Institut für Informatik
Abteilung für Programmiersprachen
Universität Freiburg

Studienarbeit

EasyOCaml: Making OCaml More Pleasant

Concepts, Implementation and Technicalities

Benus Becker

Summer 2008, et seq.

Abstract

OCaml is not a language well suited for beginners or teaching programming, because error messages are sometimes hard to understand without some practice. *EasyOCaml* equips the *OCaml* system with a new type checker for a reasonable subset of the *OCaml* language and an adapted parser to make error messages more descriptive. Plugins for reporting errors can be loaded at runtime, to produce output for different settings. Furthermore, *EasyOCaml* adds language levels and teach packs (similar to those of *DrScheme*) to make the language a good choice for teaching programming. All this is integrated in the original *OCaml* system.

This report describes the ideas and formal and notional foundations of *EasyOCaml* as well as its implementation as a manual for the user and guide to further development.

Contents

1 Objectives and Introduction

Objective Caml (Leroy et al., 2008) 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 efficient machine code compiler and there are a lot of libraries, which make it a great multi purpose programming language.

But up to now, *OCaml* is not a language well suited for *learning* and *teaching programming*: It has a very rich type system, but type errors are reported only with few information on the underlying reasons. Some practice is necessary to manage these. On the other hand, OCaml comes with tools (e.g. Stolpmann's *findlib*) which make it easy to handle libraries for developers, but it lacks a fool-proof system to use primed code in programming lessons.

The objectives of this work are, in large, to make OCaml a programming language better suited for beginners and to teach programming. We achieve this by

- improving *OCaml*'s error messages by providing a modified parser and a new type checker.
- equipping *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 toploop and compiler system to take advantage of existing libraries and *OCaml*'s code generation facilities.

The project is hosted at `http://easyocaml.forge.ocamlcore.org` where an online demo of *EasyOCaml*'s type inference and languge levels is available, too.

1.1 Similar Projects

There are some projects which heavily influenced our work:

Haack and 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 *EasyOCamlś* type checker.

Helium (Heeren et al., 2003) is a system for teaching programming in Haskell. In a similar manner, type checking is done via constraint solving. Furthermore, it features detailed error messages including hints how to fix errors based on certain heuristics.

Finally, *DrScheme* (Felleisen et al., 1998) is a programming environment for the Scheme language which is build for teaching programming. It has introduced the concept of language levels and teach packs to restrict the syntax and broaden functionality especially for exercises.

EasyOCaml uses and combines ideas of all these projects for the attempt to make *OCaml* better suited for learning and teaching programming. * *+++

Goals of this report

This report has four goals: Firstly to present *EasyOCaml* and the concepts used in and developed for it (target audience: the users) through section 2–5. Then, it describes the usage of the programs `ecaml` and `ecamlc` for usage in section 7. Afterwards it combines the concepts with the actual implementation and describes its architecture in code (target audience: the developers) in section 9. Finally, formal foundations for the type checker are given the section 10 (target audience: the interested readers).

This report is also split in a less formal part I (answering the question: *what?*) and a more formal part II which acts as a reference for users and developers (answering the question: *how?*).

2 General Functionality of EasyOCaml

This section covers some general information on architecture of *EasyOCaml* and its language for orientation in supsequent parts. It will give a broad overview on *EasyOCaml*'s course of actions first. Every step is elaborated in more detail in subsequent sections, but they might be easier to understand with the knowledge of their role in *EasyOCaml*. It will give a desription of the language supported by *EasyOCaml* afterwards.

2.1 Outline of EasyOCaml's Course

By large *EasyOCaml*'s additions to the compiler and the toplevel loop do a fairly similar job: First, they parse the command line flags. If they contain the `-easy` flag, *EasyOCaml*'s type checker is enabled and flags for defining language levels, teach packs and error printers are accepted and loaden into *EasyOCaml*.

Then, *EasyOCaml* takes over the first stage of the compiler by parsing the input with a *Camlp4* parser. The parser incorporates modifications by the language level to the availabel syntax. The resulting AST is annotated with fresh type variables.

EasyOCaml then tries to type check the program. It reads the definitions in the program and generates a set of constraints on the types of every element of the program by traversing the AST. The constraints on the node's types are based on the node's usage and their generation is detailed in the section 3 and 10. Then, the type checker tries to solve those constraints. If they are unifiable, the type checker can assign a valid typing on the elements of the AST and gives control back to *OCaml*. Otherwise, *EasyOCaml* tries to generate as many conflict sets (type errors) as possible from the constraints and reports them to the user.

2.2 Supported Language: Caml$_{-m}$

EasyOCaml targets to be usable not only for the very first steps in programming, so a reasonable subset of the *OCaml* language is supported. This language can be characterized as “Caml minus module declarations”, hence its designation *Caml*$_{-m}$.

Unlike simpler functional programming languages like *Scheme* with only a single syntactic category *expression*, *OCaml* makes a distinction between *structure items*, *expressions* and *patterns*. Here are the supported language features in more detail, which also can be pruned by teach packs. See section 8 for a complete grammar.

Structure items

OCaml programs consist of a list of *structure items*, which are used to declare values, types and exceptions. *EasyOCaml* supports

- optionally parametrized *type declarations* of type synonyms, records with optionally mutable fields and variant types.
- *exception declarations* like those in *OCaml*.
- optionally recursive (`rec`) and multiple (`and`) *value declarations*, where bindings occur with arbitrary patterns. Note, that a languge level may require type annotations for toplevel value declarations.
- Toplevel evaluations. Note that the toplevel evaluation of `e` is just syntactic sugar for the value declaration `let _ = e`.

Core types

core types are direct combinations of existing types are called in *OCaml*. *EasyOCaml* allows as core types primitive types (`int`, `char`, `string` and `float`), free (in type annotations) and bound (within type declarations) type variables, type arrows (function types), tuples and type constructors, i.e. applications of parametrized types.

Expressions

Expressions are parts of a program which can be evaluated to a *OCaml* value and occur only as part of structure items. *EasyOCaml* supports simple expressions that can be found in *MiniML*, too, like variables, functions, infix operators, conditionals and variable binding.

Despite those, it features the construction of tuples, records and variants, conditionals with optional `else` branch, `while` and `for` loops, sequences of expressions, exception handling (raising and catching) as well as type annotations.

Patterns

In *EasyOCaml*, pattern matching is possible in every place where it works in *OCaml*, i.e. in value matching[1], in variable bindings[2], in functional abstractions with the `function`

[1] `match ... with` *`pat`* `-> ... | ...`
[2] `let` *`pat`* `= ...`

keyword[3] and exception catching[4].

Pattern matching works on every possible value in *EasyOCaml*, i.e. primitive values, tuples, variants and records[5], and can be nested at every level.

