

Institut für Informatik  
Abteilung für Programmiersprachen  
Universität Freiburg

Studienarbeit

# **EasyOCaml: 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

<b>1</b>	<b>Objectives and Introduction</b>	<b>5</b>
1.1	Similar Projects . . . . .	5
<b>2</b>	<b>Supported Language</b>	<b>6</b>
<b>I</b>	<b>Concepts and design</b>	<b>8</b>
<b>3</b>	<b>Constraint Based Type Inference</b>	<b>8</b>
3.1	Haack & Wells . . . . .	8
3.2	Extensions for EasyOCaml . . . . .	9
<b>4</b>	<b>Errors and Error Reporting Adaptability</b>	<b>11</b>
4.1	The structure of errors in EasyOCaml . . . . .	11
4.2	New Errors for Camlp4 . . . . .	11
4.3	Adaptability . . . . .	12
<b>5</b>	<b>Language Levels and Teachpacks</b>	<b>13</b>
<b>6</b>	<b>Forecast and Conclusion</b>	<b>13</b>
<b>II</b>	<b>Manual, formalities and implementation</b>	<b>14</b>
<b>7</b>	<b>The user interface (manual)</b>	<b>14</b>
7.1	Command line parameters . . . . .	14
7.2	Environment variables . . . . .	14
7.3	The EasyOCaml directory . . . . .	14
<b>8</b>	<b>Grammar of Caml<sub>-m</sub></b>	<b>16</b>
<b>9</b>	<b>Details of the Implementation</b>	<b>17</b>
9.1	Outline of EasyOCaml's Pipeline . . . . .	17
9.2	Utilities and Miscellaneous . . . . .	17
9.3	Error Reporting . . . . .	18
9.4	Teachpacks and Language Levels . . . . .	18
9.5	Abstract Syntax Tree . . . . .	19
9.6	Type Constraints . . . . .	19
9.7	Typing . . . . .	19
9.8	Camlp4 . . . . .	20
<b>10</b>	<b>Typing Rules for EasyOCaml</b>	<b>21</b>
10.1	Structure items . . . . .	21

---

10.2 Rules . . . . .	22
10.3 Expressions . . . . .	23
10.4 Patterns . . . . .	27

# 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 toplevel 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 language 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 *EasyOCaml*’s 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 is split in a readable less formal part I (answering the question: *what?*) and a more formal part II which acts as a reference for the user and developer (answering the question: *how?*). 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).

## 2 Supported Language

*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 without module declarations”, hence its designation *Caml<sub>m</sub>*.

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 is 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 in *OCaml*.
- optionally recursive (`rec`) and multiple (`and`) *value declarations*, where binding occur with arbitrary patterns. Note, that a language level may require type annotations of those toplevel value declarations.
- Toplevel evaluations. Note that the toplevel evaluation of `e` is just syntactic sugar for `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 (in 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 that 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 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. value matching<sup>1</sup>, in variable bindings<sup>2</sup>, in functional abstractions with the **function** keyword<sup>3</sup> and exception catching<sup>4</sup>.

Pattern matching works for every possible value in *EasyOCaml*, i.e. primitive values, tuples, variants and records<sup>5</sup> and every can be nested in every level.

---

<sup>1</sup>`match ... with pat -> ... | ...`

<sup>2</sup>`let pat = ...`

<sup>3</sup>`function pat -> ... | ...`

<sup>4</sup>`try ... with pat -> ... | ...`

<sup>5</sup>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: For example, 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 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 reports only the location where the error became obvious to the type checker. 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.

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

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 it 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 type information on the variables and unifies it with existing type information anon. Haack & Wells' algorithm works by separating these steps.

During *constraint generation* every node of the AST gets annotated with a type variable. While traversing the AST, information on these 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 now, the collected type information is still available as a set of constraints and enables the algorithm to reexamine the errors in a second stage of *error enumeration* and *minimization*: Error enumeration is basically done by systematically removing constraints grounded at one program location from the constraint set and running unification again. 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 delimited in *EasyOCaml* to a given time amount which can be



specified by an environment variable (see 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 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.

