Typer: ML boosted with type theory and Scheme

Stefan Monnier
Université de Montréal - DIRO
monnier@iro.umontreal.ca

Abstract

We present the language Typer which is a programming language in the ML family. Its name is an homage to Scheme(r) with which it shares the design of a minimal core language combined with powerful metaprogramming facilities, pushing as much functionality as possible into libraries. Contrary to Scheme, its syntax includes traditional infix notation, and its core language is very much statically typed. More specifically the core language is a variant of the implicit calculus of constructions (ICC). We present the main elements of the language, including its Lisp-style syntactic structure, its elaboration phase which combines macro-expansion and Hindley-Milner type inference, its treatment of implicit arguments, and its novel approach to impredicativity.

1 Introduction

Typer is an experimental programming language born from a desire to have a programming language with a type system as powerful as that of Coq but with a meta-programming system like those from the Lisp family.

While Scheme with its dynamic typing may appear diametrically opposed to systems like Coq where the static typing can be extremely rigid, they both share the desire to make “everything” first class. In this sense, Coq shares some of Scheme’s design, with a minimal but very powerful language at its core (Gallina) and layers of meta-programming added on top to make it convenient for the programmer. Yet, Coq’s meta-programming does not follow the same minimalist approach, resulting in extra complexity: the meta-programs are split into tactics written in Ltac and proof scripts that call those different tactics. As a first approximation, those correspond respectively to Lisp macros and macro calls, except they don’t reuse the same language and syntax as Gallina.

Typer starts with a core language similar to that of Coq, but combines it with a macro facility where macros are written directly in and called directly from Typer, so the language can be seamlessly extended via meta-programming, just as in Lisp. Part of this is made possible by the use of a very primitive parsing technique, which is just flexible enough to support a fairly familiar infix syntax, yet simple enough that it maps straightforwardly to the equivalent of Lisp’s S-expressions.

While the core language lets us manipulate proofs, Typer is mostly meant to be used as a programming language. So we wanted the power of fully dependent types to not unduly get in the way of programs that do not make use of them. Concretely, we tried to design Typer in such a way that programs can be written with just as few extra annotations as in any other ML-family language.

Finally, while the core of Typer is also a variant of the calculus of inductive constructions [Pau93], it is different from Coq’s pCIC in various important ways, most notably in that it eschews the Prop universe, replacing it with impredicative erasable arguments.
The paper presents the following contributions of the design of Typer:

- A syntactic structure that shares many of the features of Lisp but uses operator precedence grammars to apply it to a more familiar infix syntax.
- An elaboration phase which combines HM-style type inference and macro-expansion, relying on the inferred type information to distinguish macro calls.
- An extension of ICC* [BB08] with inductive types and a new rule for impredicativity.

2 Typer primer

Before getting to Typer’s internals, we’ll give a short overview of what the language looks like. To a first approximation Typer is very similar to other languages in the ML family. It is a statically typed (pure) functional language, with basically two core elements: functions and datatypes. To define a function which adds 1 to its argument, you can write:

```markdown
add1 : Int -> Int;
add1 = lambda x -> x + 1;
```

Like in Agda [BDN09], the type of dependently typed functions is written “\((x : \tau_1) \to \tau_2\)”. The definition above could have used a bit of a syntactic sugar to become:

```markdown
add1 x = x + 1;
```

To define a new datatype to represent singly linked lists you can write:

```markdown
type List (a : Type)
  | nil
  | cons (hd : a) (tl : List a);
```

where `hd` and `tl` are optional field names: we could have written just `cons a (List a)` instead.

Functions and data constructors are curried. You can define the `map` function as follows:

```markdown
map : (a : Type) \equiv (b : Type) \equiv (a \to b) \to List a \to List b;
map f xs = case xs
  | nil => nil
  | cons x xs => cons (f x) (map f xs);
```

The type declaration is generally optional, although we currently require it for recursive definitions. The triple arrow \(\equiv\) is used for functions whose argument is implicit, which is actually called erasable in Typer.

There is just one remaining important construct in Typer, which is `let`: This allows you to introduce new locally scoped definitions. The shape of this construct is “\(\text{let \ decls \ in \ exp}\)” where `decls` is a sequence of declarations such as the ones shown above, separated by semicolons. For example, we could have defined the above map function as follows:

```markdown
map f =
let map’ xs = case xs
  | nil => nil
  | cons x xs => cons (f x) (map’ xs)
in map’
```
Typer Stefan Monnier

Typer has fundamentally two syntactic categories: expressions and declarations. A Typer file is defined as a sequence of declarations, separated by semicolons.

