

Implementing the Calculus of Inductive Constructions in the MetaPRL Framework

Natalia Novak and Yegor Bryukhov

Graduate Center, City University of New York
365 Fifth Avenue, New York, NY 10016
nnovak@gc.cuny.edu
ybryukhov@gc.cuny.edu

Abstract. The Calculus of Inductive Constructions is an underlying logic of the `Coq` proof assistant — a widely used mature proof assistant. In this paper we present our work on implementing the Calculus of Inductive Constructions in the `MetaPRL` logical framework. Rules from the `Coq` reference manual have quite unrestricted format so we have to make certain design decisions in order to express those rules in the plain Gentzen style supported by `MetaPRL`. The most complicated case-analysis and fixpoint rules have yet to be implemented. There is a working implementation with rudimentary proof automation; the toy example of inductive definition (parameterized lists) is type-checked.

1 Introduction

`MetaPRL` [5,7] is a relatively young logical framework from the PRL family [2] originated at Cornell University.

Among logical theories already defined in `MetaPRL` there are

- NuPRL-like Computational Type Theory CTT (based on Martin-Löf’s Intuitionistic Type Theory);
- the constructive set theory CZF, based on Aczel’s axiomatization;
- the First Order Logic.

`MetaPRL` was designed to address scalability and efficiency issues of NuPRL; as a result of these efforts CTT in `MetaPRL` is two decimal orders of magnitude faster than in NuPRL [6].

The `Coq` proof assistant [8] is a widely used mature logical system. Its underlying logic is the Calculus of Inductive Constructions (CIC) [8,3,4,10,1]. CIC is a rather sophisticated and powerful system. Implementing CIC in `MetaPRL` is the natural next step in developing the latter. It would be a good test for `MetaPRL`’s universality and a challenge for a fast `MetaPRL` proof engine. It could help `MetaPRL` to import `Coq`’s vast formal libraries.

In this paper we discuss our pilot implementation of CIC in the `MetaPRL` logical framework. We have a working code (rules, rewrites and tactics) that implements lambda calculus and inductive definitions. Implementation of inductive

definitions is not complete. We implemented rule about correctness of an inductive definition, typechecking of inductive types and constructors. Case-analysis and fixpoint are not supported yet.

2 MetaPRL meta-language

A brief syntax description of MetaPRL will give a better understanding of implementation problems and their solutions further in the article. *Terms* have the following syntax:

$$term ::= operator \{ bterms \}$$

where the *operator* represents the *name* of a term and *bterms* are possibly bound terms.

Bound terms have the following syntax:

$$bterm ::= term \mid vars.term$$

For bound term $v_1, \dots, v_n.t$ variables v_1, \dots, v_n are bound in t . Such binding is the part of signature (arity) of the outer operator. For example, $\forall x : T.P(x)$ can be expressed as $forall\{T; x.P[x]\}$, where *forall* has arity (0,1) — no bindings in the first subterm and one binding in the second subterm.

Variables are special terms treated specifically by the system. There are two types of variables: *first-order* variables represent variables of the *object theory*, *second-order* variables (*meta-level* variables) represent terms with substitutions.

A theory is defined by its inference rules and computational equivalences between terms. The syntax of an inference rule is

$$\mathbf{rule} \ name \ [params] : inference$$

where *name* is the name of the rule, *params* are extra parameters passed to the inference rule (optional) and *inference* is a valid inference in the defined logic. Inference is declared in the following form:

$$inference ::= term \mid term \rightarrow inference$$

Inference rules can be derived from previous rules or they can be defined as a primitive axioms of the theory.

Rewrites can be used to establish computational and/or definitional equality between certain terms. Rewrites are declared as follows:

$$\mathbf{rewrite} \ name \ [params] : [conditions] redex \leftrightarrow contractum$$

where *name* is a name, *params* are extra variables and terms needed in rewrite and if the rewrite is conditional then the condition is stated in *conditions*. Rewrite replaces redex with contractum in any context. Just like rules rewrites can be primitive or derived. Rewrites and inference rules are logical inferences of MetaPRL.