[3] `function` *pat* `-> ... | ...`

[4] `try ... with` *pat* `-> ... | ...`

[5] version 0.49 lacks the implementation for pattern matching on the latter

Part I
Concepts and design

3 Constraint Based Type Inference

The type inference currently used by *OCaml* has the algorithm $\mathcal{W}$ by Milner (1978) at its core. Although very efficient for most programs and broadly extended to *OCaml*'s requirements, it lacks sort of a memory: Infering the type of a variable is done by accumulating (unifying) information on its usages while traversing the abstract syntax tree (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 contextual reasons for a type error but it reports only the location where the error became obvious to the type checker. Much work while debugging type errors in *OCaml* comprises thus of manually searching for other usages of the mis-typed variable in the program which might have lead to the type constructor clash.

I will first loosely describe the algorithm by Haack and Wells—see their 2004 paper for a rigid explanation—and then explain the extensions we made for *EasyOCaml*.

3.1 Haack & Wells's Type Checking Algorithm

Haack and Wells (2004) describe an algorithm which exceeds algorithm $\mathcal{W}$ 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 one of them would vanish the error. Secondly, it is able to report 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 $\mathcal{W}$ does two things at once while traversing the AST: It generates information on the types of the variables and unifies it with existing type information anon. Haack and Wells' algorithm works in some sense by separating these steps.

During *constraint generation* every node of the AST is* annotated with a type variable. *?
While traversing the AST, information on those type variables is collected from the usage of each node. This information is stored as a set of constraints on the type variables. The intention is the following: If the constraints are unifiable, 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 in case of a constraint conflict, 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. Haack and Wells (2004) 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 nevertheless exponential time consumptions. Thus error enumeration is delimited in *EasyOCaml* to a given time amount which can be specified by an environment variable (see section 7.2).

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

In addition to type errors, Haack and Wells' technique also enables the type checker to collect all unbound variables in the program. Their types are assumed as free type variables to avoid a type error, but reported after solving the constraints or with the type errors after error enumeration.

3.2 Extensions for EasyOCaml

Haack and Wells (2004) comes with constraint generation rules for *MiniML*, a subset of the *ML* language only supporting variables, infix operations, functional abstraction, application and local polymorphic variable bindings. This is good to describe the algorithms involved, but we had to extend it to the language *Caml*$_{-m}$—as loosely described in section 2.2 and more formally in section 8—and *EasyOCaml*'s much richer type system. I will describe the constraint generation rules for *EasyOCaml* by example here, section 10 exposes the complete set of rules.

In the following, Δ always denotes an environment (store) for current bindings of variables, record fields and variant constructors anon, accessible by $\Delta|_{\mathrm{id}}$, $\Delta|_{\mathrm{var}}$, $\Delta|_{rec}$ and $\Delta|_{\mathrm{var}}$ respectively.

To capture the possibility to *declare* values and types, constraint generation rules for structure items have the form

$$\Delta;\ strit \Downarrow_s \langle \Delta',\ C,\ u\rangle.$$

where Δ denotes the environent which contains declarations in the program so far and *strit* denotes the current structure item. Δ' denotes the environment Δ extended by declarations in *strit* and C is a set of constraints collected in *strit*. u is a set of errors in *strit* which are desribed in more detail in section 4.1. Those declarations, constraints and errors are accumulated while generating traversing the program's structure items.

This is the rule for the declaration of a variant type:

$$\textsc{Variant Decl}\quad \frac{\Delta' = \Delta|_{\mathrm{var}}[t \mapsto \{\langle K_1, ty_1\rangle^{l_1}, \ldots, \langle K_n, ty_n\rangle^{l_n}\}^l]}{\Delta;\ (\texttt{type}\ t = K_1\ \texttt{of}^{l_1}\ ty_1\ \texttt{|} \ldots \texttt{|} K_n\ \texttt{of}^{l_n}\ ty_n)^l \Downarrow_s \langle \Delta',\ \emptyset,\ \emptyset\rangle}$$

It just extends the current environment Δ with the variant constructors K_1 to K_n with the given types. Note, that the locations are stored to make a reference on the type declaration in case of a typing error related to one of those variant constructors.

Constraint generation rules for expressions are better examples for accumulating constraint sets. As a simple starting point, we will discuss the rule for `if` expressions without an `else` branch here. *OCaml* provides the test expression to be of type $\underline{bool}$, the expression in the branch of type $\underline{unit}$ and the whole expression of type $\underline{unit}$, too. The rule implements this a follows:

$$\textsc{If-then}\quad \frac{\Delta;\ lexp_1 \Downarrow_e \langle ty_1,\ C_1,\ u_1\rangle \qquad \Delta;\ lexp_2 \Downarrow_e \langle ty_2,\ C_2,\ u_2\rangle \qquad C_0 = \{ty_1 \stackrel{l}{=} \underline{bool},\ ty_2 \stackrel{l}{=} \underline{unit},\ a \stackrel{l}{=} \underline{unit}\} \qquad a \text{ fresh}}{\Delta;\ (\texttt{if}\ lexp_1\ \texttt{then}\ lexp_2)^l \Downarrow_e \langle a,\ C,\ u_1 \cup u_2\rangle}$$

Constraint generation is applied here to $lexp_1$ (and exp_2 respectively), resulting in the type ty_1 which is in fact a type variable constraint to the result type of $lexp_1$ in C_1. The rule generates three additional constraints. The first one, $ty_1 \stackrel{l}{=} \underline{bool}$, asserts the result type ty_1 of expression $lexp_1$ to be of type $\underline{bool}$. The second, $ty_2 \stackrel{l}{=} \underline{unit}$ asserts the result type ty_2 of the expression $lexp_1$ to be of type $\underline{unit}$. The third one, $ty_3 \stackrel{l}{=} \underline{unit}$, asserts that a freshly generated type variable ty_3, which represents the type of the whole conditional expression, is of type $\underline{unit}$.

Note, that all new constraints are annotated with the location l of the overall expression. This facilitates the blaming of it in case of a type error resulting from a conflict with one of the constraints special to this conditional expression.

The constraint generation results in the generated type variable ty_3 and the union of all occurring constraint and error sets.

Furthermore, *EasyOCaml* features type annotations of the form $(lexp : ct)$. Special considerations are necessary for them: OCaml's current type checker ignores the type annotations whilst unification by using the expression *lexp*'s inferred type for further type checking and only checks its validity as to *ct* later on. So type inference and error reporting makes no use of the type annotations itself.

