $\Gamma, Y <: T \vdash U[{}^{Y}/_{\#0}] <: U[{}^{Y}/_{\#0}]$ holds – then we choose Y to be fresh both in Γ and T, and the thesis follows trivially from the induction hypothesis; using the original ALL rule is only apparently more difficult: we need to prove the same judgement for any $Y \notin dom(\Gamma)$ but, since $\forall T.U$ is well-formed in Γ, one can easily prove that no variables outside Γ can be free in U, thus the induction hypothesis is sufficient even in this case.

4.3.2. *De Bruijn nameless encoding*
While the concern about readability of terms containing nameless dummies, which is also brought against locally nameless solutions, is debatable, the fact that de Bruijn open terms must be explicitly lifted when the environment is changed, is a serious matter. The statement of theorems must be carefully tuned and while we don't feel that the readability of the proofs is compromised, the ease of formalization is impaired to some extent. Still, formalization of the properties of dangling dummies has some interest *per se*, and theorem provers provide all the tools to carry on their proofs.

The proof of reflexivity in the de Bruijn encoding is even easier than in the locally nameless encoding (exactly 3 proof steps in Matita); weakening, however, is the typical example of a theorem whose statement must be somewhat reworked in the de Bruijn encoding. The relation $\Gamma \subseteq \Delta$ we had used in the locally nameless version, denoting that Γ is extended by Δ, possibly with some entries permuted, is not meaningful for de Bruijn environments: while the entries of a locally nameless environment can be permuted consistently without updating the free names, in a de Bruijn environment the dangling dummies must be also permuted explicitly.

We are tempted to state weakening by means of environment concatenation and lifting:

For all environments Γ, Γ' *and types* S, T*, if* $\Gamma \vdash S <: T$ *and* $\Gamma, \Gamma' \vdash \Diamond$*, then* $\Gamma, \Gamma' \vdash S \uparrow |\Gamma'| <: T \uparrow |\Gamma'|$.

The statement is correct but its proof (as noted in the POPLmark specifications (Aydemir et al., 2005)) requires a permutation lemma which is precisely what we wanted to avoid in the first place.

The best we can do is to prove a strong version of weakening, implying the permutation lemma, just like we did in the locally nameless formalization. However, the notion of environment inclusion needs to be significantly refined. On the other hand, lifting is not sufficient to deal with permutations, and a generalization is needed.

DEFINITION 2. *The* map application *of a function* $f : \mathbb{N} \to \mathbb{N}$ *on* $F_{<:}$ *types is defined as follows:*

$$f \cdot T = \begin{cases} f \cdot n & = f(n) \\ f \cdot \mathtt{Top} & = \mathtt{Top} \\ f \cdot (T \to U) & = f \cdot T \to f \cdot U \\ f \cdot (\forall_T . U) & = \forall_{f \cdot T} . (f \cdot U) \end{cases}$$

DEFINITION 3. *A function* $f : \mathbb{N} \to \mathbb{N}$ *is an* environment extension map *from* Γ *to* Δ *(notation:* $\Gamma \subseteq_f \Delta$*) if and only if it is injective and for all* $n < |\Gamma|$*,* $f(n) < |\Delta|$ *and* $f \cdot (\Gamma(n) \uparrow n+1) = \Delta(f(n)) \uparrow f(n)+1$.

In simple terms, an environment extension map is a more explicit version of the $\subseteq$ relation used in the locally nameless formalization. Its definition can be paraphrased by saying that for every n, the n-th entry of Γ, relocated at the top level (by lifting) and then mapped to the environment Δ (by means of f) must be equal to the $f(n)$-th entry of Δ, relocated to the top level (again by lifting).

We can then prove weakening in the following form.

Theorem 7. (weakening (de Bruijn encoding)) For all environments Γ, Δ*, if for some* f*,* $\Gamma \subseteq_f \Delta$*,* $\Gamma \vdash S <: T$ *and* $\Delta \vdash \Diamond$*, then* $\Delta \vdash f \cdot S <: f \cdot T$.

