Design and Definition of CeXL and ξ-Calculus Version 0.9.3

Ánoq of the Sun, Hardcore Processing *

March 28, 2010

Contents

1 Introduction **7**

2 The Features of CeXL **8**
2.1 Overall Design 12

3 Examples and Motivation for CeXL **13**
3.1 Example of Polymorphic Extensible Records 13
3.2 Restrictions on Type Variables in CeXL 14
3.3 Partial Type Instantiation 15
3.4 Partial Type Instantiation Enhances Encapsulation 15
3.5 Named Type Parameters 15
3.6 Named Type Parameters Benefit Partial Type Instantiation . . . 16
3.7 Declaration of Restrictions 17
3.8 Combinable Restrictions 17
3.9 Optional Fields And Fieldcase 17
3.10 Implementing Free Record Extension in CeXL 19
3.11 Why Absent and Present are 2 Different Types 20
3.12 Why Optional Fields and Extensible Records Should be Strongly Statically Typed 20
3.13 We Only Remove Absent Fields at Val Bindings 21
3.14 Use Case of Mutual Forward and Backwards Compatibility . . . 23
3.15 Examples Comparing CeXL and ML 25
3.16 More Examples In Section 25 26

4 Previous Work **27**
4.1 Record Calculi 27
4.2 How ξ-Calculus Relates to Other Work 28
4.2.1 Extensible Records and Field Absence and Presence . . . 28
4.2.2 The Rest of ξ-Calculus 29

*© 2003-2010 Ánoq of the Sun (alias Johnny Bock Andersen)

5 Walk-through of the CeXL Definition **30**
5.1 The Languages and Calculi Presented 30
5.2 About the Separate ξ-Calculus 30
5.3 Presenting Implicitly Typed ξ-Calculus 31
5.4 Implicitly Typed ξ-Calculus 32
5.5 Syntax and Semantic Objects for the Type System 32
5.6 A Syntax for ξ-Calculus Expressions 32
5.7 Comments on the Type System 33
5.8 Static Semantics of ξ-Calculus - Unification 35
5.8.1 Formal View and Implementation of Meta Variables with Restrictions 36
5.8.2 Unification of Restrictions 36
5.9 Static Semantics of ξ-Calculus 37
5.10 Dynamic Semantics of ξ-Calculus 39
5.11 Grammar of Naked CeXL 39
5.12 The Naked CeXL to ξ-Calculus Translation 41
5.13 The Full Syntax for CeXL 41
5.14 CeXL To Naked CeXL Translation 41
5.15 Putting It All Together 41

6 Notes on Implementation **42**
6.1 Removing "Weak Typing" 42
6.2 Instantiating All Polymorphism During Compilation 42
6.3 A Compiler Will Be Another Project 42
6.4 The Implementation Provided 43

7 Results of Other Language Features Which Have Been Considered **44**
7.1 Relating Free Extension to Exactly Typed Records and Optional Fields with Fieldcase 44
7.2 Considering Polymorphic Restrictions 47
7.3 Considering the Semantics of a General Typecase 47
7.4 Considering Staticly Typed Exceptions for ξ-Calculus 48
7.5 Considering the Semantics of Record Concatenation 49

8 How Hard it was to Create This Language **50**
8.1 Early Investigation of 3D Architectures 50
8.2 An Old Version of CeXL Has Been Used Until Now 50
8.3 Development of CeXL for This Specification 51
8.4 The Ease of Implementation From Semantics 51
8.5 Conclusion And Warnings About Language Design 52

9 Grand Conclusion **53**

10 Directions for Future Work **54**
10.1 Closing Remarks About the Future 54

11 Acknowledgements **55**

12 Appendices **58**
12.1 The Languages and Calculi Presented 58
12.2 About the Separate ξ-Calculus 58

13 Implicitly Typed ξ-Calculus **59**
13.1 Syntax and Semantic Objects for the Type System 59
13.2 A Syntax for ξ-Calculus Expressions 59
13.3 Comments on the Type System 60

