The $\lambda\Pi$-calculus Modulo as a Universal Proof Language

Mathieu Boespflug
McGill University
mboes@cs.mcgill.ca

Quentin Carbonneaux
INRIA and ENPC
carbonnq@eleves.enpc.fr

Olivier Hermant
ISEP
ohermant@isep.fr

Abstract

The $\lambda\Pi$-calculus forms one of the vertices in Barendregt's λ-cube and has been used as the core language for a number of logical frameworks. Following earlier extensions of natural deduction [14], Cousineau and Dowek [11] generalize the definitional equality of this well studied calculus to an arbitrary congruence generated by rewrite rules, which allows for more faithful encodings of foreign logics. This paper motivates the resulting language, the $\lambda\Pi$-calculus modulo, as a universal proof language, capable of expressing proofs from many other systems without losing their computational properties. We further show how to very simply and efficiently check proofs from this language. We have implemented this scheme in a proof checker called Dedukti.

1 Introduction

The success of formal methods since the pioneering efforts of the teams around LCF and Automath, both as tools of practical importance and as objects of intellectual curiosity, has spawned a bewildering variety of software systems to support them. While the field has developed to maturity in academia and has registered some important successes in the industry, such as in aerospace, mass transit and smart cards to name a few, the full benefit of formal methods in an industrial setting remains largely untapped. We submit that a lack of standards and easy interoperability in the field is one explanation to this state of affairs.

Indeed, the field of formal methods has largely passed on the typical trend of other fields in computer science and engineering as they mature: the shift from systems to data. Standards have emerged for numbers, texts, images, sounds, videos, structured data, etc. These standards are not linked to a particular application, but all the applications comply to these standards, allowing the interoperability of various software, even on heterogeneous networks. When a community reaches a critical mass, standardization is unavoidable to allow exchange and collaborative improvements. As long as the use of proof systems was restricted to a small community, each system could develop its own independent language. But, this is not possible anymore: the success of proof systems and the widening of their audience put the interoperability problem to the fore.

Yet, importing and exporting lumps of proofs from system to system is a thorny issue, because different systems implement different formalisms: some are constructive, some are not, some are predicative, some are not, some are first-order, some are not, etc. Thus, a standard must not only be able to define the proofs themselves, but also to specify the formalism in which they are expressed. To define such a standard, we propose the $\lambda\Pi$-calculus modulo, a simple but very powerful extension of a well-known and well understood proof language for minimal first-order logic that embeds a congruence on types generated by a set of rewrite rules.

1.1 From deep to shallow

Avoiding the quadratic blowup in the number of adapters necessary between n different systems means introducing one common target for all these adapters. All foreign systems read and

write a single proof format. One might hope to construct a formalism so powerful that other formalisms form a fragment of this super formalism, but this is infeasible in practice. For one, different formalisms require different but incompatible features, such as predicativity or impredicativity. Moreover, justifying the consistency of such a tower of Babel might not be quite so easy. Finally, the vast majority of proofs simply would not need the full power offered by the formalism, only ever using a small fragment of it.

The alternative is to chose a simple formalism, say first-order logic, and make other formalisms the objects of discourse of this simple formalism. This is known as a *deep embedding*. One will assert appropriate axioms to introduce these objects of discourse, such as axioms characterizing the set of formulas, proof objects, validity judgments on proof objects, etc. This is the approach used by Edinburgh Logical Framework [18], Twelf [24], Isabelle [23] and others.

Proofs, however, are not interesting merely for the validity of the judgment they establish. It can be desirable to reason about the proofs themselves. Realistic proofs of interesting facts typically contain detours in the reasoning. These detours often make the proof shorter, though they will usually be taken to be equivalent in some precise sense to a canonical derivation that does not contain any detours, or *cuts*. Eliminating cuts in a proof is a procedure, *i.e.* a computation.

When proofs are objects of discourse, it is convenient to be able to reason about them modulo elimination of any cuts. Many formalisms, such as the Calculus of Constructions or the original LF, include a powerful notion of definitional equality that includes equality modulo cuts. Hence, identity of two proof objects can be established not by writing tedious reasoning steps, but rather by an appeal to this rather large definitional equality, written ($\equiv$), which can be decided by applying the cut elimination procedure. The more powerful the definitional equality, the more often derivations may appeal to it, hence the shorter they become.

Pure Type Systems (PTS), of which the Calculus of Constructions and LF are an instance, internalize definitional equality using the following *conversion* rule:

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash A : s \qquad \Gamma \vdash B : s}{\Gamma \vdash M : B} \, A \equiv B$$

where s is some PTS dependent sort. Arguing that two objects are equal can be done in only one derivation step if they are definitionally equal. The trouble here is that the notion of definitional equality is fixed once and for all, no matter the objects of discourse du jour. Hence, a deep embedding of foreign logics treats formulas, judgments and proofs of these logics as second class citizens — one can always define a judgment ($\equiv_L$) on proofs of a foreign logic L, but given two proofs P, Q in L, one can only justify $P \equiv_L Q$ using a derivation, not the native definitional equality decision procedure.

In short, deep embeddings destroy the computational behaviour of proofs, because proof identity can no longer be justified computationally. Moreover, in general desirable properties such as canonicity of values and the existence of empty types are lost due to the presence of axioms. Many of these axioms uniquely characterize functions on the objects of discourse, for example as a set of equations. If only we could turn these equations into rewrite rules instead, we would be able to directly compute the results of applying these functions, rather than having to systematically prove using derivations that a function application is related to its result.

The λΠ-calculus modulo extends the λΠ-calculus, the core calculus of the original LF, with the ability to define one's own rewrite rules. The conversion rule above is modified to take definitional equality to be an arbitrary congruence, generated by some set of user defined rewrite rules $\mathcal{R}$. In the λΠ-calculus modulo, we eschew equational axioms and instead turn them into rewrite rules. Intuitively, because the native definitional equality is now extensible, when

embedding a foreign logic within the λΠ-calculus modulo, we can tune the native definitional equality to internalize the foreign definitional equality. The computational behaviour of proofs is thus restored. The resulting embeddings are still deep embeddings, but we can introduce rewrite rules in a systematic way that essentially interpret a deep embedding into another *shallow embeddings*.

1.2 Specifying logics

In the λΠ-calculus modulo, a theory is specified by a finite set of typed constants and a finite set of typed rewrite rules. This set of rules defines the computational behavior of the set of constants. Encoding proofs of a logical system is achieved by writing a translation function from proofs of the original logical system to well typed terms of the λΠ-calculus modulo extended by the appropriate constants and rewrite rules. The same translations as the ones defined for LF could be reused since LF is a strict subset of the λΠ-calculus modulo. However, in this case no benefit is drawn from the new computational facilities offered by our framework and the computational properties of proof objects are lost.