In contrast, *EasyOCaml* assumes the expression to have the denoted type while type checking, type checks the expression isolated and tests the validity afterwards, by proving that the annotated type is a subtype of the inferred type.

$$\textsc{Type-Annot}\quad \frac{\Delta;\ lexp \Downarrow_e \langle ty,\ C_0,\ u\rangle \qquad C_1 = \{a \stackrel{l}{=} b,\ b \stackrel{l'}{=} ty',\ ty \stackrel{l}{=} c,\ c \stackrel{l'}{=} ty',\ a \succcurlyeq^l ct\} \qquad a,\ b,\ c \text{ fresh} \qquad ty' \text{ is a fresh instance of } ct}{\Delta;\ (lexp : ct^{l'})^l \Downarrow_e \langle a,\ C_0 \cup C_1,\ u\rangle}$$

That way, *EasyOCaml* assumes the programmer's annotation to be valid and meaningful during type inference of the context of *lexp* and checks for contradiction to the expression's type against the annotation afterwards.

Patterns & rules* *+++

EasyOCaml's constraint generation does a second task anon: It checks th validity of the variables, record constructors, field accesses and type constructions. The handling of those errors and others are described in the next section.

4 Errors and Error Reporting Adaptibility

EasyOCaml is essentially build for teaching programming. As such, special attention is paid to the way errors are reported to achieve the following goals:

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 of the program should be reported, which are essential to the error. Delivering more information on the underlying reasons of type errors is exactly what *EasyOCaml*'s type checker is made for.

Secondly, error reporting should be adaptable: Reading errors in a foreign language can distract or even prevent for a beginner from understanding it. So internationalization of error messages is necessary. 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 to display it in a web browser. Adaption of the language used for error reporting can be achieved by changing the environment variable `LANG`. The formatting can be given by a command line argument (see section 7 for details).

The last section has explained the improvements of *EasyOCaml* to the type error messages in the last section and this one will describe the structure of the errors handled by *EasyOCaml* as well as the changes to the parser to build a foundation to include more information with the parsing errors below and the error adaption possibilities subsequently.

4.1 The Structure of Errors in EasyOCaml

While*parsing and type checking, *EasyOCaml* can detect different errors. Differences are made between the following three classes of errors[6]: * ***

The compiler attempts to report errors as late as possible to collect as many errors as possible. *Common errors* are reported at the very end after contstraint unification. So type errors (type constructor clashes, clashes of the arities of tuples and circular types) are of course of this kind. Furthermore, unbound variables are admittedly detectey during constraint generation but provided with free type variables, such that constraint unification is possible but no extra errors are introduced. Other light errors like the attemt to change the value of immutable record fields are reported after constraint unification, too, and invalid type annotations as well.

Heavy errors are collected during and reported directly after constraint generation, because they make it impossible to assign at least a transitionally type to the term. They include among others: Incoherent record constructions (fields from different records or several bindings of a field), several bindings of a variable in a pattern, the usage of unknown variant constructors or wrong usage of type constructors (unknown or wrong number of type arguments).

Syntactical errors (parsing errors) and accessing inexistent modules are *fatal errors* and stop the compiler directly, because it prevents reasonable further processing of the

[6] the file `ezyErrors.mli` provides a complete and well documented list of errors

program.

The next section will describe our changes to *Camlp4* for more detailled parsing errors.

4.2 New Errors for Camlp4

As mentioned, the program code is parsed py a *Camlp4* parser in *EasyOCaml*. Unfortunately, *Camlp4*'s error messages are hard coded in English and never represented in data which prohibits the adaption of format and language while reporting. This is because *Camlp4* is a *OCaml* stream parser in its core, which requires parsing errors to be reported as exceptions with just a string for information[7].

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 parsing errors with more information on the state of the parser. A parsing error[8] 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 which contains the entry.

`Illegal_begin sym` is raised when the parser is not able to parse the program's toplevel categories given 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 subsection 9.8.)

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, parsing exceptions contain a string of the format "`<msg>\000<mrsh>`" where `<msg>` is the usual *Camlp4* error message and `<mrsh>` is the marshalled parsing error as just described. This string is again decomposed in *Camlp4*'s interface function for parsing[9], and reported as a parsing error[10] to the user.

And so *Camlp4*'s parsing errors are now represented in data to apply error reporting adaptabilities and to further extend them by more information on the parser's state.

[7] exception `Stream.Error of string`
[8] type `ParseError.t`
[9] function `Camlp4.Struct.Grammar.Entry.action_parse`
[10] type `ParseError.t`

4.3 Adaptibility

For internationalization of error messages and different structures 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 which EasyOCaml provides specifically for teaching programming.

5 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 taught 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*. Teach packs can specify the directories which are included for searching objects, just like the `-I` command line flag. A teach pack can contain objects itself and the specification which have to be loaded (just like putting them on the command line). Furthermore, teach packs can specify which modules are opened on startup.

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

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

The idea of teach packs and language levels is taken from DrScheme. See http://docs.plt-scheme.org/c
for more information.

6 Forecast and Conclusion

EasyOCaml in version 0.5 is not yet full ready for action. For version 1.0, we will add installation procedures for language levels and teachpacks. Furthermore, we will equip the errors with more informations such that error reporting plugins can use certain heuristics to give hints how to fix an error (i.e. to pass the variable environment to check misspellings for unbound variable errors). In a long term, a dynamically typed interpreter would be great for the very first lessons in programming without the need to think about types, as well as the support for module declarations to make *EasyOCaml* ready for our daily programming.

Acknowledgements
This project was funded in the *OCaml Summer Project 2008*. So thanks to the people at Jane Street Capital for this initiative and to make *EasyOCaml* possible! I would also like to thank Prof. Peter Thiemann and his team—especially Stefan Wehr—very much for mentoring me, giving invaluable hints and guiding me through *EasyOCaml*.

Part II
Manual, formalities and implementation

7 The User Interface (Manual)

This section describe the extensions of the `ocaml` and `ocamlc` programs.

7.1 Command Line Parameters

There are some command line flags to control *EasyOCaml*:

`-easy` This flag enables *EasyOCaml* and is obligatory for usage of all other command line flags desribed here. It enables an alternative type checking algorithm wich gives more information on the type errors.

`-easyteachpack <teachpack>` Enables a teachpack named `<teachpack>`.

7.2 Environment Variables

`EASYOCAML_ENUM_TIMEOUT` The real value controls the maximal amount of time *EasyOCaml* may use to enumerate type errors (note, that the underlying algorithm has exponential time consumptions).

7.3 The EasyOCaml Directory

EasyOCaml searches for language levels and teachpacks in a designated configuration directory.

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

EasyOCaml's configuration directory must have the following structure:

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

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

8 Grammar of Caml$_{-m}$ in rail diagrams

9 Details of the Implementation

9.1 Dependency Graph for EasyOCaml's modules

...

9.2 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:

9.3 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 ellipses.