Sequent schema language [9] is used for specifying new inference rules in a theory. The extension of the theory with sequents is conservative and derived rules can be used as primitive axioms [9]. The *sequent* syntax is:

$$\mathbf{sequent}[name]\{H_1; \dots; H_n \vdash C\}$$

where *name* is a name of a sequent (optional), which can be used to assign different semantics to differently named sequents. Each of $H_1; \dots; H_n$ is either a variable declaration (hypothesis) or a sequent context, and C is a conclusion. Contexts are meta-variables that are used as placeholders for sequences of hypotheses (again variable declarations and contexts). A variable declaration $x : T$ introduces a variable x bound in the rest of the sequent.

One can think of sequents as a special kind of terms with flexible arity, where *name* is an analogue of operator and “sequent” indicates that this is a special kind of term (with flexible arity). It is more convenient to look at sequents in this perspective for the rest of the article.

There is a discipline of specifying permitted dependencies of a context or a second-order variable on all contexts and declarations from the left of it. We say that a context Γ (a second order variable A) can depend on variable declaration $x : T$ if x is allowed to occur in Γ (in A). We indicate it by $\Gamma[x]$ and $A[x]$. If variable x declared before Γ (before A) is not listed in brackets it is interpreted as prohibition of free occurrences of x in Γ (in A).

We say that a context Δ (a second order variable A) can depend on a context Γ if it is allowed for variables potentially declared in Γ to occur freely in Δ (in A). We indicate it by $\Delta_{\{\Gamma\}}$ and $T_{\{\Gamma\}}$. If a context Γ declared before Δ (before A) and Γ is not listed in curly brackets after Δ (after A) it is prohibited for variables potentially declared in Γ to occur freely in Δ (in A). If curly brackets are not used at all it is interpreted as a dependency on *all* preceding contexts.

Sequents are legitimate terms and can be used wherever regular terms can be used. In particular nested sequents (when conclusion is again a sequent) allows to separate different kinds of contexts from each other so they will not mix:

$$\mathbf{sequent}\{\Gamma_A \vdash \mathbf{sequent}\{\Gamma_B \vdash C\}\}$$

can be thought as $\Gamma_A | \Gamma_B \vdash C$ where “|” is a marker used to enforce some structure in antecedent pattern (to separate Γ_A and Γ_B).

3 A brief description of CIC

CIC is based on a typed lambda calculus. Without inductive definitions it is a system $\lambda P\omega$ (or λC) from Barendregt’s cube. There is no syntactical differentiation between types and objects, they are just terms. Terms are built from variables, global names, constructors, abstraction, application, product and “let-in” expressions. Each term should have a type, types of types are constants called sorts. There are two basic sorts **Set** and **Prop** and a cumulative hierarchy of higher sorts **Type(0)**, **Type(1)**, \dots all containing the basic sorts. Intuitively **Prop**

is a type of all propositions and **Set** is a type of specifications (of programs) and usual types (integers, booleans, lists, etc).

We based our work on the system of rules presented in the chapter 4 “Calculus of Inductive Constructions” of The Coq Proof Assistant Reference Manual [8]. Although CIC is formulated in Gentzen style, it is not a usual plain Gentzen style system. Each CIC rule is explicitly parameterized with environment and can explicitly change it. Environment contains declarations of global constants and global assumptions. Such a non standard format is chosen because of inductive definitions — once inductive definition is verified to be correct, all types and constructors it defines are (automatically) added to the environment. Alternatively one can carry the whole inductive definition throughout the proof as a term. The latter approach is in original papers [4,10] about inductive definitions for the Calculus of Constructions; it is (at least) easier to express in the plain Gentzen style. For this reason we use the latter approach.