While the proof of the above statement is similar to its locally nameless counterpart, automation turns out to be a bit less effective.

It can be noted that lifting and environment extension maps have an interesting relation. Let $\Uparrow_k^m : \mathbb{N} \to \mathbb{N}$ be the family functions defined as follows:

$$\Uparrow_k^m (n) = \begin{cases} m+n & \text{if } k \leq n \\ n & \text{else} \end{cases}$$

We show that for all types T, $T \uparrow_k m = \Uparrow_k^m \cdot T$. As a corollary, for all environments Γ, Δ, $\Gamma \subseteq_{\Uparrow_0^{|\Delta|}} \Gamma, \Delta$: therefore the version of weakening involving environment extension maps also implies the previous statement with concatenation of environments and lifting.

Narrowing and transitivity are then proved separately, following the same strategy, if not the letter, of the locally nameless proofs.

Theorem 8. (narrowing (de Bruijn encoding)) For all typing environments Γ,Γ' and for all types U, P, M, N, if

1. $\Gamma' \vdash P <: U$
2. $\Gamma \vdash M <: N$
3. for all Γ'', S, T, if $\Gamma', \Gamma'' \vdash S <: (U \uparrow |\Gamma''|)$ and $\Gamma', \Gamma'' \vdash (U \uparrow |\Gamma''|) <: T$ imply $\Gamma', \Gamma'' \vdash S <: T$

then for all Δ s.t. $\Gamma = \Gamma', \bullet <: U, \Delta$, the judgement $\Gamma', \bullet <: P, \Delta \vdash M <: N$ holds.

Theorem 9. (transitivity (de Bruijn encoding)) Let S, T, U be types, Γ a typing environment, f a function from naturals to naturals. If $\Gamma \vdash S <: f \cdot T$ and $\Gamma \vdash f \cdot T <: U$, then $\Gamma \vdash S <: U$.

Somewhat surprisingly, the above statement of transitivity does not require f to be an environment extension map: it is sufficient for f to be a function from naturals to naturals. The particular statement of the theorem is needed in order to get an induction hypothesis which is sufficiently strong to imply the weak transitivity requirement of the previous narrowing theorem. The proof also exploits the $\{\Gamma, T\}$-induction/inversion principle, similarly to the corresponding proof in the locally nameless encoding.

The usual statements of transitivity and narrowing are then obtained as corollaries.

4.3.3. *Named variables*

Our solution using named variables follows a radically different approach from the other two: instead of proving the transitivity of subtyping directly on types with named variables, we decided to provide a translation of types with named variables to locally nameless types. This induces a translation of environments and, consequently, a translation of whole subtyping judgements: if we can prove that the subtyping judgement on types with named variables is adequate and faithful with respect to the corresponding judgement on locally nameless types, we

can obtain the transitivity on types with named variables as a corollary, from the transitivity on locally nameless types.

This kind of formalization, similar to transformations performed by actual compilers, has an interest in itself and hides some difficulties: therefore it seemed to be a good companion to the problems of the POPLmark challenge.

First, we need to define an algorithm providing the intended encoding of types with named variables into locally nameless types.

$$\llbracket T \rrbracket_\ell = \begin{cases} \llbracket \mathtt{Top} \rrbracket_\ell & = \mathtt{Top} & \\ \llbracket X \rrbracket_\ell & = \#n & \text{if } n = posn(X, \ell) \\ \llbracket X \rrbracket_\ell & = X & \text{if } X \notin \ell \\ \llbracket T' \to T'' \rrbracket_\ell & = \llbracket T' \rrbracket_\ell \to \llbracket T'' \rrbracket_\ell & \\ \llbracket \forall X_{T'}.T'' \rrbracket_\ell & = \forall_{\llbracket T' \rrbracket_\ell}.\llbracket T'' \rrbracket_{X,\ell} & \end{cases}$$