14 Static Semantics of ξ-Calculus - Unification **62**
14.1 Formal View and Implementation of Meta Variables with Restrictions 63
14.2 Unification of Restrictions 63
14.3 Restrictions on Types: $\tau|_\xi$ 63
14.4 Judgement Forms for the ξ Unification Relation 64
14.5 Inference Rules for the ξ Unification Relation 64
14.5.1 Unifying Types Different From Records and Meta Variables 64
14.5.2 Unifying Meta Variables 65
14.5.3 Unifying Records 67
14.5.4 Unifying With the Empty Record 67
14.5.5 Unifying Order Independent Records 68
14.5.6 Making Record Fields Independent of Order 70
14.5.7 Type Respects Restriction 71
14.5.8 Type Pattern Respects Restriction 72
14.5.9 ξ More Restrictive Than ξ' 72

15 Static Semantics of ξ-Calculus - Inference **73**
15.1 Environments 73
15.2 Generalizing by Type Schemes: $\sigma \succ \tau$ 73
15.3 Non-expansive Expressions 74
15.4 Scope of Explicit Type Variables 74
15.5 Closure 75
15.6 Tidying Up During Closure 76
15.7 Predefined Constructors and Type Schemes 77

16 Type Inference for ξ-Calculus **78**
16.1 Judgement Forms 78
16.2 Inference Rules 79
16.3 Rules for Implicitly Scoped Type Variables 88
16.3.1 The $\uplus$ Operator 88

17 Dynamic Semantics **93**
17.1 Semantic Objects for Values 93
17.2 Notes About the Semantic Objects 93
17.3 Primitive Functions: BasVal and $APPLY$ 94
17.4 Predefined Semantic Objects 94
17.5 Function Closures 94
17.6 Conventions for the Inferences Rules 94
17.7 Judgement Forms 96
17.8 Inference Rules 96

17.9 Comments on Dynamic Semantics 100

18 Grammar of Naked CeXL **102**
18.1 Notational Conventions 102
18.2 Grammar Productions 103
18.3 Syntactic Limitations 106
18.4 Comments on the Grammar 106

19 Naked CeXL To ξ-Calculus Translation **107**
19.1 About the Translation 107
19.2 The Prefix Variable 108
19.3 Application of Type Functions: $\zeta\theta$ 108
19.4 No Free Types: $\rhd\tau\lhd$ 109
19.5 Combination of Restrictions: $\bowtie$ 109
19.6 Predefined Semantic Objects 110
19.7 Translation Functions 111
19.8 Translation Rules 112

20 CeXL Lexical Syntax **127**
20.1 Reserved Words and Symbols 127
20.2 Special Constants 127
20.3 Comments 128
20.4 Identifiers 128
20.5 Lexical Analysis 130

21 Full CeXL Grammar **131**
21.1 Notational Conventions 131
21.2 The Grammar Productions 132
21.3 Syntactic Limitations 136
21.4 Displaying Types to the User 137

22 CeXL To Naked CeXL Translation **138**
22.1 Atomic Expressions 138
22.2 Infix Expressions 139
22.3 Expressions 139
22.4 Atomic Patterns 139
22.5 Patterns 139
22.6 Pattern Rows 139
22.7 Declarations 140
22.8 Type Parameters 140
22.9 Function-value Bindings 140
22.10Type Expressions 140
22.11Type Pattern in Restrictions 140

23 Initial Environments **141**
23.1 Environment for Static Semantics of ξ-Calculus 141
23.2 Environment for Dynamic Semantics of ξ-Calculus 142
23.3 Environment for Naked CeXL To ξ-Calculus Translation 143
23.4 Extending to a Complete Basis Library 144

24 Type-check and Execution of CeXL Programs **145**

25 A Few Regression Tests **146**
25.1 Demonstrating Value Restriction 146
25.2 Demonstrating Scoping of Type Variables 147
25.3 Demonstrating Inference of Most General Unifier 149
25.4 Demonstrating Requirement for Dynamic Allocation of Exception Names 150
25.5 Tests for Extensible Records 151
25.6 Examples of Type Inference in CeXL 157
25.6.1 Legal Programs and Their Types 157
25.6.2 Illegal Programs 157

Preface

- This document and an implementation is maintained online at:
  `http://www.CeX3D.net/cexl/`

Certain parts of this document is almost a verbatim copy from *The Definition of Standard ML (Revised)* [SML97]. There is no need to rephrase well-written text :-)

This "we" and "I" Thing

Whenever the text reads "I" it refers to me, the author of this document. This is usually where I have made a choice that you, as the reader, are not involved in. The use of "we" refers both to you as the reader and to me as the author, which means that it refers to things we can both do. At these points it is thus expected that you, the reader, follow me in the description.