4 Implementation problems and their solutions

Coq’s implementation of CIC operates with the notion of *environment* (or to be more precise *global environment*). It is an ordered list of declarations of global names, such as names of new operators and types. Of course MetaPRL maintains something similar internally but it is not available for the direct control of the user. It also seems that explicit global environment was introduced primarily for efficiency reason — to mention inductive definitions only once and later only refer to them. We prefer the global environment. So we modified all rules not to use the global environment explicitly.

There is also a notion of *context* (or more precisely *local context*) where the names of variables are declared. Contexts are native entities of the MetaPRL meta-language so we are fine here.

There are two official kinds of judgements in CIC:

- $E[\Gamma] \vdash t : T$ means that the term t is well-typed and has type T in the environment E and context Γ
- $\mathcal{WF}(E)[\Gamma]$ means that the environment E is well-formed and the context Γ is a valid context in this environment

But in the actual rules we find one more kind of judgement:

- $D \in E$

where D is either inductive definition $\text{Ind}(\Gamma)[\Gamma_P]\{\Gamma_I := \Gamma_C\}$ or constant declaration $c : T$ and E is an environment. It means that E is well-formed and contains D (or if $D \in E$ is the conclusion of the rule, D is added to E).

4.1 \mathcal{WF} -judgement

More traditional formulation of calculus of constructions [1] does not use \mathcal{WF} -judgement:

$$\vdash \text{Prop} : \text{Type}(i) \quad \vdash \text{Set} : \text{Type}(i) \quad \vdash \text{Type}(i) : \text{Type}(j) \quad \text{axioms, } i < j$$

$$\frac{\Gamma \vdash A : s}{\Gamma; x : A \vdash x : A} \text{ start, } x \notin \Gamma \quad \frac{\Gamma \vdash A : B \quad \Gamma \vdash C : s}{\Gamma; x : C \vdash A : B} \text{ weakening, } x \notin \Gamma$$

The problem is that if you want to pull some declaration from the middle of an antecedent to the succedent $\Gamma; x : A; \Delta \vdash x : A$ you need to type-check A and whole Δ using the weakening rule. This is not practical and not desirable if you want to prove something like $\Gamma; x : A; \Delta \vdash x : A$ about arbitrary Δ .

Coq has the rule

$$\frac{\mathcal{WF}(E)[\Gamma; x : T; \Delta[x]]}{E[\Gamma; x : T; \Delta[x]] \vdash x : T} \quad (\text{Var})$$

but then you can hardly prove something like $E[\Gamma] \vdash t : T$ for t and T not depending on Γ without assuming $\mathcal{WF}(E)[\Gamma]$. So this kind of assumption would precede any theorem.

We decided to use the following set of rules:

$$\Gamma \vdash \text{Prop} : \text{Type}(i) \quad \Gamma \vdash \text{Set} : \text{Type}(i) \quad \Gamma \vdash \text{Type}(i) : \text{Type}(j) \quad (\text{axioms, } i < j)$$

$$\frac{\Gamma; \Delta \vdash T : s}{\Gamma; x : T; \Delta \vdash x : T} \quad (\text{Var})$$

$$\frac{\Gamma; \Delta \vdash A : B \quad \Gamma; \Delta \vdash C : s}{\Gamma; x : C; \Delta \vdash A : B} \quad (\text{Weak})$$

So unlike rules in [1] we allow to insert new declarations in the middle of hypotheses list. We also allow nonsense in hypotheses (because of our choice of axioms) but it seems alright — falsum derives anything.

4.2 Lambda Calculus

Implementation of the lambda part of CIC is pretty straightforward, after we settled with treating of \mathcal{WF} and do not tell anything about environment E .

We did not implement “let-in” construction and definition $x := t : T$ because first of all they seem redundant. Secondly, the majority of the rules do not distinguish definition $x := t : T$ and variable declaration $x : T$, so for now we decided not to complicate our implementation with such a polymorphism.