We propose to use shallow embeddings as described above. Not only do proofs enjoy smaller representations but consistency of theories can be proved with more generic techniques using, for instance, super-consistency based approaches as described in [15]. Specifying full theories with rewrite rules only, as opposed to traditional axiomatic encodings, allows to show semantic properties of the logical system by checking properties on the set of rewrite rules. For example, if a theory can be expressed in λΠ-calculus modulo with a set of rewrite rules $\mathcal{R}$ and if the rewriting modulo $\mathcal{R} + \beta$ is well-behaved, then the theory is consistent and constructive proofs have the disjunction property and the witness property. Moreover, these shallow embeddings present theories in ways which share a generic notion of cut, rather than a theory specific notion of cut.

1.3 A new proof checker

To support this language, we propose a new proof-checker called Dedukti. A number of proof checkers for similar logical frameworks have been proposed, such as Twelf [24] and LFSC [27], alternately aiming for relative simplicity and correctness, or for focusing more on performance but hopefully also maintaining correctness. LFSC in particular focuses on checking proofs of very large size or many proofs with respect to a fixed signature. Within the design space, we focus on checking small but computationally intensive proofs, through translations of proofs into higher order data and into functional programs. We obtain a very simple implementation to boot, that directly corresponds to the bidirectional typing rules that form its specification.

1.4 Overview

This paper will first present the λΠ-calculus modulo and motivate its use as a target language for many other systems (Section 2). We will then describe informally the rationale behind a purpose built proof-checker we have developped. A companion paper [8] describes in details the encoding of the calculus of constructions inside the λΠ-calculus modulo, as well as practical results in checking Coq's standard library.

$\boxed{\Gamma \text{ WF}}$ Context Γ is well-formed

$$(empty)\ \frac{}{\cdot \text{ WF}} \qquad (decl)\ \frac{\Gamma \text{ WF} \qquad \Gamma \vdash A : s \qquad x \notin \Gamma}{\Gamma, x{:}A \text{ WF}}\ s \in \{\mathsf{Type}, \mathsf{Kind}\}$$

$\boxed{\Gamma \vdash M : A}$ Term M is of type A in context Γ

$$(sort)\ \frac{\Gamma \text{ WF}}{\Gamma \vdash \mathsf{Type} : \mathsf{Kind}} \qquad (var)\ \frac{\Gamma \text{ WF} \qquad x{:}A \in \Gamma}{\Gamma \vdash x : A}$$

$$(prod)\ \frac{\Gamma \vdash A : \mathsf{Type} \qquad \Gamma, x{:}A \vdash B : s}{\Gamma \vdash \Pi x{:}A.\ B : s}\ s \in \{\mathsf{Type}, \mathsf{Kind}\}$$

$$(abs)\ \frac{\Gamma \vdash A : \mathsf{Type} \qquad \Gamma, x{:}A \vdash B : s \qquad \Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda x{:}A.\ M : \Pi x{:}A.\ B}\ s \in \{\mathsf{Type}, \mathsf{Kind}\}$$

$$(app)\ \frac{\Gamma \vdash M : \Pi x{:}A.\ B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : \{N/x\}B}$$

$$(conv)\ \frac{\Gamma \vdash M : A \qquad \Gamma \vdash A : s \qquad \Gamma \vdash B : s}{\Gamma \vdash M : B}\ A \equiv B$$

Figure 1: Typing rules for the λΠ-calculus (modulo)

2 The λΠ-calculus modulo

2.1 The λΠ-calculus

The λΠ-calculus modulo is a familiar formalism — it is a variation on the λΠ-calculus, which under the guise of many names (LF, λP, etc) has been used with great success to specify other formalisms, from logics to programming languages. The λΠ-calculus is a simple proof language for minimal first-order logic, whose syntax for pre-terms can be given succinctly as follows, provided we have an infinite set X of variable names:

$$M, N, A, B ::= x \mid \lambda x{:}A.\ M \mid \Pi x{:}A.\ B \mid M\ N \mid \mathsf{Type} \mid \mathsf{Kind}$$

The notation $A \to B$ stands for $\Pi x{:}A.\ B$ when x does not appear in B. In the tradition of Pure Type Systems (PTS) [3], we conflate the language of formulas, proofs and sorts (or types, terms and kinds if we are looking from the other side of Curry-Howard lens) into a single language. Within the set of *pre-terms* (or *raw terms*) we distinguish the set of *terms*, which are well-typed.

The type system for the λΠ-calculus, given in figure 1, does enforce a stratification between sublanguages, that of terms, of types and of kinds: terms are typed by a type, types are typed by a kind and kinds are of type Kind.

Notice that in general types can be indexed by terms, giving a *family* of types, and that one can λ-abstract not only over terms but also over types. For these two reasons, β-redexes can appear at the level of types and it is convenient to quotient them by β-equivalence (written $(\equiv)$) through the conversion rule. In figure 1, all the contexts are lists of pairs of a variable name and a type:

$$\Gamma, \Delta ::= \cdot \mid \Gamma, x{:}A$$

Figure 1 serves just as well as a presentation of the rules of the λΠ-calculus modulo — the only difference between the two formalisms is in the conversion rule. In the λΠ-calculus, the

congruence is generated by the β-reduction rule alone, while in the $\lambda\Pi$-calculus modulo it is generated by β-reduction *and* the rewrite rules, as presented below.

2.2 Adding rewrite rules: the $\lambda\Pi$-calculus modulo

Well-typed terms live within a context. One can segregate this context into a global shared context and a local context. We will write Σ for this global context, which we call a *signature* and we assume to be well-formed. The assumptions of a signature are *constants*.

A system of rewrite rules is declared over constants of some signature Σ. Each rewrite rule pertains to one constant of the signature. A typed rewrite rule is a rewrite rule paired with a typing context Γ that assigns types to all the free variables appearing in the left hand side of the rule. Both sides of a rewrite rule must be typable with the same type A.

Definition 1 (Rewrite rule)**.** Let Σ be a context. A *rewrite rule* in Σ is a quadruple $l \longrightarrow^{\Gamma,A} r$ composed of a context Γ, and three β-normal terms l, r, A, such that it is *well-typed*:

- the context Σ, Γ is well-formed,
- and $\Sigma, \Gamma \vdash l : A$ and $\Sigma, \Gamma \vdash r : A$ are derivable judgments.

In order to use the rewrite rule, we must rewrite the instances of the left member by instances of the right member. Instances are done through *typed substitutions*.

Definition 2. Let Σ, Γ and Δ be contexts. A substitution binding the variables declared in Γ is said to be of type $\Gamma \rightsquigarrow \Delta$ if for any $x{:}T \in \Gamma$, we can derive $\Sigma, \Delta \vdash \theta x : \theta T$.

Lemma 3. *Let Σ and Δ be contexts, $l \longrightarrow^{\Gamma,T} r$ be a rewrite rule in Σ and θ be a substitution of type $\Gamma \rightsquigarrow \Delta$. Then $\Sigma, \Delta \vdash \theta l : \theta T$ and $\Sigma, \Delta \vdash \theta r : \theta T$ and we say that θl rewrites to θr.*

For Σ a context and $\mathcal{R}$ a set of rewrite rules in Σ, we let $\equiv_{\mathcal{R}}$ be the smallest congruence generated by $\mathcal{R}$. We write $(\longrightarrow)$ and $(\equiv)$ (respectively) for the contextual closure and the smallest congruence generated by β-reduction and $\mathcal{R}$.