If these are all the constructs, you might wonder how macros are defined. They’re defined simply as values with a dedicated type Macro:

```plaintext
if_then_else_ : Macro;
if_then_else_ = macro ifthen;
```

where macro is the constructor of the Macro type and ifthen is the function which performs the expansion. Ignoring the types, this is very similar to how it is done in Emacs Lisp. The ifthen function could be defined as follows:

```plaintext
ifthen : List Sexp -> ME Sexp;
ifthen args =
    let e1 = nth 0 args error_sexp;
    e2 = nth 1 args error_sexp;
    e3 = nth 2 args error_sexp;
    code = (quote
        (case (uquote e1)
            | true => (uquote e2)
            | false => (uquote e3)))
    in return code;
```

where quote is a macro similar to the backquote/quasiquote in Lisp macros (and uquote corresponds to the comma in those systems). Wherever the above macro is in scope, the programmer can write:

```plaintext
... if_then_else_ x "x is true" "x is false" ...
```

Being purely functional, Typer resorts to the usual monadic technique to get access to a side effecting world, just as is done in Haskell. In the above code, ME is the macro-expansion monad, used for the same purpose as the one in Template Haskell [SP02], and return is the unit of that monad.

3 Syntactic structure

Once lexical analysis is performed, rather than performing the syntactic analysis in one step, Typer further subdivides the syntactic analysis phase into two steps. The first step does a rudimentary analysis that only extracts a generic tree structure, called S-expression. The shape of S-expressions could be described with the datatype shown in Figure 1. Note that contrary to the Lisp S-expression syntax, parentheses are only used for grouping purposes, so (x) will produce the same Sexp as just x. And we use () as the printed representation of the zero-length symbol (which we call espilon).

Note how, at this stage, the representation of the code has no notion of bindings, functions, types, or function calls. It’s only in a second step that S-expressions are analyzed to distinguish the various constructs such as macro calls, function calls, let bindings, variable references, etc.

Any S-expression written using an infix or mixfix operator can also be written some other way, following the underscore convention of Agda’s mixfix [DN08]. In the case of Typer, instead of:

```plaintext
let x = a * b + c in x
```
Typer Stefan Monnier

let_in_ (_=_ x (_+_ (_*_ a b) c)) x

and these two notations result in identical S-expressions.

3.1 Operator precedence grammar

Typer’s external notion of S-expression is more flexible than Lisp’s, since it allows infix notation. It relies on operator precedence grammars (OPG) [Flo63] for that. An OPG is a very restrictive subset of context free grammars, much more restrictive than LALR, for example.

You can think of the job of an OPG parser from the point of view of someone trying to add parentheses to render the document’s structure explicit: whenever the parser sees something of the form “kw₁ e kw₂” (where kw₁ and kw₂ are keywords and e is a sequence of non-keyword tokens or fully parenthesized sub-trees), it just needs to decide whether that should be parenthesized as “kw₁ (e kw₂)” or “kw₁ e kw₂”. For example, when starting with:

... g + f(5) * 6 - x ...

The parser will look at “+ f(5) *” and add an open paren because it decides that the “f(5)” should be attached to the “*” keyword:

... g + (f(5) * 6) - x ...

then it will see “* 6 -” and add a close paren this time:

... g + (f(5) * 6) - x ...

Then it will consider “* (f(5) * 6) -” and add an open paren:

... g + ((f(5) * 6) - x ...

and so on and so forth. What sets OPG apart here is that it makes these choices without considering e nor the surrounding context: it bases its decision only on the pair of keywords.

In Typer, the grammar is represented by simply associating to each keyword two precedence levels: one for its left side and another for its right side. Then parsing uses the following rule: when we see “kw₁ e kw₂”, we lookup the right precedence of kw₁ and the left precedence of kw₂, and we then attach e to whichever is higher. If the precedences are equal, then we consider those two keywords as part of a mixfix.

For example, given the default grammar, we can define the new form “if e₁ then e₂ else e₃” by setting the precedences as follows:

define-operator "if" () 2;
define-operator "then" 2 1;
define-operator "else" 1 66;

After which such a form gets parsed identically to "$\text{if} \ e_1 \ \text{then} \ e_2 \ \text{else} \ e_3\$". Note that the modification of the grammar is independent from the definition of $\text{if} \ \text{then} \ \text{else}$ as a macro: the grammar can be changed for infix functions and new macros can be defined without changing the grammar.