In addition to type errors, Haack and Wells' technique also enables the type checker to collect unbound variables in the program while generating the constraints. Their types are assumed as a free type variable 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<sub>m</sub>* as loosely described in section 2 and more formally in section 8.

I will describe the constraint generation rules for *EasyOCaml* by example here, take a look into section 10 for the complete set of rules. In the following,  $\Delta$  always denotes an environment for bindings of variables, record fields and variant constructors anon, accessible by  $\Delta|_{\text{id}}$ ,  $\Delta|_{\text{var}}$ ,  $\Delta|_{\text{rec}}$  and  $\Delta|_{\text{var}}$  respectively.

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

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

$\Delta$  denotes the environment which contains declarations in the program so far, *strit* denotes the current structure item.  $\Delta'$  denotes the environment  $\Delta$  extended by declarations in *strit* and  $C$  is a set of constraints collected in *strit*. Those constraints declarations and constraints are accumulated while generating the constraints for the program's structure items. Finally,  $u$  is a set of errors in *strit* which are described in more detail in section 4.1.

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

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

It just extends the current environment  $\Delta$  with the variant constructors  $K_1$  to  $K_n$  with the given types. The locations are stored to make a reference on the type declaration in case of a typing error.

Constraint generation rules for expressions are better examples for accumulating constraint sets. Here is the rule for **if** expressions without an **else** branch as a simple starting point.

IF-THEN

$$\frac{\Delta; \text{lexp}_1 \Downarrow_e \langle ty_1, C_1, u_1 \rangle \quad \Delta; \text{lexp}_2 \Downarrow_e \langle ty_2, C_2, u_2 \rangle \quad C_0 = \{ty_1 \stackrel{l}{=} \underline{bool}, ty_2 \stackrel{l}{=} \underline{unit}, a \stackrel{l}{=} \underline{unit}\} \quad a \text{ fresh}}{\Delta; (\text{if } \text{lexp}_1 \text{ then } \text{lexp}_2)^l \Downarrow_e \langle a, C, u \rangle}$$

Constraint generation is applied here to  $\text{lexp}_1$  (and  $\text{exp}_2$  respectively), resulting in the type  $ty_1$  which is in fact a type variable constraint to the result type of  $\text{lexp}_1$  in  $C_1$ . This rule generates three additional constraints. The first one,  $ty_1 \stackrel{l}{=} \underline{bool}$ , asserts the result type of expression  $\text{lexp}_1$  to be of type  $\underline{bool}$ . The second,  $ty_2 \stackrel{l}{=} \underline{unit}$  asserts the result of the expression  $\text{lexp}_1$  to be of type  $\underline{unit}$  as required by a conditional without **else** branch. The third on,  $ty_3 \stackrel{l}{=} \underline{unit}$ , asserts that a freshly generated type variable  $ty_4$ , 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 conditional expression, in order to blame this expression in case of a type error. The constraint generation results in the freshly generated type variable  $ty_3$  and the union of all occurring constraint and error sets.

EasyOCaml features type annotations of the form  $(e:ct)$ . Special considerations are necessary for them: OCaml's current type checker ignores the type annotations whilst unification by using the term's inferred type and only checks its validity later on. So type inference and error reporting makes no use of the type annotations itself. In contrast, EasyOCaml assumes terms to have the denoted type while type checking, type checks the term isolated and tests the validity afterwards by proving that the annotated type is a subtype of the inferred type.

TYPE-ANNOT

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

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

*EasyOCaml* facilitates references and mutable data.

...

*EasyOCaml* collects as many errors as possible.

- records ...
- variants ...

- type annotations ...
- references ...

These examples should simplify understanding the complete list of inference rules used in *EasyOCaml* as given in section 10.

## 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:

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

Secondly, error reporting should be adaptable: For a beginner, reading errors in a foreign language can distract or even prevent him from understanding it. 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).