4.3 Inductive Definitions

Inductive definitions allow us to introduce new types and constructors of these types. $\text{Ind}(\Gamma)[\Gamma_P](\Gamma_I := \Gamma_C)$ is a formal representation of an inductive definition valid in context Γ with parameters Γ_P , a context of definitions Γ_I and a context of constructors Γ_C . Γ_I actually contains types defined by the inductive definition.

Example Parameterized lists is defined as follows:

$$\text{Ind}()[A : \text{Set}](\text{List} : \text{Set} := \text{nil} : \text{List}, \text{cons} : A \rightarrow \text{List} \rightarrow \text{List})$$

List is a new inductive type, **nil** (an empty list) and **cons** (a concatenation of an element and a list) are the constructors of type **List**; A is a parameter of type **Set**. **List** A is a type of lists with elements of type A .

Since **Ind** has contexts as parameters it has to have flexible arity. As it was mentioned, in **MetaPRL** the only construct with flexible arity is sequent term. But we should not simply write $\Gamma; \Gamma_P; \Gamma_I; \Gamma_C \vdash \cdot$, because there is no way to tell later which hypotheses are from context Γ , which hypotheses are from context Γ_P , etc. Of course we can reserve special terms to separate those contexts but **MetaPRL** allows nested sequents so we can write:

$$\mathbf{sequent}\{\Gamma \vdash \mathbf{sequent}\{\Gamma_P \vdash \mathbf{sequent}\{\Gamma_I \vdash \mathbf{sequent}\{\Gamma_C \vdash A\}\}\}\}$$

because we use nested sequents all over the place we label all sequents generously:

$$\mathbf{sequent}\{\Gamma \vdash \mathbf{sequent}[\mathbf{IndParams}]\{\Gamma_P \vdash \mathbf{sequent}[\mathbf{IndTypes}]\{\Gamma_I \vdash \mathbf{sequent}[\mathbf{IndConstrs}]\{\Gamma_C \vdash A\}\}\}\}$$

We do not label the outermost sequent because Γ really plays role of hypotheses so outermost sequent is really logical, whereas all other sequents here are merely placeholders with an arbitrary arity. Using display forms we can easily give it a “traditional” format

$$\begin{aligned} & \mathbf{Ind}(\Gamma)[\Gamma_P](\Gamma_I := \Gamma_C)A \\ & \text{or} \\ & \mathbf{Ind}(\Gamma)[\Gamma_P](\Gamma_I := \Gamma_C) \vdash A \\ & \text{or} \\ & \Gamma \vdash \mathbf{Ind}[\Gamma_P](\Gamma_I := \Gamma_C)A \end{aligned}$$

which we will use for the rest of the paper. Here A is the actual meaning of the term $\mathbf{Ind}[\Gamma_P](\Gamma_I := \Gamma_C)A$ but A can refer to the inductive definition it is wrapped in. Note that due to the nesting, variables declared in an outer sequent’s antecedent are bound in all inner sequents but that is exactly what we want.

Types of inductive types and constructors are described by the following two rules. For the rest of the paper we assume that Γ_P is $[p_1 : P_1; \dots; p_r : P_r]$, Γ_I is $[I_1 : A_1; \dots; I_k : A_k]$, and Γ_C is $[c_1 : C_1; \dots; c_n : C_n]$.

$$\frac{\mathbf{Ind}(\Gamma)[\Gamma_P](\Gamma_I := \Gamma_C) \in E}{(I_j : (p_1 : P_1) \dots (p_r : P_r)A_j) \in E} \quad (j = 1 \dots k)$$

$$\frac{\mathbf{Ind}(\Gamma)[\Gamma_P](\Gamma_I := \Gamma_C) \in E}{(c_i : (p_1 : P_1) \dots (p_r : P_r)C_i\{I_j/(I_j p_1 \dots p_r)\}_{j=1 \dots k}) \in E} \quad (i = 1 \dots k)$$

here $(x : S)T$ is a dependent product type (or dependent function type) and it associates to the right.