2.3 How to encode formalisms in the $\lambda\Pi$-calculus modulo

The claim of this paper is that the $\lambda\Pi$-calculus modulo can serve as a melting pot, capable of expressing all manner of other type theories. This is in particular the case for all functional PTS [11]. By way of example, let us see how to encode polymorphism (*à la* System F).

The typing rules of System F are identical to the rules of the $\lambda\Pi$-calculus that are presented in figure 1, except for the rules (*prod*) and (*abs*) that become:

$$(\textit{F-prod})\ \frac{\Gamma \vdash A : s \qquad \Gamma, x : A \vdash B : \mathsf{Type}}{\Gamma \vdash \Pi x : A.B : \mathsf{Type}}\ s \in \{\mathsf{Type}, \mathsf{Kind}\}$$

$$(\textit{F-abs})\ \frac{\Gamma \vdash A : s \qquad \Gamma, x : A \vdash B : \mathsf{Type} \qquad \Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x{:}A.\ M : \Pi x{:}A.\ B}\ s \in \{\mathsf{Type}, \mathsf{Kind}\}$$

System F is also stratified into kinds (Type is the only kind), types and terms. For brevity and to lift any ambiguities, in the following we write $\star$ (resp. $\Box$) for the constant Type (resp. Kind) of System F. A typical example of term that is typable in System F is the polymorphic identity:

$$\vdash \lambda A{:}\star\ .\ \lambda x{:}A.\ x : \Pi A{:}\star\ .\ A \to A$$

It is not possible to write this term in the pure $\lambda\Pi$-calculus.

To encode System F in the λΠ-calculus modulo, we introduce four constants. $U_\star, U_\Box$, also called *universe* constants reflect the types $\star$ and $\Box$ of System F in the λΠ-calculus modulo. To each universe is associated a *decoding* function ε from System F types and function spaces to the native λΠ-calculus modulo types and function spaces.

$$\begin{array}{ll} U_\Box : \mathsf{Type} & \varepsilon_\Box : U_\Box \to \mathsf{Type} \\ U_\star : \mathsf{Type} & \varepsilon_\star : U_\star \to \mathsf{Type} \end{array}$$

We also introduce a constant $\star : U_\Box$ reflecting the fact that $\star : \Box$ and the following rewrite rule:

$$\varepsilon_\Box \ \star \ \longrightarrow \ U_\star$$

Now we introduce a constant that represents a deep embedding of the Π-types of System F as higher order abstract syntax into the λΠ-calculus modulo:

$$\begin{array}{rcl} \dot{\Pi}_{\langle \Box,\star,\star\rangle} & : & \Pi X{:}U_\Box.\ (\varepsilon_\Box\ X \to U_\star) \to U_\star \\ \dot{\Pi}_{\langle \star,\star,\star\rangle} & : & \Pi X{:}U_\star.\ (\varepsilon_\star\ X \to U_\star) \to U_\star \end{array}$$

Together with rewrite rules extending the decoding function to these System F types:

$$\begin{array}{rcl} \varepsilon_\star\ (\dot{\Pi}_{\langle \Box,\star,\star\rangle}\ X\ Y) & \longrightarrow & \Pi x{:}(\varepsilon_\Box\ X).\ \varepsilon_\star\ (Y\ x) \\ \varepsilon_\star\ (\dot{\Pi}_{\langle \star,\star,\star\rangle}\ X\ Y) & \longrightarrow & \Pi x{:}(\varepsilon_\star\ X).\ \varepsilon_\star\ (Y\ x) \end{array}$$

Both rewrite rules have type Type (see Definition 1). They contain the free variables X and Y, so they are defined (respectively) in the contexts:

$$\begin{array}{c} X : U_\Box,\ Y : (\varepsilon_\Box\ X) \longrightarrow U_\star \\ X : U_\star\ ,\ Y : (\varepsilon_\star\ X) \longrightarrow U_\star \end{array}$$

Cousineau and Dowek [11] go on to define a translation from terms and types of System F to terms and types using the above constants, and proving conservativity results for this translation. Since our goal here is didactic, we will only give the example of the translation of the previous polymorphic identity, that can now be written in our settings as (keeping the global signature made up of the above constants implicit):

$$\vdash (\lambda A{:}(\varepsilon_\Box\ \star).\ \lambda x{:}(\varepsilon_\star\ A).\ x)\ :\ \varepsilon_\star\ \left(\dot{\Pi}_{\langle \Box,\star,\star\rangle}\ \star\ \left[\lambda A : (\varepsilon_\Box\ \star).\ (\dot{\Pi}_{\langle \star,\star,\star\rangle}\ A\ (\lambda x : (\varepsilon_\star\ A).\ A))\right]\right)$$

which, after simplification of $\varepsilon_\Box\ \star$ in $U_\star$ and rewriting of the $\dot{\Pi}_{\langle \Box,\star,\star\rangle}$ constant gives the term:

$$\vdash \lambda A{:}U_\star.\ \lambda x{:}(\varepsilon_\star\ A).\ x\ :\ \Pi a{:}(\varepsilon_\Box\ \star).\ \varepsilon_\star\left(\left[\lambda A{:}U_\star.\ (\dot{\Pi}_{\langle \star,\star,\star\rangle}\ A\ (\lambda x{:}(\varepsilon_\star\ A).\ A))\right]\ A\right)$$

after β-reduction and rewriting of $(\varepsilon_\Box\ \star)$, we get:

$$\vdash \lambda A{:}U_\star.\ \lambda x{:}(\varepsilon_\star\ A).\ x\ :\ \Pi a{:}U_\star.\ \varepsilon_\star\ \left(\dot{\Pi}_{\langle \star,\star,\star\rangle}\ A\ (\lambda x{:}(\varepsilon_\star\ A).\ A)\right)$$

and after another step of rewriting of the $\dot{\Pi}_{\langle \star,\star,\star\rangle}$:

$$\vdash \lambda A{:}U_\star.\ \lambda x{:}(\varepsilon_\star\ A).\ x\ :\ \Pi a{:}U_\star.\ \Pi y{:}(\varepsilon_\star\ A).\ (\varepsilon_\star\ ((\lambda x{:}(\varepsilon_\star\ A).\ A)\ y))$$

After β-reduction and α-conversion, we have:

$$\vdash \lambda A{:}U_\star.\ \lambda x{:}(\varepsilon_\star\ A).\ x\ :\ \Pi A{:}U_\star.\ \Pi x{:}(\varepsilon_\star\ A).\ (\varepsilon_\star\ A)$$

a judgment for which one can easily construct a derivation in the λΠ-calculus modulo.

Remark in particular how the use of higher order abstract syntax in the encoding of System F types affords us substitution on those types for free. Also, polymorphism has been introduced in this term, through the constant $\varepsilon_\star$ that behave as a lifting from terms to types and from *deep to shallow* embeddings: this is clearly allowed by its nonempty computational content. Finally, conceivably one could along the same lines encode the λΠ-calculus into the λΠ-calculus modulo through the same mechanisms described above. This encoding would *not* be the identity (we would still have ε and "dotted" constants appearing in types).