\* \*\*

I have explained the improvements of *EasyOCaml* to type error messages in the last section and 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*

### 4.2 New Errors for *Camlp4*

As mentioned, *EasyOCaml* parses its input with a *Camlp4* parser. Unfortunately, *Camlp4*'s error messages are hard coded in English and never represented in data. This is because *Camlp4* is a *OCaml* stream parser in its core, and this requires parsing errors to be reported as exceptions which only allow just a string for information<sup>6</sup>.

Nevertheless, we supplied *Camlp4* with a new error reporting system, up to now just to make error reporting adaptable, but it should be possible now to augment the information of parse errors with more information on the state of the parser. A parsing error<sup>7</sup> is one of the following:

---

<sup>6</sup>exception `Stream.Error` of `string`

<sup>7</sup>type `ParseError.t`

`Expected (entry, opt_before, context)` is raised if when the parser stuck while parsing a phrase: `entry` describe the categories of the possible, expected sub-phrases, `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 top categories described by `sym`.

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

`Specific_error err` Beside the generic parsing errors just mentioned, it is possible to extend the parsing errors per language by “artificial” errors which are specific to a language, e.g. curried 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, the parsing exception contains 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 the interface function for parsing<sup>8</sup>, and reported as a parsing error<sup>9</sup> to the user.

And so *Camlp4*’s parsing errors now represented in data to apply error reporting adaptabilities and to further extend it to contain more informations.

### 4.3 Adaptability

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.

<sup>8</sup>function `Camlp4.Struct.Grammar.Entry.action_parse`

<sup>9</sup>type `ParseError.t`

## 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/drsch> 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 support for module declarations to make EasyOCaml ready in our daily programming.

### Acknowledgements

This project is 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 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 described 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<sub>-m</sub>



## 9 Details of the Implementation

### 9.1 Outline of EasyOCaml's Pipeline

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

1. Command line flags are evaluated to check the “`-easy`” flag and an error printer and possibly load a teach pack and/or language levels. See section 7.1 for information on the command line flags.
2. `EzyCamlgrammar` parses the AST from the input, possibly respecting restrictions from the language level which yields an `EzyAst.imported_structure`.
3. `EzyGenerate` generates constraints from the AST (involving type information from the default environment and modules loaded by command line or teach packs/language levels). This yields a quadruple `generated_structure * AtConstrSet.t * PostProcess.t` where  
`generated_structure` is the AST annotated with type variables and unique identifiers.  
`AtConstrSet.t` is a set of constraints on the type variables in the AST.  
`PostProcess.t` is build gradually during constraint generation and contains sets of different types of errors (from `EzyErrors`) as well as checks which can only processed after constraint unification (i.e. type annotations)  
`EzyEnv.t` is used to keep track of local variables and after constraint generation contains information on the global types and values of the program.
4. `EzyTyping.solve` tries to solve the generated constrains. If solving succeeds, the program is typed by a substitution on the variables in the generated AST and the environment and contains type errors otherwise.
5. The last step, reimporting the `EzyEnrichedAst.generated_structure` with typing information given by type substitution to OCaml's original typed tree is not yet done. We type the code again with OCaml's original type checker and compare the result to verify its correctness.

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

### 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 Easy-OCaml'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 Typing 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  $\Delta; \text{lexp} \Downarrow_e \langle ty, C, u \rangle$
- for structure items  $\Delta; \text{strit} \Downarrow_s \langle \Delta, C, u \rangle$
- for rules  $\Delta; \text{pat} \rightarrow \text{lexp} \mid \text{rules} \Downarrow_r \langle ty_p, ty_e, C, u \rangle$  (just an auxiliary)
- for patterns  $\Delta; \text{pat} \Downarrow_p \langle ty, C, b \rangle$  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  $\Delta$  a general environment. A general environment  $\Delta$  encapsulates environments for lookup of the

- type of a variable:

$$\Delta|_{\text{id}}(lid) = \langle ty, \varpi, C \rangle^l$$

where  $\varpi \in \{\text{mono}, \text{poly}\}$  and  $lid$  has been bound accordingly.