Aside from giving certain types to inductive types and constructors these rules say that if an inductive definition was given all types and constructors from it are injected in the environment (thus becoming accessible for the later use).

Of course we have to give some explicit meaning for all “...” in those rules and for “massive” simultaneous substitution $I_j/(I_j p_1 \dots p_r)$. Again we use sequents to express something with flexible arity.

Example We define $(x_1 : T_1) \dots (x_n : T_n)S$ using two rewrites over sequent term **sequent**[longProduct] $\{x_1 : T_1; \dots; x_n : T_n \vdash S\}$. For readability we will write **longProduct** $\{x_1 : T_1; \dots; x_n : T_n \vdash S\}$:

$$\begin{aligned} \text{longProduct}\{\vdash S\} &\longleftrightarrow S && \text{base case, } n = 0 \\ \text{longProduct}\{\Gamma; x : T \vdash S\} &\longleftrightarrow \text{longProduct}\{\Gamma \vdash (x : T)S\} && \text{rec. step} \end{aligned}$$

on each iteration rightmost declaration $x : T$ is taken from the context Γ and used to form a function type $(x : T)S$ to the result S of the previous iteration.

For the latter rule we need to give definitions of massive application, product and substitution simultaneously because all bindings in the rule have to be preserved correctly. It unfolds to 8 rewrites that act as one recursive function on contexts (basically base case and recursive step for each operation which is 6 already plus some glue).

The next rule tells us if inductive definition is correct.

$$\frac{(E[\Gamma; \Gamma_P] \vdash A_j : s'_j)_{j=1 \dots k} \quad (E[\Gamma; \Gamma_P; \Gamma_I] \vdash C_i : s_{p_i})_{i=1 \dots n}}{\mathcal{WF}(E; \text{Ind}(\Gamma)[\Gamma_P](\Gamma_I := \Gamma_C))[\Gamma]}$$

providing the following side conditions hold:

- $k > 0$, I_j, c_i are different names for $j = 1 \dots k$ and $i = 1 \dots n$
- for $j = 1 \dots k$ we have A_j is an arity of sort s_j and $I_j \notin \Gamma \cup E$
- for $i = 1 \dots n$ we have C_i is a type of constructor of I_{p_i} which satisfies the positivity condition for $I_1 \dots I_k$ and $c_i \notin \Gamma \cup E$

As you can see this rule has a few side conditions. We need to formalize those side conditions via rules and/or rewrites. Side conditions of this rule operate with notions:

- A_j is an arity of sort s_j
- C_i is a type of constructor of I_{p_i}
- C_i satisfies the positivity condition for a constant X
- constant X occurs strictly positively in T

Example The constant X occurs *strictly positively* in T in the following cases:

- X does not occur in T
- T converts to $(X t_1 \dots t_n)$ and X does not occur in any of t_i

- T converts to $(x : U)V$ and X does not occur in type U but occurs strictly positively in type V

actually there is a fourth case but it is too complicated for the discussion.

And the formalization of this definition in MetaPRL looks as follows:

$$\frac{}{\Gamma; x : T; \Delta \vdash \text{strictly_pos}\{x; S\}} \quad (\text{base case})$$

here x does not occur freely in S because according to MetaPRL syntax we would have to say $S[x]$ in order to allow free occurrence of x in S .

$$\frac{}{\Gamma; x : T; \Delta \vdash \text{strictly_pos}\{x; \text{appContext}\{\Sigma \vdash x\}\}} \quad (\text{application case})$$

here again x does not occur freely in Σ because according to MetaPRL syntax we would have to say $\Sigma[x]$ in order to allow free occurrence of x in Σ .

$$\frac{\Gamma; x : T; \Delta; y : U \vdash \text{strictly_pos}\{x; V[y; x]\}}{\Gamma; x : T; \Delta \vdash \text{strictly_pos}\{x; y : U \rightarrow V[y; x]\}} \quad (\text{function case})$$

again x does not occur freely in U .