While it enjoys a simple and efficient implementation\(^1\) the motivation behind the choice of an OPG parser was not efficiency but rather the following aspects:

- Extensible grammars suffer from an inherent lack of modularity, since the combination of two extensions can always lead to conflicts or ambiguities, sometimes in ways that are very difficult to understand (as anyone who had to fix a reduce/reduce conflict in an LALR parser can attest). While OPG’s simplicity means that conflicts are more frequent, they also tend to be much more superficial and hence easier to understand.

- More importantly, OPG grammars are “strongly context free”: a given stream of tokens will be parsed in the same way regardless of the surrounding context.

The second point is what makes them particularly suitable for S-expressions, since at the time of parsing, we do not know yet what role a given S-expression will play: we do not yet know if it is a type, a pattern, a declaration, an expression, an lvalue, or anything else for that matter since it depends on the definition of the macros in which it appears.

For example, in OCaml the following piece of code:

```
let x = a; b in x = a; b
```

is parsed as

```
let (x = (a; b)) in ((x = a); b)
```

Notice that the ‘;’ has higher precedence than ‘=’ before the “in” keyword (i.e. in a declaration context) but it’s the opposite after the “in” (i.e. in an expression context). So if Typer supported a similar grammar, when faced with a macro call of the form

```
mymacro (x = a; b)
```

the parser would need to know if the macro’s argument should be parsed as a declaration or as an expression. We did not want to make the parser depend so tightly on the semantics of the language: by restricting ourselves to an OPG grammar we disallow these context-dependent parsing rules, such that we do not need to know what is a macro call, let alone figure out what is that macro’s definition.

### 4 Elaboration

Elaboration is the phase in Typer’s compiler which turns an S-expression into an expression in Typer’s core λ-calculus. We want most of this phase to be itself implemented in Typer so that we can prove properties such as the correctness of the compilation of pattern matching [CA18].

Figure 2 shows the (simplified) representation used internally for that calculus. Notice that $Lexp$ represents both what is usually considered expressions (such as let, fun, ...) as well as

---

\(^1\)as well as some other interesting properties such as the ability to parse backward.
what is usually considered as types (e.g. \(\text{arw} \ x \ t_1 \ t_2\) which represents the function type \((x : t_1) \rightarrow t_2\), or \(\text{adt}\) which is the representation of an abstract data type) since this core language is a kind of Pure Type System (PTS) [Bar91].

Elaboration performs the following tasks:

- Finish the syntactic analysis: decide what is a function call, a macro call, a \textit{let} definition, a case analysis, ...
- Macro expand the macro calls.
- Infer and verify the types.

This is the heart of Typer’s front-end and requires a fair bit of supporting functionality: type inference needs to perform unification between arbitrary \textit{Lexp} terms as well as compare them for equality, which involves normalizing them; macro expansion requires evaluation of arbitrary Typer code, either via a small interpreter, or via the complete compiler and runtime system.

### 4.1 Final syntactic analysis

Elaboration has to distinguish the different constructs of the language. Figure 3 shows a pseudocode of how it works (where \textit{Ltype} is actually an alias for \textit{Lexp}). In the case of Typer, just as is the case of Scheme, we do it without hard coding the meaning of any identifier. More specifically, elaboration will not check to see if an \textit{Sexp node} has a symbol named for example \textit{let in} as its head. Instead, when it encounters a \textit{node}, it does the following:

1. Elaborate the head, which will also return the inferred type of that expression.
2. If the returned type is \textit{Special-Form}, it means this expression is one of the “built-in” ones and we want to call the corresponding special form’s elaboration function, found in a global table. There is a special form for each core syntactic construct, such as \textit{let in} and \textit{case}.
elaborate : Ctx → Sexp → Pair Lexp Ltype;
elaborate c sexp =
case sexp
    | symbol s ⇒ elab_variable_reference c s
    | number n ⇒ elab_immediate_value n
    | node head args ⇒
      let (e, t) = elaborate c head in
case t
    | "Macro" ⇒ elaborate c (macroexpand e c args)
    | "Special-Form" ⇒ elab_special_form c e args
    | _ ⇒ elab_funcall c t e args;

Figure 3: Sketch of the elaboration

3. If the returned type is Macro, then it is a macro call, and we expand it, as detailed below.
4. Otherwise, it should be a function call and we recurse on each element of the node.

Note that at step 2 above, we have to double check that the elaborated head is of the form “prim ..”, because the source code could be

(\if x then let_in_ else case_\) 42

in which case the head is a valid expression of type Special-Form but we do not know which special form it is, so we have to reject such meaningless code.

The way keywords like let_in_ get their special meaning is simply by binding them to the corresponding special form primitive in the initial environment. The programmers are free to rebind those identifiers if they want, or to bind the corresponding primitive to other identifiers.