So I am in fact neither speaking about myself in the plural nor being inconsistent about when to use "we" and when to use "I" :-)

1 Introduction

The purpose of this document is to present a formal specification for a programming language which I call *CeXL* (pronounced as "Sex El"). I will also compare the work with other relevant work and give motivation and examples of what the language is useful for.

The design of the features of CeXL and early implementations have actually been a big undertaking ranging over several years. During the design of CeXL, several alternatives for features and semantics have been investigated. Many caused problems and were rejected. Towards the end of the document I present the results of these experiences. I also make a claim which justifies that given the design constraints for the language, the most central part of the design can most likely not be done any better. This claim also puts a lot of other relevant research into perspective.

Prerequisities of the Reader

To understand the whole document the reader should have a firm understanding of *operational semantics* for programming languages and *compiler technology*. Courses such as *Dat1E*, *Dat2V-Programming Languages* and *Advanced Compiler Construction* from *DIKU*, The Computer Science Department of the University of Copenhagen, would be appropriate as minimum background knowledge. Even though everything should be described here, the reader would definitely also be better off by having understood *The Definition of Standard ML* from 1997 [SML97] and at least some of the articles mentioned about extensible records.

That being said, the document starts gently with some concrete examples of use of the programming language and this should be accessible to anyone who knows how to write Standard ML programs - although the examples are not a thorough beginner's guide. The thoroughness is in the rest of the specification. The syntax and grammar for the CeXL language should also be accessible to people who know how to read *BNF*-grammars and the likes. But for the most important and essential parts of the specification, i.e. the semantics - solid understanding of the relevant theory is essential.

2 The Features of CeXL

A good starting point for the design of the language is taking Standard ML '97 [SML97] and modifying it appropriately, since it is well-defined, well-known and has many good properties. There are many of ML's features that I want to retain. This includes type inference. The features that the language is required to have in addition to those found in Standard ML include polymorphic and extensible records with optional fields and the ability to write programs which react on whether a field is present or absent. I will also remove some features from Standard ML to simplify an implementation and because the available time for this project is limited.

The list below presents the features of CeXL - especially in comparison to Standard ML '97. The section following this one will demonstrate these features and give further motivation for them. The features are named and I will refer to them frequently later in the document.

- *Limitations Compared to Standard ML '97*:
  - There are a few, one might say rather inessential, syntactical details which are allowed in Standard ML that we don't allow in CeXL. This includes:
    - We don't allow arbitrarily many semicolons to separate declarations
    - We don't allow notation such as `let rec rec val x = ...`
  - We require patterns to be exhaustive in val bindings and the fieldcase construct. This is mostly to simplify things but also to keep programmers from writing non-exhaustive patterns in these places. It means that the evaluation of a pattern-match in a declaration or a fieldcase will never raise an exception.
  - We have only a subset of all the Standard ML features. In particular we do not support: abstype, withtype in datatypes, open, local in end, infix operators and equality types. We also only support putting the prefix **op** in front of infix operators in expressions. So **op** is not allowed anywhere else.
  - At each val binding when taking the closure in the semantics we insist that completely parameterized type schemes result at each val binding (except for type variables which are already scoped by an enclosing val binding). This is subject to the value restriction.

    This might give some extra limitations compared to Standard ML - but hopefully it is rare in practice. This problem should usually be fixable in CeXL programs by adding explicit type constraints.
  - We do not support the full Standard ML '97 module system. However, writing simple nested structures are supported to enhance the name space of identifiers. Signatures and functors are not supported at all.
- We don't support explicit scoping of type variables at `val` bindings. However we support the notation: `val ('a, 'b) f : 'a -> 'b` which is the notation for explicit scoping of type variables in Standard ML. In CeXL it

has another meaning though. It is used only for specifying that the type variables `'a` and `'b` does not have any restrictions on them (the description of what "restrictions" are follow further down).

The scoping of type variables is always done implicitly in CeXL. It happens at the `val` bindings in CeXL - but not necessarily at the *same* `val` binding as where the type variables are explicitly written.

The notation of type variables at val bindings is thus supported but *has a slightly different meaning than in ML.*

- *Polymorphic Extensible Records*: This is a central feature of the language. It allows one to make operations on only some of the fields of a record while the rest of the record is unknown. The semantics of this in CeXL aims to give as few limitations as possible and yet keep the extensible record types completely statically typed. We will give examples of this feature in the following section. Section 4 about previous work will do a more in-depth comparison with earlier research of this topic. The important claim in section 7.1 is also relevant for this feature.