3 An abstract type checking algorithm

3.1 Bidirectional type checking

Dedukti's internal representation of terms is domain-free, as introduced in [5], meaning that abstractions are not annotated by the type of the variable that they bind. This information is largely redundant if one adopts a bidirectional typing discipline, a technique going back to [10]. Hence, input terms are smaller at no cost to the complexity or size of the type checker. Bidirectional type systems split the usual type judgments into two forms: checking judgments and synthesis judgments. Dropping the domains on abstractions means that some terms cannot have their type readily inferred — these terms can only be *checked* against a given type, their type cannot be *synthesized*.

Contrary to the usual presentation of λΠ-calculus as a PTS, given in Section 2, the bidirectional type system we present in Figure 2 is syntax directed. As such, the choice of rule to apply at each step of the derivation is entirely determined by the shape of the term that is being typed. In particular, conversion only happens during a phase change between checking mode and synthesis mode. The moment at which this phase change is allowed to occur is guided by the "inputs" to the judgments. As such, the type system presented here is deterministic. One can therefore readily extract an algorithm from these typing rules: checking rules should be read as functions taking a context, a term and type as input and answering true or false; synthesis rules should be read as functions taking a context and term as input and producing a type as output.

3.2 Context-free type checking

One particularly thorny issue when checking dependently typed terms is the treatment of renaming of variables to ensure that contexts remain well-formed and the implementation of substitution. In keeping with the spirit of the de Bruijn criterion [13], we strive to obtain an implementation of a type checker that is as simple and small as possible. In Dedukti, we have done so using a higher-order abstract representation (HOAS) of terms, which allows us to piggy-back substitution over terms on the substitution already provided by the implementation language. In such a setting, abstractions are encoded as functions of the implementation language. They are therefore opaque structures that cannot be directly analyzed. Functions are black boxes whose only interface is that they can be applied. In HOAS, it is therefore necessary to draw the parameter/variable distinction common in presentations of first-order logic — *variables* are always bound and *parameters* (or *eigenvariables*) are variables that are free. As

we move under a binding, such as an abstraction, the bound variable becomes a parameter. In HOAS, this means that we must substitute a piece of data representing a new parameter for the previously bound variable. In this section we do not address the implementation of a concrete algorithm (postponed to Section 4) but elaborate a type system that capture the essence of the way the implementation of Section 4 works.

If in order to proceed to the body of abstractions it is necessary to unfold implementation-level functions by feeding them parameters, we might as well feed more than just a parameter to these functions. We can indeed pass the type of the parameter along with it. In other words, we substitute a *box* — a pair of a term and its type — for bound variable as we move between bindings. We arrive in this way at a *context-free* type system, so-called because we no longer need to carry around a typing context in judgments, provided all free variables (parameters) in terms are always boxed.

3.3 Putting it all together

The resulting type system, bidirectional and context-free, is given in Figure 2. A straightforward mapping of these rules into a purely functional program is discussed in Section 4. The syntax over which these rules are defined is given below.

In its essence, a proof script consists of a sequence of non-recursive declarations of the form $x : A$. Later declarations have earlier declarations in scope. Proof scripts are type checked following the order of the sequence. Given a term M, checking that this term has some given type A will first assume that A has already been type checked, just as the types of constants appearing in A may be assumed to already be checked when checking A. Therefore adding premises ensuring that contexts are well-formed to the leaf rules and mixing premises about types in the deduction rules for term typing as is done in the standard presentation of Section 2 does not accurately reflect how type checking works in practice. In the rules of Figure 2, the type A in a judgment $\vdash M \Leftarrow A$ is assumed to be a proper type or kind.

For the needs of type checking we can therefore use two distinct representations: one for the *subject* of a judgment (to the left of $\Rightarrow$ or $\Leftarrow$), underlined,

$$\underline{M}, \underline{N}, \underline{A}, \underline{B} ::= x \mid [y : \overline{A}] \mid \lambda x.\ \underline{M} \mid \Pi x{:}\underline{A}.\ \underline{B} \mid \underline{M}\ \underline{N} \mid \mathsf{Type}$$

and one for the *classifier* of a judgment (to the right of $\Rightarrow$ or $\Leftarrow$), overlined,

$$\overline{M}, \overline{N}, \overline{A}, \overline{B} ::= x \mid y \mid \lambda x.\ \overline{M} \mid \Pi x{:}\overline{A}.\ \overline{B} \mid \overline{M}\ \overline{N} \mid \mathsf{Type} \mid \mathsf{Kind}$$

We reuse the same syntactic elements for both representations. The essential difference between the two representations is that in the first, parameters y are tagged with their type, whereas in the second parameters are bare. This is because it is not the classifiers that we are type checking, but only the subjects of typing judgments. The type of a parameter in a classifier is therefore not needed. We split the infinite set of variables into two disjoint infinite sets of (bound) variables x on the one hand and parameters y on the other. There is no way of binding parameters, because they stand in place of free variables.

In the rules of Figure 2, the judgment $\vdash M \Rightarrow A$ reads "the expression N synthesizes type A" and the judgment $\vdash M \Leftarrow A$ reads "the expression M checks against type A". Within judgments, we omit making the representation explicit through overlining and underlining, because which representation we mean is evident from the position in a judgment. In some rules we observe a crossover of subparts of the subject into the classifier and *vice versa*. In the (app^b) rule, the argument part N crosses over to the right of the synthesis arrow. In the (abs^b) rule, the type A appears both in the subject part of a premise, but in the classifier part

$\boxed{\vdash M \Rightarrow A}$ Term M synthesizes type A

$$(sort^b)\ \frac{}{\vdash \mathsf{Type} \Rightarrow \mathsf{Kind}} \qquad (var^b)\ \frac{}{\vdash [x : A] \Rightarrow A}$$

$$(app^b)\ \frac{\vdash M \Rightarrow C \qquad C \longrightarrow_w^* \Pi x{:}A.\ B \qquad \vdash N \Leftarrow A}{\vdash M\ N \Rightarrow \{N/x\}B}$$

