More On Implicit Syntax*

Marko Luther

Universität Ulm
Fakultät für Informatik
D-89069 Ulm, Germany
luther@informatik.uni-ulm.de

Abstract. Proof assistants based on type theories, such as Coq and Lego, allow users to omit subterms on input that can be inferred automatically. While those mechanisms are well known, ad-hoc algorithms are used to suppress subterms on output. As a result, terms might be printed identically although they differ in hidden parts. Such ambiguous representations may confuse users. Additionally, terms might be rejected by the type checker because the printer has erased too much type information. This paper addresses these problems by proposing effective erasure methods that guarantee successful term reconstruction, similar to the ones developed for the compression of proof-terms in Proof-Carrying Code environments. Experiences with the implementation in Typelab proved them both efficient and practical.

1 Implicit Syntax

Type theories are powerful formal systems that capture both the notion of computation and deduction. Particularly the expressive theories, such as the Calculus of Constructions ($\mathcal{CC}$) [CH88] which is investigated in this paper, are used for the development of mathematical and algorithmic theories since proofs and specifications are representable in a very direct way using one uniform language.

There is a price to pay for this expressiveness: abstractions have to be decorated with annotations, and type applications have to be written explicitly, because type abstraction and type application are just special cases of λ-abstraction and application. For example, to form a list one has to provide the element type as an additional argument to instantiate the polymorphic constructors *cons* and *nil* as in (*cons* $\mathbb{N}$ 1 (*nil* $\mathbb{N}$))). Also, one has to annotate the abstraction of n in $\lambda n{:}\mathbb{N} \,.\, n+1$ with its type $\mathbb{N}$ although this type is determined by the abstraction body. These excessive annotations and applications make terms inherently verbose and thus hard to read and write. Without such explicit type information, decidability of type-checking may be lost [Miq01,Wel99].

Proof assistants are programs that have primarily been built to support humans in constructing proofs represented as λ-terms. They combine a (mostly) interactive proof-development system with a type-checker that can check those

* This research has partly been supported by the "Deutsche Forschungsgemeinschaft" within the "Schwerpunktprogramm Deduktion".

R. Goré, A. Leitsch, and T. Nipkow (Eds.): IJCAR 2001, LNAI 2083, pp. 386–400, 2001.
© Springer-Verlag Berlin Heidelberg 2001

proofs. Moreover, most proof assistants support a syntax that is more convenient to use than the language of the underlying type theory.

Besides some purely syntactic enhancements, such as the possibility to use operations in infix notation, most systems allow to leave out subterms that the system can infer as a remedy to the redundancy of explicit types. The *explicit language* (a variant of $\mathcal{CC}$ in our case) is complemented by an *implicit language* defined in terms of the underlying type theory through some inference mechanism. From the viewpoint of the user, the implicit language acts as an alternative grammar for the type theory. It improves the expressiveness of the original theory as more terms have types, but no additional types are inhabited. This pragmatic approach to simplify the user interface of proof assistants is referred to as *implicit syntax* [Pol90]. Motivated by the examples above, the inference mechanisms that we focus on in this paper are *argument synthesis*, to avoid explicit polymorphic instantiations, and *(partial) term reconstruction*, to suppress annotations on abstractions. For these, we use the term *elaboration*. The inverse process that removes redundant subterms is called *erasure*.

Ad-hoc argument synthesis, implemented in the proof assistant Coq [Bar99], uses explicit placeholders to mark omitted subterms that should be inferred. The above example can be written in Coq as ($cons$? 1 ($cons$? 2 (nil ?))) using the placeholder symbol "?". In addition, Coq supports the automatic insertion of placeholders. This is done by analyzing the types of global constants. Parameters occurring free in the type of at least one of the succeeding parameters are assumed to be inferable (e. g., the first parameter of *cons*). We may write ($cons$ 1 ($cons$ 2 (nil ?)))[1]. To decide whether a term can be hidden on output or not, the same oversimplified[2] analysis is used. This might lead to representations of terms without unique elaborations. Especially if two terms are printed identically although they are not identical internally, resulting from different hidden arguments, users may get confused [Miq01]. Even in cases where the automatic detection and erasure work correctly, the system tends to hide arguments one wants to see explicitly although they would be inferable.

A finer control over implicit positions is possible through *uniform argument synthesis* as implemented in Lego [LP92]. The user can mark parameter positions, using abstractions of a second 'color', at which arguments can be left implicit. Explicit arguments trigger the elaboration of arguments at preceding hidden positions[3]. To allow the specialization of polymorphic functions there is a syntactic facility to overwrite argument synthesis and to supply arguments 'forced' at implicit positions by preceding them with "!". At the internal representations of terms Lego uses annotations that correspond to forced marks and colored parameters of the user language. These annotations are used to decide which arguments to hide on output. Unfortunately, there are cases in which also Lego suppresses arguments that cannot be reconstructed. Furthermore, Lego

[1] Note that we still have to apply a placeholder to the empty list.

[2] A free occurrence of a parameter in another parameter type does not generally guarantee a successful elaboration. One reason for this is that argument synthesis is based on unification, which is not decidable in the higher-order case [Gol81].

[3] This rules out the inference of the type argument of *nil*.

sometimes does not hide arguments at marked positions, even when they are inferable. Both defects result from difficulties to define reduction for a language with forced arguments properly since uniqueness of elaboration cannot be decided locally [HT95].

This paper solves the problems and limitations caused by the usage of implicit syntax in current proof assistants by proposing a stronger elaboration algorithm and by complementing it with an erasure algorithm that guarantees unique elaboration. The elaboration algorithm is stronger than the mentioned ones since it allows the inference of implicit arguments at marked positions triggered also by the outer term context, while doing universal argument synthesis (e. g., our algorithm accepts $(\mathit{cons}\ 1\ \mathit{nil})$). In addition, the algorithm avoids the inconvenience of having to attach type information to all abstracted variables by doing (partial) term reconstruction. This allows the omission of type annotations[4] that can be inferred by propagating type information (e. g., the annotation of n in $(\lambda n\,.\,n+1)$). The erasure algorithm does a global analysis of terms and reconstructs forced marks (if necessary) instead of just propagating them.

The rest of this paper is organized as follows. After introducing a bicolored variant of $\mathcal{CC}$ in the next section, we present the elaboration algorithm (Sect. 3). Guided by the strategy of this elaboration algorithm, we develop in Sect. 4 an erasure algorithm that is both effective in removing most subterms and practical as shown by experimental results (Sect. 5). Finally, we report on the adoption to more realistic languages and comment on related work (Sect. 6).