Because we do not use \mathcal{WF} -judgement we need some special treatment for the conclusion of the last rule. We use another judgement

$$\Gamma \vdash \text{IndWF}[\Gamma_P](\Gamma_I := \Gamma_C)$$

which sole purpose is to claim correctness of the inductive definition.

As it was said we do not add types and constructors from inductive definitions to the global environment hence we carry whole inductive definitions everywhere we use it.

Example Using inductive definition of the parameterized lists we say:

$$\begin{aligned} \text{List} &::= \text{Ind}[A : \text{Set}](\text{List} : \text{Set} := \text{nil} : \text{List}; \text{cons} : A \rightarrow \text{List} \rightarrow \text{List})\text{List} \\ \text{nil} &::= \text{Ind}[A : \text{Set}](\text{List} : \text{Set} := \text{nil} : \text{List}; \text{cons} : A \rightarrow \text{List} \rightarrow \text{List})\text{nil} \\ \text{cons} &::= \text{Ind}[A : \text{Set}](\text{List} : \text{Set} := \text{nil} : \text{List}; \text{cons} : A \rightarrow \text{List} \rightarrow \text{List})\text{cons} \end{aligned}$$

To support this approach our implementation has three rewrites:

$$\text{Ind}[\Gamma_P](\Gamma_I := \Gamma_C)t_{\{\}} \leftrightarrow t$$

$$\begin{aligned} \text{Ind}[\Gamma_P](\Gamma_I := \Gamma_C; x : T; \Delta_C)t[x] &\leftrightarrow \text{Ind}[\Gamma_P](\Gamma_I := \Gamma_C; x : T; \Delta_C) \\ &\quad t[\text{Ind}[\Gamma_P](\Gamma_I := \Gamma_C; x : T; \Delta_C)x] \end{aligned}$$

$$\begin{aligned} \text{Ind}[\Gamma_P](\Gamma_I; x : T; \Delta_I := \Gamma_C[x])t[x] &\leftrightarrow \text{Ind}[\Gamma_P](\Gamma_I; x : T; \Delta_I := \Gamma_C[x]) \\ &\quad t[\text{Ind}[\Gamma_P](\Gamma_I; x : T; \Delta_I := \Gamma_C[x])x] \end{aligned}$$

The first rewrite says that if term t under inductive definition does not really depend on it, we can get rid of inductive definition and use just t . Second and third rewrite say that any occurrence of inductive type or constructor (under inductive definition) can be wrapped additionally with one more layer of that inductive definition. Having in mind that rewrites are bidirectional we can prove such trivial facts as:

$$\begin{aligned} \text{List} &\in \text{Set} \rightarrow \text{Set} \\ \text{nil} &\in (A : \text{Set})(\text{List}A) \\ \text{cons} &\in (A : \text{Set})(A \rightarrow \text{List}A \rightarrow \text{List}A) \end{aligned}$$

Up to this point we were describing actually working implementation. It includes all the necessary rules, rewrites and tactics for rudimentary proof automation. The example of the parameterized lists is proved correct and simple facts given above are proved. The implementation is available for download under GPL license from the MetaPRL CVS server <http://cvs.metapr1.org:12000/cvsweb/metapr1/theories/cic/>.

4.4 Implementation of Cases and Fixpoint

Besides defining inductive types, establishing their sorts and types of constructors one needs means for case analysis of such types and recursion over inductive types. In CIC (Coq) there are two separate operations — case analysis and recursion (fixpoint) each accompanied with a certain number of rules governing it.