$$\llbracket \Gamma \rrbracket = \begin{cases} \llbracket \emptyset \rrbracket & = \emptyset \\ \llbracket \Gamma', x : T \rrbracket & = \llbracket \Gamma' \rrbracket, x : \llbracket T \rrbracket \\ \llbracket \Gamma', X <: T \rrbracket & = \llbracket \Gamma' \rrbracket, X <: \llbracket T \rrbracket \end{cases}$$

where ℓ is a list of names used to trace non-locally bound variables. The encoding of a type is obtained beginning with ℓ being empty and it is denoted by $\llbracket \cdot \rrbracket$; the list is updated with a new variable whenever we enter the scope of a quantifier; the encoding $\llbracket X \rrbracket_\ell$ is X when $X \notin \ell$ (meaning X is a free variable); if $X \in \ell$, the encoding $\llbracket X \rrbracket_\ell$ is $\#n$, where n is the position of X in ℓ (meaning X is a bound variable, and the "distance" of its binder is n); the encoding of a type commutes with all other constructs. The encoding of an environment is the same environment where every (sub)typing bound has been replaced by its encoding.

We can also show that this translation is surjective: for every locally closed type T in the locally nameless representation, there exists a type T' in the named variables representation, such that $\llbracket T' \rrbracket = T$.

The key lemma we need to prove adequacy is the following:

Theorem 10. For all variables X,Y and types T, if $X \in FV(T) \Rightarrow X = Y$, then $\llbracket (X\ Y) \cdot T \rrbracket = \llbracket T \rrbracket_Y [{}^X/_{\#0}]$.

Proof. Actually, the theorem is obtained as a corollary from a stronger statement:

> *Given a list of variables ℓ, if X and Y are not in ℓ and $X \in \mathrm{FV}(T) \Rightarrow X = Y$, then $\llbracket (X\ Y) \cdot T \rrbracket_\ell = \llbracket T \rrbracket_{\ell,Y} [{}^X/_{\#|\ell|}]$.*

Our proof is by induction on the *weight* of T, then by case analysis again on T. The weight of T is defined as follows:

$$\|T\| = \begin{cases} \|X\| &= 0 \\ \|\mathtt{Top}\| &= 0 \\ \|U \to V\| &= \max(\|U\|, \|V\|) + 1 \\ \|\forall X <: U.V\| &= \max(\|U\|, \|V\|) + 1 \end{cases}$$

It is necessary to go through a large number of cases, depending on whether variables in T are equal to X, Y or to some variable in ℓ. The full proof is beyond the scope of this paper.

Theorem 11. (adequacy and faithfulness)

1. Let Γ be a typing environment, T, U types in the named presentation. If $\Gamma \vdash T <: U$, then $[\![\Gamma]\!] \vdash [\![T]\!] <: [\![U]\!]$.

2. Let Γ be a typing environment, T, U types in the locally nameless encoding. Let Γ', T', U' such that $\Gamma = [\![\Gamma']\!]$, $T = [\![T']\!]$ and $U = [\![U']\!]$. If $\Gamma \vdash T <: U$, then $\Gamma' \vdash T' <: U'$.

Proof. Adequacy is proved by a straightforward induction on the derivation of $\Gamma \vdash T <: U$. Almost all the cases are easy (and are proved automatically by Matita), except for the "for all" case, requiring us to prove prove that

$$[\![\Gamma]\!] \vdash \forall_{[\![S_1]\!]}.[\![S_2]\!]_X <: \forall_{[\![T_1]\!]}.[\![T_2]\!]_Y$$

under the following induction hypotheses

$$\begin{array}{ll} IH_1: & [\![\Gamma]\!] \vdash [\![T_1]\!] <: [\![S_1]\!] \\ IH_2: & \text{for all } Z \notin \mathrm{FV}(\Gamma): \\ & (Z \in \mathrm{FV}(S_2) \Rightarrow Z = X) \Rightarrow \\ & (Z \in \mathrm{FV}(T_2) \Rightarrow Z = Y) \Rightarrow \\ & [\![\Gamma]\!], Z <: [\![T_1]\!] \vdash [\![(Z\ X) \cdot S_2]\!] <: [\![(Z\ Y) \cdot T_2]\!] \end{array}$$