4.2 Macro expansion

As explained above, a macro call is recognized simply by the fact that the head of the node has type Macro. As before with Special-Form, the mere fact that the head has type Macro does not guarantee that this is a valid macro. Again we may just be looking at a source code of the form:

(\if x then mymacro else yourmacro\) 42

where the head may be a valid expression of type Macro but is not really a macro we can expand because x will only be known at runtime. So, to make sure we do have a macro, we additionally need to verify that the head expression is closed: it can refer to let-bound variables, as long as these are themselves closed, but it cannot refer to a function’s formal argument. Once established that it is closed, we can reduce it (by regular evaluation) to a value out of which we can finally extract the Typer function to call to perform the expansion. Since Typer is pure, the evaluation of the head has no visible side-effects and its result can be cached (contrary to the macroexpansion itself, which is done in a monad that can perform arbitrary side-effects).

This approach naturally supports lexically scoped macros or even higher-order macros. Its downside is that it needs to know the type of the head of a node to detect a macro call. This makes it virtually impossible to expand all macros in a separate phase before we infer
Typer

And we can’t infer types before we have expanded the macros either, so we are forced to interleave macro expansion and type inference within one big elaboration phase. While this can be a significant downside, performing macro expansion from inside the type inference phase has the advantage that macros can get access to the complete type context.

In the context of macros that provide syntax extensions, this is often of little benefit, but it makes it possible to write other kinds of macros which act more like proof tactics, where the type environment represents the set of valid hypotheses, and the expected return type is the proposition one wants to prove. While Typer is a programming language at heart, its core language is very similar to that of proof assistants, so it can also be used to write and manipulate propositions and proofs.

4.3 Type checking

Typer uses a bidirectional type checking [PT00] approach to minimize the required type annotations. Usually, this means that elaboration is split into two mutually recursive functions:

- `infer` takes a type environment and an `Sexp` and returns the corresponding elaborated `Lexp` along with its type (also an `Lexp`).
- `check` takes a type environment, an `Sexp`, and its expected type (an `Lexp`), and returns the elaborated form, of type `Lexp`.

And each language construct is either handled in `infer` or in `check` (so `check` defers to `infer` when faced with a construct it does not know how to handle). For example, typically function calls are handled in `infer` as follows:

1. When a function call is encountered, `infer` is called on the function part.
2. The returned type is verified to be that of a function and then split into the argument type and the return type.
3. Then `check` is called on the argument since we now know its expected type.
4. Finally we can construct the `app` node and return it along with its type.

In our case, this division of labor would result in too much code duplication since both `infer` and `check` would need to look for special forms and macro calls, so we want to have just a single `elaborate` function which plays both roles. To that end, we took the `elaborate` shown in Figure 3, whose type corresponds to what `infer` would need, and changed it to also cover the needs of `check`. Its resulting type is:

\[
\text{elaborate} : \text{Ctx} \to \text{Sexp} \to \text{Maybe Ltype} \to \text{Pair Lexp Ltype};
\]

We can then trivially define `check` and `infer` on top of it:

\[
\begin{align*}
\text{infer } c \ s & = \text{elaborate } c \ s \ \text{nothing}; \\
\text{check } c \ s \ t & = \text{fst} (\text{elaborate } c \ s \ (\text{just } t));
\end{align*}
\]

4.4 Type inference

Bidirectional type checking is helpful to propagate existing type information, such as that given in top-level type annotations, but it is no substitute for real type inference. In order to provide
the same kind of experience as in other members of the ML family, Typer also uses a form of
Hindley-Milner (HM) unification-based type inference with let-polymorphism.

HM inference is defined in a much restricted language than Typer, so it required some adjust-
ments. For lack of a nice inference algorithm with principal types in ICC, Typer uses an ad-hoc
algorithm which tries to be simple enough to be understandable for the user and works about
as well as HM on those programs that fall into the HM subset.

HM performs specialization (i.e. introduction of implicit (type) applications) every time an
identifier is used, and every time an identifier is used it is fully specialized: all type arguments
are instantiated. In the context of Typer this is sometimes too eager and at the same time it is
insufficient: it can be too eager because an identifier can be passed to a function which expects
an erasable (“polymorphic”) function argument in which case the identifier should be passed
as-is without specializing it. And it can be insufficient because a normal function can return an
erasable function so specialization can be needed also at other places than where identifiers are
used. So, instead Typer performs specialization as a kind of coercion: whenever an expression
with erasable function type is used in a context which does not expect an erasable function,
that expression in specialized (i.e. Typer inserts a type application).