2 Bicolored Calculus of Constructions

The bicolored Calculus of Constructions ($\mathcal{CC}^{\mathrm{bi}}$) to be used as explicitly typed language, is a variant of the Calculus of Construction ($\mathcal{CC}$) [CH88]. Terms are built up from a set $\mathcal{V}$ of variables, sort constants $s \in \{\mathit{Prop}, \mathit{Type}\}$, dependent function types $(\mathit{\Pi} x{:}A\,.\,B)$, λ-abstractions $(\lambda x{:}A\,.\,M)$ and function applications $(M\ N)$ as in $\mathcal{CC}$. To this we add abstractions of the form $\mathit{\Pi} x|A\,.\,B$ and $\lambda x|A\,.\,B$ using the vertical bar as a color to mark implicit argument positions. If the color is irrelevant we use the symbol "$\|$" to stand for either a colon or a bar and we abbreviate $\mathit{\Pi} x{:}A\,.\,B$ by $A \to B$ if x does not occur free in B.

We denote the set of free variables of a term M by $\mathcal{FV}(M)$, the term resulting from substituting N for each free occurrence of x in M by $M[x{:=}N]$, syntactic equality, the one-step β-reduction and the β-conversion relation by $\equiv$, $\to_\beta$ and $\simeq$, respectively. For a term M the weak head normal form (whnf) is denoted by $\overline{M}$, the normal form by $|M|$, and the application head by $\mathit{head}(M)$. As usual, we will consider terms up to α-conversions.

The colors of abstractions have no essential meaning in the explicit language, only the distinction is important. So, implicit λ-abstractions behave analogously to the corresponding explicit variant with respect to reduction.

$$((\lambda x|A\,.\,M)\ \ N) \to_\beta M[x{:=}N]$$

[4] Note that even if some annotations can be left implicit we demand all variables to be introduced explicitly, to avoid confusions caused by typos [GH98].

The typing rules of $\mathcal{CC}$ are augmented by the following rules which differ only in the coloring from the related uncolored rules of $\mathcal{CC}$.

$$\frac{\Gamma \vdash A{:}s_1 \quad \Gamma, x{:}A \vdash B{:}s_2}{\Gamma \vdash \Pi x|A \,.\, B : s_2} \qquad \frac{\Gamma, x{:}A \vdash M{:}B \quad \Gamma \vdash \Pi x|A \,.\, B : s}{\Gamma \vdash \lambda x|A \,.\, M : \Pi x|A \,.\, B} \qquad \frac{\Gamma \vdash N{:}A \quad \Gamma \vdash M : \Pi x|A \,.\, B}{\Gamma \vdash (M\ N) : B[x{:=}N]}$$

The consistency of $\mathcal{CC}^{\mathrm{bi}}$ under β follows immediately from that of $\mathcal{CC}$. Note that while $\mathcal{CC}$ is still confluent with respect to $\beta\eta$-reduction [Geu92], this property is lost in the bicolored system. The term $(\lambda x{:}A \,.\, (\lambda y|A \,.\, y)\ x)$ yields the critical pair $\langle \lambda x{:}A \,.\, x \,, \lambda y|A \,.\, y \rangle$ under $\beta\eta$-reduction.

2.1 Unification Variables

The elaboration algorithm, to be described below, maps *partial terms*[5] (i. e., terms of the implicit language) to terms of $\mathcal{CC}^{\mathrm{bi}}$. A partial term M' is translated to an *open term* M of the explicit language that contains unsolved *unification variables*. These are solved in turn by unification during the elaboration process.

The explicit language extended by unification variables, which are syntactically distinguished from other variables by a leading "?", is a variant of the one introduced by Strecker [SLvH98,Str99]. Unification variables are handled as constants with respect to $\mathcal{FV}$ and reduction. For a term M, $\mathcal{UV}(M)$ denotes the set of unification variables occurring in M. A unification variable depends on a context Γ and has a type A, as expressed by the suggestive notation $\Gamma \vdash ?n{:}A$. *Sort unification variables*, $\Gamma \vdash ?n{:}*$, are a special flavor of unification variables where instantiation is restricted to terms of type *Prop* or *Type*.

An *instantiation* is a function, mapping a finite set of unification variables to terms. When instantiating a unification variable, it is replaced by the solution term without renaming of bound variables. Instantiations are inductively extended to terms and contexts.

There can be complex dependencies among unification variables in calculi with dependent types. Therefore, a context and a type are not invariantly assigned to a unification variable $?n$, but they are determined by the *elaboration state* under consideration. An elaboration state $\mathcal{E}$ consists of a finite set $\Phi_{\mathcal{E}}$ of unification variables, a function $ctxt_{\mathcal{E}}$ assigning a context to each $?n \in \Phi_{\mathcal{E}}$, and a function $type_{\mathcal{E}}$ assigning a type to each $?n \in \Phi_{\mathcal{E}}$. Our elaboration algorithm will only produce *well-typed elaboration states* $\mathcal{E}$, where the dependencies among unification variables $?n \in \Phi_{\mathcal{E}}$ are not cyclic[6] and the typing constraints imposed by $ctxt_{\mathcal{E}}$ and $type_{\mathcal{E}}$ are 'internally consistent'.

For well-typed elaboration states reduction is confluent and strongly normalizing. Type inference and type checking are decidable and subject reduction holds [Str99].

---

[5] As a convention we prime metavariables that stand for terms of the implicit language, as M', to distinguish them syntactically from those standing for fully explicit terms.

[6] Note that unification variables do not have to be linearly ordered, as in the calculus of Muñoz [Muñ01], since this would restrict elaboration considerably.

2.2 Unification

Unification tries to find an instantiation that, when applied to two terms, makes them equal. Equality is taken modulo convertibility, thus, the unification problems we obtain will in general be higher-order. We use a 'colored' version of the unification algorithm defined by Strecker [Str99]. It essentially carries out first-order unification by structural comparison of weak head normal forms. Unification judgments are of the form $\langle \mathcal{E}_0; \Gamma \vdash t_1 \approx t_2 \rangle \Rightarrow \mathcal{E}_1; \iota$, which express that the unification problem $\langle \mathcal{E}_0; \Gamma \vdash t_1 \approx t_2 \rangle$ can be solved by instantiation ι, leaving open the unification variables of $\mathcal{E}_1$. The resulting elaboration state $\mathcal{E}_1$ is guaranteed to be well typed if the elaboration state $\mathcal{E}_0$ is well typed. For presentation purposes we use the simplified notation $\Gamma \vdash t_1 \approx t_2$ for unification judgments. We assume that all instantiations are immediately applied to all terms and we keep the elaboration states implicit.

For us, the key properties are that unification is decidable for unification problems that produce only disagreement pairs of the simple rigid-rigid kind, and that solvable unification problems of this kind have most general unifiers (MGUs). Stronger decidable unification algorithms computing MGUs for the pattern fragment of higher-order unification [Mil91] depend on the η-relation, which we have to rule out to keep $\mathcal{CC}^{\mathrm{bi}}$ confluent.

3 Elaboration

The implicit user language is an extension of the explicit language by Curry-style abstractions of both colors and applications with forced arguments.