$\boxed{\vdash M \Leftarrow A}$ Term M checks against type A

$$(abs^b)\ \frac{C \longrightarrow_w^* \Pi x{:}A.\ B \qquad \vdash \{[y : A]/x\}M \Leftarrow \{y/x\}B}{\vdash \lambda x.\ M \Leftarrow C}$$

$$(prod^b)\ \frac{\vdash A \Leftarrow \mathsf{Type} \qquad \vdash \{[y : A]/x\}B \Leftarrow s}{\vdash \Pi x{:}A.\ B \Leftarrow s}\ s \in \{\mathsf{Type}, \mathsf{Kind}\}$$

$$(conv^b)\ \frac{\vdash N \Rightarrow B}{\vdash N \Leftarrow A}\ A \equiv B$$

Figure 2: Bidirectional context-free type checking for the $\lambda\Pi^b$-calculus modulo

of the conclusion. We are therefore implicitly applying a coercion between representations, determined by the following relation between subject and classifier representations:

$$\frac{}{x \sim x} \qquad \frac{}{[y : \overline{A}] \sim y} \qquad \frac{}{\mathsf{Type} \sim \mathsf{Type}} \qquad \frac{}{\mathsf{Kind} \sim \mathsf{Kind}}$$

$$\frac{\underline{M} \sim \overline{M}}{\lambda x.\ \underline{M} \sim \lambda x.\ \overline{M}} \qquad \frac{\underline{A} \sim \overline{A} \qquad \underline{B} \sim \overline{B}}{\Pi x{:}\underline{A}.\ \underline{B} \sim \Pi x{:}\overline{A}.\ \overline{B}} \qquad \frac{\underline{M} \sim \overline{M} \qquad \underline{N} \sim \overline{N}}{\underline{M}\ \underline{N} \sim \overline{M}\ \overline{N}}$$

This relation is left-total and right-unique, so that it determines a function from subject representations to classifier representations. Synthesis rules should be read top-down and checking rules bottom-up, so we don't need a coercion in the opposite direction.

In the (abs^b) and $(prod^b)$ rules, the new parameter y introduced in the premises must be chosen fresh. This condition can always be satisfied since terms are of finite size and the set of parameters is infinite.

Example 4. Consider the polymorphic identity as described in section 2. A type derivation for this term in the $\lambda\Pi^b$-calculus modulo can be given as follows:

$$\frac{\Pi A.\ \ldots \longrightarrow_w^* \Pi A.\ \ldots \qquad \dfrac{\Pi x{:}(\varepsilon_\star\ y_1).\ (\varepsilon_\star\ y_1) \longrightarrow_w^* \Pi x{:}(\varepsilon_\star\ y_1).\ (\varepsilon_\star\ y_1) \qquad \dfrac{\dfrac{}{\vdash [y_2 : \varepsilon_\star\ y_1] \Rightarrow \varepsilon_\star\ y_1}}{\vdash [y_2 : \varepsilon_\star\ y_1] \Leftarrow \varepsilon_\star\ y_1}}{\vdash \lambda x.\ x \Leftarrow \Pi x{:}(\varepsilon_\star\ y_1).\ (\varepsilon_\star\ y_1)}}{\vdash \lambda A.\ \lambda x.\ x \Leftarrow \Pi A{:}U_\star.\ \Pi x{:}(\varepsilon_\star\ A).\ (\varepsilon_\star\ A)}$$

3.4 Reduction on types

The algorithm presented here relies on finding the weak head normal form of types in order to proceed. This imposes that the set of rewrite rules used to generate the congruence ($\equiv$) verifies a certain number of properties for the algorithm to be deterministic and complete. Notice that reduction happens only on the classifier parts of a judgment. Reduction hence does not need to concern itself with dealing with boxes.

Definition 5 (Weak reduction)**.** Given a set $\mathcal{R}$ of rewrite rules, we define weak reduction ($\longrightarrow_w$) as the contextual closure of the union of the usual β-rule with $\mathcal{R}$ under the reduction context

$$C ::= [] \mid C\ M \mid M\ C \mid \Pi x{:}C.\ B$$

Its reflexive and transitive closure is written ($\longrightarrow_w^*$) and its reflexive, transitive and symmetric closure is written ($\equiv^w$).

Definition 6 (Standardization)**.** A relation on terms, provided it is confluent, is said to be standardizing if:

1. $M \equiv x\ N_1\ \ldots\ N_n$ implies $M \longrightarrow_w^* x\ N_1'\ \ldots\ N_n'$ and $N_1 \equiv N_1', \ldots, N_n \equiv N_n'$;
2. $M \equiv \lambda x.\ N$ implies $M \longrightarrow_w^* \lambda x.\ N'$ and $N' \equiv N$;
3. $M \equiv \Pi x{:}A.\ B$ implies $M \longrightarrow_w^* \Pi x{:}A'.\ B'$ and $A' \equiv A$ and $B' \equiv B$;
4. $M \equiv \mathsf{Type}$ implies $M \longrightarrow_w^* \mathsf{Type}$;
5. $M \equiv \mathsf{Kind}$ implies $M \longrightarrow_w^* \mathsf{Kind}$.

If ($\longrightarrow$) is confluent, standardizing and strongly normalizing, then it is sufficient to compute weak head normal forms [2], which are the normal forms of the weak reduction relation defined above. The reduction relation ($\longrightarrow_w$) may in general be non-deterministic — in that case one can fix a specific reduction strategy. This can also be a way to ensure confluence and strong normalization.

Remark 7. The typing rules of this section are closer to an actual implementation of a type checker, but a legitimate question to ask is how do we know that they are sound and complete with respect to the typing rules of Figure 1. Coquand [10] presents essentially the same algorithm as above (though not context-free) for a simple dependent type theory where $\mathsf{Type} : \mathsf{Type}$, and proves it sound through semantic means. Abel and Altenkirch [2] prove soundness and partial completeness of a very similar algorithm to that of Coquand through syntactic means. Their proof relies only on confluence and standardization properties of β-reduction and can readily be adapted to a type system such as the λΠ-calculus modulo provided the rewrite rules respect the same properties. The first author has formalized soundness and completeness results about various context-free type systems [7].

3.5 Rewriting and dependent types

Rewrite rules can be classified according to the head constant to which they pertain. One can read the set of rewrite rules for one same head constant as clauses of a functional program, the left hand side of which contains a number of patterns.

However, contrary to an ordinary functional program, in the presence of dependent types matching one term might force the shape of another term. This will typically happen when a constructor forces one of the indexes of the type family in its result type. One simple example of this situation is the head function on vectors, defined in the following signature:

$$\begin{array}{rcl@{\qquad}rcl@{\qquad}rcl}
\mathsf{Nat} & : & \mathsf{Type} & \mathsf{Z} & : & \mathsf{Nat} & \mathsf{S} & : & \mathsf{Nat} \to \mathsf{Nat} \\
\mathsf{Vector} & : & \Pi n : \mathsf{Nat}.\ \mathsf{Type} & \mathsf{Nil} & : & \mathsf{Vector\ Z} & \mathsf{Cons} & : & \Pi n{:}\mathsf{Nat}.\ \mathsf{Nat} \to \mathsf{Vector}\ n \to \mathsf{Vector}\ (\mathsf{S}\ n)
\end{array}$$

The type and the rewrite rule associated to the head function are:

$$\mathsf{head} \ : \ \Pi n{:}\mathsf{Nat}.\ \mathsf{Vector}\ n \to \mathsf{Nat} \qquad\qquad \mathsf{head}\ \{\mathsf{S}\ n\}\ (\mathsf{Cons}\ n\ h\ tl) \ \longrightarrow \ h$$

A non-linear pattern seems to be needed because n appears twice in its left hand side. We cannot just generalize the first pattern in this rewrite rule, because then the left hand side would be ill-typed. A non-linear pattern would mean that we must check that the instances for the two occurrences of n are convertible. Moreover, in general inductive type families might be indexed by higher-order terms, meaning that we would need to implement higher-order matching. However, what we actually want to express is that any instance to the first pattern of the rewrite rule above is uniquely determined by a matching instance of the second pattern. Indeed, matching on the Cons constructor forces the dependent first argument to head to be S n. As such, the first argument to head plays no operational role; it is merely there for the sake of typing. Following Agda [22], in Dedukti we support marking parts of the left hand side of a rule as operationally irrelevant using curly braces — actual arguments are not pattern matched against these operationally irrelevant parts, hence we can avoid having to implement full higher-order matching, which carries with it an inherent complexity that we would rather keep out of a proof checking kernel.

