Travail pratique

IFT-6172

October 5, 2023

1 Overview

In this project, you will extend an existing language with an extension of your choice. The existing language on which you will build your project is a simplified version of the experimental language Typer, a mix of Coq and Lisp.

2 Typer

The implementation of Typer is written in OCaml; it is structured as follows:

1. Lexical analysis.
2. Syntactic analysis, which returns an abstract syntax tree of type `sexp` (a variant of Lisp's S-expressions).
3. Elaboration, which transforms the `sexp` into a `lexp` ("lambda-expression"), which is the "core language" of Typer, a Pure Type System (PTS) with a few extensions.
4. Type checking.
5. Erasure, takes a `lexp` and transforms it into a simplified form called `elexp` where type annotations have been erased.
6. `elexp` interpreter.

2.1 Lexical Analysis

The lexical analysis is very simple: it recognizes comments, strings, numbers, and everything else is split into "symbols" separated by whitespace, with the exception of a few characters which are their own symbols.

The only parameter that you may want to change is the map `default_stt`, defined in `src/grammar.ml` which indicates which characters are their own symbols. Currently these are the parentheses, the comma, and the semi-colon. For example `a+2:` is considered as 1 symbol.

2.2 Syntactic analysis

The syntactic analysis is also very simple, based on *operator precedence grammars* (OPG), a very restrictive class of grammars. It is parameterized by a grammar in the form of a map giving the precedence levels of each "keyword", where the default table is `default_grammar` which is found in `src/grammar.ml`. Note that this table actually comes from `typer-smie-grammar`, in `emacs/typer-mode.el` (where it's also used by the Emacs mode to aid in navigation and auto-indentation of the code), which is generated mechanically from a more-or-less BNF representation.

It is used to allow the use of infix notation, but the programmer can also use a Lisp-style prefix notation, which is 100% équivalent. E.g. When the programmer writes:

```
type List (a : Type)
 | nil
 | cons a (List a)
```

the result is the same as if he had written:

```
type_ (_|_ (List (_:_ a Type))
           nil
           (cons a (List a)))
```

This is because the grammar gives to `type` a left precedence of `None`, which indicates that it is a prefix keyword, whereas `|` has the same left and right precedence, which indicates that is infix and that it combines with itself (i.e. `A | B | C` gives `_|_ A B C` rather than `_|_ A (_|_ B C)` or `+_|_ (_|_ A B) C`).

The result of the analysis is a data structure of type `sexp`, defined in dans `src/sexp.ml`:

```
type sexp =
  | Block of location * pretoken list * location
  | Symbol of symbol
  | String of location * string
  | Integer of location * integer
  | Float of location * float
  | Node of sexp * sexp list
```

Note that at this stage there is no notion of semantic, it's just a tree of symbols; this part of the code does not know what is a function, a function call, a type definition, ... [1]

[1] The `Block` elements will likely not be important for your project, but if you need to know, they correspond to blocks of text delimited by braces: they are kept "as is" without performing syntactic analysis (this makes it possible for macros to later perform syntactic analysis on them with a grammar of their choice).

2.3 Elaboration

This is the heart of Typer, which can be found in `src/elab.ml`. It takes a `sexp` and transforms it into a lambda-expression `lexp` (defined in `src/lexp.ml`), which requires propagating type information (without really performing type inference, but rather a bi-directionnel propagation) as well as taking care of macro expansion.

```
type ltype = lexp
 and lexp =
   | Imm of sexp (* Used for strings, ...  *)
   | Sort of U.location
   | Builtin of symbol * ltype
   | Var of vref
   | Susp of lexp * subst
   | Let of U.location * (vname * lexp * ltype) list * lexp
   | Arrow of vname * ltype * U.location * ltype
   | Lambda of vname * ltype * lexp
   | Call of lexp * lexp list
   | Inductive of U.location * ((vname * ltype) list) SMap.t
   | Cons of lexp * symbol
   | Case of U.location * lexp
             * ltype
             * (U.location * vname list * lexp) SMap.t
             * (vname * lexp) option
```

The type `U.location` keeps track of source code position information. The constructors `Arrow`, `Lambda`, and `Call` correspond respectively to the type, constructor and elimination forms of fonctions (note that `Call` is *curried*: the functions only take a single argument at a time).

The `Let` describes a list of definitions, which can be mutually recursive; This also allows defining types and type-level functions, without any of the restrictions which could guarantee some form of normalization. So we cannot use this language as a logic (it would be inconsistent) and type checking is not decidable but that does not bother us.

The `Imm` corresponds to "imm"ediate constants like numbers or strings; the `Builtin` is a reference to a function or type implemented in the OCaml code. `Sort` is the *sort* in the PTS sens, where we have a single sort called `Type`, which is its own type.

The `Susp(e,s)` is a "suspended substitution", i.e. a `lexp e` where we still need to apply the substitution `s` (we do that to apply substitutions more lazily). Every time you encounter such a term, you should pass it to `push_susp e s` which will return the expression hidden behind this suspension.

The `Var` is of course a variable reference. A `vref` (defined in `src/util.ml`) is made up of two parts:

```
type vref = (location * string list) * db_index
```

The first part holds the name(s) of the variable and the second is an integer: it is the *De Bruijn index* which is the position of the variable in the context.

¡IMPORTANT! the name is not significant, only the index is: the name of variables in `lexp` is always optional and only serves for debugging purposes and to print more understandable error messages. Your code cannot/should not not trust variable names to do its job (e.g., never search variable by name in the context, always use its de Bruijn index instead).