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

9.4 Teachpacks and Language Levels

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

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

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

EzyTeachpack Shortcut to `EzyFeatures` and registering of the teach pack. Actual teach packs 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 teach packs and language levels to other parts of EasyOCaml.

9.5 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. 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. Furthermore, each syntactic category can be some "dots" which is only used in type error slicing.

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

9.6 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.

9.7 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`.

9.8 Camlp4

OcamlSpecificErrors

10 Constraint Generation Rules for EasyOCaml

This section describes the rules for type inference used in EasyOCaml. See section 3.2 for some introducing text. The rules have the following form.

- for expressions $\Delta;\ lexp \Downarrow_e \langle ty,\ C,\ u\rangle$
- for structure items $\Delta;\ strit \Downarrow_s \langle \Delta,\ C,\ u\rangle$
- for rules $\Delta;\ pat$ `->` $lexp \mid rules \Downarrow_r \langle ty_p,\ ty_e,\ C,\ u\rangle$ (just an auxiliary)
- for patterns $\Delta;\ pat \Downarrow_p \langle ty,\ C,\ b\rangle$ where b maps identifiers to $\langle ty\rangle^l$.

We use the following notations, to keep the rules short: C denotes a set of constraints, a a type variable, ty a type, u a set of identifiers and Δ a general environment. A general environment Δ encapsulates environments for lookup of the

- type of a variable:
$$\Delta|_{\text{id}}(lid) = \langle ty,\ \varpi, C\rangle^l$$
where $\varpi \in \{\mathsf{mono}, \mathsf{poly}\}$ and lid has been bound accordingly.
- types of record fields:
$$\Delta|_{rec}(f) = \langle ty_r,\ ty_f,\ \mu\rangle$$
where ty'_f is the type of field f in record type ty'_r and ty_f, ty_r are fresh variants of ty'_f, ty'_r. $\mu \in \{\mathsf{mutable},\ \mathsf{immutable}\}$
- types of variants
$$\Delta|_{\text{var}}(K) = \langle ty_r,\ [ty_1,\ \ldots,\ ty_n]\rangle$$
where $ty'_1,\ \ldots,\ ty'_n$ are the arguments for variant k of type ty'_r and $ty_1,\ \ldots,\ ty_n, ty_r$ are fresh variants of $ty'_1,\ \ldots,\ ty'_n, ty'_r$.

$\Delta|_X[x \mapsto y]$ designates the general environment Δ where x is substituted by y in the encapsulated environment X.

$\Delta|_{\text{id}}[b,\ C,\ \varpi]$ is a shorthand for $\Delta|_{\text{id}}\left[id \mapsto \langle ty,\ \varpi,\ C\rangle^l \mid id \mapsto \langle ty\rangle^l \in b\right]$, i.e. the substitution of all bindings of b in Δ with constraints C where b maps identifiers to $\langle ty\rangle^l$.

10.1 Structure items

Eval

$$\frac{\Delta;\ lexp \Downarrow_e \langle ty,\ C,\ u\rangle}{\Delta;\ lexp \Downarrow_s \langle \Delta,\ C,\ u\rangle}$$

VALUE DECL

$$\frac{\begin{array}{cc}\Delta;\,pat_i \Downarrow_p \langle ty_{p,i},\ C_{p,i},\ b_i\rangle & \Delta;\,exp_i \Downarrow_e \langle ty_{e,i},\ C_{e,i},\ u_i\rangle \\ \varpi_i := \mathsf{poly}\text{ if } \mathit{value}\ lexp_i \text{ else } \mathsf{mono} & C_{x,i} := C_{p,i} \cup C_{e,i} \cup \{ty_{p,i} \stackrel{l}{=} ty_{e,i}\} \\ \Delta' := \Delta|_{\mathrm{id}}[b_i,\ C_{x,i},\ \varpi_i \mid i = 1,\ldots,n] & \mathrm{dom}(b_i) \cap \mathrm{dom}(b_j) = \emptyset \text{ for all } i \neq j\end{array}}{\Delta;\,(\texttt{let}\ pat_1\ \texttt{=}\ lexp_n\ \texttt{and}\ \ldots\ \texttt{and}\ pat_n\ \texttt{=}\ lexp_n)^l \Downarrow_s \langle \Delta',\ \bigcup_{i=1}^{n} C_{x,i},\ \bigcup_{i=1}^{n} u_i\rangle}$$

REC VALUE DECL

$$\frac{\begin{array}{c}\Delta|_{\mathrm{var}}[x_j \mapsto \langle a_j,\ \mathsf{mono},\ \emptyset\rangle^l \mid j = 1,\ldots,n];\,lexp_i \Downarrow_e \langle ty_i,\ C_i,\ u_i\rangle \\ \varpi_i := \mathsf{poly}\text{ if } \mathit{value}\ lexp_i \text{ else } \mathsf{mono} \\ \text{for i=1,}\ldots\text{,n} \qquad \Delta' := \Delta|_{\mathrm{var}}[x_i \mapsto \langle ty_i,\ \varpi_i,\ C_i \cup \{ty_i \stackrel{l}{=} a_i\}\rangle^l \mid \text{for } i = 1,\ldots,n] \\ a_1,\ \ldots,\ a_n \text{ fresh} \qquad x_i = x_j \text{ iff } i = j\end{array}}{\Delta;\,(\texttt{let rec}\ x_1\ \texttt{=}\ lexp_n\ \texttt{and}\ \ldots\ \texttt{and}\ x_n\ \texttt{=}\ lexp_n)^l \Downarrow_s \langle \Delta',\ \bigcup_{i=1}^{n} C_{x,i},\ \bigcup_{i=1}^{n} u_i\rangle}$$

RECORD DECL

$$\frac{\Delta' = \Delta|_{rec}[t \mapsto \{\langle f_1, ty_1\rangle^{l_1}, \ldots, \langle f_n, ty_n\rangle^{l_n}\}^l]}{\Delta;\,(\texttt{type}\ t = \{f_1\texttt{:}^{l_1} ty_1\texttt{;}\ \ldots\texttt{;}\ f_n\texttt{:}^{l_n}\ ty_n\})^l \Downarrow_s \langle \Delta',\ \emptyset,\ \emptyset\rangle}$$

VARIANT DECL

$$\frac{\Delta' = \Delta|_{\mathrm{var}}[t \mapsto \{\langle K_1, ty_1\rangle^{l_1}, \ldots, \langle K_n, ty_n\rangle^{l_n}\}^l]}{\Delta;\,(\texttt{type}\ t = K_1\ \texttt{of}^{l_1}\ ty_1\ \texttt{|}\ \ldots\ \texttt{|}\ K_n\ \texttt{of}^{l_n}\ ty_n)^l \Downarrow_s \langle \Delta',\ \emptyset,\ \emptyset\rangle}$$