- types of record fields:

$$\Delta|_{\text{rec}}(f) = \langle ty_r, ty_f, \mu \rangle$$

where  $ty'_f$  is the type of field  $f$  in record type  $ty'_r$  and  $ty_f, ty_r$  are fresh variants of  $ty'_f, ty'_r$ .  $\mu \in \{\text{mutable}, \text{immutable}\}$

- types of variants

$$\Delta|_{\text{var}}(K) = \langle ty_r, [ty_1, \dots, ty_n] \rangle$$

where  $ty'_1, \dots, ty'_n$  are the arguments for variant  $k$  of type  $ty'_r$  and  $ty_1, \dots, ty_n, ty_r$  are fresh variants of  $ty'_1, \dots, ty'_n, ty'_r$ .

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

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

### 10.1 Structure items

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

VALUE DECL

$$\begin{array}{c}
\Delta; \text{pat}_i \Downarrow_p \langle \text{ty}_{p,i}, C_{p,i}, b_i \rangle \quad \Delta; \text{exp}_i \Downarrow_e \langle \text{ty}_{e,i}, C_{e,i}, u_i \rangle \\
\varpi_i := \text{poly if value lexp}_i \text{ else mono} \quad C_{x,i} := C_{p,i} \cup C_{e,i} \cup \{\text{ty}_{p,i} \stackrel{l}{=} \text{ty}_{e,i}\} \\
\Delta' := \Delta|_{\text{id}}[b_i, C_{x,i}, \varpi_i \mid i = 1, \dots, n] \quad \text{dom}(b_i) \cap \text{dom}(b_j) = \emptyset \text{ for all } i \neq j \\
\hline
\Delta; (\text{let pat}_1 = \text{lexp}_n \text{ and } \dots \text{ and pat}_n = \text{lexp}_n)^l \Downarrow_s \langle \Delta', \bigcup_{i=1}^n C_{x,i}, \bigcup_{i=1}^n u_i \rangle
\end{array}$$

REC VALUE DECL

$$\begin{array}{c}
\Delta|_{\text{var}}[x_j \mapsto \langle a_j, \text{mono}, \emptyset \rangle^l \mid j = 1, \dots, n]; \text{lexp}_i \Downarrow_e \langle \text{ty}_i, C_i, u_i \rangle \\
\varpi_i := \text{poly if value lexp}_i \text{ else mono} \\
\text{for } i=1, \dots, n \quad \Delta' := \Delta|_{\text{var}}[x_i \mapsto \langle \text{ty}_i, \varpi_i, C_i \cup \{\text{ty}_i \stackrel{l}{=} a_i\} \rangle^l \mid \text{for } i = 1, \dots, n] \\
a_1, \dots, a_n \text{ fresh} \quad x_i = x_j \text{ iff } i = j \\
\hline
\Delta; (\text{let rec } x_1 = \text{lexp}_n \text{ and } \dots \text{ and } x_n = \text{lexp}_n)^l \Downarrow_s \langle \Delta', \bigcup_{i=1}^n C_{x,i}, \bigcup_{i=1}^n u_i \rangle
\end{array}$$

RECORD DECL

$$\begin{array}{c}
\Delta' = \Delta|_{\text{rec}}[t \mapsto \{\langle f_1, \text{ty}_1 \rangle^{l_1}, \dots, \langle f_n, \text{ty}_n \rangle^{l_n}\}^l] \\
\hline
\Delta; (\text{type } t = \{f_1 :^{l_1} \text{ty}_1; \dots; f_n :^{l_n} \text{ty}_n\})^l \Downarrow_s \langle \Delta', \emptyset, \emptyset \rangle
\end{array}$$

VARIANT DECL

$$\begin{array}{c}
\Delta' = \Delta|_{\text{var}}[t \mapsto \{\langle K_1, \text{ty}_1 \rangle^{l_1}, \dots, \langle K_n, \text{ty}_n \rangle^{l_n}\}^l] \\
\hline
\Delta; (\text{type } t = K_1 \text{ of}^{l_1} \text{ty}_1 \mid \dots \mid K_n \text{ of}^{l_n} \text{ty}_n)^l \Downarrow_s \langle \Delta', \emptyset, \emptyset \rangle
\end{array}$$