- *Optional Record Fields with a Fieldcase Construct*: This is another central feature of the language which is tightly coupled with the feature *Polymorphic Extensible Records*. It allows us to make some of the fields of a record optional. Again we try to give as few limitations as possible while keeping everything completely statically typed in the semantics. There will be examples of this in the following section and a thorough comparison with earlier research in section 4 about previous work.

  An important fact about how this is supported is that we are able to specify the type of an optional field and the optionalness of the field separately. This allows several fields to have the same type but differing optionalness or to have the same optionalness but differing types. This is important and most other research proposals related to record cacluli cannot do this. The claim in section 7.1 is also relevant here.

  We have a fieldcase construct which allows us to "case out" an optional field and write 2 different pieces of code for when the field is absent or present in a record respectively. This is the feature that allows us to really make use of the optional fields. Examples follow in the next section.

- *Restrictions on Type Parameters*: We support restrictions on type parameters and have some new syntax for this. The restrictions for type parameters is specified by syntax like `['a : <restr>, 'b; 'c : <restr>]` where `<restr>` represents the syntax for restrictions. The syntax allows a restriction to be either of the form `~{lab1, lab2, lab3}` or of the form `tycon1 | tycon2 | tycon3`. The first form of restriction specifies that the type parameter may only be instantiated with records which do *not* contain any of the labels `lab1`, `lab2` or `lab3`. Anywhere from zero to a finite number of labels may be specified. The second restriction specifies that the type parameter may only be instantiated with *either* the type `tycon1` *or* the type `tycon2` *or* the type `tycon3`. There may be anywhere from 2 to a finite number of types (but they must be monotypes and disjoint). Each type may even have type arguments (but this would most likely be rare in practice) so the restriction `int list | real` is valid.

This is an essential feature for the way we support *Polymorphic Extensible Records* and *Optional Record Fields*. It is also what makes the fieldcase construct work, where we use the restriction `absent | present`. The fairly general notion of these restrictions even allows overloaded operations to be well-typed in CeXL. For instance the operator `+` gets the value description[1]:

```
val ['a : real | int | word] + : 'a * 'a -> 'a
```

Functions which use such operators also continue to be both well-typed and to retain the overloading properties of the operators used. We will see examples of this in the next section.

- *Specifying Restrictions On Types*: The introduction of the feature *Restrictions on Type Parameters* requires that the programmer state all restrictions on any type variables mentioned explicitly. In CeXL this is done at `val` bindings or the derived `fun` bindings. Any types occuring within a value binding and which mention a type variable get the restriction on the type variable as specified at the `val` binding. So in the binding `fun ['a : ~{lab}] f (x : 'a) = x` the `'a` in the parameter for `f` gets the restriction `~{lab}`.

  As mentioned, type variables are implicitly scoped at the `val` bindings. However it may be that the scoping really occurs at some nested or some enclosing val binding rather than where the variables are mentioned.

  This scoping is *a difference from Standard ML.*

- *Named Type Parameters*: We have a slightly more general interpretation of type parameter names. We use the *names* of the type parameters to identify them rather than just the *order* of the parameters as done in Standard ML. As an example, consider the following two types:

  ```
  type ('a, 'b) t1 = 'a -> 'b
  type ('b, 'a) t2 = 'a -> 'b
  ```

  These two types behave differently in ML when instantiated but they are equivalent in CeXL. This *Named Type Parameters* feature introduces some new syntax for instantiating types. Instead of `(int, real) t1` we can also write `['a = int, 'b = real] t1` in CeXL.

  This is *only backwards compatible with Standard ML '97 in some cases.* Most notably the cases where the names of type parameters are `'a`, `'b`, `'c` etc. - and *in that order.* More examples of this follow in the next section.

- *Partial Type Instantiation*: The type inference algorithm has been strengthened at a subtle point compared to that of Standard ML. When using parameterized types in type expressions we allow some or all of the type arguments to be omitted. From the declaration of the type the parameters of the type and their restrictions are already known. So if the programmer does not have any particular constraints on them it is no problem to omit them. For instance we may write:

[1] The word *value description* is from the Standard ML Module System, found as part of signatures in [SML97]

```
fun f (l : list) = (l, l)
```

We do not have to specify that the type argument for `list` has to be any particular type variable.