SEQUENCE

$$\frac{\Delta;\,strit_1 \Downarrow_s \langle \Delta',\ C_1,\ u_1\rangle \qquad \Delta';\,strit_2 \Downarrow_s \langle \Delta'',\ C_2,\ u_2\rangle}{\Delta;\,strit_1\ \texttt{;;}\ strit_2 \Downarrow_s \langle \Delta'',\ C_1 \cup C_2,\ u_1 \cup u_2\rangle}$$

10.2 Rules

$$\frac{\Delta;\,pat \Downarrow_p \langle ty_p,\ C_p,\ b\rangle \qquad \Delta|_{\mathrm{id}}[b,\ C_p,\ \mathsf{mono}];\,lexp \Downarrow_e \langle ty_e,\ C_e,\ u\rangle}{\Delta;\,pat\ \texttt{->}\ lexp \Downarrow_r \langle ty_p,\ ty_e,\ C_p \cup C_e,\ u\rangle}$$

$$\frac{\begin{array}{c}\Delta;\,pat\ \texttt{->}\ lexp \Downarrow_r \langle ty_{p,1},\ ty_{e,1},\ C_1,\ u_1\rangle \qquad \Delta;\,rules \Downarrow_r \langle ty_{p,2},\ ty_{e,2},\ C_2,\ u_2\rangle \\ C = \{a_p \stackrel{l}{=} ty_{p,1},\ a_p \stackrel{l}{=} ty_{p,2},\ a_e \stackrel{l}{=} ty_{e,1},\ a_e \stackrel{l}{=} ty_{e,2}\} \cup C_1 \cup C_2 \qquad a_p,\ a_e \text{ fresh}\end{array}}{\Delta;\,(pat\ \texttt{->}\ lexp)^l\ |\ rules \Downarrow_r \langle a_p,\ a_e,\ C,\ u_1 \cup u_2\rangle}$$

10.3 Expressions

Var-Mono

$$\frac{\Delta(x) = \langle ty, \mathsf{mono}, C\rangle^{l'} \qquad a,\ a_x \text{ fresh}}{\Delta;\ x^l \Downarrow_e \langle a,\ C \cup \{a_x \stackrel{l}{=} a,\ ty \stackrel{l'}{=} a_x\},\ \emptyset\rangle}$$

Var-Poly

$$\frac{\Delta(x) = \langle ty, \mathsf{poly}, C\rangle^{l'} \qquad a,\ a_x \text{ fresh} \qquad \langle ty', C'\rangle \text{ fresh variant of } \langle ty, C\rangle}{\Delta;\ x^l \Downarrow_e \langle a,\ \{a_x \stackrel{l}{=} a,\ ty' \stackrel{l'}{=} a_x\} \cup C',\ \emptyset\rangle}$$

Var-Undef

$$\frac{x \notin \mathrm{dom}(\Delta) \qquad a \text{ fresh}}{\Delta;\ x^l \Downarrow_e \langle a,\ \emptyset,\ \{x^l\}\rangle}$$

Const

$$\frac{C_0 = \{ty \stackrel{l}{=} a\} \qquad a \text{ fresh} \qquad ty \text{ type of constant } c}{\Delta;\ c^l \Downarrow_e \langle a,\ C_0,\ \emptyset\rangle}$$

Abstr

$$\frac{\Delta;\ rules \Downarrow_r \langle ty_p,\ ty_e,\ C_0,\ u\rangle \qquad C_1 = \{a \stackrel{l}{=} ty_p \to ty_e\}}{\Delta;\ (\texttt{function}\ rules)^l \Downarrow_e \langle a,\ C_0 \cup C_1,\ u\rangle}$$

App

$$\frac{\begin{array}{c}\Delta;\ lexp_i \Downarrow_e \langle ty_i,\ C_i,\ u_i\rangle \text{ for } i = 0, \ldots, n \\ C' = \{a'_{i-1} \stackrel{l-l_i}{=\!=} a_i \to a'_i,\ ty_i \stackrel{l}{=} a_i \mid i = 1, \ldots, n\} \cup \{ty_0 \stackrel{l}{=} a'_0,\ a \stackrel{l}{=} a'_n\} \\ a, a_1, \ldots, a_n, a'_0, \ldots, a'_n \text{ fresh}\end{array}}{\Delta;\ (lexp_0\ lexp_1^{l_1}\ \ldots\ lexp_n^{l_n})^l \Downarrow_e \langle a,\ C' \cup \bigcup_{i=0}^{n} C_i,\ \bigcup_{i=1}^{n} u_i\rangle}$$

Let

$$\dfrac{\begin{array}{c}\Delta;\ pat_i \Downarrow_p \langle ty_{p,i},\ C_{p,i},\ b_i\rangle \\ \Delta;\ lexp_i \Downarrow_e \langle ty_{e,i},\ C_{e,i},\ u_i\rangle \qquad \varpi_i := \textsf{poly}\text{ if }\mathit{value}\ lexp_i\text{ else }\textsf{mono} \\ \text{for } i=1,\dots,n \qquad \Delta' = \Delta|_{\mathrm{id}}[b_i,\ C_{p,i}\cup C_{e,i}\cup\{ty_{p,i} \stackrel{l}{=} ty_{e,i}\},\ \varpi_i \mid \text{for } i=1,\dots,n] \\ \Delta';\ lexp_{n+1} \Downarrow_e \langle ty_{n+1},\ C_{n+1},\ u_{n+1}\rangle \\ C_0 = \{a \stackrel{l}{=} ty_{n+1}\} \cup \{ty_{p,i} \stackrel{l}{=} ty_{e,i} \mid i=1,\dots,n\} \\ \mathrm{dom}(b_i)\cap\mathrm{dom}(b_j)=\emptyset \text{ for all } i\neq j \qquad a \text{ fresh}\end{array}}{\Delta;\ (\texttt{let}\ x_1\ \texttt{=}\ lexp_1\ \texttt{and}\ \dots\ \texttt{and}\ x_n\ \texttt{=}\ lexp_n\ \texttt{in}\ lexp_{n+1})^l \Downarrow_e \langle a,\ \bigcup_{i=0}^{n+1} C_i,\ \bigcup_{i=1}^{n+1} u_i\rangle}$$

