EasyOCaml: Concepts, Implementation and Teachnicallities

Benus Becker[*]

Summer 2008

Contents

[*]Institut fur Informatik, Abteilung Programmiersprachen, Uni Freiburg

1 Objectives of EasyOCaml and Introduction

OCaml is a programming language which unifies functional, imperative and object oriented concepts in a ML-like language with a powerful and sound type system. Its main implementation (http://caml.inria.fr) ships with an platform independend byte code compiler and an effient machine code compiler and there are a lot of existing libraries, which make really multi purpose programming language.

But by now, OCaml is not a language well suited for *teaching programming*: Foremost, it features a complex type system but type errors, for example, are reported only by a message and a single location in the code without hints on reasons for it (This is mainly because of used algorithm which is very efficient, though). The objectives of this work are, in large, to make OCaml a programming language better suited for beginners and teaching programming. We

- improved OCaml's error messages by providing a modified parser and a new type checker.
- equiped OCaml with an infrastructure to make it adaptable for teaching programming, in meams of restricting the supported features of the language, or providing code and the startup environment in a simple way of distribution (language levels).
- we integrated all that into OCaml's original REPL and compiler system to take advantage on existing libraries and OCaml's code generation facilities.

1.1 Supported Lanugage

EasyOCaml supports a strict subset of OCaml, namely Caml without module declarations (called "Caml-*m*"). Take a look into the file easyocaml-features.pdf for more details on the supported features.

1.2 Similar Projects

There are some projects which heavily infuenced our work:

Haack & Wells have decribed and implemented a technique to produce more descriptive type error messages in a subset of SML. Their work is seminal for constraint based type checking with attention on good error reporting.

Helium is a system for teaching progamming in Haskell. In a similar manner, type checking is done via constraint solving. Furthermore, it features detailed error messages including hints how to fix certain errors (based on heuristics).

Finally, *DrScheme* is a programming environment for the Scheme language that is build for teaching programming. It has introduced the concept of language levels and teachpacks to restrict the syntax and broaden functionality espacially for excercises.

This report has three targets: Firstly, to present EasyOCaml and the concepts used and developed for it (audience: users) through section 2 – 4. Secondly, it combines these concepts with the actual implementation and describes its architecture in code (audience: developer) in section 5. Thirdly, formal foundations for the type checker are given the appendix (audience: the interested reader) in section 6.

2 Constraint based type inference

The type inference currently used by OCaml is a Damas & Milner algorithm in its core. Although it is very efficient and broadly extended to OCaml's requirements, it lacks memory: For examples, type inference for variables works by accumulating (unifying) information on their usage while traversing the AST. Broadly spoken, a type constructor clash is detected as the usage just inspected contradicts the information gathered so far. Therefore, OCaml's type checker cannot report any *reasons* for a type error but reports only the location where the error became obvious to the type checker. An important work while debugging type errors in OCaml persists of manually searching other usages of the mis-typed variable in the program which lead to the type constructor clash.

2.1 Haack & Wells

Haack & Wells described an algorithm which exceeds a Damas & Milner-style type inference (algorithm T) engine in two ways: Firstly, every type error reported contains information on exactly those locations in the program which are essential to the error, by means of dropping it would vanish the error. Secondly, it reports all type errors in a program at once (whilst locations which partake in several type errors are most notable a source of the errors, by the way).

In a sense, algorithm T does two things at once while traversing the AST; it generates type information on the variables and unifies it with existing type information anon. Haack & Wells' algorithm achieves its goals by seperating these steps:

Constraint generation Every node of the AST is annotated with a type variable. While traversing the AST, information on those type variables are collected from the usage and context of each node. The information is stored as a set of constraints on the type variables.

The intention is the following: If those constraints are unifiable, the resulting substitution represents a valid typing of the program with respect to the nodes' type variables. Otherwise, the program has at least one type error and the representation of the gathered type information as a set of constraints enables the algorithm to reexamine the errors in a second stage:

Error enumeration and minization ...

2.2 Extensions for EasyOCaml

See section 6 for a complete list of inference rules used in EasyOCaml.

3 Errors and Error Reporting Adaptibility

EasyOCaml is essentially build for usage in programming teachings. As this, special attention is paid to the way it reports its errors.

Firstly, errors should provide a *right* amount of details, too less information is of course insufficient, but too much information can be confusing, too. So, for example in type constructor clashes exactly those locations are reported, which are essential to the error. Delivering more information on the reasons of type errors is EasyOCaml's type checker is exactly what it is made for.

Secondly, error reporting should be adaptable: For a beginner, reading errors in a foreign language can distract or even prevent him from fixing an error. Furthermore, the error output should be adaptable in its overall structure to serve as the input for different kinds of presentation, e.g. plain text on command line or HTML for displaying it visually in a web browser.