Note that the braces could equally have been placed around the n that is fed as argument to Cons. The only important property to verify is that one and only one occurrence of n must not have braces. This occurrence will be the one that binds n.

4 An implementation of the type checking algorithm

We have by now developed an abstract characterization of a proof checking algorithm. However, we have not so far been explicit about how to implement substitution on terms, nor have we discussed how to decide definitional equality algorithmically. Ideally, whatever implementation we choose, it should be both simple and efficient. Indeed, encodings in the style of Section 2.3 introduce many more redexes in types than were present in the original input. Moreover, computing the weak head normal form of some types may require an arbitrarily large number of reductions, even before any encoding overhead, such as can be the case in proofs by reflection.

Normalization by evaluation (NbE) is one particularly easy to implement normalization scheme that alternates phases of weak reduction (*i.e.* evaluation) and *reification* phases (or *readback* phases [17]). Weak reduction is significantly easier to implement than strong reduction, which in general requires reduction under binders. Evaluation conveniently side steps the traditionally vexing issues of substitution such as renaming of variables to avoid capture, because evaluation only ever involves substituting closed terms for variables[1]. What's more, if evaluation is all that is required, we can pick efficient off-the-shelf evaluators for stock functional programming languages to do the job. These evaluators are invariably much more efficient than we could feasibly implement ourselves — their implementations have been fine-tuned and well studied for the benefit of faster functional programs over many years. What's more, we achieve a better separation of concerns by using off-the-shelf components: we are in the business of writing type checkers, not evaluators. The size of the trusted base arguably does not decrease if we are to rely on an existing evaluator, but trust is all the better for it nonetheless, because a mature off-the-shelf component has likely been stress tested by many more users in many more settings than any bespoke implementation we can come up with.

[1] A term may contain parameters but those cannot be captured: there are no binding forms for parameters in the syntax of terms.

$$
\begin{aligned}
||x|| &= x \\
||y|| &= \mathsf{Var}_C\ y \\
||\lambda x.\ \overline{M}|| &= \mathsf{Lam}_C\ (\lambda x \to ||\overline{M}||) \\
||\Pi x{:}\overline{A}.\ \overline{B}|| &= \mathsf{Pi}_C\ ||\overline{A}||\ (\lambda x \to ||\overline{B}||) \\
||\overline{M}\ \overline{N}|| &= \mathsf{app}\ ||\overline{M}||\ ||\overline{N}|| \\
||\mathsf{Type}|| &= \mathsf{Type}_C \\
||\mathsf{Kind}|| &= \mathsf{Kind}_C
\end{aligned}
$$

(a)

$$
\begin{aligned}
|x| &= x \\
|[y : \overline{A}]| &= \mathsf{Box}_T\ y \\
|\lambda x.\ \underline{M}| &= \mathsf{Lam}_T\ (\lambda x \to |\underline{M}|) \\
|\Pi x{:}\underline{A}.\ \underline{B}| &= \mathsf{Pi}_T\ |\underline{A}|\ (\lambda x \to |\underline{B}|) \\
|\underline{M}\ \underline{N}| &= \mathsf{App}_T\ |\underline{M}|\ |\underline{N}| \\
|\mathsf{Type}| &= \mathsf{Type}_T
\end{aligned}
$$

(b)

Figure 3: Deep embedding of (a) classifiers and (b) subjects into Haskell.

For simplicity, we use an untyped variant of NbE, in the style of [6] (see also [16, 20, 9]). We proceed by translating terms of the λΠ-calculus modulo into Haskell programs. Assume the following datatype of evaluated terms:

$$
\begin{aligned}
\textbf{data}\ \mathsf{Code} = {} & \mathsf{Var}_C\ \mathsf{Int} \mid \mathsf{App}_C\ \mathsf{Code}\ \mathsf{Code} \\
\mid {} & \mathsf{Lam}_C\ (\mathsf{Code} \to \mathsf{Code}) \mid \mathsf{Pi}_C\ \mathsf{Code}\ (\mathsf{Code} \to \mathsf{Code}) \\
\mid {} & \mathsf{Type}_C \mid \mathsf{Kind}_C
\end{aligned}
$$

One can view the values of this datatype as code because these values are interesting for their computational behavior. Translation of terms into values of this type is given in Figure 3. Because we only ever need to compute with classifiers during type checking, this translation is defined over the classifier representation of terms, not the subject representation.

The actual identity of a bound variable is usually quite irrelevant[2]. What is relevant is to be able to conveniently substitute another term for all free occurrences of a variable under its binder. Paramaters are dual, in that we never need to substitute for a parameter, nor do they have scope since they cannot be bound, but their identity is crucial. During type-checking, we must be able to ask whether this parameter is the same as this other parameter.

We therefore map (bound) variables to variables of the target language, allowing us to piggy back substitution and scope management onto that of the target language. We get substitution essentially for free. Parameters, on the other hand, are a ground piece of data: a name. Here, for simplicity, we use integers for names.

In a similar spirit, we wish to reuse the target language's computational facilities to reduce terms to normal forms. Therefore applications are mapped to target language applications. However, our embedding of terms in the target language is not quite shallow, in that functions and so on are not represented directly as functions of the target language but rather as values of type Code. This embedding into a monomorphic datatype is essentially an untyped embedding, thus avoiding the impedence mismatch between the type system of the λΠ-calculus modulo and that of the target language, in our case Haskell. Therefore we must introduce app, a wrapper function around the native application operation, which essentially decapsulates functions embedded in values of type Code:

$$\mathsf{app} :: \mathsf{Code} \to \mathsf{Code} \to \mathsf{Code}$$

[2]Indeed this is why formal developments in the presence of binding internalize Barendregt's *variable convention* in some form.

$$\begin{aligned}
&\mathsf{app}\ (\mathsf{Lam}_C\ f)\ t &&= f\ t\\
&\mathsf{app}\ t_1 \qquad\quad t_2 &&= \mathsf{App}_C\ t_1\ t_2
\end{aligned}$$

If the first argument of app is a function, then we can proceed with the application. If not, we are stuck, and the result is a neutral term.