By SA-All and IH_1, we reduce to the problem of proving

$$\text{for all } W \notin \mathrm{FV}(\Gamma): [\![\Gamma]\!], W <: [\![T_1]\!] \vdash [\![S_2]\!]_X[^W/_{\#0}] <: [\![T_2]\!]_Y[^W/_{\#0}]$$

This follows easily from IH_2 by means of theorem 10.

The proof of faithfulness basically mirrors that of adequacy and is performed by providing an algorithm to compute the backwards encoding of a locally closed type.

5. Conclusions

The POPLmark challenge proved to be a valuable test-bench for the Matita theorem prover. During the development of the solution a few bugs and malfunctioning have been detected and corrected, especially concerning tactics like *inversion* or *destruct*, less frequently used in developments of a more mathematical nature. Moreover, we have identified a new proof principle, that we called *induction/inversion* and that we plan to implement in Matita. The principle seems to have been implicitly adopted in several solutions, but never made explicit before. We believe that our proof where it is explicit is not only easier to understand, but also more faithful to the informal proof of the POPLmark specification.

We must stress that the absence of agreed principles to evaluate solutions to the challenge makes it difficult to draw conclusions. As a purely indicative data, in Table I we compare the size of our solution with other solutions written in Coq (whose syntax is the closest to Matita). Remarkably, the size of our locally nameless solution is 350 lines (slightly less than 4kB compressed), which is the most compact among those based on the same encoding. That is to say that, in spite of a sensibly simpler architecture design resulting in about half of the code of Coq, the functionalities offered by Matita and the expressiveness of its tactical language are fully comparable with the former. Therefore, we believe that Matita brilliantly overcame the test, proving to be a competitive tool for the verification of properties of programming languages and the automation of formal reasoning.

Our de Bruijn solution has not been tuned for compactness and is still relatively lengthy (with a size comparable to that of the second locally nameless solution): in general, de Bruijn formalizations have proved to be more synthetic than locally nameless ones, and we believe that automation might help reducing its size significantly.

The solution using named variables is not comparable to the other ones, since it is based on a completely different proof strategy. Its size amounts to 1270 lines (9706 bytes compressed), which confirms that handling of named variables comes at a cost.

As for the encoding issue, as already pointed out by other authors, we agree that the locally nameless approach leads to proofs which are more readable (in comparison with de Bruijn's representation): this is due to the fact that we do not have to deal with free indices. We also believe that while it may be possible to obtain a smaller solution using pure de Bruijn's approach, the proofs tend to become much less linear, making the locally nameless approach still preferable.

Table I. Size comparison between solutions of the challenge in Matita (ours) and in Coq (all others)

Author	Lines	GZip'ed	Encoding technique
Hirschowitz & Maggesi	293	2906	de Bruijn (nested datatypes)
Ricciotti	**350**	**3931**	**locally nameless**
Charguéraud	465	4058	de Bruijn
Ricciotti	**576**	**5382**	**de Bruijn**
Charguéraud	630	5272	locally nameless (cofinite)
Vouillon	796	6231	de Bruijn
Chlipala	923	4729	locally nameless
Stump	1256	8025	named variables
Leroy	1528	13488	locally nameless
Sallinens	2045	12384	de Bruijn

The biggest drawback with the locally nameless approach is that typing rules which deal with binders are required (reasoning backwards) to make disappear indices, in such a way that they are not fully structural on the types. This means that where the paper proof would use a straight induction on a type, we are required to use an induction on its well-formedness derivation. For the same reasons, the cases with binders are also the most difficult to deal with in the adequacy proof of the named variables encoding with respect to the locally nameless encoding.

