EasyOCaml: Concepts, Implementation and Teachnicalities

Benus Becker, Stefan Wehr, Peter Thiemann*

Summer 2008

Contents

*Institut fur Informatik, Abteilung Programmiersprachen, Uni Freiburg

1 Objectives and Introduction

OCaml (Leroy et al., 2007) 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 a platform independent byte code compiler and an effient machine code compiler and there are a lot of existing libraries, which make it a real multi purpose programming language.

But up to now, OCaml is not a language well suited for *teaching programming*: Foremost, it has a complex type system. But type errors, for example, are reported only by a message and a single location in the code without giving any reasons for it (this is mainly due to the utilized 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 to teach programming. We achive this by

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

1.1 Supported Language

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 influenced our work:

Haack & Wells (2004) have described 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 and builds the foundation for EasyOCaml's type checker.

Helium (Heeren et al., 2003) 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 certain heuristics.

Finally, *DrScheme* (Felleisen et al., 1998) 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. These are included in EasyOCaml.

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

2 Constraint Based Type Inference

The type inference currently used by OCaml has the algorithm by Damas-Milner (1982)* at its core. Although it is very efficient and broadly extended to OCaml's requirements, it lacks a memory: For example, type inference for variables works by accumulating (unifying) information on their usages while traversing the AST. Broadly spoken, a type constructor clash is detected as the usage just inspected contradicts the information collected 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. Much work while debugging type errors in OCaml persists of manually searching for other usages of the mis-typed variable in the program which lead to the type constructor clash.

*??? *

2.1 Haack & Wells

Haack & Wells (2004) have described an algorithm which exceeds a Damas-Milner-style type inference (algorithm W) in two ways: Firstly, every type error report 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 are involved in several type errors are most notable the 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.

During *constraint generation*, every node of the AST gets annotated with a type variable. While traversing the AST, information on those type variables is 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 *unification* on those constraints succeeds, the resulting substitution represents a valid typing of the program with respect to the type variables of the nodes.

Otherwise, the program has at least one type error. But now, the collected type information is still available as a set of constraints and enables the algorithm to reexamine the errors in a second stage of *error enumeration and minimization*: Error enumeration is basically done by systematically removing constraints grounded at one program location from the constraint set and running unification again on the result. Haack & Wells also present an iterative version of this algorithm which is implemented in EasyOCaml. Although it avoids recomputation of the same errors over and over again, error enumeration has neverthess exponential time consumptions. Thus error enumeration delimited to a given time amount which can be specified by the environment variable `EASYOCAML_ENUM_TIMEOUT`.

The result of error enumeration is a set of errors each represented as a complete set of locations whose nodes in the AST have yielded to the error (*complete* in being a superset of the constraints which caused the type error). By application of error minimization on each error, the algorithm guarantees *minimality* of the reported errors in the sense that removing the constraints annotated with a location would vanish the error itself.

In addition to type errors, Haack & Wells technique also enables the type checker to collect unbound variables in the program while generating. Their types are assumed as a free variable to avoid a type error, but unbound variables are reported with the type errors after error enumeration.

2.2 Extensions for EasyOCaml

Haack & Wells' description comes with constraint generation rules only for variables, infix operations, functional abstraction, application and local polymorphic variable bindings. Although this is a good starting point to describe the involved algorithms, we had to extend it to the Caml-m.

- records ...
- variants ...
- type annotations ...

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

* *+++

3 Errors and Error Reporting Adaptibility

EasyOCaml is essentially build for teaching programming . Therefore, special attention is paid to the way errors are reported.

Firstly, errors should provide a *right* amount of details, too few information is of course insufficient, but also too much information can be confusing. 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 exactly what EasyOCaml's type checker 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 visually display it in a web browser.

The last section has explained the improvements of EasyOCaml to type error messages. The following will describe improvements to the parser.

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 supplied Camlp4 with a new error reporting system, up to now just to make error reporting adaptable, but it should be possible now to augment the information of parse errors by the parser's current state. Now, a parsing error `ParseError.t` is one of the following:

`Expected (entry, opt_before, context)` is raised if when the parser stucks while parsing a phrase: `entry` describe the categories of the possible, expected subphrases, `opt_before` might describe the category of the entry just parsed and `context` denotes the category of the phrase currently parsed.

`Illegal_begin sym` is raised when the parser is not able to parse the top categories described by `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, which 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 interface of Camlp4: 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 again decomposed in the interface function for parsing (namely `Camlp4.Struct.Grammar.Entry.action_parse`), and reported as a `ParseError.t` to the user.

3.2 Adaptibility

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

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

The following section describes the tools, EasyOCaml provides specifically for teaching programming.

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 so far, to avoid 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 to EasyOCaml's AST.

Settings of *path inclusion* and *object loading and opening*. 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 have to be loaded (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 and teachpack to use by the `-easylevel` and `-easyteachpack` command line parameter respectivly. 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
           ...
```

Each language level and teachpack contains a module `LANG_META` which is loaded into EasyOCaml.

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 of the Implementation

5.1 Outline of EasyOCaml's Pipeline

Here is a rough outline of EasyOCaml's pipeline which is quite similar for both the compiler and the toploop:

1. Command line flags are evaluated to check the "`-easy`" flag and an error printer and possibly load a teachpack and/or language levels.

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

3. `EzyGenerate` generates constraints 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 during 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 after constraint generation contains 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 substitution on the 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