SEQUENCE

$$\begin{array}{c}
\Delta; \text{strit}_1 \Downarrow_s \langle \Delta', C_1, u_1 \rangle \quad \Delta'; \text{strit}_2 \Downarrow_s \langle \Delta'', C_2, u_2 \rangle \\
\hline
\Delta; \text{strit}_1 ;; \text{strit}_2 \Downarrow_s \langle \Delta'', C_1 \cup C_2, u_1 \cup u_2 \rangle
\end{array}$$

## 10.2 Rules

$$\begin{array}{c}
\Delta; \text{pat} \Downarrow_p \langle \text{ty}_p, C_p, b \rangle \quad \Delta|_{\text{id}}[b, C_p, \text{mono}]; \text{lexp} \Downarrow_e \langle \text{ty}_e, C_e, u \rangle \\
\hline
\Delta; \text{pat} \rightarrow \text{lexp} \Downarrow_r \langle \text{ty}_p, \text{ty}_e, C_p \cup C_e, u \rangle
\end{array}$$

$$\begin{array}{c}
\Delta; \text{pat} \rightarrow \text{lexp} \Downarrow_r \langle \text{ty}_{p,1}, \text{ty}_{e,1}, C_1, u_1 \rangle \quad \Delta; \text{rules} \Downarrow_r \langle \text{ty}_{p,2}, \text{ty}_{e,2}, C_2, u_2 \rangle \\
C = \{a_p \stackrel{l}{=} \text{ty}_{p,1}, a_p \stackrel{l}{=} \text{ty}_{p,2}, a_e \stackrel{l}{=} \text{ty}_{e,1}, a_e \stackrel{l}{=} \text{ty}_{e,2}\} \cup C_1 \cup C_2 \quad a_p, a_e \text{ fresh} \\
\hline
\Delta; (\text{pat} \rightarrow \text{lexp})^l \mid \text{rules} \Downarrow_r \langle a_p, a_e, C, u_1 \cup u_2 \rangle
\end{array}$$

### 10.3 Expressions

VAR-MONO

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

VAR-POLY

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

VAR-UNDEF

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

CONST

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

ABSTR

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

APP

$$\frac{\begin{array}{c} \Delta; \text{lexp}_i \Downarrow_e \langle ty_i, C_i, u_i \rangle \text{ for } i = 0, \dots, n \\ C' = \{a'_{i-1} \stackrel{l-l_i}{=} a_i \rightarrow a'_i, ty_i \stackrel{l}{=} a_i \mid i = 1, \dots, n\} \cup \{ty_0 \stackrel{l}{=} a'_0, a \stackrel{l}{=} a'_n\} \\ a, a_1, \dots, a_n, a'_0, \dots, a'_n \text{ fresh} \end{array}}{\Delta; (\text{lexp}_0 \text{lexp}_1^{l_1} \dots \text{lexp}_n^{l_n})^l \Downarrow_e \langle a, C' \cup \bigcup_{i=0}^n C_i, \bigcup_{i=1}^n u_i \rangle}$$

LET

$$\begin{array}{c}
\Delta; \text{pat}_i \Downarrow_p \langle ty_{p,i}, C_{p,i}, b_i \rangle \\
\Delta; \text{lexp}_i \Downarrow_e \langle ty_{e,i}, C_{e,i}, u_i \rangle \quad \varpi_i := \text{poly if value lexp}_i \text{ else mono} \\
\text{for } i = 1, \dots, n \quad \Delta' = \Delta|_{\text{id}}[b_i, C_{p,i} \cup C_{e,i} \cup \{ty_{p,i} \stackrel{l}{=} ty_{e,i}\}, \varpi_i \mid \text{for } i = 1, \dots, n] \\
\Delta'; \text{lexp}_{n+1} \Downarrow_e \langle ty_{n+1}, C_{n+1}, u_{n+1} \rangle \\
C_0 = \{a \stackrel{l}{=} ty_{n+1}\} \cup \{ty_{p,i} \stackrel{l}{=} ty_{e,i} \mid i = 1, \dots, n\} \\
\text{dom}(b_i) \cap \text{dom}(b_j) = \emptyset \text{ for all } i \neq j \quad a \text{ fresh} \\
\hline
\Delta; (\text{let } x_1 = \text{lexp}_1 \text{ and } \dots \text{ and } x_n = \text{lexp}_n \text{ in lexp}_{n+1})^l \Downarrow_e \langle a, \bigcup_{i=0}^{n+1} C_i, \bigcup_{i=1}^{n+1} u_i \rangle
\end{array}$$