References

Asperti, A., W. Ricciotti, C. Sacerdoti Coen, and E. Tassi: 2009, 'A compact kernel for the Calculus of Inductive Constructions'. *Sadhana* **34**(1), 71–144.

Asperti, A., C. Sacerdoti Coen, E. Tassi, and S. Zacchiroli: 2006, 'Crafting a Proof Assistant'. In: *Proceedings of Types 2006: Conference of the Types Project.*

Asperti, A., C. Sacerdoti Coen, E. Tassi, and S. Zacchiroli: 2007, 'User Interaction with the Matita Proof Assistant'. *Journal of Automated Reasoning, Special Issue on User Interfaces for Theorem Proving.* To appear.

Aydemir, B., A. Bohannon, M. Fairbairn, J. Foster, B. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic: 2005, 'Mechanized Metatheory for the Masses: The POPLmark Challenge'. In: *Proceedings of the Eighteenth International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2005).*

Aydemir, B. E., A. Charguéraud, B. C. Pierce, R. Pollack, and S. Weirich: 2008, 'Engineering formal metatheory'. In: G. C. Necula and P. Wadler (eds.): *POPL*. pp. 3–15.

Cardelli, L., S. Martini, J. C. Mitchell, and A. Scedrov: 1991, 'An Extension of System F with Subtyping (TACS)'. In: T. Ito and A. R. Meyer (eds.): *Proc. of 1st Int. Symp. on Theor. Aspects of Computer Software, TACS'91, Sendai, Japan, 24–27 Sept 1991*, Vol. 526 of *?* Berlin: Springer-Verlag, pp. 750–770.

Coq: 2004, 'The Coq Proof Assistant Reference Manual – Version 8.0'. The Coq Development Team.

Cornes, C. and D. Terrasse: 1996, 'Automating Inversion of Inductive Predicates in Coq'. In: *TYPES '95: Selected papers from the International Workshop on Types for Proofs and Programs*. London, UK, pp. 85–104.

Gabbay, M. J. and A. M. Pitts: 1999, 'A New Approach to Abstract Syntax Involving Binders'. In: *14th Annual Symposium on Logic in Computer Science*. Washington, DC, USA, pp. 214–224.

Guidi, F.: 2006, 'Lambda Types on the Lambda Calculus with Abbreviations'. Research report UBLCS-2006-25, Department of Computer Science, University of Bologna.

Leroy, X.: 2007, 'A locally nameless solution to the POPLmark challenge'. Research report 6098, INRIA.

McBride, C.: 2002, 'Elimination with a Motive'. In: P. Callaghan, Z. Luo, J. McKinna, and R. Pollack (eds.): *Types for Proofs and Programs (Proceedings of the International Workshop, TYPES'00)*, Vol. 2277 of *LNCS*.

Pitts, A. M.: 2003, 'Nominal Logic: A First Order Theory of Names and Binding'. *Inf. Comput.* **186**(2), 165–193.

Pollack, R.: 1993, 'Closure Under Alpha-Conversion'. In: H. Barendregt and T. Nipkow (eds.): *Proceedings of the Workshop on Types for Proofs and Programs*. Nijmegen, The Netherlands, pp. 313–332.

Sacerdoti Coen, C., E. Tassi, and S. Zacchiroli: 2006, 'Tinycals: step by step tacticals'. In: *Proceedings of User Interface for Theorem Provers 2006*.

Urban, C. and R. Pollack: 2007, 'Strong Induction Principles in the Locally Nameless Representation of Binders'. In: *Workshop on Mechanized Metatheory*.

Werner, B.: 1994, 'Une Théorie des Constructions Inductives'. Ph.D. thesis, Université Paris VII.