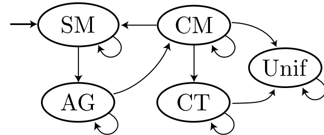
$$\mathcal{T} \;::=\; \ldots \;\mid\; \lambda\mathcal{V}\|.\mathcal{T} \;\mid\; \Pi\mathcal{V}\|.\mathcal{T} \;\mid\; (\mathcal{T}\ n!\mathcal{T}) \qquad n \in \mathbb{N}$$

The value of n indicates implicit arguments preceding the forced one. The notation $(M\ !N)$ abbreviates $(M\ 0!N)$ and we write $(M\ \|N)$ to subsume all variants of applications. We assume a function $@(M)$ that decomposes the term M in its application head and the list of its arguments in reverse order. For example, $@(f\ a\ 3!b\ c) = \langle f\,,\, c :: 3!b :: a :: \cdot \rangle$.

3.1 Bidirectional Elaboration Algorithm

We present the algorithm at an abstract level through five judgments making use of unification as introduced above. In- and output parameters are separated by "⇒" and the flow of typing information is indicated by up and down arrows.

Main Elaboration	$\Gamma \vdash M' \Rightarrow M{:}N$
Synthesis Mode (SM)	$\Gamma \vdash M' \Rightarrow M \uparrow N$
Argument Generation (AG)	$\Gamma \vdash L'; l \Rightarrow M{:}N$
Checking Mode (CM)	$\Gamma \vdash M' \downarrow N \Rightarrow M$
Coerce to Type (CT)	$\Gamma \vdash M_1{:}N \downarrow O \Rightarrow M_2$



Elaboration works in two distinct modes: SM, where type information is propagated upward from subexpressions, and CM, where information is propagated downward from enclosing expressions[7]. Unification variables are generated for implicit arguments through the AG function and are solved later on by the CT function which essentially calls unification.

The main elaboration judgment is a partial function taking a context Γ and a partial term M', producing the elaboration M of M' and its type N. Elaboration always starts with an empty elaboration state $\mathcal{E}$, $\Phi_{\mathcal{E}} = \emptyset$, in synthesis mode. As for unification, we keep instantiations and elaboration states implicit in the presentation of rules and judgments.

In the rest of this section we define the four remaining judgments as a collection of syntax-directed inference rules. The judgments are mutually dependent according to the above control flow graph. All rules assume a valid context Γ and check that context extensions maintain validity.

Elaboration, Synthesis Mode: $\Gamma \vdash M' \Rightarrow M \uparrow N$
The synthesizing (partial) elaboration function (Fig. 1) takes a context Γ and a partial term M' and produces the corresponding explicit term M and its type N. For example, the following judgment is derivable.

$$\Gamma \vdash (\mathit{cons}\ 1\ \mathit{nil}) \Rightarrow (\mathit{cons}\ \mathbb{N}\ 1\ (\mathit{nil}\ \mathbb{N})) \uparrow (\mathit{List}\ \mathbb{N})$$

The function implements essentially a colored version of the most natural type inference algorithm for $\mathcal{CC}$ known as the Constructive Engine [Hue89] and is used if nothing is known about the expected type of an expression. It makes use of the abbreviating judgment $\Gamma \vdash M' \overset{\text{wh}}{\Rightarrow} M \uparrow N$, which is identical to the SM judgment but assumes N to be in whnf. Note that the syntactic test in the side condition $B \not\equiv \mathit{Type}$ is strong enough to ensure the Π-abstraction to be well typed even in our calculus extended with unification variables. The differences with respect to the Constructive Engine are the addition of the Curry-style abstraction rules, $(\lambda^*\uparrow)$ and $(\Pi^*\uparrow)$, which generate new unification variables for the missing abstraction types, and a modified application rule $(@\uparrow)$. This application rule embodies a simple heuristic: always synthesize the type of the function, and then use the resulting information to switch to checking mode for the argument expression by calling the argument generation function.

Argument Generation: $\Gamma \vdash L'; l \Rightarrow M{:}N$
Argument generation essentially calculates the type of the application head L' and elaborates the list of arguments l under the expected parameter types in checking mode (Fig. 2). If the type of the head is not functional (e. g., a unification variable) elaboration fails. Unification variables are introduced at hidden positions unless overwritten by a forced argument. The result is the elaborated application M of type N. The example derivation from the last paragraph contains a subderivation of the following argument generation judgment.

$$\Gamma \vdash \mathit{cons}; \mathit{nil} :: 1 :: \cdot \Rightarrow (\mathit{cons}\ \mathbb{N}\ 1\ (\mathit{nil}\ \mathbb{N})) : (\mathit{List}\ \mathbb{N})$$

---

[7] The basic idea of bidirectional checking is not new and is for example used by the ML type-inference algorithm known as Algorithm $\mathcal{M}$ [LY98].

$$(\textsc{Prop}\uparrow)\ \frac{}{\Gamma \vdash \mathit{Prop} \Rightarrow \mathit{Prop} \uparrow \mathit{Type}} \qquad (\textsc{Var}\uparrow)\ \frac{}{\Gamma_1, x{:}A, \Gamma_2 \vdash x \Rightarrow x \uparrow A}$$

$$(\lambda\uparrow)\ \frac{\Gamma \vdash A' \overset{\text{wh}}{\Rightarrow} A \uparrow s \qquad \Gamma, x{:}A \vdash M' \Rightarrow M \uparrow B}{\Gamma \vdash \lambda x \| A' \,.\, M' \Rightarrow \lambda x \| A \,.\, M \uparrow \Pi x \| A \,.\, B}\ [B \not\equiv \mathit{Type}]$$

$$(\lambda^*\uparrow)\ \frac{\Gamma \vdash ?a{:}* \qquad \Gamma, x{:}?a \vdash M' \Rightarrow M \uparrow B}{\Gamma \vdash \lambda x \| \,.\, M' \Rightarrow \lambda x \| ?a \,.\, M \uparrow \Pi x \| ?a \,.\, B}\ [B \not\equiv \mathit{Type}] \qquad (@\uparrow)\ \frac{@(M' \ \| N') = \langle L' \,,\, l \rangle \qquad \Gamma \vdash L'; l \Rightarrow U{:}V}{\Gamma \vdash (M' \ \| N') \Rightarrow U \uparrow V}$$

$$(\Pi\uparrow)\ \frac{\Gamma \vdash A' \overset{\text{wh}}{\Rightarrow} A \uparrow s_1 \qquad \Gamma, x{:}A \vdash B' \overset{\text{wh}}{\Rightarrow} B \uparrow s_2}{\Gamma \vdash \Pi x \| A' \,.\, B' \Rightarrow \Pi x \| A \,.\, B \uparrow s_2} \qquad (\Pi^*\uparrow)\ \frac{\Gamma \vdash ?a{:}* \qquad \Gamma, x{:}?a \vdash B' \overset{\text{wh}}{\Rightarrow} B \uparrow s}{\Gamma \vdash \Pi x \| \,.\, B' \Rightarrow \Pi x \| ?a \,.\, B \uparrow s}$$