HM performs generalization (i.e. introduction of implicit (type) abstractions) whenever a value
is defined in a let-binding. Typer does the same when the variable had a type annotation, but
with the following difference: if a free meta-variable is used in a non-erasable way, we signal an
error since generalizing it with an erasable abstraction would lead to invalid code.

For definitions that come with a type annotation, Typer also provides a form of generalization:
first, when elaborating a type annotation, all remaining free meta variables are generalized into
erasable arrows, so the user can write:

```
map : (a => b) => List a => List b;
```

where we use the “?” prefix for user-written meta-variables, so Typer will add ?a and ?b as
two additional erasable arguments. This reproduces the same behavior as that used in systems
such as Twelf [PS99].

Second, when a λ abstraction is elaborated in a context that expects an erasable function, we
wrap it into an additional erasable λ. So if the previous type annotation is followed by:

```
map f x = . . .
```

the elaboration will automatically add the two additional (erasable) λ corresponding to ?a and
?b.

We believe this behaves just like HM inference for the corresponding sublanguage, but have not
shown it yet. Also we do not know whether this inference algorithm is guaranteed to terminate
in theory, but it seems to perform well in practice. Given that macro-expansion is allowed to
perform arbitrary side-effects, we have already given up the idea of guaranteeing termination
of elaboration anyway.

5 Core language

Typer’s core language is based on ICC* [BB08]. We start with a λ-calculus that is a sort of
pure type system (PTS) [Bar91] extended with annotations to indicate which arguments are
'n’ormal and which are ‘e’rasable:

\[
\begin{align*}
\text{(level)} & \quad \ell \in \mathbb{N} \\
\text{(var)} & \quad x, y, t \in \mathcal{V} \\
\text{(sort)} & \quad s ::= \text{Type } \ell \\
\text{(argkind)} & \quad k ::= n | e \\
\text{(exp)} & \quad e, \tau ::= s \mid x \mid (x:\tau_1) \rightarrow^k \tau_2 \mid \lambda x: \tau \rightarrow e \mid e_1 \oplus e_2 \\
\end{align*}
\]

Those annotations are similar to those of Bernardy et al.’s colored PTS (CPTS) [BJP12], in that the annotation on a function or function call has to match the annotation of the function’s type. The rules of the CPTS corresponding to Typer’s core calculus are the following:

\[
\begin{align*}
\mathcal{S} &= \{ \text{Type } \ell \mid \ell \in \mathbb{N} \} \\
\mathcal{A} &= \{ (\text{Type } \ell : \text{Type } (\ell + 1)) \mid \ell \in \mathbb{N} \} \\
\mathcal{R} &= \{ (e, s, \text{Type } \ell, \text{Type } \ell) ; s \in \mathcal{S}, \ell \in \mathbb{N} \} \\
&\quad \cup \{ (n, \text{Type } \ell_1, \text{Type } \ell_2, \text{Type } (\ell_1 \sqcup \ell_2)) \mid \ell_1, \ell_2 \in \mathbb{N} \} \\
\end{align*}
\]

Where \(\mathcal{S}\) is the set of possible sorts (i.e. types of types), \(\mathcal{A}\) is the set of axioms, and \(\mathcal{R}\) specifies the set of allowed abstractions: \((k, s_1, s_2, s_3)\) means that an arrow of color \(k\) can go from an argument in sort \(s_1\) to a result in sort \(s_2\), and that this arrow will live in sort \(s_3\).

Compared to a normal CPTS, we use a slightly different typing rule for erasable functions:

\[
\frac{\Gamma \vdash \tau_1 : s \quad \Gamma, x : \tau_1 \vdash e : \tau_2 \quad x \not\in \text{fv}(e^*)}{\Gamma \vdash \lambda x : \tau_1 \rightarrow^k e : (x : \tau_1) \rightarrow^k \tau_2}
\]

The difference compared to the rule for normal functions is the addition of the test that \(x\) doesn’t appear in \(e^*\) which is the erasure of \(e\). The erasure function \((\cdot)^*\) erases type annotations as well as all erasable arguments:

\[
\begin{align*}
\text{\(s^* = s\)} & \\
\text{\(x^* = x\)} & \\
\text{\(((x:\tau_1) \rightarrow^k \tau_2)^* = (x:\tau_1^*) \rightarrow^k \tau_2^*\)} & \\
\text{\((\lambda x: \tau \rightarrow e)^* = \lambda x \rightarrow e^*\)} & \\
\text{\((\lambda x: \tau \rightarrow e)^* = e^*\)} & \\
\text{\((e_1 \oplus e_2)^* = e_1^* \oplus e_2^*\)} & \\
\text{\((e_1 \oplus e_2)^* = e_1^*\)} & \\
\end{align*}
\]

This expresses the fact that erasable arguments do not influence evaluation. So far, this is exactly like ICC*. But Typer extends this with impredicativity and with inductive types.