Let rec

$$\dfrac{\begin{array}{c}\Delta|_{\mathrm{id}}\left[x_j \mapsto \langle a_j,\ \textsf{mono},\ \emptyset\rangle^l \mid j=1,\dots,n\right];\ lexp_i \Downarrow_e \langle ty_i,\ C_i,\ u_i\rangle \\ \varpi_i := \textsf{poly}\text{ if }\mathit{value}\ lexp_i\text{ else }\textsf{mono} \\ \text{for } i=1,\dots,n \qquad \Delta' := \Delta|_{\mathrm{id}}[x_j \mapsto \langle ty_j,\ \varpi_j,\ C_j \cup \{a_j \stackrel{t}{=} y_j\}\rangle^l \mid j=1,\dots,n] \\ \Delta';\ lexp_{n+1} \Downarrow_e \langle ty_{n+1},\ C_{n+1},\ u_{n+1}\rangle \\ C_0 = \{a \stackrel{l}{=} ty_{n+1}\} \qquad x_i = x_j \text{ iff } i=j \qquad a,\ a_1,\ \dots,\ a_n \text{ fresh}\end{array}}{\Delta;\ (\texttt{let rec}\ x_1\ \texttt{=}\ lexp_1\ \texttt{and}\ \dots\ \texttt{and}\ x_n\ \texttt{=}\ lexp_n\ \texttt{in}\ lexp_{n+1})^l \Downarrow_e \langle a,\ \bigcup_{i=0}^{n+1} C_i,\ \bigcup_{i=1}^{n+1} u_i\rangle}$$

Tuple

$$\dfrac{\Delta;\ lexp_i \Downarrow_e \langle ty_i,\ C_i,\ u_i\rangle \text{ for } i=1,\ \dots,\ n \qquad C_0 = \{a \stackrel{l}{=} (ty_1,\ \dots,\ ty_n)\} \qquad a \text{ fresh}}{\Delta;\ (lexp_1\texttt{,}\ \dots\texttt{,}\ lexp_n)^l \Downarrow_e \langle a,\ \bigcup_{i=0}^{n} C_i,\ \bigcup_{i=1}^{n} u_i\rangle}$$

Record construction

$$\dfrac{\begin{array}{c}\Delta;\ lexp_i \Downarrow_e \langle ty_i,\ C_i,\ u_i\rangle \qquad \Delta|_{rec}(f_i) = \langle ty_r,\ ty_{f,i},\ .\rangle \\ \text{for } i=1,\dots,n \qquad C_0 = \{a \stackrel{l}{=} ty_r\} \cup \{ty_i \stackrel{l}{=} ty_{f,i} \mid i=1,\dots,n\} \\ a \text{ fresh} \qquad \{f_i \mid i=1,\dots,n\} \text{ are the fields of record } ty_r\end{array}}{\Delta;\ \{f_1\ \texttt{=}\ lexp_1\texttt{;}\ \dots\texttt{;}\ f_n\ \texttt{=}\ lexp_n\}^l \Downarrow_e \langle a,\ \bigcup_{i=0}^{n} C_i,\ \bigcup_{i=1}^{n} u_i\rangle}$$

$$\dfrac{\begin{array}{c}\Delta;\ lexp_i \Downarrow_e \langle ty_i,\ C_i,\ u_i\rangle \text{ for } i=0,\dots,n \qquad \Delta|_{rec}(f_i) = \langle ty_r,\ ty_{f,i},\ \cdot\rangle \text{ for } i=1,\dots,n \\ C_0 = \{a \stackrel{l}{=} ty_r,\ ty_0 \stackrel{l}{=} ty_r\} \cup \{ty_{f,i} \stackrel{l}{=} ty_i \mid i=1,\dots,n\} \qquad a \text{ fresh}\end{array}}{\Delta;\ \{lexp_0\ \texttt{with}\ f_1\ \texttt{=}\ lexp_1\texttt{;}\ \dots\texttt{;}\ f_n\ \texttt{=}\ lexp_n\}^l \Downarrow_e \langle a,\ \bigcup_{i=0}^{n} C_i,\ \bigcup_{i=1}^{n} u_i\rangle}$$

Record Access

$$\frac{\Delta;\ lexp \Downarrow_e \langle ty,\ C,\ u\rangle \qquad \Delta|_{rec}(f) = \langle ty_f,\ ty,\ \cdot\rangle \qquad C_0 = \{a \stackrel{l}{=} ty_f,\ ty \stackrel{l}{=} ty_r\} \qquad a \text{ fresh}}{\Delta;\ (lexp\,.\,f)^l \Downarrow_e \langle a,\ C_0 \cup C,\ u\rangle}$$

Record field assignment

$$\frac{\Delta;\ lexp_1 \Downarrow_e \langle ty_1,\ C_1,\ u_1\rangle \qquad \Delta;\ lexp_2 \Downarrow_e \langle ty_2,\ C_2,\ u_2\rangle \qquad \Delta|_{rec}(f) = \langle ty_r,\ ty_f, \mathsf{mutable}\rangle \qquad C_0 = \{a \stackrel{l}{=} \underline{unit},\ ty_r \stackrel{l}{=} ty_1,\ ty_f \stackrel{l}{=} ty_2\}}{\Delta;\ (lexp_1\,.\,f\ \texttt{<-}\ lexp_2)^l \Downarrow_e \langle a,\ C_0 \cup C_1 \cup C_2,\ u_1 \cup u_2\rangle}$$

Variant

$$\frac{\Delta|_{\mathrm{var}}(K) = \langle ty_r,\ [ty_{a,1},\ \ldots,\ ty_{a,n}]\rangle \qquad \Delta;\ lexp_i \Downarrow_e \langle ty_i,\ C_i,\ u_i\rangle \text{ for } i = 1,\ldots,n \qquad C_0 = \{a \stackrel{l}{=} ty_r\} \cup \{ty_i \stackrel{l}{=} ty_{a,i} \mid i = 1,\ldots,n\} \qquad a \text{ fresh}}{\Delta;\ (K\ lexp_1\ \ldots\ lexp_n)^l \Downarrow_e \langle a,\ \bigcup_{i=0}^{n} C_i,\ \bigcup_{i=1}^{n} u_i\rangle}$$

NB The distinction between $K\ \ lexp_1\ \ \ldots\ \ lexp_n$ (n arguments) and $K\ \ (lexp_1,\ \ldots,\ lexp_n)$ (an n-tuple as the single argument) is actually made by an `explicit_arity` flag in the AST.