LET REC

$$\begin{array}{c}
\Delta|_{\text{id}} \left[ x_j \mapsto \langle a_j, \text{mono}, \emptyset \rangle^l \mid j = 1, \dots, n \right]; \text{lexp}_i \Downarrow_e \langle ty_i, C_i, u_i \rangle \\
\varpi_i := \text{poly if value lexp}_i \text{ else mono} \\
\text{for } i = 1, \dots, n \quad \Delta' := \Delta|_{\text{id}}[x_j \mapsto \langle ty_j, \varpi_j, C_j \cup \{a_j \stackrel{t}{=} y_j\} \rangle^l \mid j = 1, \dots, n] \\
\Delta'; \text{lexp}_{n+1} \Downarrow_e \langle ty_{n+1}, C_{n+1}, u_{n+1} \rangle \\
C_0 = \{a \stackrel{l}{=} ty_{n+1}\} \quad x_i = x_j \text{ iff } i = j \quad a, a_1, \dots, a_n \text{ fresh} \\
\hline
\Delta; (\text{let rec } x_1 = \text{lexp}_1 \text{ and } \dots \text{ and } x_n = \text{lexp}_n \text{ in lexp}_{n+1})^l \Downarrow_e \langle a, \bigcup_{i=0}^{n+1} C_i, \bigcup_{i=1}^{n+1} u_i \rangle
\end{array}$$

TUPLE

$$\begin{array}{c}
\Delta; \text{lexp}_i \Downarrow_e \langle ty_i, C_i, u_i \rangle \text{ for } i = 1, \dots, n \quad C_0 = \{a \stackrel{l}{=} (ty_1, \dots, ty_n)\} \quad a \text{ fresh} \\
\hline
\Delta; (\text{lexp}_1, \dots, \text{lexp}_n)^l \Downarrow_e \langle a, \bigcup_{i=0}^n C_i, \bigcup_{i=1}^n u_i \rangle
\end{array}$$

RECORD CONSTRUCTION

$$\begin{array}{c}
\Delta; \text{lexp}_i \Downarrow_e \langle ty_i, C_i, u_i \rangle \quad \Delta|_{\text{rec}}(f_i) = \langle ty_r, ty_{f,i}, \cdot \rangle \\
\text{for } i = 1, \dots, n \quad C_0 = \{a \stackrel{l}{=} ty_r\} \cup \{ty_i \stackrel{l}{=} ty_{f,i} \mid i = 1, \dots, n\} \\
a \text{ fresh} \quad \{f_i \mid i = 1, \dots, n\} \text{ are the fields of record } ty_r \\
\hline
\Delta; \{f_1 = \text{lexp}_1; \dots; f_n = \text{lexp}_n\}^l \Downarrow_e \langle a, \bigcup_{i=0}^n C_i, \bigcup_{i=1}^n u_i \rangle \\
\hline
\Delta; \text{lexp}_i \Downarrow_e \langle ty_i, C_i, u_i \rangle \text{ for } i = 0, \dots, n \quad \Delta|_{\text{rec}}(f_i) = \langle ty_r, ty_{f,i}, \cdot \rangle \text{ for } i = 1, \dots, n \\
C_0 = \{a \stackrel{l}{=} ty_r, ty_0 \stackrel{l}{=} ty_r\} \cup \{ty_{f,i} \stackrel{l}{=} ty_i \mid i = 1, \dots, n\} \quad a \text{ fresh} \\
\hline
\Delta; \{\text{lexp}_0 \text{ with } f_1 = \text{lexp}_1; \dots; f_n = \text{lexp}_n\}^l \Downarrow_e \langle a, \bigcup_{i=0}^n C_i, \bigcup_{i=1}^n u_i \rangle
\end{array}$$