2.3.1 Algebraic data types

`Inductive` describes an algebraic data type: the `Smap` is a map indexed by the name of the constructor which indicates the list of arguments of the constructor.

`Cons` is a reference to a particular constructor of an algebraic data type: its first argument is the algebraic data type and the second is the name of the constructor to which it refers.

The type declaration:

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

is really a call to the macro `type_` which transforms this code into:

```
List : (a : Type) -> Type;
List (a : Type) = typecons nil (cons a (List a));
nil (a : Type) = datacons (List a) nil;
cons (a : Type) = datacons (List a) cons;
```

where `typecons` translates directly to an `Inductive` and `datacons` translates to a `Cons`.

The `Case(l, target, ret, branches, default)` corresponds of course to a case analysis term, where `target` is the expression that we want to analyse, `ret` is the return type (hence also the type of every branch), `branches` is a table mapping each constructor name to its corresponding code, and `default` is a default branch (for the case where `branches` does not cover all the possible constructors).

2.3.2 Macros

When the elaborator encounters an expression $\texttt{e}_1$ $\texttt{e}_2$ $\texttt{e}_3$ `...` which looks like a function call but where $\texttt{e}_1$ is an expression of type `Macro`, it is a macro invocation, and the elaborator then calls the function contained in the $\texttt{e}_1$ object, passing to it the `sexp`s provided as arguments $\texttt{e}_2$ $\texttt{e}_3$ `...`, which then returns the expansion in the form of a new `sexp` on which elaboration continues. In other words, this works very similarly to the `defmacro` of Lisp.

2.4 Type checking

Elaboration has to propagate types and thus basically has to perform the job of checking types. Yet, it is a relatively complex phase that we would rather not trust too much. So after elaboration we pass the code to `check` , defined in `src/opslexp.ml`. This function tries to check the code as thoroughly and simply as possible, trying to stay as close as possible to the theoretical presentation of the typing rules, to minimize the risk of errors.

2.5 Erasure

This phase is very simple and simply erases the type annotations that the interpreter does not need. It is implemented in `erase_type`, defined in `src/opslexp.ml`. Since types can be manipulated like values, it can happen that some types still remain in this phase, but they don't "do" anything any more, so we only keep them because it's easier them to remove them and they can be printed sometimes for debugging purposes. The result is of type `elexp` ("erased" lexp), defined in `src/elexp.ml`.

2.6 Interpreter

Finally the code is executed by a straightforward interpreter implemented in `src/eval.ml`. That's also where the bulk of the builtin primitives are defined.

2.7 Differences with Typer

Do not confuse Typer with Typer: the one you use for this cours is a bit simpler than the official Typer. You will probably encounter references to functionalities of the official Typer along the way, so here's a list of those that have been removed:

- Implicit and erasable arguments, which use functions with arrows of the form `=>` et `≡>`.
- Type inference.
- Monads: the official Typer is a pure functional language, which relies on monads to confine side effects, whereas the Typer you're using is not pure, so it is more like OCaml than like Haskell.
- Universe polymorphism.
- The equality type (which allows the use of *cast*).
- A (very primitive) module system.

3 The assignment

This project can be done alone or in groups of two. It is decomposed into three stages:

1. Choosing an extension.
2. Design of the extension.
3. Implementation.

3.1 Choice

You are free to choose whichever extension you'd like to add to Typer, regardless of other people's choices. Of course, it has to be an extension to the language and not just its implementation (this is not a compilation course; we're not interested in generating more efficient code, for example).

Examples of extensions:

- Add exceptions.
- Add automatic coercions.
- Add a notion of subtyping.
- Add a class-based object system.
- Add a concept of aspect.
- Add a notion of linear types or of *ownership* types.
- Gradual typing.
- Hygienic macros.
- Delimited continuations.
- A module system.
- Type classes.
- ...

At this stage, you will just need a brief description (maximum 1 page) of what the extension you're considering could look like, with some examples of code and a brief description of their intended meaning. This will let me give you feedback if your extension is sufficient (or too ambitious) and to give you some directives to try and adjust the complexity.

Beware: adding new functionality to an existing language is much more difficult than it seems, so don't aim too high.

3.2 Design

This is the most important stage. You will have to hand in a report in LaTeX(source code, not PDF or something else, maximum 5 pages) which shows how your extension integrates with the rest of the Typer language, with a formal description of the added (or modified) syntactic elements, their static and dynamic semantics (i.e. typing rules and reduction rules), as well as a description of changes needed to the pre-existing rules (if needed). This formal presentation should be accompanied with a few examples where it will be very important to explain what those examples are expected to do.

It is possible/likely (depending on your choices) that it be difficult to integrate your extension in a fully satisfactory way. So an important part of the design will be to be aware of the limitations and to mention them in the report.

Although this stage does not include any code, at that point of the work you should obviously already be working on the code, since it's also while coding that you will encounter problems that will shape your design.

3.3 Implementation

In this final stage you have to hand in the code (in the form of a *patch*) as well as a report (again in LaTeX, maximum 10 pages) which describes the final design (potentially identical to that of the previous stage) as well as the approach, the choices, and the limitations of the implementation.

Adding such extensions to an implementation like that of Typer can be challenging, so your implementation will likely be a limited prototype which only covers the core features. It's normal, but it's important to clearly describe those limitations in the report.

Please write your code cleanly, properly indented, do not use more than 80 columns, ...

4 Grade

This project counts for 50% of the final grade. Those 50% are decomposed as follows: 5% for the choice, 20% for the design, 10% for the final report and 15% for the implementation.