3.1 New Errors for Camlp4

As mentioned, EasyOCaml parses its input with a Camlp4 parser. Unfortunatly, Camlp4's error messages are hard coded in the parser's code in english and never represented in data. The reason is that Camlp4 is a OCaml stream parser in its core, and this requires parsing errors to be reported as `Stream.Error of string` exceptions.

Nevertheless, we furnished Camlp4 with a new error reporting system, up to now just to make error reporting adaptable, but it should be possible now to extend the information of parse errors by the parser's current state.

`Expected (entry, opt_before, context)` is raised if when the parser stucks while parsing a phrase: `entry` describes the category of the subphrase to be expected, `opt_before` might describe the category of the entry just parsed and `context` denotes the category of the phrase which is just parsed.

`Illegal_begin sym` is raised when the parser is not able to parse the top category `sym`.

`Failed` is raised only in `Camlp4.Struct.Grammar.Fold`.

`Specific_error err` Beside the generic parsing errors just mentioned, it is possible to extend the parsing errors per language by "artificial" errors which are specific to a language, e.g. `Currified constructor` in OCaml wich is not represented in the grammar but checked in code. (further errors for EasyOCaml are specified in `Camlp4.Sig.OCamlSpecificError`.)

Now, how are these errors represented in the string information of the `Stream.Error`? Not without a hack, which is luckily hidden behind the Camlp4 interface: Internally, camlp4 throws `Stream.Error` exceptions but the string has the following format: "<`msg`>\000<`mrsh`>" where <`msg`> is the usual Camlp4 error message and <`mrsh`> is a marshalled parsing error as just described. The string of a `Stream.Error` is decomposed again in the interface function for parsing (namely `Camlp4.Struct.Grammar.Entry.action_parse`), and reported as a `ParseError` to the user.

3.2 Adaptibility

For internationalization of error messages and different structure of error messages for different settings, EasyOCaml provides adaptability of error messages by a plugin system. Error reporting plugins should use `EzyError`'s internationalized functions to output the text describing the error message (`EzyErrors.print*_desc`) to keep them uniform but can print it any structure: Currently, a plain text format is default but a HTML printer which highlights the locations of an error in source code and a XML/Sexp printer for usage in an IDE are provided.

The user can register an error printer via the command line flag `-easyerrorprinter`. The module is dynamically linked and registers itself in `EzyErrors` where it overwrites the appropriate functions.

4 Language Levels and Teachpacks

Language levels are a facility to describe the initial state of the EasyOCaml compiler or toplevel system in means of the environment which is accessible to the user and the available syntax. Language levels are useful for teaching programming, as they can be designed just for specific exercises – probably providing an easy interface to some advanced API and restrictions on the syntactic elements tought sofar, to not have syntax errors regarding unknown syntactic elements.

Here is in more detail, what language levels can define:

- The available syntactic features. One can specify the syntactic elements which are allowed for patterns, expressions, structure items and type declarations, in high detail. Currently, most of these restrictions are implemented by deleting the according entries from the grammar (thanks to the power of Camlp4!). However, some features like mandatory type annotations for toplevel values are checked afterwards while importing the AST into a type of `EzyAst`.
- Settings of path inclusion and object loading. Teachpacks can specify the directories which are included for searching objects, just like the `-I` command line flag. A teachpack can contain objects itself and the specification which are to load (just like putting them on the command line). Furthermore, teachpacks can specify which modules are opened on startup.

Teachpacks can specify the settings for path inclusion and object loading. Whereas only one language level can be loaded, teachpacks can extend a possible language level.

The user can specify which language level respectively teachpack to use by the `-easylevel` respectively `-easyteachpack` command line parameter. EasyOCaml then searches for it in the following directories:

There is a global and a user configuration directory. First, EasyOCaml searches the user then the global configuration directory. Here's how the global configuration directory is determined (in descending preference):

1. Environment variable EASYOCAML_GLOBAL_DIR
2. Compile-time option

Here's how the user configuration directory is determined (in descending preference):

1. Environment variable EASYOCAML_USER_DIR
2. $HOME/.easyocaml

Layout of the easyocaml configuration directory:

```
language-levels/level-1
                level-2
                ...
teachpacks/tp-1
           tp-2
           ...
```

The idea of teachpacks and language levels is taken from DrScheme. See http://docs.plt-scheme.org/drscheme/extending-drscheme.html for more information.

5 Details on the Implementation

5.1 Outline of EasyOCaml's Pipeline

Here is a rough outline of EasyOCaml's machinery (in both the compiler and the toploop):

1. Command line flags are evaluated to check the "`-easy`" flag and an error printer, load language level and/or teachpack might be loaded.

2. `EzyCamlgrammar` parses the AST from the input, possibly respecting restrictions from the language level which yields an `EzyAst.imported_structure`.

3. `EzyGenerate` generate constrains from the AST (involving type information from the default environment and modules loaded by command line or teachpacks/language levels). This yields a quadruple `generated_structure * AtConstrSet.t * PostProcess.t * EzyEnv.t` where

   `generated_structure` is the AST annotated with type variables and unique identifiers.