## RECORD ACCESS

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

## RECORD FIELD ASSIGNMENT

$$\frac{\Delta; \text{lexp}_1 \Downarrow_e \langle ty_1, C_1, u_1 \rangle \quad \Delta; \text{lexp}_2 \Downarrow_e \langle ty_2, C_2, u_2 \rangle \quad \Delta|_{\text{rec}}(f) = \langle ty_r, ty_f, \text{mutable} \rangle \quad C_0 = \{a \stackrel{l}{=} \underline{\text{unit}}, ty_r \stackrel{l}{=} ty_1, ty_f \stackrel{l}{=} ty_2\}}{\Delta; (\text{lexp}_1.f <- \text{lexp}_2)^l \Downarrow_e \langle a, C_0 \cup C_1 \cup C_2, u_1 \cup u_2 \rangle}$$

## VARIANT

$$\frac{\Delta|_{\text{var}}(K) = \langle ty_r, [ty_{a,1}, \dots, ty_{a,n}] \rangle \quad \Delta; \text{lexp}_i \Downarrow_e \langle ty_i, C_i, u_i \rangle \text{ for } i = 1, \dots, n \quad C_0 = \{a \stackrel{l}{=} ty_r\} \cup \{ty_i \stackrel{l}{=} ty_{a,i} \mid i = 1, \dots, n\} \quad a \text{ fresh}}{\Delta; (K \text{ lexp}_1 \dots \text{lexp}_n)^l \Downarrow_e \langle a, \bigcup_{i=0}^n C_i, \bigcup_{i=1}^n u_i \rangle}$$

NB The distinction between  $K \text{ lexp}_1 \dots \text{lexp}_n$  ( $n$  arguments) and  $K (\text{lexp}_1, \dots, \text{lexp}_n)$  (an  $n$ -tuple as the single argument) is actually made by an `explicit_arity` flag in the AST.

## IF-THEN-ELSE

$$\frac{\Delta; \text{lexp}_1 \Downarrow_e \langle ty_1, C_1, u_1 \rangle \quad \Delta; \text{lexp}_2 \Downarrow_e \langle ty_2, C_2, u_2 \rangle \quad \Delta; \text{lexp}_3 \Downarrow_e \langle ty_3, C_3, u_3 \rangle \quad C = \{ty_1 \stackrel{l}{=} \underline{\text{bool}}, a \stackrel{l}{=} ty_3, a \stackrel{l}{=} ty_2\} \cup C_1 \cup C_2 \cup C_3 \quad a \text{ fresh}}{\Delta; (\text{if } \text{lexp}_1 \text{ then } \text{lexp}_2 \text{ else } \text{lexp}_3)^l \Downarrow_e \langle a, C, u_1 \cup u_2 \cup u_3 \rangle}$$

## IF-THEN

$$\frac{\Delta; \text{lexp}_1 \Downarrow_e \langle ty_1, C_1, u_1 \rangle \quad \Delta; \text{lexp}_2 \Downarrow_e \langle ty_2, C_2, u_2 \rangle \quad C_0 = \{ty_1 \stackrel{l}{=} \underline{\text{bool}}, ty_2 \stackrel{l}{=} \underline{\text{unit}}, a \stackrel{l}{=} \underline{\text{unit}}\} \quad a \text{ fresh}}{\Delta; (\text{if } \text{lexp}_1 \text{ then } \text{lexp}_2)^l \Downarrow_e \langle a, C, u \rangle}$$

## MATCHING

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

## ARRAY CONSTRUCTION

$$\frac{\Delta; \text{lexp}_i \Downarrow_e \langle ty_i, C_i, u_i \rangle \quad C_0 = \{a \stackrel{l}{=} b \text{ array}\} \cup \{ty_i \stackrel{l}{=} b \mid i = 1, \dots, n\} \quad a, b \text{ fresh}}{\Delta; ([\text{lexp}_1; \dots; \text{lexp}_n])^l \Downarrow_e \langle a, \bigcup_{i=0}^n C_i, \bigcup_{i=1}^n u_i \rangle}$$

