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 without some practice error messages are sometimes hard to understand. 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 are loaded at runtime to produce output for different settings. Error messages are internationalized. Furthermore, EasyOCaml adds language levels and teach packs to make the language a good choice for teaching programming. This is completely integrated into the original OCaml system.

This report describes the ideas as well as the formal and notional foundations of EasyOCaml. It serves as an introduction and manual for users and as a guide to its principles and implementation for 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 language of the ML family with a powerful and sound type system. Its main implementation[1] 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 EasyOCaml are in large to make OCaml a programming language better suited for beginners and for teaching 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 and 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`. The website features an online demo of EasyOCaml's type inference system and some language levels.

Similar Projects

There are some projects which heavily influenced our work:

Haack and Wells (2003) 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.

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 built for teaching programming. Beside the integration of the editor and the REPL as well as an easy-to-use debugger (*stepper*), DrScheme has first introduced the concept of language levels and teach packs to restrict the syntax and to widen functionality especially for exercises.

[1] `http://caml.inria.fr`

Here is the way EasyOCaml puts together those elements: Its type checker is an extension of Haack and Wells (2003)'s algorithm to a significant subset of OCaml. It is fairly similar to DrScheme in scope, as it features language levels and teach packs for usage in programming courses. Although not yet implemented, EasyOCaml can build the basis for a tutoring system like Helium by providing more information to give advises in case of an error.

Goals Of This Report

This report is split in a rather "narrative" 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?*.

The report has several goals: First to present EasyOCaml and the concepts used in and developed for it (target audience: the users). Beside general informations in section 2 it covers the implemented type inference algorithm in section 3, error handling in section 4 and language levels in section 5. Then, it describes the usage of the implemented programs `ecaml` and `ecamlc` in section 7 by means of a manual. The grammar of EasyOCaml's language is given in rail road diagrams in section 8. Afterwards formal foundations for the type checker are given the section 9 (target audience: the interested readers). Finally, section 10 combines the concepts of the first sections with the actual implementation and describes the scheme of the code (target audience: the developers).

2 General Functionality of EasyOCaml

This section covers some general information on EasyOCaml's architecture and its language for orientation in subsequent sections. It will give a broad overview on the sequence plan. Every step is elaborated in more detail in subsequent sections, but they might be easier to understand with the knowledge of their role within EasyOCaml. This section ends with a description of the supported language.

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, both programs parse the command line flags. On recognizing the `-easy` flag, EasyOCaml's type checker is enabled, flags for defining language levels, teach packs and error printers are accepted and the according code is loaded 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 available syntax. The resulting abstract syntax tree (AST) is annotated with fresh type variables for the next stage.

EasyOCaml then tries to infer the types in the program. It reads the definitions in the program and generates a set of constraints on the types of every term 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 9. 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 passes control back to OCaml[2]. 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 significant 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*. The options for each syntactic category which will be described in the following and can be pruned by language levels in high detail, as given in section 5. See section 8 for the 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*.
- optionally recursive (`rec`) and multiple (`and`) *value declarations*, where bindings can occur in arbitrary patterns for non-recursive declarations and identifiers otherwise. Language levels may enforce mandatory type annotations for toplevel value declarations.
- *Toplevel evaluations*. Note that a toplevel evaluation `e` is just syntactic sugar for the dummy value declaration `let _ = e`.

Core types

Direct combinations of existing types are called *core types* 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.

[2]Note that up to now the program is type checked again by OCaml and the results are compared to validate our type checker. This might change in upcoming versions of EasyOCaml.

Expressions

Expressions are parts of a program which can be evaluated to a OCaml value and occur only as part of a structure item. 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 at every site where it works in OCaml, i.e. in value matching, in variable bindings, in functional abstractions with the `function` keyword, and exception catching.

Pattern matching works on every possible value in EasyOCaml, i.e. primitive values, tuples, variants and records[3], and can be nested at every level.

[3]The current version of EasyOCaml still 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 Damas and Milner (1982) at its core. Although very efficient for most programs and broadly extended to OCaml's requirements, it lacks sort of a memory: Inferring 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 when 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. Thus much work while debugging type errors in OCaml comprises of manually searching for other usages of the mis-typed variable in the program which might have lead to the type constructor clash.

This section first gives a loose description of Haack and Wells's type inference algorithm which covers the very problems just mentioned —see their 2003 paper for a rigid explanation. It explains the extensions we made for EasyOCaml afterwards.

3.1 Haack & Wells's Type Checking Algorithm

Haack and Wells (2003) describe an algorithm which exceeds algorithm $\mathcal{W}$ in two ways: First, 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. Second, 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, the original algorithm $\mathcal{W}$ deals with two things at once while traversing the AST: It generates information on the types of the current variables and unifies it with existing type information anon. Haack and Wells' algorithm works in some sense by separating those steps.

During *constraint generation* every node of the AST is first 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 contains 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 targets to find as many type errors as possible. This is basically done by systematically removing constraints grounded at one program location from the constraint set and running unification again such that the conflict continues to exist. Haack and Wells (2003) also present an iterative version of this algorithm which is implemented in EasyOCaml. Although it avoids repeated computation 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 actually caused the type error). By application of error minimization to each error, the algorithm further guarantees *minimality* of the reported errors, in the sense that removing a single constraint would vanish the constraint conflict, i.e. the error itself. So, the reported type error contains exactly those locations of the program which have lead to the error.