If-then-else

$$\frac{\Delta;\ lexp_1 \Downarrow_e \langle ty_1,\ C_1,\ u_1\rangle \qquad \Delta;\ lexp_2 \Downarrow_e \langle ty_2,\ C_2,\ u_2\rangle \qquad \Delta;\ lexp_3 \Downarrow_e \langle ty_3,\ C_3,\ u_3\rangle \qquad C = \{ty_1 \stackrel{l}{=} \underline{bool},\ a \stackrel{l}{=} ty_3,\ a \stackrel{l}{=} ty_2\} \cup C_1 \cup C_2 \cup C_3 \qquad a \text{ fresh}}{\Delta;\ (\texttt{if}\ lexp_1\ \texttt{then}\ lexp_2\ \texttt{else}\ lexp_3)^l \Downarrow_e \langle a,\ C,\ u_1 \cup u_2 \cup u_3\rangle}$$

If-then

$$\frac{\Delta;\ lexp_1 \Downarrow_e \langle ty_1,\ C_1,\ u_1\rangle \qquad \Delta;\ lexp_2 \Downarrow_e \langle ty_2,\ C_2,\ u_2\rangle \qquad C_0 = \{ty_1 \stackrel{l}{=} \underline{bool},\ ty_2 \stackrel{l}{=} \underline{unit},\ a \stackrel{l}{=} \underline{unit}\} \qquad a \text{ fresh}}{\Delta;\ (\texttt{if}\ lexp_1\ \texttt{then}\ lexp_2)^l \Downarrow_e \langle a,\ C,\ u\rangle}$$

Matching

$$\frac{\Delta;\ lexp \Downarrow_e \langle ty_0,\ C_0,\ u_0\rangle \qquad \Delta;\ rules \Downarrow_r \langle ty_p,\ ty_e,\ C_1,\ u_1\rangle \qquad C = \{ty_0 \stackrel{l}{=} ty_p,\ a \stackrel{l}{=} ty_e\} \cup C_1 \cup C_2 \qquad u = u_0 \cup u_1 \qquad a \text{ fresh}}{\Delta;\ (\texttt{match}\ lexp\ \texttt{with}\ rules)^l \Downarrow_e \langle a,\ C,\ u\rangle}$$

Array construction

$$\frac{\Delta;\ lexp_i \Downarrow_e \langle ty_i,\ C_i,\ u_i\rangle \qquad C_0 = \{a \stackrel{l}{=} b\ \underline{array}\} \cup \{ty_i \stackrel{l}{=} b \mid i = 1, \ldots, n\} \qquad a,\ b \text{ fresh}}{\Delta;\ (\texttt{[|}lexp_1\texttt{;}\ \ldots\texttt{;}\ lexp_n\texttt{|]})^l \Downarrow_e \langle a,\ \bigcup_{i=0}^{n} C_i,\ \bigcup_{i=1}^{n} u_i\rangle}$$

Array access

$$\frac{\Delta;\ lexp_1 \Downarrow_e \langle ty_1,\ C_1,\ u_1\rangle \qquad \Delta;\ lexp_2 \Downarrow_e \langle ty_2,\ C_2,\ u_2\rangle \qquad C_0 = \{ty_1 \stackrel{l}{=} a\ \underline{array},\ ty_2 \stackrel{l}{=} \underline{int}\} \qquad a \text{ fresh}}{\Delta;\ (lexp_1\texttt{.(}lexp_2\texttt{)})^l \Downarrow_e \langle a,\ C_0 \cup C_1 \cup C_2,\ u_1 \cup u_2\rangle}$$

While

$$\frac{\Delta;\ lexp_i \Downarrow_e \langle ty_i,\ C_i,\ u_i\rangle \text{ for } i = 1,2 \qquad C_0 = \{ty_1 \stackrel{l}{=} \underline{bool},\ ty_2 \stackrel{l}{\longleftrightarrow} \underline{unit},\ a \stackrel{l}{=} \underline{unit}\} \qquad a \text{ fresh}}{\Delta;\ (\texttt{while}\ lexp_1\ \texttt{do}\ lexp_2\ \texttt{done})^l \Downarrow_e \langle a,\ C_0 \cup C_1 \cup C_2,\ u_1 \cup u_2\rangle}$$

For

$$\frac{\Delta;\ lexp_i \Downarrow_e \langle ty_i,\ C_i,\ u_i\rangle \text{ for } i = 1,2 \qquad \Delta' = \Delta|_{\mathrm{id}}\left[var \mapsto \langle \underline{a_{var}},\ \mathsf{mono},\ \{a_{var} \stackrel{l}{=} \underline{int}\}\rangle^l\right] \qquad \Delta';\ lexp_3 \Downarrow_e \langle ty_3,\ C_3,\ u_3\rangle \qquad C_0 = \{a \stackrel{l}{=} \underline{unit},\ ty_1 \stackrel{l}{=} int,\ ty_2 \stackrel{l}{=} \underline{int},\ ty_3 \stackrel{l}{\longleftrightarrow} \underline{unit}\} \qquad a, a_{var} \text{ fresh}}{\Delta;\ (\texttt{for}\ var\ \texttt{=}\ lexp_1\ \texttt{to/downto}\ lexp_2\ \texttt{do}\ lexp_3\ \texttt{done})^l \Downarrow_e \langle a,\ C_0 \cup C_1 \cup C_2,\ u_1 \cup u_2\rangle}$$

Sequence

$$\frac{\Delta;\ lexp_i \Downarrow_e \langle ty_i,\ C_i,\ u_i\rangle \text{ for } i = 1,2 \qquad C_0 = \{a \stackrel{l}{=} ty_2,\ ty_1 \stackrel{l}{\longleftrightarrow} \underline{unit}\}}{\Delta;\ (lexp_1\texttt{;}\ lexp_2)^l \Downarrow_e \langle a,\ C_0 \cup C_1 \cup C_2,\ u_1 \cup u_2\rangle}$$

Raise

$$\frac{\Delta;\ lexp \Downarrow_e \langle ty,\ C,\ u\rangle \qquad a \text{ fresh}}{\Delta;\ (\texttt{raise}\ lexp)^l \Downarrow_e \langle a,\ C \cup \{ty \stackrel{l}{=} \underline{exc}\},\ u\rangle}$$

Try

$$\frac{\Delta;\ lexp \Downarrow_e \langle ty,\ C_1,\ u_1\rangle \qquad \Delta;\ rules \Downarrow_r \langle ty_p,\ ty_e,\ C_2,\ u_2\rangle \qquad C_0 = \{ty_p \stackrel{l}{=} \underline{exc},\ a \stackrel{l}{=} ty,\ a \stackrel{l}{=} ty_e\} \qquad a \text{ fresh}}{\Delta;\ (\texttt{try}\ lexp\ \texttt{with}\ rules)^l \Downarrow_e \langle a,\ C_0 \cup C_1 \cup C_2,\ u_1 \cup u_2\rangle}$$

ASSERT

$$\frac{\Delta;\ lexp \Downarrow_e \langle ty,\ C_1,\ u\rangle \qquad C_2 = \{ty \overset{l}{=} \underline{bool},\ a \overset{l}{=} \underline{unit}\} \qquad a \text{ fresh}}{\Delta;\ (\texttt{assert}\ lexp)^l \Downarrow_e \langle a,\ \emptyset,\ \emptyset\rangle}$$

$$\frac{}{\Delta;\ (\texttt{assert false})^l \Downarrow_e \langle a,\ \emptyset,\ \emptyset\rangle}$$