Fig. 1. Elaboration, Synthesis Mode

$$(\bullet)\ \frac{\Gamma \vdash M' \Rightarrow M \uparrow N}{\Gamma \vdash M'; \cdot \Rightarrow M{:}N} \qquad (@|)\ \frac{\Gamma \vdash M'; l \overset{\text{wh}}{\Rightarrow} L : \Pi x | A \,.\, B \quad \Gamma \vdash ?n{:}A}{\Gamma \vdash M'; N' :: l \Rightarrow (L\ ?n) : B[x{:=}?n]}$$

$$(@:)\ \frac{\Gamma \vdash M'; l \overset{\text{wh}}{\Rightarrow} L : \Pi x{:}A \,.\, B \qquad \Gamma \vdash N' \downarrow \overline{A} \Rightarrow N}{\Gamma \vdash M'; N' :: l \Rightarrow (L\ N) : B[x{:=}N]} \qquad (@!)\ \frac{\Gamma \vdash M'; l \overset{\text{wh}}{\Rightarrow} L : \Pi x | A \,.\, B \qquad \Gamma \vdash N' \downarrow \overline{A} \Rightarrow N}{\Gamma \vdash M'; 0!N' :: l \Rightarrow (L\ N) : B[x{:=}N]}$$

$$(@n!)\ \frac{\Gamma \vdash M'; (n-1)!N' :: l \overset{\text{wh}}{\Rightarrow} L : \Pi x | A \,.\, B \quad \Gamma \vdash ?n{:}A}{\Gamma \vdash M'; n!N' :: l \Rightarrow (L\ ?n) : B[x{:=}?n]}\ [n{>}0]$$

Fig. 2. Argument Generation

Elaboration, Checking Mode: $\Gamma \vdash M' \downarrow N \Rightarrow M$
The checking mode, described by the rules of Fig. 3, is used if the surrounding term context determines the type of the expression. The partial term M' is elaborated to M under the given expected type N, which is assumed to be in whnf. The resulting term M is guaranteed to be of type N. There is no side condition $B \not\equiv \mathit{Type}$ in the λ-abstraction rules, since the expected type is known to be valid. Note further, that the expected type in rule ($@\downarrow$) cannot be propagated down to elaborate the function part of an application since the result type of a function in $\mathcal{CC}$ depends on the actual arguments and not only on their number. Thus, to ensure soundness a final unification, essentially done by a call to the coerce to type function, is necessary. The argument *nil* of the one-element list (*cons* 1 *nil*) is elaborated in CM by a derivation of the following judgment.

$$\Gamma \vdash \mathit{nil} \downarrow (\mathit{List}\ \mathbb{N}) \Rightarrow (\mathit{nil}\ \mathbb{N})$$

Coerce to Type: $\Gamma \vdash M_1{:}N \downarrow O \Rightarrow M_2$
The coerce to type function tries to convert the given term M_1 of type N, with $\Gamma \vdash M_1{:}N$, to a related term M_2 of the expected type O (Fig. 4). The rule (Unif) just checks if the given and the expected type are unifiable and therefore,

$$(\textsc{Prop}\!\downarrow)\ \frac{}{\Gamma \vdash \mathit{Prop}\!\downarrow\!\mathit{Type} \Rightarrow \mathit{Prop}} \qquad (\textsc{Var}\!\downarrow)\ \frac{\Gamma \vdash x \Rightarrow x \uparrow A \quad \Gamma \vdash x{:}\overline{A} \downarrow B \Rightarrow M}{\Gamma \vdash x \downarrow B \Rightarrow M}$$

$$(\lambda\!\downarrow)\ \frac{\Gamma \vdash A' \overset{\mathrm{wh}}{\Rightarrow} A \uparrow s \quad \Gamma \vdash A \approx A_{\mathrm{e}} \quad \Gamma, x{:}A \vdash M' \downarrow \overline{B} \Rightarrow M}{\Gamma \vdash \lambda x \| A' \,.\, M' \downarrow \Pi x \| A_{\mathrm{e}} \,.\, B \Rightarrow \lambda x \| A \,.\, M} \qquad (@\!\downarrow)\ \frac{\Gamma \vdash (M' \;\| N') \Rightarrow U \uparrow V \quad \Gamma \vdash U{:}\overline{V} \downarrow O \Rightarrow Q}{\Gamma \vdash (M' \;\| N') \downarrow O \Rightarrow Q}$$

$$(\lambda^*\!\downarrow)\ \frac{\Gamma, x{:}A \vdash M' \downarrow \overline{B} \Rightarrow M}{\Gamma \vdash \lambda x \| \,.\, M' \downarrow \Pi x \| A \,.\, B \Rightarrow \lambda x \| A \,.\, M} \qquad (\Pi^*\!\downarrow)\ \frac{\Gamma \vdash ?a{:}* \quad \Gamma, x{:}?a \vdash B' \downarrow s \Rightarrow B}{\Gamma \vdash \Pi x \| \,.\, B' \downarrow s \Rightarrow \Pi x \| ?a \,.\, B}$$

$$(\Pi\!\downarrow)\ \frac{\Gamma \vdash A' \overset{\mathrm{wh}}{\Rightarrow} A \uparrow s_1 \quad \Gamma, x{:}A \vdash B' \downarrow s_2 \Rightarrow B}{\Gamma \vdash \Pi x \| A' \,.\, B' \downarrow s_2 \Rightarrow \Pi x \| A \,.\, B} \qquad (\mathrm{UV}\!\downarrow)\ \frac{\Gamma \vdash M' \Rightarrow M \uparrow N \quad \Gamma \vdash M; \overline{N} \downarrow ?n \Rightarrow O}{\Gamma \vdash M' \downarrow ?n \Rightarrow O}$$

Fig. 3. Elaboration, Checking Mode

$$(\textsc{Unif})\ \frac{\Gamma \vdash N \approx O}{\Gamma \vdash M{:}N \downarrow O \Rightarrow M} \qquad (\mathrm{NTI})\ \frac{\Gamma \vdash (\Pi x | A \,.\, B) \not\approx O \quad \Gamma \vdash ?n{:}A \quad \Gamma \vdash (M\ ?n) : \overline{B[x{:=}?n]} \downarrow O \Rightarrow P}{\Gamma \vdash M : \Pi x | A \,.\, B \downarrow O \Rightarrow P}$$

Fig. 4. Coerce to Type

the given term M_1 has not to be modified apart from resulting instantiations. If this fails and the given type is an implicit Π-abstraction, a newly created unification variable is applied to M_1 through *nil-type inference* using rule (NTI) and the result is recursively checked. In the other cases elaboration fails. While this strategy enables the inference of the type argument of *nil*, it rules out the possibility to collect unification constraints first and solve them later. Solving constraints immediately seems to be more efficient anyway [BN00].