Contrary to this approach, other type checkers report whole subtrees of the program as the reason of an error. This often yields to non-minimality of the reported error as the subtree might contain locations which do not contribute to the error. To correctly display only the minimal (and complete) set of program points, Haack and Wells use *error slices*: All subtrees of the program, which do not contribute to the error are pruned and replaced by ellipsises, leaving only a scaffold of the program consisting of the blamed program points.

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 during type inference to avoid an artificial type error. But they are reported after unifying the constraints or with the type errors after error enumeration.

3.2 Extensions for EasyOCaml

Haack and Wells (2003) describe the constraint generation rules for *MiniML*, a subset of the *ML* language supporting only 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 its much richer type system. This section will describe the constraint generation rules for EasyOCaml by example here, section 9 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

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

where Δ denotes the environment 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 the set of constraints collected in *strit*. u is a set of errors in *strit* which are described in more detail in section 4.1. Those declarations, constraints and errors are accumulated while traversing the program's structure items.

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

Variant Decl

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

It just extends the current environment Δ with information on 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. This information is available in $\Delta|_{\text{var}}$ during constraint generation of the subsequent program code and can be used to assert the proper argument and result types of variant constructors.

Constraint generation rules for expressions are a better example for the process of 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. This is implemented in the following rule:

If-then

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

Constraint generation is applied here to $lexp_1$ (and exp_2 respectively), resulting in the type ty_1 (and ty_2) which is in fact a type variable constrained to the result type of $lexp_1$ in C_1 (and $lexp_2$ in C_2). The rule generates three additional constraints. The first one, $ty_1 \overset{l}{=} \underline{bool}$, asserts the result type ty_1 of expression $lexp_1$ to be of type $\underline{bool}$. The second, $ty_2 \overset{l}{=} \underline{unit}$ asserts the result type ty_2 of the expression $lexp_1$ to be of type $\underline{unit}$. The third one, $ty_3 \overset{l}{=} \underline{unit}$, asserts a freshly generated type variable ty_3, which represents the type of the whole conditional expression, to be of type $\underline{unit}$.

Note, that all new constraints are annotated with the location l of the overall expression. This facilitates the accusation 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.

A more particular feature of EasyOCaml are type annotations of the form $(lexp : ct)$. Special considerations are necessary for them: OCaml's current type checker ignores the type annotations during unification by using the expression $lexp$'s inferred type for further type checking and only checks its validity as to ct afterwards. 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.

Type-Annot

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

A freshly generated type variable a represents the type of the overall expression. It is constrained to a fresh instance of the annotated type. The constrain $ty \succcurlyeq^l ct$ captures the validity of the annotated expression's type ty with respect to the annotated type ct to catch up on after constraint unification. That way, EasyOCaml assumes the programmer's annotation to be valid and meaningful during type inference of the context of the overall expression and checks for a contradiction to the expression's type against the annotation afterwards.

Constraint generation rules for patterns have the form

$$\Delta;\ pat \Downarrow_p \langle ty,\ C,\ b\rangle$$

and provide the type of the values that can be matched with it as well as a set of constraints on it and all nested patterns. But contrary to constraint generation rules for expressions, all occurring variables are provided as bound to given parts of the matched value. b is a mapping of variables in the pattern to their according type variable constrained by C. Note that constraint generation rules for compound patterns like

Tuple

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

assert the sets of bound variables for sub-patterns to be mutually exclusive.

Additionally, there is a kind of auxiliary constraint generation rules:

$$\Delta;\ pat\ \texttt{->}\ lexp \mid rules \Downarrow_r \langle ty_p,\ ty_e,\ C,\ u\rangle.$$

This covers the mapping of patterns to expressions as used in value matching, functional abstraction with the keyword `function` and exception catching to deal with different variants of a value. The rule arranges the bound variables of the patterns to be available while generating constraints on the right hand side expressions and further generates constraints on the equality of the pattern's as well as the expression's types.

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

4 Errors and Error Reporting Adaptability

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

First, errors should provide a right amount of details: Needless to say that too few information are insufficient. But too much information can be just as confusing. So, for example in type constructor clashes exactly those locations of the program should be reported which are essential to the error. Delivering the right amount of information on the underlying reasons of a type error is exactly what EasyOCaml's type checker is made for.

Second, error reporting should be adaptable: Reading errors in a foreign language can distract a beginner or even prevent him from understanding it. So internationalization of error messages is necessary. Furthermore, the error output should be adaptable in its overall structure to serve for different kinds of presentation, e.g. on command line, or in a web browser.

The last section has explained the improvements of EasyOCaml to the type error messages. This section will first 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, and then the error adaption possibilities.

4.1 The Structure of Errors in EasyOCaml

While parsing and type checking a program, EasyOCaml can detect different errors. Those errors can be separated into three classes differing by the stage of the compilation process where they occur and in the impact on the subsequent compilation process. This section describes the error classes and gives some examples for the participating errors. The file `ezyErrors.mli` provides a complete and well documented list of the errors.