### 5.1 Inductive types

Rather than decompose inductive types into separate unions and products as suggested by Bernardo in [Ber09], Typer keeps inductive types as a “hardcoded” combination of a sum of products:

\[
\begin{align*}
\text{(label)} & \quad l \quad \in \quad \mathcal{L} \\
\text{(exp)} & \quad e, t ::= \ldots \mid \text{ind } l \mapsto x : \ell \tau \mid \text{con}(e, l) \mid \text{case } e \text{ in } l \ y \ x^k \Rightarrow e_l \mid \text{fix } x : \tau = \ell \text{ in } e
\end{align*}
\]
This mostly follows the approach used in Giménez [Gim94] but with the following differences: constructors are selected by labels instead of by position; fields can be annotated as erasable; inductive types cannot have indices; in \texttt{ind} each alternative is specified by the list of its field types rather than by the curried type of the constructor; \texttt{ind} does not take a variable \( x \) to be able to refer to itself, because we let \texttt{fix} play this role instead; \texttt{case} does not specify its own return type.

5.2 Equality type

An important pragmatic issue with inductive types and GADTs is how to provide type refinement in the branches of the \texttt{case} statement. Following the de Bruijn principle we want our core language’s typing rule for \texttt{case} to be fairly primitive and leave it up to the elaboration phase to provide a more transparent form of refinement.

The careful reader has probably noted how we solved this problem: our inductive types simply cannot have indices, in other words our inductive types are like plain old algebraic data types rather than GADTs, so there is simply no refinement to be had in \texttt{case} branches.

This means that Typer’s core language cannot directly express the obligatory length-indexed list type:

\[
\begin{align*}
\text{type } \text{NList} & (a : \text{Type}) : (n : \text{Nat}) \to \text{Type} \\
& | \text{nil} : \text{NList} a \text{ zero} \\
& | \text{cons} a (\text{NList} a n) \to \text{NList} a (\text{succ} n)
\end{align*}
\]

To make up for it, Typer provides an additional built-in equality type “\texttt{Eq } e_1 e_2\texttt{ “}, with its customary \texttt{Eq.refl} constructor and \texttt{Eq.cast} eliminator, the eliminator encoding Leibniz equality. Armed with this equality type, Typer can now define the obligatory length-indexed list type as follows:

\[
\begin{align*}
\text{type } \text{NList} & (a : \text{Type}) (n : \text{Nat}) \\
& | \text{nil} (P ::\text{ Eq } n \text{ zero}) \\
& | \text{cons} a (\text{NList} a n') (P ::\text{ Eq } (\text{succ} n') n)
\end{align*}
\]

where ::\texttt{ is used to denote an erasable field. The two ways to define such a type are basically equivalent, and while the approach we chose was fairly common back when GADTs started to appear (e.g. in [SP04]), it is the exception rather than the norm nowadays. The factors that made us choose this approach are the following:

- It eliminates the subtle distinction between parameters and indices to inductive types:
  Notice how in the first definition of \texttt{NList} above, the two arguments \( a \) and \( n \) are presented differently because \( a \) is a parameter while \( n \) is an index. Here, it’s pretty clear which argument will be a parameter and which an index, but this is not always the case. Furthermore the typing rules can be slightly different for the two cases in terms of universe constraints.
• As mentioned, it eliminates the need for \texttt{case} to provide a form of refinement for type indices, simplifying the typing rule of the \texttt{case} construct as well as its syntax since there is no need for the syntax of \texttt{case} to specify its return type.

• Instead, the constructors directly carry the equality witnesses we need to implement the same refinement.

This last point means that when we perform a \texttt{case} on an object of type \texttt{NList} defined as above:

```
case (e : NList a n)
| nil => <en>
| cons x xs => <ec>
```

the branch \texttt{en} receives an (erasable and by default invisible) witness that “\texttt{Eq n zero}”. We can then use it to adjust our return type according to the needed refinement. Compared to Coq’s \texttt{match} construct, this also eliminates the need to resort to the \texttt{convoy pattern} [Chl13] when this equality proof is needed for other reasons than to refine the return type of the branch.

In order for the return type of \texttt{case} to be able to depend on the value of the object analyzed, each branch additionally receives another equality witness. For instance the branch \texttt{en} above also receives an (erasable and invisible) witness that “\texttt{Eq e nil}”.