The elaboration of the argument *nil* under the expected type $(\mathit{List}\ \mathbb{N})$ is done using the rule (NTI), deriving the following judgment.

$$\Gamma \vdash \mathit{nil} : \Pi T | \mathit{Prop} \,.\, (\mathit{List}\ T) \downarrow (\mathit{List}\ \mathbb{N}) \Rightarrow (\mathit{nil}\ \mathbb{N})$$

3.2 Properties

Proposition 1. *If* $\Gamma \vdash M_1{:}N \downarrow O_1 \Rightarrow M_2$ *then* $\Gamma \vdash M_2{:}O_2$ *with* $O_1 \simeq O_2$.

Proposition 2 (Soundness). *If* $\Gamma \vdash M' \Rightarrow M{:}N$ *then* $\Gamma \vdash M{:}N$.

Proof. By (mutual) induction on the derivation trees of the elaboration and argument generation judgments using Proposition 1, correctness of unification, subject reduction and correctness of types.

Proposition 3 (Partial Completeness). *If $\Gamma \vdash M{:}N_1$ and M' corresponds to the term M with forced applications at all implicit parameter positions then $\Gamma \vdash M' \Rightarrow M{:}N_2$ and $N_1 \simeq N_2$.*

Proof. Since the term M' has no missing subterms our algorithm never generates unification variables. Therefore, all derivations can be translated into derivations without CM judgments and unification reduces to conversion leading to derivations identically to those of the (bicolored) Constructive Engine, which is known to be complete [Hue89].

Partial completeness essentially enables the user to give just as much explicit type information as needed. This is necessary, because we cannot expect elaboration to be complete.

Generally, the elaboration algorithm calculates only one of several possible elaborations. For example, assuming a constant id of type $\Pi T|Prop\,.\,T \to T$ the partial term (id id 1) has the two elaborations (id ($\mathbb{N} \to \mathbb{N}$) ($id$ $\mathbb{N}$) 1) and (id ($\Pi T|Prop\,.\,T \to T$) id $\mathbb{N}$ 1) which are not convertible. Our algorithm would generate the second elaboration.

4 Erasure

The erasure algorithm is supposed to remove as many type annotations and arguments at implicit positions from a given term as possible, without losing any information. We propose an algorithm (Fig. 5) that mimics elaboration in an abstract way to predict its behavior. The erasure judgments, of the form $\Gamma \vdash M'; \mathcal{C} \overset{\mathrm{M}}{\Leftarrow} M$, compute for a unification variable free term M, $\mathcal{UV}(M) = \emptyset$, the erasure M' and the set $\mathcal{C}$ of variables that are erased in CM on first occurrence. It works again in one of two modes, $\mathrm{M} \in \{\mathrm{SM}, \mathrm{CM}\}$, corresponding roughly to those of the bidirectional elaboration algorithm. In contrast to elaboration, erasure works also in synthesis mode if the expected type on elaboration would be an open term, conservatively assuming it not to contain any structural information. Only if the expected type on elaboration is known to be unification variable free, erasure works in checking mode.

Type annotations of λ-abstractions are always left implicit if erasure is in checking mode, since those can be read off the fully explicit expected type without even generating a unification variable using rule ($\lambda^*\!\downarrow$) of Fig. 3. Other type annotations are only left implicit if the first reference to the corresponding abstraction variable x is erased in checking mode (i. e., $x \in \mathcal{C}$). Both cases are shown by the following two derivable judgments.

$$\Gamma, f : (\mathbb{N} \to \mathbb{N}) \to \mathbb{N} \vdash f\ (\lambda n\,.\,n); \emptyset \overset{\mathrm{SM}}{\Leftarrow} f\ (\lambda n{:}\mathbb{N}\,.\,n) \qquad \Gamma \vdash \lambda n\,.\,n{+}1; \{n\} \overset{\mathrm{SM}}{\Leftarrow} \lambda n{:}\mathbb{N}\,.\,n{+}1$$

Arguments at marked position are left implicit if they are determined by the first depending argument type in elaboration order or by the expected type, if in CM (Fig. 6). Otherwise, erasure represents these arguments explicit as forced arguments.

$$(\textsc{Prop})\ \frac{}{\Gamma \vdash \mathit{Prop}; \emptyset \stackrel{\mathrm{M}}{\Leftarrow} \mathit{Prop}} \qquad (\textsc{Var})\ \frac{}{\Gamma \vdash x; \emptyset \stackrel{\mathrm{SM}}{\Leftarrow} x} \qquad (\textsc{Var}^*)\ \frac{}{\Gamma \vdash x; \{x\} \stackrel{\mathrm{CM}}{\Leftarrow} x}$$

$$(\lambda^*)\ \frac{\Gamma, x{:}A \vdash M'; \mathcal{C} \stackrel{\mathrm{CM}}{\Leftarrow} M}{\Gamma \vdash \lambda x \| \,.\, M'; \mathcal{C} \stackrel{\mathrm{CM}}{\Leftarrow} \lambda x \| A \,.\, M} \qquad (@)\ \frac{\Gamma \vdash L'; P; i; \mathcal{S}; \mathcal{C} \stackrel{\mathrm{M}}{\Leftarrow} (M\ N); 0}{\Gamma \vdash L'; \mathcal{C} \stackrel{\mathrm{M}}{\Leftarrow} (M\ N)}$$

$\mathcal{Q} \in \{\lambda, \Pi\}$

$$(\mathcal{Q}^*)\ \frac{x \in \mathcal{C} \quad \Gamma, x{:}A \vdash M'; \mathcal{C} \stackrel{\mathrm{M}}{\Leftarrow} M}{\Gamma \vdash \mathcal{Q}x \| \,.\, M'; \mathcal{C} \stackrel{\mathrm{M}}{\Leftarrow} \mathcal{Q}x \| A \,.\, M} \qquad (\mathcal{Q})\ \frac{\Gamma \vdash A'; \mathcal{C}_1 \stackrel{\mathrm{SM}}{\Leftarrow} A \quad \Gamma, x{:}A \vdash M'; \mathcal{C}_2 \stackrel{\mathrm{M}}{\Leftarrow} M \quad x \notin \mathcal{C}_2 \quad \mathcal{C} = \mathcal{C}_1 \cup (\mathcal{C}_2 \setminus \mathcal{FV}(A))}{\Gamma \vdash \mathcal{Q}x \| A' \,.\, M'; \mathcal{C} \stackrel{\mathrm{M}}{\Leftarrow} \mathcal{Q}x \| A \,.\, M}$$

Fig. 5. Bidirectional Erasure

Argument erasure judgments are of the form $\Gamma \vdash M'; N; i; \mathcal{S}; \mathcal{C} \stackrel{\mathrm{M}}{\Leftarrow} M; n$, where M is the application term to be erased and n is the number of additional arguments. It calculates the erasure M' and the type N of M, the number of preceding implicit argument positions i, a set $\mathcal{S}$ of positions which should be erased in SM since an implicit argument has to be inferred from the corresponding argument type and the set $\mathcal{C}$ as described above. Note, that arguments are identified here by the number of consecutive arguments. The erasure mode of an argument is computed from the set $\mathcal{S}$ as follows.