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, i.e. after constraint unification. So type errors (type constructor clashes, clashes of the arities of tuples and circular types) are quite natural of this kind. Furthermore, unbound variables are admittedly detected during constraint generation but provided with free type variables, such that constraint unification is possible without introducing any artificial errors. Other light errors like the attempt to change the value of immutable record fields are reported after constraint unification, too, as well as invalid type annotations.

Heavy errors are collected during and reported directly after constraint generation, because they prohibit even a transitionally type assignment to the corresponding term. They include among others: Incoherent record constructions (e.g. fields from different records or several bindings of a single field), several bindings of the same variable in a pattern, the usage of unknown variant constructors or the 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 immediately because they prevent any reasonable continuation of the compilation process.

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

4.2 New Errors for Camlp4

We have chosen Camlp4 as the parser generator for EasyOCaml because it is the only system known to the authors that facilitates manipulation of the grammar at runtime: The whole grammar is defined as an OCaml program which dynamically generates an OCaml stream parser (Pouillard, 2007). Thus, it is possible to modify the grammar by deleting certain rules at runtime. We use this in the first place to prune a full-fledged OCaml parser which is shipped with Camlp4 to Caml$_{-m}$ and then to implement the subset of the language specified by the language level.

But the foundation of Camlp4 as an OCaml stream parser yields to another problem: Stream parsers allow only a single exception type for passing parsing errors (internally and in the user interface). This exception can contain just a string for information on the detected error (`exception Stream.Error of string`). Camlp4's actual phrasing of the errors is hard-coded deep in the parser code in the English language which prevents the internationalization and format adaption necessary for EasyOCaml.

Nevertheless, we supplied Camlp4 with a new error reporting system, up to now just to make error reporting adaptable. But now it should be possible to augment the information of parsing errors with more information on the state of the parser.

Errors are represented in a variant type distinguishing the following forms of parsing errors.

`Expected (entry, opt_before, context)` describes the most common error when the parser recognizes a certain syntactic category in the program which does not match the category given by the grammar. `entry` describes the expected category, `opt_before` optionally describes the category of the entry just parsed and `context` denotes the category of the phrase which contains the questionable entry.

`Illegal_begin sym` is raised when the parser is not even able to parse the program's toplevel category denoted by `sym`.

`Failed` is raised only in `Camlp4.Struct.Grammar.Fold` and included in the error type just for consistency.

`Specific_error err` is raised for language-specific, "artificial" errors. Every grammar is parameterized on the type of `err` in our fork of Camlp4. This type covers conflicts with the language which can not be expressed by the Camlp4 grammar but are checked in plain OCaml code while parsing. This includes three errors for OCaml: Currified constructor, errors forcing an expression to be an identifier and bad directives on the REPL (the corresponding error is given in `Camlp4.Sig.OCamlSpecificError`).

But 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 with format "`<msg>\000<mrsh>`" where `<msg>` is the usual Camlp4 error message and `<mrsh>` is the marshaled version of the parsing error as just described. This string is decomposed again in Camlp4's interface function for parsing[4], and reported as a parsing error to the user.

So Camlp4's parsing errors are now represented in structured data to apply EasyOCaml's error reporting adaptabilities to them and to further extend them with more information on the parser's state.

4.3 Adaptability

For internationalization of error messages and adjusting the structure of the error messages to different display settings, EasyOCaml provides two possibilities to adapt the error reporting.

First, the language of the error messages is chosen via an environment variable. The functions for phrasing the errors in different languages are part of EasyOCaml's error system. This has two reasons. On the one hand, OCaml's format adaption does not capture different orders of the arguments. This has forced us to provide this functionality in OCaml program code. On the other hand, keeping the actual phrasing in the mainstream code suggests the error format plugins to use common phrases for the messages. This simplifies comparisons of errors under different display settings. See section 7.2 for more details on choosing the language.

Second, the format of the error messages can be adapted by so-called "error reporting plugins". A plugin consists of OCaml code which defines arbitrary error formatting functions and registers them in EasyOCaml's error system. As mentioned, those functions should use the provided functions for phrasing the errors to enforce conformity. But they can print them in any structure. Currently, a plain text formatter is the default for the usage on a command line. A HTML printer which highlights the locations of an error in the source code with funny colors is delivered with EasyOCaml for clear display with a web browser, and a XML/Sexp printer for future usage in an IDE. See section 7.4 for more details on providing custom error reporting plugins.

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

5 Language Levels and Teachpacks

Language levels and teach packs are didactic tools to simplify a language and its handling for beginners. They were introduced by DrScheme (Felleisen et al., 1998) with two goals in mind: First, they split the Scheme language into a "tower of languages"—that is different levels of syntactic and semantic richness. And language levels can specify the available language. Second, language levels and teach packs can provide arbitrary code to the user in a simple and encapsulated way. In contrast, teach packs contain only additional code but can extend a currently active language level.

[4]function `Camlp4.Struct.Grammar.Entry.action_parse`

Language levels and teach pack are a mean to adapt the language to the knowledge of students at a dedicated state of the class: Reasonable levels of the tower of languages are e.g. a first-order functional language, a higher-order functional language or a language with imperative features and mutable date.