## ARRAY ACCESS

$$\frac{\Delta; \text{lexp}_1 \Downarrow_e \langle ty_1, C_1, u_1 \rangle \quad \Delta; \text{lexp}_2 \Downarrow_e \langle ty_2, C_2, u_2 \rangle \quad C_0 = \{ty_1 \stackrel{l}{=} a \text{ array}, ty_2 \stackrel{l}{=} \text{int}\} \quad a \text{ fresh}}{\Delta; (\text{lexp}_1.(\text{lexp}_2))^l \Downarrow_e \langle a, C_0 \cup C_1 \cup C_2, u_1 \cup u_2 \rangle}$$

## WHILE

$$\frac{\Delta; \text{lexp}_i \Downarrow_e \langle ty_i, C_i, u_i \rangle \text{ for } i = 1, 2 \quad C_0 = \{ty_1 \stackrel{l}{=} \text{bool}, ty_2 \stackrel{l}{\leftrightarrow} \text{unit}, a \stackrel{l}{=} \text{unit}\} \quad a \text{ fresh}}{\Delta; (\text{while } \text{lexp}_1 \text{ do } \text{lexp}_2 \text{ done})^l \Downarrow_e \langle a, C_0 \cup C_1 \cup C_2, u_1 \cup u_2 \rangle}$$

## FOR

$$\frac{\Delta; \text{lexp}_i \Downarrow_e \langle ty_i, C_i, u_i \rangle \text{ for } i = 1, 2 \quad \Delta' = \Delta|_{\text{id}} \left[ \text{var} \mapsto \langle a_{\text{var}}, \text{mono}, \{a_{\text{var}} \stackrel{l}{=} \text{int}\} \rangle^l \right] \quad \Delta'; \text{lexp}_3 \Downarrow_e \langle ty_3, C_3, u_3 \rangle \quad C_0 = \{a \stackrel{l}{=} \text{unit}, ty_1 \stackrel{l}{=} \text{int}, ty_2 \stackrel{l}{=} \text{int}, ty_3 \stackrel{l}{\leftrightarrow} \text{unit}\} \quad a, a_{\text{var}} \text{ fresh}}{\Delta; (\text{for } \text{var} = \text{lexp}_1 \text{ to/down to } \text{lexp}_2 \text{ do } \text{lexp}_3 \text{ done})^l \Downarrow_e \langle a, C_0 \cup C_1 \cup C_2, u_1 \cup u_2 \rangle}$$

## SEQUENCE

$$\frac{\Delta; \text{lexp}_i \Downarrow_e \langle ty_i, C_i, u_i \rangle \text{ for } i = 1, 2 \quad C_0 = \{a \stackrel{l}{=} ty_2, ty_1 \stackrel{l}{\leftrightarrow} \text{unit}\}}{\Delta; (\text{lexp}_1; \text{lexp}_2)^l \Downarrow_e \langle a, C_0 \cup C_1 \cup C_2, u_1 \cup u_2 \rangle}$$

## RAISE

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

## TRY

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

ASSERT

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

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

TYPE-ANNOT

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

## 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

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

TUPLE

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

VARIANT

$$\frac{\begin{array}{l} \Delta|_{\text{var}}(K) = \langle ty_r, [ty_{a,1}, \dots, ty_{a,n}] \quad \Delta; pat_i \Downarrow_p \langle ty_i, C_i, b_i \rangle \text{ for } i = 1, \dots, n \\ C_0 = \{a \stackrel{l}{=} ty_r\} \cup \{ty_i \stackrel{l}{=} ty_{a,i} \mid i = 1, \dots, n\} \quad \text{dom}(b_i) \cap \text{dom}(b_j) = \emptyset \text{ for } i \neq j \end{array}}{\Delta; (K \text{ pat}_1 \dots \text{pat}_n)^l \Downarrow_p \langle a, \bigcup_{i=0}^n C_i, \bigcup_{i=1}^n b_i \rangle}$$

RECORD

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

OR

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

ALIAS

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

TYPE-ANNOT

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

## 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>.