$$\mathit{mode}(n, \mathcal{S}) = \begin{cases} \mathrm{SM} & \text{if } n \in \mathcal{S} \\ \mathrm{CM} & \text{else} \end{cases}$$

On erasing the term $(\mathit{cons}\ \mathbb{N}\ 1\ (\mathit{nil}\ \mathbb{N}))$ a derivation of the following judgment is constructed. The resulting set $\mathcal{S} = \{1\}$ determines the second argument, 1, to be erased in SM while the last argument, $(\mathit{nil}\ \mathbb{N})$, can be erased in CM since nothing has to be inferred from its type.

$$\Gamma \vdash \mathit{cons}; \Pi T | \mathit{Prop} \,.\, T \to (\mathit{List}\ T) \to (\mathit{List}\ T); 1; \{1\}; \emptyset \stackrel{\mathrm{SM}}{\Leftarrow} (\mathit{cons}\ \mathbb{N}); 2$$

The calculation of the determining information source, if any, for arguments at implicit parameter positions is done by the function $\mathit{dpos}_{\mathrm{M}}$.

Definition 4 (Determining Position). *The function $\mathit{dpos}_{\mathrm{M}}$ for a mode* M*, a context Γ, a term $T \equiv \Pi x | A \,.\, B$ with $|B| \equiv \Pi x_1 \| A_1, \ldots, x_l \| A_l \,.\, C$, $C \not\equiv \Pi y \| D \,.\, E$, another term N with $\Gamma \vdash N{:}A$ and $n \in \mathbb{N}$ is specified as follows.*

$$\mathit{dpos}_{\mathrm{M}}(\Gamma, T, N, n) = \begin{cases} n - r & \mathit{if}\ \exists\, r \in \mathbb{N} \,.\, 0 < r \leq \min(l, n),\ x \notin \bigcup_{i=1}^{r-1} \mathcal{FV}(|A_i|) \\ & \quad \mathit{and}\ x \in \mathcal{SF}(\mathit{dom}(\Gamma), A_r) \\ \blacksquare & \mathit{if}\ \mathrm{M} = \mathrm{CM},\ n \leq l,\ x \notin \bigcup_{i=1}^{n} \mathcal{FV}(|A_i|),\ T \not\simeq B[x := N] \\ & \quad \mathit{and}\ x \in \mathcal{SF}(\mathit{dom}(\Gamma), \Pi x_{n+1} \| A_{n+1}, \ldots, x_l \| A_l \,.\, C) \\ \bigstar & \mathit{else} \end{cases}$$

The result of $\mathit{dpos}_{\mathrm{M}}$ is a number $d \in \mathbb{N}$ indicating the determining argument, or one of the symbols "$\blacksquare$", "$\bigstar$" if the argument can be inferred from the expected type or cannot be inferred at all, respectively. The condition $T \not\simeq B[x := N]$

$$(\textsc{NoArg})\ \frac{\Gamma \vdash M{:}N \qquad \Gamma \vdash M';\mathcal{C} \overset{\text{SM}}{\Leftarrow} M}{\Gamma \vdash M';\overline{N};0;\emptyset;\mathcal{C} \overset{\text{M}}{\Leftarrow} M;n}\ {\scriptstyle [M \not\equiv (M_1\ M_2)]}$$

$$(\textsc{Visible})\frac{\Gamma \vdash M';\Pi x{:}A\,.\,B;i;\mathcal{S};\mathcal{C}_1 \overset{\text{M}}{\Leftarrow} M;n+1 \qquad \Gamma \vdash N';\mathcal{C}_2 \overset{m}{\Leftarrow} N}{\Gamma \vdash (M'\ N');\overline{B[x:=N]};0;\mathcal{S};\mathcal{C}_1 \cup (\mathcal{C}_2 \setminus \mathcal{FV}(M)) \overset{\text{M}}{\Leftarrow} (M\ N);n}\ {\scriptstyle [m=mode(n,\mathcal{S})]}$$

$$(\textsc{Forced})\frac{\begin{array}{c}\Gamma \vdash M';\Pi x|A\,.\,B;i;\mathcal{S};\mathcal{C}_1 \overset{\text{M}}{\Leftarrow} M;n+1 \qquad \Gamma \vdash N';\mathcal{C}_2 \overset{m}{\Leftarrow} N\\ n \in \mathcal{S} \vee dpos_{\text{M}}(\Gamma,\Pi x|A\,.\,B,N,n) = \bigstar\end{array}}{\Gamma \vdash (M'\ i!N');\overline{B[x:=N]};0;\mathcal{S};\mathcal{C}_1 \cup (\mathcal{C}_2 \setminus \mathcal{FV}(M)) \overset{\text{M}}{\Leftarrow} (M\ N);n}\ {\scriptstyle [m=mode(n,\mathcal{S})]}$$

$$(\textsc{Implicit})\ \frac{\begin{array}{c}\Gamma \vdash M';\Pi x|A\,.\,B;i;\mathcal{S};\mathcal{C} \overset{\text{M}}{\Leftarrow} M;n+1\\ dpos_{\text{M}}(\Gamma,\Pi x|A\,.\,B,N,n) = d \quad n \notin \mathcal{S}\end{array}}{\Gamma \vdash M';\overline{B[x:=N]};i+1;\mathcal{S}\cup\{d\};\mathcal{C} \overset{\text{M}}{\Leftarrow} (M\ N);n}\ {\scriptstyle [d \neq \bigstar]}$$

Fig. 6. Argument Erasure

ensures the applicability of the rule (NTI) on elaboration by forcing the term M applied to the argument N to change its type. This is for example not the case for any term of type $\Pi T|Prop\,.\,T$ applied to its own type.

The definition of $dpos_{\text{M}}$ depends on the set $\mathcal{SF}$ of free variables of a term, solvable by unification with a fully explicit term. This set is defined as follows, assuming nothing is ever substituted to variables of the set $\mathcal{V}_0$ during unification.

Definition 5 (Solvable and Dangerous Free Variables). *The set of* solvable free variables*, $\mathcal{SF}(\mathcal{V}_0, M)$, of a term M relative to variables $\mathcal{V}_0$ is defined mutual dependent with the set of* dangerous free variables*, $\mathcal{DF}(\mathcal{V}_0, M)$, as follows.*

$$\mathcal{SF}(\mathcal{V}_0, M) = \mathcal{SF}^*(\mathcal{V}_0, \overline{M}, \emptyset) \qquad \mathcal{DF}(\mathcal{V}_0, M) = \mathcal{DF}^*(\mathcal{V}_0, \overline{M}, \emptyset)$$