To use a full-fledged parser and to check for the syntactical restraints afterwards would facilitate syntax errors regarding syntactical categories not included in the current language level—very confusing for beginners. EasyOCaml, however, directly manipulates the parser to the requirements of the language level. The parser itself does not even "know" about the syntactical categories not part of the language level and does not report errors regarding to them in the sequel.

EasyOCaml further grants fine grained control over the available syntax. Every option for non-terminal nodes in the grammar (structure items, type declarations expressions, patterns) can be switched on and off. This can be even done independently for different usages, e.g. the patterns in value matching can be configured in a different way than the patterns in functional abstractions. This is implemented by deleting the minimal common disabled set of options from the Camlp4 grammar. At a second stage (namely while transforming the OCaml AST into EasyOCaml's), all remaining restrictions are examined.

As mentioned, language levels and teach packs can be used to provide code at start-up time of the compiler and toplevel loop in a simple manner. They contain a list of modules which ought to be available to the user. Each goes with an annotation if they are opened at start-up time. Section 7.3 shows how to define custom language levels and teach packs.

The user can specify which language level and teach pack to use by some command line parameter as described in section 7.1. EasyOCaml then searches for it in a dedicated directory as described in section 7.5.

6 Forecast and Conclusion

EasyOCaml in version 0.5 is not yet completely ready for action. For version 1.0, we will add installation procedures for language levels and teach packs. 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 current variable environment to unbound variable errors to check misspellings). 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.

Acknowledgments
This project was funded by the *OCaml Summer Project 2008*. So thanks to the people at Jane Street Capital for this initiative and to make EasyOCaml possible! I am deeply grateful to Prof. Peter Thiemann and his team—especially Stefan Wehr—for mentoring and guiding me through the course of 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

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

`-easyerrorprinter <printer>` Loads an error formatting plugin into the compiler. This must be an OCaml object that calls functions as given in section 7.4.

`-easylevel <level>` Enables a language level named `<level>` which is installed in a configuration directory (see section 7.5).

`-easyteachpack <teachpack>` Enables a teach pack named `<teachpack>` which is installed in a configuration directory (see section 7.5).

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).

`EASYOCAML_ONLY_TYPECHECK` Exit the compiler after type checking without any code generation.

`LANG, LANGUAGE` Controls internationalization of error messages. Possible values: “en”, “fr”, “de”.

`EASYOCAML_LOGLEVEL` Controls logging details for debugging. Possible values are “error”, “warn”, “info”, “debug”, and “trace”.

`EASYOCAML_GLOBAL_DIR` Specifies the global configuration directory as described below.

`EASYOCAML_USER_DIR` Specifies the user configuration directory as described below.

7.3 Defining Languge Levels and Teach Packs

A language level must specify the syntax and the modules available to the user. This is done in EasyOCaml in the following way. Each language level consists of a directory in the configuration directory as described in section 7.5. The name of the directory defines the name of the language level. The directory must contain a file name `LANG_META.ml` which actually defines the language level. This is done by calling the function `EzyLangLevel.configure`. This function expects the specification of the available syntax and modules as arguments.

Syntactic features are given by record values for every syntactic category. Functions to generate minimal and maximal specifications are available for convenience.

The available modules are specified just by two lists. The first one is a list of OCaml module names paired with a boolean value designating if they are opened on start-up (like `Pervasives` is by default). The second constists of the file names which contain those modules.

Teach packs are defined in an appropriate directory as described in section 7.5 and must contain a file called `TEACHPACK_META.ml`. The actual teach pack in defined by a call to `EzyTeachpack.configure` which expects the modules descriptions in two lists like just described.

Note that this fashion to define language levels and teach packs might change in future versions of EasyOCaml. A plain text file is sufficient because OCaml code is necessary neither to specify the syntax nor to specify the available modules.

7.4 Defining Error Formatter

Error formatter for EasyOCaml are given in compiled OCaml code. They must call `EzyErrors.register` to register functions to print common, heavy and fatal errors. After compiling the code with access to EasyOCaml's code, the resulting object file can be loaden into the compiler at runtime with help of the `-easyerrorprinter` command line flag.

It is recommended to use the functions `print_errors`, `print_heavies` and `print_fatal` from the module `EzyErrors` to actually phrase the error and to define only the layout with the plugin. This is favorable, because the former functions are already internationalized and provide a common phrasing between the different error printing plugins.

7.5 The Configuration 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`

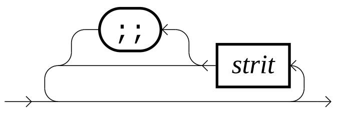
EasyOCaml's configuration directory must have the following structure:

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

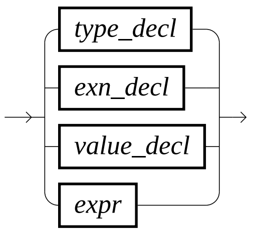
Each language level contains a module `LANG_META` which is loaded into EasyOCaml. Teach packs go with a module `TEACHPACK_META`.