Erasable arguments are usually not visible in the source code: they are “invisible” (or anonymous) variables. But they are very much present in the core language and the elaboration phase can make use of them just like normal variables. When needed, the source code can also make them visible, for example the pattern \texttt{nil (P := iszero)} would let \texttt{iszero} refer to the (still erasable) proof that “\texttt{Eq n zero}”.

5.3 Erasable impredicative arguments

The other important difference between Typer and ICC* is the treatment of impredicativity. ICC* follows the approach of making every universe predicative except for the bottom universe, called \texttt{Prop} or \texttt{Set}, which is impredicative. Usually \texttt{Prop} corresponds to an impredicative universe that can be erased during program extraction and \texttt{Set} is its non-erasable counterpart.

We hope that the erasability of \texttt{Prop} would be somewhat redundant with ICC*'s own notion of erasability, so we did not want to distinguish \texttt{Set} from \texttt{Prop}. Also, while the typing rules of the calculus of constructions are not made more complex by \texttt{Prop} and \texttt{Set}'s impredicativity, the same is not true in the presence of inductive types where soundness requires disallowing strong elimination on large inductive types.

So we decided to introduce impredicativity differently: Typer does not come with an impredicative universe like \texttt{Set} or \texttt{Prop}, and instead it lets its erasable functions be impredicative. To see what this means, let’s consider the rules of the colored pure type system of ICC*:

\[
\mathcal{S} = \{ \texttt{Prop; Type } \ell | \ell \in \mathbb{N} \} \\
\mathcal{A} = \{ (\texttt{Prop : Type } 0); (\texttt{Type } \ell : \texttt{Type } (\ell + 1)) | \ell \in \mathbb{N} \} \\
\mathcal{R} = \{ (k, \texttt{Prop}, s, s); (k, s, \texttt{Prop}, \texttt{Prop}) | s \in \mathcal{S} \} \\
\cup \{ (k, \texttt{Type } \ell_1, \texttt{Type } \ell_2, \texttt{Type } (\ell_1 \sqcup \ell_2)) | \ell_1, \ell_2 \in \mathbb{N} \}
\]

The \((k, s, \texttt{Prop}, \texttt{Prop})\) is the relevant rule above that allows impredicativity for \texttt{Prop}. One can divide this rule into two: \((e, s, \texttt{Prop}, \texttt{Prop})\) and \((n, s, \texttt{Prop}, \texttt{Prop})\). if you consider Typer’s \texttt{Type}_0 as ICC's \texttt{Prop}, the first rule is included in Typer but not the second. As it turns out, the second, is actually redundant in ICC*. More specifically:
Lemma 5.1 (Erasability of impredicative arguments). In ICC*, if $\Gamma \vdash x : \tau_e : \text{Type } \ell$ and $\Gamma \vdash e : \tau_e : \text{Prop}$, then $x$ can only appear in $e*$ within arguments to functions of type $(y:\tau_1) \xrightarrow{k} \tau_2$ where $\tau_2 : \text{Prop}$ and $\tau_1 : \text{Type } \omega$.

Proof. By induction on the derivation of $\Gamma_e \vdash e : \tau_e$. Since $\tau_e : \text{Prop}$, clearly $e$ can neither be a sort nor an arrow type and it cannot be $x$ itself either, so it can only be either a $\lambda y:\tau_y \to e_y$ or an application $e_1 e_2$. We can apply the induction hypothesis to $e_y$ and to $e_1$. As for $e_2$, there are two cases: either $e_1$ takes an argument of type $\tau_1 : \text{Type } \ell'$ in which case we’re done, or it takes an argument of type $\tau_1 : \text{Prop}$ in which case we can again apply the induction hypothesis.

Corollary: ICC*'s rule $(n,s,\text{Prop},\text{Prop})$ is redundant since we could convert all the impredicative functions that use it to functions that use $(e,s,\text{Prop},\text{Prop})$ instead.

5.4 Strong elimination of large inductive types

If we extend ICC* with Coq-style inductive types, Lemma 5.1 does not hold any more because we can perform a case analysis on an argument in universe $\text{Type } \ell$ and return something in universe $\text{Prop}$. For this reason, while the restriction of impredicativity to erasable functions does not make Typer weaker than ICC* it does make it in this respect weaker than CIC. But Typer is incomparable to CIC because in another respect it allows things that CIC does not.

As mentioned before, CIC has a special restriction that large inductive types (i.e. inductive types that belong to a universe that is smaller than some of the values it carries) cannot be used in a strong elimination (i.e. a case analysis that returns a type in a universe larger than that of the object analyzed).