with

$$\mathcal{SF}^*(\mathcal{V}_0, M, \mathcal{V}) = \begin{cases} \{x\} & \textit{if } M \equiv x \textit{ and } x \notin \mathcal{V}_0 \cup \mathcal{V} \\ \mathcal{SF}^*(\mathcal{V}_0, \overline{A}, \mathcal{V}) & \textit{if } M \equiv \mathcal{Q}x\|A\,.\,N, \\ \quad \cup\, \mathcal{SF}^*(\mathcal{V}_0, \overline{N}, \mathcal{V} \cup \{x\}) \setminus \mathcal{DF}^*(\mathcal{V}_0, \overline{A}, \mathcal{V}) & \quad \mathcal{Q} \in \{\lambda, \Pi\} \\ \mathcal{SF}^*(\mathcal{V}_0, M_1, \mathcal{V}) & \textit{if } M \equiv (M_1\ M_2), \\ \quad \cup\, \mathcal{SF}^*(\mathcal{V}_0, \overline{M_2}, \mathcal{V}) \setminus \mathcal{DF}^*(\mathcal{V}_0, M_1, \mathcal{V}) & \quad head(M_1) \in \mathcal{V}_0 \cup \mathcal{V} \\ \emptyset & \textit{else} \end{cases}$$

and $\quad \mathcal{DF}^*(\mathcal{V}_0, M, \mathcal{V}) = \mathcal{FV}(|M|) \setminus \mathcal{SF}^*(\mathcal{V}_0, M, \mathcal{V})$

The definition of the set $\mathcal{SF}$ mimics unification in that terms are kept essentially in normal form through stepwise reduction to whnf. The condition $head(M_1) \in \mathcal{V}_0 \cup \mathcal{V}$ ensures that the heading of the term M remains unchanged under application of any substitution with the implication that all $x \in \mathcal{SF}$ have residuals in every reduction of every substitution instance of M.

To illustrate the last two definitions consider the erasure of $(nil\ \mathbb{N})$ in CM; the last argument of the above example. $dpos_{\text{CM}}(\Gamma, \Pi T|Prop\,.\,(List\ T), \mathbb{N}, 0)$ yields

$\blacksquare$ since $\mathcal{SF}(dom(\Gamma), (List\ T)) = \{T\}$ assuming $List \in dom(\Gamma)$. This allows to derive the following argument erasure judgment using rule (Implicit) of Fig. 6.

$$\Gamma \vdash nil; (List\ \mathbb{N}); 1; \{\blacksquare\}; \emptyset \overset{\text{CM}}{\Leftarrow} (nil\ \mathbb{N}); 0$$

4.1 Properties

Proposition 6. *If $\Gamma \vdash M_1{:}N$, $\Gamma \vdash M_2{:}N$, $M_1 \simeq M_2$, $\mathcal{UV}(M_2) = \emptyset$, $\Gamma \vdash x{:}A$, $x \in \mathcal{SF}(dom(\Gamma), M_1), \Gamma \vdash ?n{:}A$ then $\Gamma \vdash M_1[x := ?n] \approx M_2$ yields the most general instantiation ι with $?n \in dom(\iota)$.*

It is always possible to reconstruct the original term from the erasure using the elaboration algorithm of Section 3.

Proposition 7 (Invertibility). *If $\Gamma \vdash M_1{:}N_1$ and $\Gamma \vdash M'; \mathcal{C} \overset{\text{SM}}{\Leftarrow} M_1$ then $\Gamma \vdash M' \Rightarrow M_2{:}N_2$ with $M_1 \simeq M_2$.*

Invertibility essentially holds, since all unification variables generated by elaboration for erased subterms are solved by the first typing constraints they participate in. It can be verified that all generated unification problems are such that one of the terms to be unified does not contain any unification variable and the other term does only contain solvable ones, which are guarantied to be instantiated (Proposition 6).

Since type annotations on parameters can help to make expressions more readable serving as checked documentation, erasure prefers implicit arguments over implicit annotations. Consider an (explicit) polymorphic function f of type $\Pi T|Prop\,.\,(T \to Prop) \to Prop$. The algorithm above computes the erasure $(f\ (\lambda x{:}\mathbb{N}\,.\,\mathbb{N}))$ rather than $(f\ !\mathbb{N}\ (\lambda x\,.\,\mathbb{N}))$ for the explicit term $(f\ \mathbb{N}\ (\lambda x{:}\mathbb{N}\,.\,\mathbb{N}))$. Note further, that two explicit terms that are structurally equal, can still have different erasures and that different explicit terms can lead to the same erasure, but only if both terms occur in different term contexts.

5 Experimental Results

We have implemented and tested several variants of the elaboration and erasure algorithms discussed above as part of the proof assistant Typelab [vHLS97]. For evaluation purposes we analyzed terms of different sizes up to 15,000 abstract syntax tree nodes. The terms were arbitrarily selected from definitions and proofs of the standard Typelab library.

We found that compression factors are independent of the fully explicit term size. For that, we calculated the percentage reduction in the total size of all terms but separated the results for definitions from the results for proofs (Table 1). On average, our combined erasure algorithm almost reduces the representation in half the size while the compression is slightly more effective for proof terms.

It has to be asked whether one could find an erasure algorithm which yields much smaller representations. To answer this question we determined the arguments at implicit positions that our combined erasure algorithm presented explicitly. We found that on average, terms could only be reduced by another 1.1%

Table 1. Reduction in abstract syntax tree size

	Implicit Arguments	Implicit Annotations	Combination
Proofs	22.1%	32.3%	49,4%
Definitions	24.7%	26.3%	39.5%

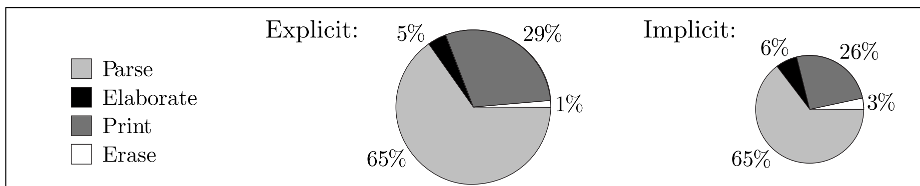


Fig. 7. Effect of implicit syntax on the full turnaround time

through blindly erasing all those arguments while only 20% of them had enough information to be reconstructed. Erasing in addition all remaining type annotations from abstractions reduced the representation by another 7,6%, but none of those implicit terms could be reconstructed. We conclude that our combined erasure algorithm removes the vast majority of redundant subterms, respecting the given color information, thus leaving little room for further improvement.

To analyze the performance benefits gained using implicit syntax we measured for all terms the times[8] needed for a full turnaround including parsing, elaboration, erasure and the final printing of the implicit representation. Since we found all timings to be linear in the size of the fully explicit term, we averaged the results again. Fig. 7 shows the results for the fully explicit representation compared with our combined implicit representation. The pies are sized by area, which corresponds to the absolute representation size. We can conclude that, in practice, the additional costs produced by the erasure algorithm are smaller than the savings gained from dealing with reduced representations.

6 Disscussion

We have presented algorithms that improve the usability of implicit syntax for proof assistants. Our inference algorithm is stronger than the one of Coq or Lego, since it allows to omit more subterms. Furthermore, our erasure algorithm generates only implicit representations that allow the reconstruction of the original terms, in contrast to the ad-hoc erasure algorithms implemented in Coq and Lego. The experimental results presented in the previous section provide evidence that our algorithms, while still being efficient, save considerable bandwidth between the user and the system.