8 Grammar of Caml$_{-m}$ in Rail Diagrams

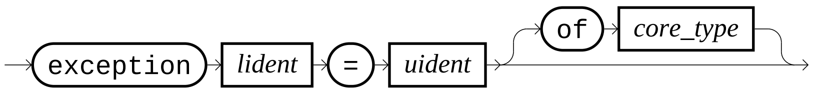
program



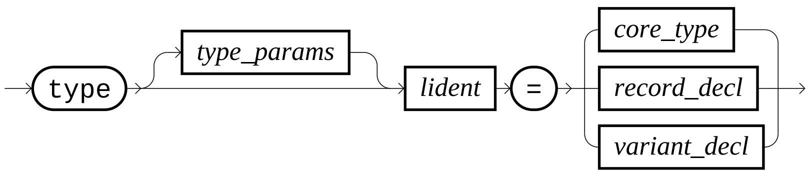
strit



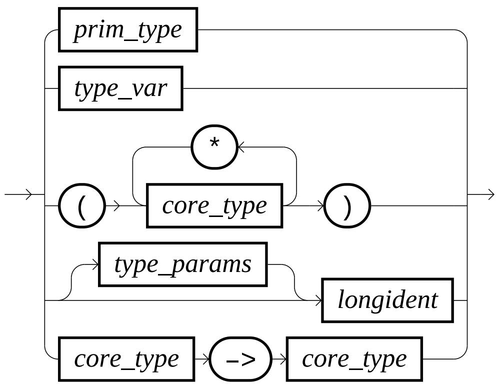
exn_decl



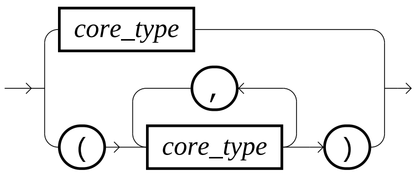
type_decl



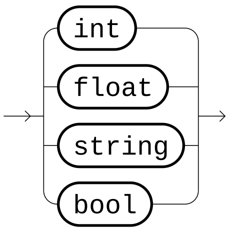
core_type



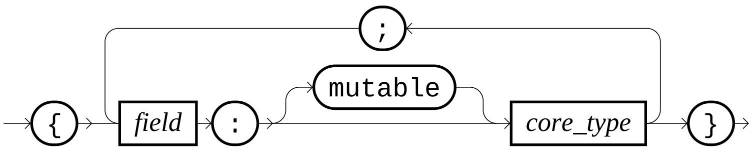
type_params



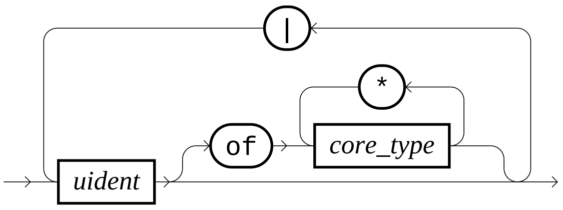
prim_type



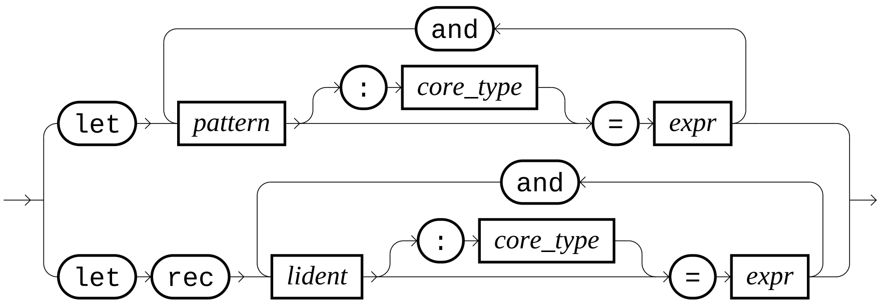
record_decl



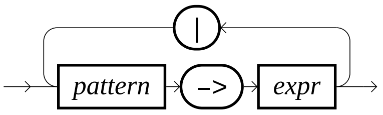
variant_decl



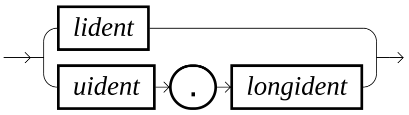
value_decl



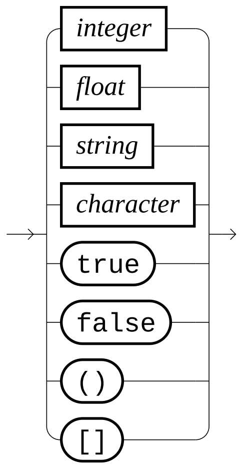
rules