Unfortunately we again face the problem of formalizing side conditions. Consider an inductive definition with several mutually defined types. The case analysis rule has to collect all constructors for one of those types from the list of all constructors of that inductive definition. This was the place where we have got stuck. Although the above condition seems expressible as a collection of rules we do not know any elegant (and efficient) approach. So we decided there is no point in formalizing case analysis and fixpoint rules if it would be too slow and no competitor to Coq.

We do consider an alternative approach. It is possible to wrap each rule in a tactic and implement too complicated side conditions in the tactic. Such tactics will check too intricate syntactical conditions and pre-compute parameters for rules (e.g. extract all appropriate constructors for case analysis rule). Those tactics should be considered as a part of the trusted core but we will get much better efficiency. Such an implementation would be no less reliable than Coq because (as far as we understand) in Coq this logic is also hard-coded and not explicitly written as a system of rules.

5 Future work

Presently we are at the crossroad of several treatments for the case-analysis and fixpoint rules, which are:

- Find a way to represent side conditions of those rules as rules and rewrites. This will most likely lead to a significant drop in the speed comparing with Coq, but MetaPRL trusted core will not be extended.
- Wrap each rule in a tactic and implement too complicated side conditions in the tactic. This would probably boost the performance. But such tactics would actually extend MetaPRL trusted core.
- Find a formal generic notation that would allow us to implement case-analysis and fixpoint rules nicely. If successful this might be a good tradeoff between performance and extension of the trusted core. And we would get an extra bonus — improve the expressiveness of the MetaPRL meta-language.

After the decision is made the rest of the CIC core and basic proof automation will be implemented. Then we will benchmark our implementation against Coq. If successful, more steps towards compatibility with the existing Coq-libraries will be made. The ultimate goal is to support import or direct access to Coq library files.

References

1. Henk P. Barendregt. *Handbook of Logic in Computer Science*, volume 2, chapter Lambda Calculi with Types, pages 118–310. Oxford University Press, 1992.
2. Robert L. Constable, Stuart F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panagaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the NuPRL Proof Development System*. Prentice-Hall, NJ, 1986.
3. Thierry Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.
4. Thierry Coquand and Christine Paulin-Mohring. Inductively defined types, preliminary version. In *COLOG '88, International Conference on Computer Logic*, volume 417 of *Lecture Notes in Computer Science*, pages 50–66. Springer, Berlin, 1990.
5. Jason Hickey, Aleksey Nogin, Robert L. Constable, Brian E. Aydemir, Eli Barzilay, Yegor Bryukhov, Richard Eaton, Adam Granicz, Alexei Kopylov, Christoph Kreitz, Vladimir N. Krupski, Lori Lorigo, Stephan Schmitt, Carl Witty, and Xin Yu. MetaPRL — A modular logical environment. In David Basin and Burkhart Wolff, editors, *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003)*, volume 2758 of *Lecture Notes in Computer Science*, pages 287–303. Springer-Verlag, 2003.
6. Jason J. Hickey and Aleksey Nogin. Fast tactic-based theorem proving. In J. Harrison and M. Aagaard, editors, *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*, pages 252–266. Springer-Verlag, 2000.
7. Jason J. Hickey, Aleksey Nogin, Alexei Kopylov, et al. MetaPRL home page. <http://metaprl.org/>.
8. INRIA. *The Coq Proof Assistant Reference Manual*, 2003.
9. Aleksey Nogin and Jason Hickey. Sequent schema for derived rules. In Victor A. Carreño, César A. Muñoz, and Sophiène Tahar, editors, *Proceedings of the 15th International Conference on Theorem Proving in Higher Order Logics (TPHOLs*

- 2002), volume 2410 of *Lecture Notes in Computer Science*, pages 281–297. Springer-Verlag, 2002.
10. Christine Paulin-Mohring. Inductive definitions in the system Coq; rules and properties. In J. F. Groote M. Bezem, editor, *Typed Lambda Calculi and Applications*, Lecture Notes in Computer Science. Springer-Verlag, 1993.