Deciding the equality between two terms can now be done near structurally:

$$\begin{aligned}
&\mathsf{conv} :: \mathsf{Int} \to \mathsf{Code} \to \mathsf{Code} \to \mathsf{Bool}\\
&\mathsf{conv}\ n\ (\mathsf{Var}_C\ x)\ (\mathsf{Var}_C\ x') = x \equiv x'\\
&\mathsf{conv}\ n\ (\mathsf{Lam}_C\ t)\ (\mathsf{Lam}_C\ t') = \mathsf{conv}\ (n+1)\ (t\ (\mathsf{Var}_C\ n))\ (t'\ (\mathsf{Var}_C\ n))\\
&\mathsf{conv}\ n\ (\mathsf{Pi}_C\ ty_1\ ty_2)\ (\mathsf{Pi}_C\ ty_3\ ty_4) = \mathsf{conv}\ n\ ty_1\ ty_3 \wedge\\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{conv}\ (n+1)\ (ty_2\ (\mathsf{Var}_C\ n))\ (ty_4\ (\mathsf{Var}_C\ n))\\
&\mathsf{conv}\ n\ (\mathsf{App}_C\ t_1\ t_2)\ (\mathsf{App}_C\ t_3\ t_4) = \mathsf{conv}\ n\ t_1\ t_3 \wedge \mathsf{conv}\ n\ t_2\ t_4\\
&\mathsf{conv}\ n\ \mathsf{Type}_C\ \mathsf{Type}_C = \mathsf{True}\\
&\mathsf{conv}\ n\ \mathsf{Kind}_C\ \mathsf{Kind}_C = \mathsf{True}\\
&\mathsf{conv}\ n\ _\ _ = \mathsf{False}
\end{aligned}$$

Descending under binding structures gives the weak head normal form of their body, which is computed automatically according to the semantics of the target language. The first argument serves as a supply of fresh parameter names; these need only be locally unique so a local supply is sufficient.

We now have a succinct and yet efficient way of deciding equality of two terms that we know to be well-typed. The translation of Figure 3 for classifiers isn't appropriate, however, for subjects of type checking, because contrary to classifiers, subjects are not *a priori* well-typed. Therefore, we cannot assume that any kind of normal form exists for subjects, lest we compromise the decidability of type checking. We therefore introduce an alternative translation for subjects, where terms are not identified modulo any reduction. This translation is given in part (b) of Figure 3. Note that it is a completely standard mapping of terms to higher order abstract syntax, into values of following datatype:

$$\begin{aligned}
&\textbf{data}\ \mathsf{Term} = \mathsf{Box}_T\ \mathsf{Code}\ \mathsf{Code} \mid \mathsf{App}_T\ \mathsf{Term}\ \mathsf{Term}\\
&\qquad\quad \mid \mathsf{Lam}_T\ (\mathsf{Term} \to \mathsf{Term}) \mid \mathsf{Pi}_T\ \mathsf{Term}\ (\mathsf{Term} \to \mathsf{Term})\\
&\qquad\quad \mid \mathsf{Type}_T
\end{aligned}$$

One particularly nice feature of this translation is that it differs from that for classifiers in one single place: the interpretation of applications.

In Dedukti, to control the amount of code that we generate, and to avoid writing wasteful coercions between values of type Term and values of type Code, which we would have to invoke at every application node in the term that we are checking, we generate a single Haskell source for a single proof script, containing both the subject translation and classifier translation for every subterm of the proof script. That is, Term and Code representations are tied together for every subterm, through a system of pairing (or *gluing*). This way no coercion is necessary, but if we performed this gluing naively, it would obviously provoke and explosion in the size of the generated code. The size would grow exponentially with the tree depth of terms, because we would be duplicating subtrees at every level.

For this reason, in the actual implementation, we first transform terms into a linear form, namely A-normal form. In this form, all subterms are named, so that it becomes possible to share substructures of translations at every level, hence avoiding any kind of exponential blowup. The rationale for this translation is similar to that of let-insertion to avoid blowups during partial evaluation [12]. We also perform closure conversion to further improve sharing.

$$||x||^{pat} = x$$
$$||c\ N_1\ \ldots\ N_n||^{pat} = \mathsf{App}\ (\ldots(\mathsf{App}\ (\mathsf{Var}\ c)\ ||N_1||^{pat})\ldots)\ ||N_n||^{pat}$$

$$\left|\left| \begin{array}{ccc} c\ N_{11}\ \ldots\ N_{1n} & \longrightarrow & M_1 \\ \vdots & & \vdots \\ c\ N_{m1}\ \ldots\ N_{mn} & \longrightarrow & M_m \end{array} \right|\right| = \begin{array}{ccccc} c & ||N_{11}||^{pat} & \ldots & ||N_{1n}||^{pat} & = & ||M_1|| \\ & \vdots & & \vdots & & \vdots \\ c & ||N_{m1}||^{pat} & \ldots & ||N_{mn}||^{pat} & = & ||M_m|| \\ c & \_ & \ldots & \_ & = & ||c\ x_1\ \ldots\ x_n||^{pat} \end{array}$$

Figure 4: Translation of rewrite rules as functions over classifiers.

Both of these standard code transformations can be performed selectively, only for subterms where both the subject and classifier representations are needed (both forms are need only when checking applications $M\ N$, and then only when the type synthesized for M is of the form $\Pi x{:}A.\ B$ where x *does* occur in B).

4.1 Extending the algorithmic equality

If we commit to a particular rewrite strategy, and for convenience we commit to a particular rewrite strategy that matches the evaluation order of the target language, then we can extend the scheme given previously to gracefully handle custom rewrite rules. The idea is to read the rewrite rules for one same head constant as the clauses of a functional program. This is the idea behind the translation of rewrite rules as functional programs over values of type Code given in Figure 4. The resulting functional program implements the rewrite rules in exactly the same way that the app function implements β-reduction; if for a given subterm none of the rewrite rules apply, then the result is a neutral term.

5 Conclusion

We have presented in this paper a simple and extensible calculus that works well as a common format for a large number of other proof systems. The essential difference with previous proposals is that through an extensible definitional equality, the λΠ-calculus modulo can act as a logical framework that respects the computational properties of proofs. We showed how this calculus can be implemented very succinctly by leveraging existing evaluation technology.

Our approach is firmly grounded in type theory; computation only happens as a result of checking for convertibility between two types, just as in other type theories. Other recent efforts for a common proof format, such as the proposal of Miller [21] grounded in proof theory, use *focusing* to structure proofs and to formulate customized macro proof rules in terms of some primitive rules. This framework captures a more general notion of computation than the one presented here: Miller's is formulated as proof search rather the functional programming centric approach we present here (our implementation views rewrite rules as inducing a recursive, pattern matching function). Computations in Miller's framework need not be deterministic and can indeed backtrack. The cost of this generality is that the proof checker needs to know how to perform unification and backtracking, whereas in our approach the trusted base is arguably smaller but effects such as backtracking need to be encoded as a pure functional program, for example by using a monad [26].

We have an implementation of a proof checker, Dedukti, which expects the system of rewrite rules provided by the user to be reasonably well behaved. This proof checker is therefore only one small piece of the larger puzzle: confidence in the results of this tool are contingent upon trusting that the provided rewrite rules do not compromise logical consistency, or that these rewrite rules indeed form a confluent and terminating system. Dedukti only checks types, it does not check the rewrite rules. Future work will focus providing these other pieces to support our methodology: tools to (semi-)automatically verify properties about rewrite rules, for example.