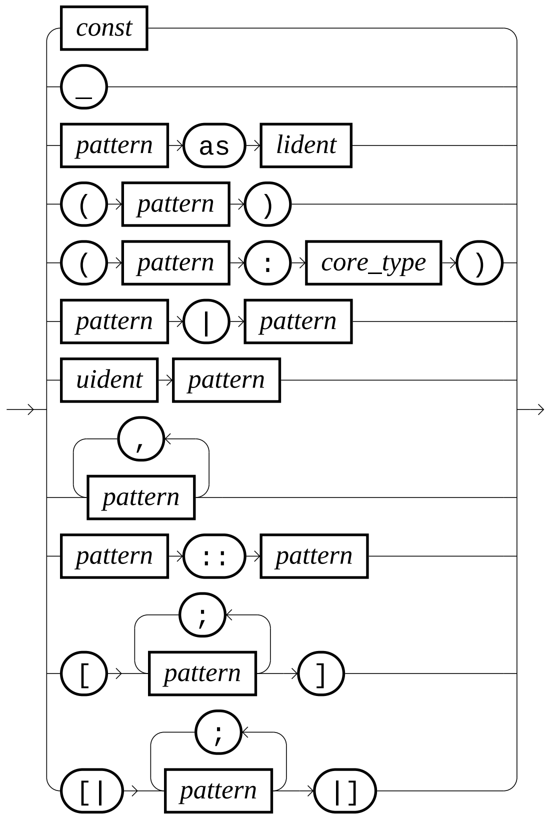
longident



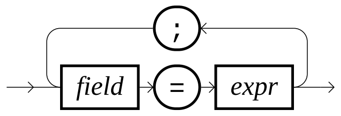
const



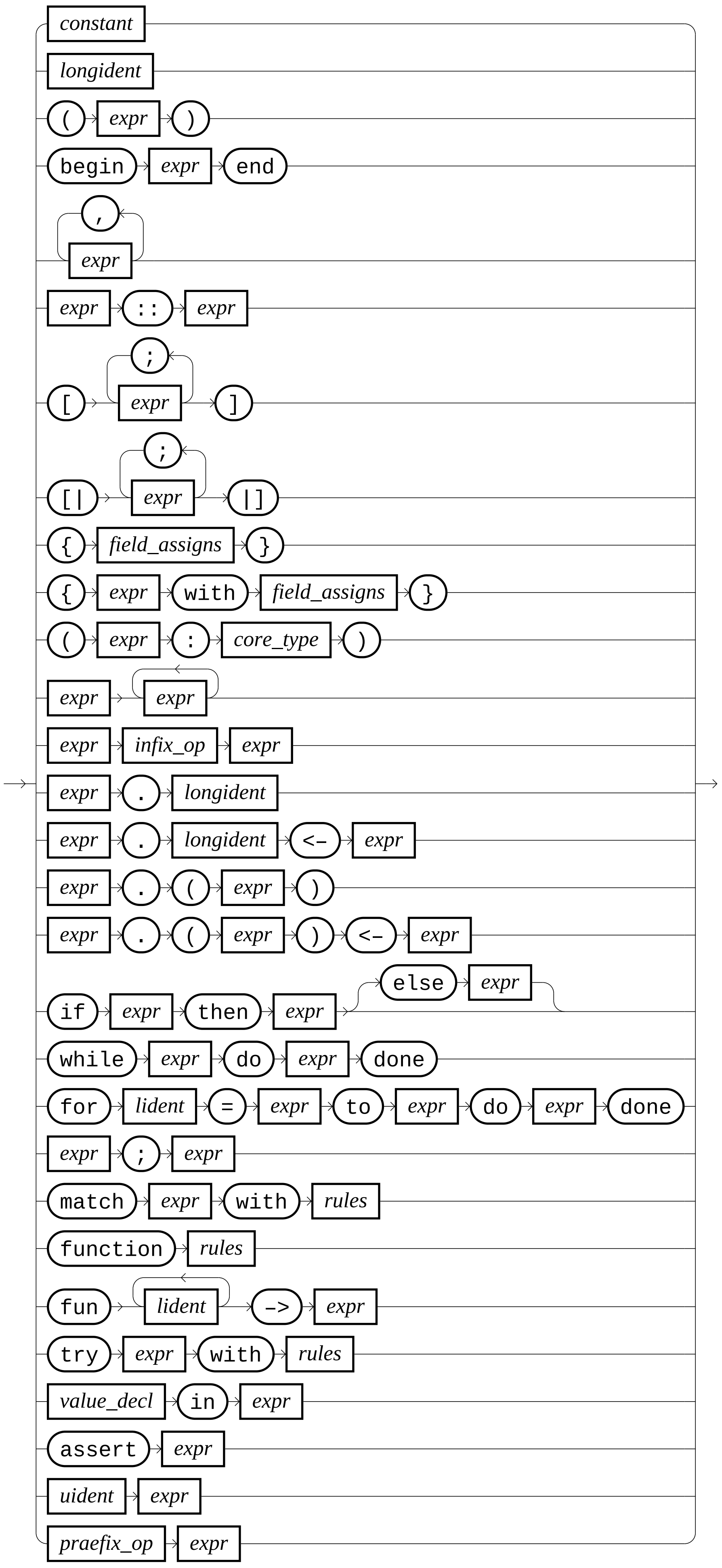
pattern



field_assigns



expr


constant
longident
(expr)
begin expr end
, expr
expr :: expr
[; expr]
[| ; expr |]
{ field_assigns }
{ expr with field_assigns }
(expr : core_type)
expr expr
expr infix_op expr
expr . longident
expr . longident <- expr
expr . (expr)
expr . (expr) <- expr
if expr then expr else expr
while expr do expr done
for lident = expr to expr do expr done
expr ; expr
match expr with rules
function rules
fun lident -> expr
try expr with rules
value_decl in expr
assert expr
uident expr
praefix_op expr