To implement the algorithms of this paper for the assistant Typelab we had to consider additional aspects. Deciding when to expand *notational definitions* is subtle for unification algorithms. Stepwise expansion, as done by most proof assistants, may return a unifier which is not most general and hence renders

[8] We found similar results about the space requirements.

unification incomplete even for the first-order case [PS99] and thus would limit our elaboration algorithm. To adapt the erasure algorithm to the language of TYPELAB with *recursive definitions*, the set $\mathcal{SF}$ had to be restricted since applications with recursively defined heads are potentially 'instable' with respect to substitution at recursive parameter positions.

The local erasure algorithm in this paper hides only arguments that can be elaborated by local methods [PT98], while our elaboration algorithm allows the global distribution of unification variables. Consider the polymorphic operation *append* on lists of type $\Pi T|Prop\,.\,(List\ T) \to (List\ T) \to (List\ T)$ and a list l of type $(List\ \mathbb{N})$. The implicit term $(append\ nil\ l)$ is elaborated into the explicit term $(append\ \mathbb{N}\ (nil\ \mathbb{N})\ l)$, but erasure would produce the wordy representation $(append\ (nil\ !\mathbb{N})\ l)$. We have also implemented an erasure algorithm that does not force elaboration to solve implicit arguments on the first opportunity completely. It works with an additional erasure mode, CM^*, where the expected type is allowed to be an open term. This considerably more complex algorithm[9] computes the optimal erasure $(append\ nil\ l)$ for the explicit term above.

One drawback of implicit syntax is the enlarged trusted code-base of the proof-checker. Fortunately, for sensible applications internal terms can always be rechecked by a small trusted or even verified checker for the base calculus.

6.1 Related Work

Berghofer and Nipkow describe a dynamic compression algorithm for proof terms in ISABELLE [BN00]. Their algorithm searches for optimal representations by essentially doing elaboration on erasure and seems not to be efficient enough for interactive usage.

The problem of redundancy has been addressed also by Necula and Lee [NL98] in the context of Proof-Carrying Code systems. They analyze the combination of ad-hoc argument synthesis with implicit type annotations for canonical first-order proof objects represented as fully applied LF terms in long $\beta\eta$-normal form, given a fully explicit expected type. This special setting enables the precomputing of large parts of the erasure for constants.

While a language generated by implicit syntax is natural to humans, it seems rather difficult to give it a direct foundation. Hagiya and Toda [HT95] have implemented an implicit version of $\mathcal{CC}$ directly using a typed version of β-reduction defined on the implicit language. Several complicated syntactic restrictions have to be imposed to ensure decidability of type inference and to avoid dynamic type checking during reduction. On the theoretical side, Miquel defined a Curry-style version of $\mathcal{CC}$ [Miq01]. Although the metatheory of this calculus is not yet fully developed, it is strongly conjectured that type checking is undecidable. For that reason, this calculus seems to be a poor basis for proof assistants.

Acknowledgments I thank Martin Strecker for developing large parts of the TYPELAB system.

[9] Details are subject of current research.

References

[Bar99] B. Barras et al. The Coq proof assistant reference manual – Version 6.3.1. Technical report, INRIA, France, 1999.

[BN00] S. Berghofer, T. Nipkow. Proof terms for simply typed higher order logic. In *Proc. of TPHOLs'00*, volume 1869 of *LNCS*, pp. 38–53. Springer, 2000.

[CH88] Th. Coquand, G. Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3):95–120, 1988.

[Geu92] H. Geuvers. The Church-Rosser property for $\beta\eta$-reduction in typed λ-calculi. In *Proc. of LICS'92*, pp. 453–460. IEEE Press, 1992.

[GH98] W. O. D. Griffioen, M. Huisman. A comparison of PVS and Isabelle/HOL. In *Proc. of TPHOLs'98*, volume 1479 of *LNCS*, pp. 123–142. Springer, 1998.

[Gol81] W. Goldfarb. The undecidability of the second-order unification problem. *Theoretical Computer Science*, 13:225–230, 1981.

[HT95] M. Hagiya, Y. Toda. On implicit arguments. Technical Report TR-95-1, Department of Information Science, University of Tokyo, 1995.

[Hue89] G. Huet. The Constructive Engine. In *A Perspective in Theoretical Computer Science*. World Scientific Publishing, Singapore, 1989.

[LP92] Z. Luo, R. Pollack. *LEGO Proof Development System.* University of Edingburgh, 1992. Technical Report ECS-LFCS-92-211.

[LY98] O. Lee, K. Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM TOPLAS*, 20(4):707–723, 1998.

[Mil91] D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *J. Logic Comput.*, 1(4):497–536, 1991.

[Miq01] A. Miquel. The implicit calculus of constructions. In *Proc. of the Conf. TLCA'01, Krakow, Poland, May 2–5, 2001*, LNCS. Springer, Berlin, 2001.

[Muñ01] C. Muñoz. Proof-term synthesis on dependent-type systems via explicit substitutions. *Theoretical Computer Science*, 2001. To appear.

[NL98] G. Necula, P. Lee. Efficient representation and validation of proofs. In *Proc. of LICS'98*, pp. 93–104. IEEE Press, 1998.

[Pol90] R. Pollack. Implicit syntax. In G. Huet, G. Plotkin, eds., *Informal Proc. of the 1st Workshop on Logical Frameworks (LF'90), Antibes.* 1990.

[PS99] F. Pfenning, C. Schürmann. Algorithms for equality and unification in the presence of notational definitions. In T. Altenkirch, W. Naraschewski, B. Reus, eds., *Proc. of TYPES'98*, volume 1657 of *LNCS*, pp. 179–193. Springer, 1999.

[PT98] B. Pierce, D. Turner. Local type inference. In *Conf. Record of POPL'98*, pp. 252–265. ACM Press, 1998.

[SLvH98] M. Strecker, M. Luther, F. von Henke. Interactive and automated proof construction in type theory. In W. Bibel, P. Schmitt, eds., *Automated Deduction — A Basis for Applications*, volume I, chapter 3. Kluwer, 1998.

[Str99] M. Strecker. *Construction and Deduction in Type Theories.* Ph. D. thesis, Fakultät für Informatik, Universität Ulm, 1999. `<http://www.informatik.uni-ulm.de/ki/Strecker/phd.html>`

[vHLS97] F. von Henke, M. Luther, M. Strecker. Typelab: An environment for modular program development. In M. Bidoit, M. Dauchet, eds., *Proc. of the 7th Intl. Conf. TAPSOFT'97*, volume 1214 of *LNCS*, pp. 851–854. Springer, 1997.

[Wel99] J. B. Wells. Typability and type checking in System F are equivalent and undecidable. *Ann. Pure & Appl. Logics*, 98(1–3):111–156, 1999.