For given translations, a system of rewrite rules or a schema of rewrite rules need only be proven to behave properly once, rather than every time for every results of these translations. However, we could also envisage making the λΠ-calculus modulo the core of a full fledged proof environment for end users, in which users can define their own domain specific rewrite rules, in which case automated certification of properties about the resulting system of rewrite rules becomes essential.

We are currently working on adapting size-based termination techniques [1, 25, 4]. This would give a general criterion that would work for the rewrite rules that encodes inductive or coinductive structures. This criterion would be more powerful than purely syntactic criteria such as Coq's guard condition [4].

Finally, we currently type check by first translating proofs and formulas into a functional program in Haskell, which we compile to native code using the GHC compiler. However, the compilation process is slow. Its cost is only amortized given proofs with significant computational content and even then, we would rather not have the compiler waste time optimizing parts of the proof that are computationally completely irrelevant. Other systems such as Coq and Isabelle ask the user to decide exactly when to use an optimizing compiler and when to compute using a much more lightweight interpreter. The advantage of our approach is its simplicity, both for the user and for the implementation. We would like to keep this model, but to make it scale, we are investigating translating terms to a language that supports "just in time" compilation, whereby hotspots that are worth compiling to native code are identified at runtime and automatically, still without any intervention from the user.

A stable release of Dedukti is available at `https://www.rocq.inria.fr/deducteam/Dedukti/index.html`, however some of the improvements discussed here (in particular, the dot patterns of section 3.5) have been integrated only in the development version of Dedukti, that can be found on Github: `https://github.com/mpu/dedukti`.

References

[1] Andreas Abel. Miniagda: Integrating sized and dependent types. In Ana Bove, Ekaterina Komendantskaya, and Milad Niqui, editors, *PAR*, volume 43 of *EPTCS*, pages 14–28, 2010.

[2] Andreas Abel and Thorsten Altenkirch. A partial type checking algorithm for type: Type. *Electr. Notes Theor. Comput. Sci.*, 229(5):3–17, 2011.

[3] Henk Barendregt. Introduction to generalized type systems. *J. Funct. Program.*, 1(2):125–154, 1991.

[4] Gilles Barthe, Maria João Frade, E. Giménez, Luis Pinto, and Tarmo Uustalu. Type-based termination of recursive definitions. *Math. Struct. Comp. Sci.*, 14(1):97–141, 2004.

[5] Gilles Barthe and Morten Heine Sørensen. Domain-free pure type systems. In *J. Funct. Program.*, pages 9–20. Springer, 1993.

[6] Mathieu Boespflug. Conversion by evaluation. In Manuel Carro and Ricardo Peña, editors, *PADL*, volume 5937 of *LNCS*, pages 58–72. Springer, 2010.

[7] Mathieu Boespflug. *Conception d'un noyau de vérification de preuves pour le λΠ-calcul modulo*. PhD thesis, Ecole polytechnique, January 2011.

[8] Mathieu Boespflug and Guillaume Burel. CoqInE: Translating the calculus of inductive constructions into the λΠ-calculus modulo. In David Pichardie and Tjark Weber, editors, *PxTP*, 2012.

[9] Mathieu Boespflug, Maxime Dénès, and Benjamin Grégoire. Full reduction at full throttle. In Jouannaud and Shao [19], pages 362–377.

[10] Thierry Coquand. An algorithm for type-checking dependent types. *Sci. Comput. Program.*, 26(1-3):167–177, 1996.

[11] Denis Cousineau and Gilles Dowek. Embedding pure type systems in the lambda-pi-calculus modulo. In Simona Ronchi Della Rocca, editor, *TLCA*, volume 4583 of *LNCS*, pages 102–117. Springer, 2007.

[12] Olivier Danvy. Pragmatics of type-directed partial evaluation. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Dagstuhl Seminar on Partial Evaluation*, volume 1110 of *LNCS*, pages 73–94. Springer, 1996.

[13] NG De Bruijn. A plea for weaker frameworks. In *Logical frameworks*, pages 40–67. Cambridge University Press, 1991.

[14] Gilles Dowek. Proof normalization for a first-order formulation of higher-order logic. In Elsa L. Gunter and Amy P. Felty, editors, *TPHOLs*, volume 1275 of *LNCS*, pages 105–119. Springer, 1997.

[15] Gilles Dowek and Olivier Hermant. A simple proof that super-consistency implies cut-elimination. *Notre-Dame Journal of Formal Logic*, 2012. to appear.

[16] Andrzej Filinski and Henning Korsholm Rohde. A denotational account of untyped normalization by evaluation. In Igor Walukiewicz, editor, *FoSSaCS*, volume 2987 of *LNCS*, pages 167–181. Springer, 2004.

[17] B. Grégoire and X. Leroy. A compiled implementation of strong reduction. In Mitchell Wand and Simon L. Peyton Jones, editors, *ICFP*, pages 235–246. ACM, 2002.

[18] Robert Harper, Furio Honsell, and Gordon D. Plotkin. A framework for defining logics. *J. ACM*, 40(1):143–184, 1993.

[19] Jean-Pierre Jouannaud and Zhong Shao, editors. *Certified Programs and Proofs - First International Conference, CPP 2011, Kenting, Taiwan, December 7-9, 2011. Proceedings*, volume 7086 of *LNCS*. Springer, 2011.

[20] S. Lindley. *Normalisation by evaluation in the compilation of typed functional programming languages*. PhD thesis, 2005.

[21] Dale Miller. A proposal for broad spectrum proof certificates. In Jouannaud and Shao [19], pages 54–69.

[22] Ulf Norell. Dependently typed programming in agda. In Pieter W. M. Koopman, Rinus Plasmeijer, and S. Doaitse Swierstra, editors, *Advanced Functional Programming*, volume 5832 of *LNCS*, pages 230–266. Springer, 2008.

[23] Lawrence C. Paulson. Isabelle: The next seven hundred theorem provers. In Ewing L. Lusk and Ross A. Overbeek, editors, *CADE*, volume 310 of *LNCS*, pages 772–773. Springer, 1988.

[24] Frank Pfenning and Carsten Schürmann. System description: Twelf - a meta-logical framework for deductive systems. In Harald Ganzinger, editor, *CADE*, volume 1632 of *LNCS*, pages 202–206. Springer, 1999.

[25] Cody Roux. *Terminaison à base de tailles: Sémantique et généralisations*. Thèse de doctorat, Université Henri Poincaré — Nancy 1, April 2011.

[26] Philip Wadler. How to replace failure by a list of successes: A method for exception handling, backtracking, and pattern matching in lazy functional languages. In *FPCA*, pages 113–128, 1985.

[27] Michael Zeller, Aaron Stump, and Morgan Deters. Signature compilation for the edinburgh logical framework. *Electr. Notes Theor. Comput. Sci.*, 196:129–135, 2008.