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

9.1 Structure items

Eval

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

Value decl

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

Rec value decl

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

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

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

9.2 Rules

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

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

9.3 Expressions

Var-Mono

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

Var-Poly

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

Var-Undef

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

Const

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

Abstr

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

App

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

Let

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

Let rec

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

Tuple

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

Record construction

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

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

Record Access

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

Record field assignment

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

Variant

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

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

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

If-then

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

Matching

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

Array construction

$$\frac{\Delta;\ \mathit{lexp}_i \Downarrow_e \langle \mathit{ty}_i,\ C_i,\ u_i\rangle \qquad C_0 = \{a \overset{l}{=} b\ \underline{\mathit{array}}\} \cup \{\mathit{ty}_i \overset{l}{=} b \mid i = 1,\ldots,n\} \quad a,\ b \text{ fresh}}{\Delta;\ (\texttt{[|}\mathit{lexp}_1\texttt{;}\ \ldots\texttt{;}\ \mathit{lexp}_n\texttt{|]})^l \Downarrow_e \langle a,\ \bigcup_{i=0}^{n} C_i,\ \bigcup_{i=1}^{n} u_i\rangle}$$

Array access

$$\frac{\Delta;\ \mathit{lexp}_1 \Downarrow_e \langle \mathit{ty}_1,\ C_1,\ u_1\rangle \qquad \Delta;\ \mathit{lexp}_2 \Downarrow_e \langle \mathit{ty}_2,\ C_2,\ u_2\rangle \qquad C_0 = \{\mathit{ty}_1 \overset{l}{=} a\ \underline{\mathit{array}},\ \mathit{ty}_2 \overset{l}{=} \underline{\mathit{int}}\} \quad a \text{ fresh}}{\Delta;\ (\mathit{lexp}_1\texttt{.(}\mathit{lexp}_2\texttt{)})^l \Downarrow_e \langle a,\ C_0 \cup C_1 \cup C_2,\ u_1 \cup u_2\rangle}$$

While

$$\frac{\Delta;\ \mathit{lexp}_i \Downarrow_e \langle \mathit{ty}_i,\ C_i,\ u_i\rangle \text{ for } i = 1,2 \qquad C_0 = \{\mathit{ty}_1 \overset{l}{=} \underline{\mathit{bool}},\ \mathit{ty}_2 \overset{l}{\longleftrightarrow} \underline{\mathit{unit}},\ a \overset{l}{=} \underline{\mathit{unit}}\} \quad a \text{ fresh}}{\Delta;\ (\texttt{while}\ \mathit{lexp}_1\ \texttt{do}\ \mathit{lexp}_2\ \texttt{done})^l \Downarrow_e \langle a,\ C_0 \cup C_1 \cup C_2,\ u_1 \cup u_2\rangle}$$

For

$$\frac{\begin{array}{c}\Delta;\ \mathit{lexp}_i \Downarrow_e \langle \mathit{ty}_i,\ C_i,\ u_i\rangle \text{ for } i = 1,2 \\ \Delta' = \Delta|_{\text{id}}\left[\mathit{var} \mapsto \langle \underline{a_{\mathit{var}}},\ \mathsf{mono},\ \{a_{\mathit{var}} \overset{l}{=} \underline{\mathit{int}}\}\rangle^l\right] \qquad \Delta';\ \mathit{lexp}_3 \Downarrow_e \langle \mathit{ty}_3,\ C_3,\ u_3\rangle \\ C_0 = \{a \overset{l}{=} \underline{\mathit{unit}},\ \mathit{ty}_1 \overset{l}{=} \mathit{int},\ \mathit{ty}_2 \overset{l}{=} \underline{\mathit{int}},\ \mathit{ty}_3 \overset{l}{\longleftrightarrow} \underline{\mathit{unit}}\} \quad a, a_{\mathit{var}} \text{ fresh}\end{array}}{\Delta;\ (\texttt{for}\ \mathit{var}\ \texttt{=}\ \mathit{lexp}_1\ \texttt{to/downto}\ \mathit{lexp}_2\ \texttt{do}\ \mathit{lexp}_3\ \texttt{done})^l \Downarrow_e \langle a,\ C_0 \cup C_1 \cup C_2,\ u_1 \cup u_2\rangle}$$

Sequence

$$\frac{\Delta;\ \mathit{lexp}_i \Downarrow_e \langle \mathit{ty}_i,\ C_i,\ u_i\rangle \text{ for } i = 1,2 \qquad C_0 = \{a \overset{l}{=} \mathit{ty}_2,\ \mathit{ty}_1 \overset{l}{\longleftrightarrow} \underline{\mathit{unit}}\}}{\Delta;\ (\mathit{lexp}_1\texttt{;}\ \mathit{lexp}_2)^l \Downarrow_e \langle a,\ C_0 \cup C_1 \cup C_2,\ u_1 \cup u_2\rangle}$$

Raise

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

Try

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

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

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

This notion of validity of an instantiation constraint is closely connected to value type enrichment as explained in Stefan Wehr's diploma thesis (Wehr, 2005, pages 15, 16).