TYPE-ANNOT

$$\frac{\begin{array}{c}\Delta;\ lexp \Downarrow_e \langle ty,\ C_0,\ u\rangle \qquad C_1 = \{a \overset{l}{=} b,\ b \overset{l'}{=} ty',\ ty \overset{l}{=} c,\ c \overset{l'}{=} ty',\ a \succcurlyeq^l ct\} \\ a,\ b,\ c \text{ fresh} \qquad ty' \text{ is a fresh instance of } ct\end{array}}{\Delta;\ (lexp : ct^{l'})^l \Downarrow_e \langle a,\ C_0 \cup C_1,\ u\rangle}$$

10.4 Patterns

WILDCARD

$$\frac{a \text{ fresh}}{\Delta;\ \_ \Downarrow_p \langle a,\ \emptyset,\ \emptyset\rangle}$$

VARIABLE

$$\frac{a \text{ fresh}}{\Delta;\ x^l \Downarrow_p \langle a,\ \emptyset,\ \{x \mapsto \langle a\rangle^l\}\rangle}$$

INT

$$\frac{C_0 = \{int \overset{l}{=} a\} \qquad a \text{ fresh}}{\Delta;\ n^l \Downarrow_p \langle a,\ C_0,\ \emptyset\rangle}$$

TUPLE

$$\frac{\begin{array}{c}\Delta;\ pat_i \Downarrow_p \langle ty_i,\ C_i,\ b_i\rangle \text{ for } i = 1, \ldots, n \\ C_0 = \{a \overset{l}{=} (ty_1, \ldots, ty_n)\} \qquad \mathrm{dom}(b_i) \cap \mathrm{dom}(b_j) = \emptyset \text{ for } i \neq j \qquad a \text{ fresh}\end{array}}{\Delta;\ (pat_1\texttt{,}\ \ldots\texttt{,}\ pat_n)^l \Downarrow_p \langle a,\ \bigcup_{i=0}^{n} C_i,\ \bigcup_{i=1}^{n} b_i\rangle}$$

Variant

$$\frac{\Delta|_{\mathrm{var}}(K) = \langle ty_r,\ [ty_{a,1},\ \ldots,\ ty_{a,n}] \qquad \Delta;\ pat_i \Downarrow_p \langle ty_i,\ C_i,\ b_i\rangle \text{ for } i = 1,\ldots,n \quad C_0 = \{a \stackrel{l}{=} ty_r\} \cup \{ty_i \stackrel{l}{=} ty_{a,i} \mid i = 1,\ldots,n\} \qquad \mathrm{dom}(b_i) \cap \mathrm{dom}(b_j) = \emptyset \text{ for } i \neq j}{\Delta;\ (K\ pat_1\ \ldots\ pat_n)^l \Downarrow_p \langle a,\ \bigcup_{i=0}^{n} C_i,\ \bigcup_{i=1}^{n} b_i\rangle}$$

Record

$$\frac{\Delta|_{rec}(f_i) = \langle ty_{r,i},\ ty_{f,i}\rangle \qquad \Delta;\ pat_i \Downarrow_p \langle ty_i,\ C_i,\ b_i\rangle \quad \text{for } i = 1,\ldots,n \qquad C_0 = \{a \stackrel{l}{=} ty_{r,i},\ ty_i \stackrel{l}{=} ty_{f,i} \mid \text{ for } i = 1,\ldots,n\} \quad \mathrm{dom}(b_i) \cap \mathrm{dom}(b_j) = \emptyset \text{ for } i \neq j \qquad a \text{ fresh}}{\Delta;\ \{f_1\texttt{=}pat_1\texttt{;}\ \ \ldots\texttt{;}\ \ f_n\texttt{=}pat_n\}^l \Downarrow_p \langle a,\ \bigcup_{i=0}^{n} C_i,\ \bigcup_{i=1}^{n} b_i\rangle}$$

Or

$$\frac{\Delta;\ pat_i \Downarrow_p \langle ty_i,\ C_i,\ b_i\rangle \text{ for } i = 1,2 \quad \mathrm{dom}(b_1) = \mathrm{dom}(b_2) \qquad b = \{id \mapsto \langle a_{id}\rangle^l \mid id \in \mathrm{dom}(b_1)\} \qquad C_0 = \{a \stackrel{l}{=} ty_1,\ a \stackrel{l}{=} ty_2\} \quad C'_i = \{a_{id} \stackrel{l'}{=} ty \mid id \mapsto \langle ty\rangle^{l'} \in b_i\} \cup C_i \text{ for } i = 1,2 \qquad a,\ a_{id} \text{ fresh for } id \in \mathrm{dom}(b_1)}{\Delta;\ (pat_1\ \texttt{|}\ pat_2)^l \Downarrow_p \langle a,\ C_0 \cup C'_1 \cup C'_2,\ b\rangle}$$

Alias

$$\frac{\Delta;\ pat \Downarrow_p \langle ty,\ C,\ b\rangle \qquad a \text{ fresh}}{\Delta;\ (pat\ \texttt{as}\ x)^l \Downarrow_p \langle a,\ \{a \stackrel{l}{=} ty\} \cup C,\ b[x \mapsto \langle ty\rangle]\rangle}$$

Type-Annot

$$\frac{\Delta;\ pat \Downarrow_p \langle ty,\ C_0,\ b\rangle \qquad C_1 = \{a \stackrel{l}{=} b,\ b \stackrel{l'}{=} ty',\ ty \stackrel{l}{=} c,\ c \stackrel{l'}{=} ty',\ a \not\succeq^l ct\} \quad a,\ b,\ c \text{ fresh} \qquad ty' \text{ fresh instance of } ct}{\Delta;\ (pat : ct^{l'})^l \Downarrow_p \langle a,\ C_0 \cup C_1,\ b\rangle}$$

References

Felleisen, M., Findler, R. B., Flatt, M., and Krishnamurthi, S. (1998). The Drscheme Project: An Overview. *SIGPLAN Notices*, 33:17–23.

Haack, C. and Wells, J. B. (2004). Abstract type error slicing in implicitly typed higher-order languages.

Heeren, B., Leijen, D., and van IJzendoorn, A. (2003). Helium, for learning Haskell. In *ACM Sigplan 2003 Haskell Workshop*, pages 62 – 71, New York. ACM Press.

Leroy, X., Doligez, D., Garrigue, J., Rémy, D., and Vouillon, J. (2008). The objective caml system release 3.11. `http://caml.inria.fr/pub/docs/manual-ocaml/index.html`.

Milner, R. (1978). A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375.

Stolpmann, G. (2008). Findlib. `http://projects.camlcity.org/projects/findlib.html`.