   `AtConstrSet.t` is a set of constraints on the type variables in the AST.

   `PostProcess.t` is build gradually while constraint generation and contains sets of different types of errors (from `EzyErrors`) as well as checks which can only processed after constraint unification (i.e. type annotations)

   `EzyEnv.t` is used to keep track of local variables and contains after constraint generation information on the global types and values of the program.

4. `EzyTyping.solve` tries to solve the generated constrains. If solving succeeds, the program is typed by a type substitution on the type variables in the generated AST and the environment and contains type errors otherwise.

5. The last step, reimporting the `EzyEnrichedAst.generated_structure` with typing information given by type substitution to OCaml's original Typedtree is not yet done. We type the code again with OCaml's original type checker and compare the result to verify its correctness.

The goal of this section is to describe roughly the modules implemented for EasyOCaml and locate functions for the EasyOCaml's steps in 1.

5.2 Utilities and Miscellaneous

Two rather independent modules for code used in EasyOCaml

EzyUtils Functionality which is not specific to EasyOCaml, but extends the standard library (String, Set, Map). It contains also code copies from existing Libraries (from Core: Option, Monad, T2, T3, T4, such that EasyOCaml adds no dependencies at bootstrap time) and new code for Logging and some more (lexical comparison, tools on functions).

EzyMisc EasyOCaml-specific code which is used at different locations in the project.

EzyOcamlmodules Extensions of the modules from the standard OCaml system (e.g. Location, Path, Longident, Types, ...) as well as sets and maps over these.

The rest of the modules contains the code for the EasyOCaml implementation:

5.3 Error Reporting

EasyOCaml offers sophisticated facilities to represent errors, to allow as detailed error reporting as possible. Furthermore, new error reporting plugins can be registered.

EzyErrorReportUtils Code for type error slicing (described in Haack & Wells), i.e. slicing an AST to only contain nodes from locations given in a set, substituting the rest with elipses.

EzyErrors Representation (types) of errors wich can occur in EasyOCaml, functions for pretty printing errors as well as functions for error reporting plugins to register themselfes.

5.4 Teachpacks and Language Levels

EzyConfig Constants of the teachpack system (e.g. the name of the module describing the teachpack or language level) and functions to find a teach pack or language level in the file system.

EzyDynload Superset of functionality for loading teachpacks and language levels (used by `EzyLang`)

EzyLang Functions for loading language levels and teachpacks (used by `EzySetup`)

EzyTeachpack Shortcut to `EzyFeatures` and registering of the teach pack. Actual teachpacks should only need to link against this module.

EzyLangLevel Shortcut to `EzyFeatures` and registering of the language level. Actual language levels should only need to link against this module.

EzySetup Process command line flags regarding language levels and teach packs and provide the actual setup of features, modules, included directories and object files given by teachpacks and language levels to other parts of EasyOCaml.

5.5 Abstract Syntax Tree

The following modules contain representation, manipulation, parsing and restrictions on EasyOCaml's AST.

EzyFeatures In EasyOCaml, the available syntax can be restricted. This module contains types to describe these restrictions and some functions to generate defaults (i.e. settings where everything is forbidden or allowed).

EzyAsttypes Adaption of Asttypes from the standard OCaml system.

EzyAst Representation of the AST in EasyOCaml. This is a bit overloaden as each node is parametrized on some data it contains. This is `unit` for a parsed tree and typing information (mainly the type variable) for a parsed tree after constraint generation.

EzyCamlgrammar The EasyOCaml Parser as a Camlp4 extension of `Camlp4OCamlParser`. It just deletes some entries in the latter (partially depending on the given features.

EzyEnrichedAst This module directly belongs to `EzyAst` but we had to outsource it because of module dependencies between `EzyErrors`. It contains

- definitions of the AST after constraint generation import functions from
- OCaml's standard Parsetree respecting given restrictions from `EzyFeatures`
- comparison of two ASTs which is used to compare OCaml's typing and EasyOCaml's typing afterwards

5.6 Type Constraints

EzyTypingCoreTypes Contains base types for the constraints and their generation, closely related to the data described in Haack & Wells (type variables, types, type substitutions, intersection types, type environments)

EzyConstraints Here are constraints annotated with only one location (`AtConstr.t`) and constraints with sets of locations (Constr.t) defined, as well as set and maps of those. Furthermore a derived environment as described in Haack & Wells is defined.

EzyGenerate There is a function for every syntactic category to generate constraints and/or errors.

5.7 Typing

EzyTyping Unification of constraint set which yield a substitution on the variables and error enumeration and minimization as described by Haack & Wells. It furthermore contains the typing functions for structures which are used in the compiler and toplevel.

EzyEnv The `EzyEnv.t` is the typing environment for EasyOCaml. Information on declared types and types of local and global variables is hold. It is build up while constraint generation (`EzyGenerate`) in combination with the type variable substitution resulting from `EzyTyping.solve`.

6 Typing Rules for EasyOCaml