9.4 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

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

Record

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

Or

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

Alias

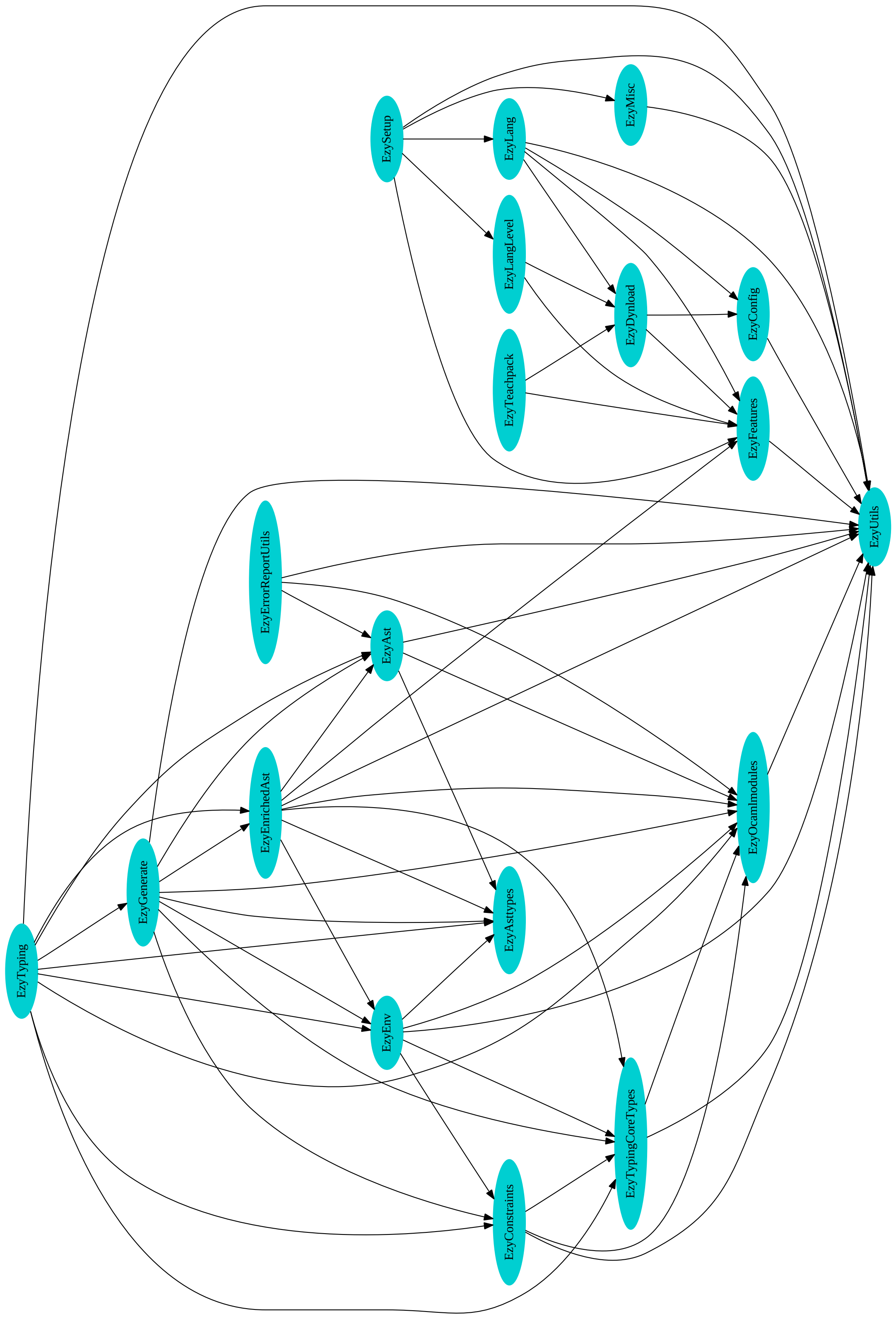
$$\frac{\Delta;\ pat \Downarrow_p \langle ty,\ C,\ b\rangle \qquad a \text{ fresh}}{\Delta;\ (pat\ \texttt{as}\ x)^l \Downarrow_p \langle a,\ \{a \overset{l}{=} ty\} \cup C,\ b[x \mapsto \langle ty\rangle]\rangle}$$

Type-Annot

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

10 Details of the Implementation

10.1 Dependency Graph of EasyOCaml's modules



10.2 Short Descriptions of the Modules

10.2.1 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 copies code 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 of these.

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

10.2.2 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 and Wells (2003)), i.e. slicing an AST to only contain nodes from locations given in a set, substituting the rest with ellipsis.

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

10.2.3 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.

10.2.4 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 an ellipsis 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 features specified by the language level.

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

10.2.5 Type Constraints

EzyTypingCoreTypes Contains base types for the constraints and their generation, closely related to the data described in Haack and Wells (2003) (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 and Wells (2003) is defined.

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

10.2.6 Typing

EzyTyping Unification of a set of constraints which yield a substitution on the variables and error enumeration and minimization as described by Haack and Wells (2003). It furthermore contains convenient 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`.

References

Damas, L. and R. Milner (1982). Principal Type-schemes for Functional Programs. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, New York, NY, USA, pp. 207–212. ACM.

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

Haack, C. and J. B. Wells (2003). Type Error Slicing in Implicitly Typed, Higher-order Languages. In *Sci. Comput. Programming*, pp. 284–301. Springer-Verlag.

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

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

Pouillard, N. (2007). Stream parser. `http://brion.inria.fr/gallium/index.php/Stream_Parser`.

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

Wehr, S. (2005, November). ML modules and Haskell type classes: A constructive comparison. Master's thesis, Albert-Ludwigs-Universität, Freiburg, Germany.