This restriction means for example that while we can define in Coq a large inductive type like:

```coq
Inductive Ω : Set :=
| int : Ω
| arw : Ω -> Ω -> Ω
| all : forall k:Set, (k -> Ω) -> Ω.
```

we cannot prove properties such as the following (which we needed while working on [Mon07]):

```coq
forall K_1 K_2 F_1 F_2 P, all K_1 F_1 = all K_2 F_2 -> P K_1 F_1 -> P K_2 F_2.
```

This important restriction significantly reduces the applicability of large inductive types, but is needed because it would be otherwise possible to “smuggle” a large element within an inductive object of a smaller universe and take it back out later, resulting in unsoundness [Coq86].

Since Typer’s impredicativity is limited to erasable elements, those large elements cannot really be taken back out later anyway, by virtue of their erasability. For this reason, we conjecture that our form of impredicativity does not require this restriction on strong elimination. As a consequence, in Typer we can define the above inductive type (with an erasable $k$) and prove its property (again with erasable $K_1$ and $K_2$).

The weak justification behind it, is a philosophical one: erasable arguments are not significant, so a function that takes an erasable argument could be considered as a mere “schema” or “prototype” which stands for all the specialized versions of the function. A similar argument is discussed by Fruchart and Longo in [FL96].
6 Related work

Like all languages, Typer takes inspiration from too many of its predecessors to be able to list them all. We will try and limit ourselves to some recent systems which share enough of their design or their goals here.

**Honu** [RF12] is a programming language in the Racket system which provides an extensible infix/mixfix syntax integrated with Racket’s metaprogramming facilities. **Typed Racket** [THSAC+11] uses an extension of Scheme’s macro system to implement a statically typed variant of Racket as a sort of embedded DSL, thus implementing the type checker as part of a macro. It shares with Typer the characteristic of mixing Lisp-style macros and static typing, and generally the Racket system shares with Typer the goal of a being a “language workbench” on top of which other languages can easily be defined, Typed Racket and Honu being just some examples. The **Star** language [MS13] is a statically typed programming language which also makes it easy to define embedded DSLs via syntactic and macro expansion facilities. **Scala** also provides sophisticated meta programming [Bur13] and staged computation [RSA+13] facilities used in novel ways. **OCaml** offers extensible syntax and metaprogramming facilities in various forms, such as via its Camlp4 system [dR03] and more recently with extension points, which work like macros, by mapping OCaml AST to OCaml AST.

**Template Haskell** [SP02] is an extension of Haskell to allow compile time metaprogramming. One of the main contribution of Template Haskell is to implement a metaprogramming system on top of a strongly typed purely functional language. Typer’s interleaving of type inference and macro expansion is very similar to that of Template Haskell. But Typer and Template Haskell differ on how the macros are used by the programmer: in Template Haskell, macro calls are made explicit in the source file by preceding them with a ‘$’ sign rather than being determined by their type. Also Template Haskell is not meant to add new binding forms to the language: arguments to the macro are type checked before being passed to the macro.

**Idris** [Bra13] and **F-Star** [SHK+16] are programming languages with dependent types. They shares many of Typer’s goals and also offers metaprogramming facilities, although these facilities are more aimed at writing proofs, while Typer’s metaprogramming facilities are more currently geared toward syntactic extensions. **Zombie** [CSW14] is an experimental programming language with dependent types. One of its most novel features is to eschew automatic reductions at the type level and require manual cast operations instead. This is a bit like of Typer’s intentionally weak typing rule for case, relying on explicit cast operations using type equality witnesses for type refinement, but pushed yet further.

**Agda** [BDN09] is a proof assistant with a syntax similar to Haskell’s but with the possibility of adding mixfix and not just infix operators. Their use of mixfix operators like if,then,else, as a way to add new syntactic forms is what gave us the idea of adding mixfix to S-expressions in Typer using operator precedence grammar. For a more detailed and formal discussion on mixfix operators and Agda, see [DN08]. **Coq** [HPM+00] has syntactic extensions similar to mixfix as well as a sophisticated metaprogramming language known as Ltac [Del00]. More recently other metaprogramming languages have been designed for it such as Mtac [ZDK+13] and Rtac [MB16].
7 Conclusion

Typer is a new experimental language in the family of dependently typed functional programming languages. Its design is generally conservative in that it mostly uses existing solutions, but tries to streamline them and combine them in ways which hopefully simplify the overall system while making it more flexible at the same time.

While it has not been officially released yet, its code can be found at https://gitlab.com/monnier/typer.

Acknowledgments

This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) grant No 298311/2012. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the NSERC.

References