Naturally this requires slightly less program code to be written by the programmer. More importantly it also benefits the encapsulation and abstraction principles which should be followed for good software development. We will see examples of this in the following section.

If we didn't have the feature *Named Type Parameters* above we would be forced to either include all arguments or omit all arguments - or figure out some other ways of specifying restrictions only for some of the parameters and not for others. We will also see examples of this in the following section.

- *Declaration of Restrictions*: We allow declaration of identifiers to which restrictions are bound. Just like the feature *Partial Type Instantiation* this is to enhance encapsulation of parameterized types with restrictions on the type parameters. We may declare a restriction with:

  ```
  res opt = absent | present
  ```

  where `res` is the keyword for restriction declaration. We may use the declared restriction `opt` by writing e.g. `['a : res opt]`. More examples of this follow.

- *Combinable Restrictions*: It is possible to combine restrictions to make new restrictions. The combined restrictions will have all the restrictions of the restrictions that it was combined from. Restrictions are combined with an infix `+`. So the restriction `~{lab1, lab2} + ~{lab1, lab3}` is equivalent to the restriction `~{lab1, lab2, lab3}`. Using the `opt` from before the restriction `res opt + absent | present | int` is equivalent to `absent | present`.

  This feature enhances encapsulation of parameterized types. More examples follow.

- *Almost Complete Backwards Compatibility with a Subset of Standard ML '97*: As mentioned the feature *Named Type Parameters*, is not always backwards compatible with Standard ML '97. The syntax of type variables at val bindings also has a slightly different meaning than in ML. The closure operations at val bindings are also slightly more restrictive in that we quantize all free types as type variables at each point of closure - subject to the value restriction.

  However, everything else which is not new features is backwards compatible with a subset Standard ML '97. In particular all of the *syntax* is backwards compatiple and it is the hope that most ML programs within the subset of CeXL will not change their semantic meaning either when considered CeXL programs.

2.1 Overall Design

For this language design it has been kept in mind that eventually we must be able to write a compiler for the language which generates efficient programs. This is why the language is kept completely *statically* and *strongly* typed.

The fact that the language is both statically and strongly typed also prevents many programmer mistakes from happening at runtime. All errors involving illegal types are reported at compile-time. This means that programs written in CeXL which successfully pass the type-check cannot crash the computer running the programs nor make runtime type errors, assuming that the implementation is correct and that the programs are not given access to functions performing illegal operations (such as writing a pixel value directly to the screen memory where it turns out that the value is actually written outside the allowed screen memory). The property that "well-typed programs do not go wrong" is known as *soundness* and has not been proved for CeXL.[2] Soundness *has* been proved for Standard ML though and I firmly believe that it holds for CeXL as well.

The following deserves mentioning about the type system of CeXL:

- Overloaded operators are typed as parametrized types with restrictions similar to `real | int | word`. Such operators will have to know with which type they are instantiated in order to work. This can be considered a form of weak typing. However it is not of the kind which postpones programmer mistakes until runtime.
- The fieldcase construct for optional fields relies on parametrized types with the restriction `absent | present`. This is the only language construct where the runtime excution depends on the instantiation of a type. However, in this case there will always be a constructor in the dynamic semantics to perform this check. Thus the dynamic semantics does not need to know the instantiation type to work.

All of the above features where the runtime execution depends on a type can usually be specialized away at compile-time as we will consider later in section 6. In particular they can *always* be specialized away for the intended use of CeXL. So we can in fact generate very efficient programs for CeXL.

[2] This is intensionally left as an excercise for the reader ;-)

3 Examples and Motivation for CeXL

CeXL is intended to be used as a general purpose programming language in a 3D computer graphics application. Therefore I will try to give some examples of program code which it is very conceivable that one would write in CeXL for this domain specific purpose. I will also give some examples which are only for illustrating the strengths of CeXL in comparison to Standard ML. As we will see, some of the primary issues that the language solves actually relate to software maintenance and mutual backwards and forward compatibility between 2 independent pieces of code.

3.1 Example of Polymorphic Extensible Records

A *3D polygon* is a common datastructure in a 3D computer graphics application. It is a set of points in 3D space which forms the corners of a geometric figure in a plane in 3D space. First, let's define a 3D point in CeXL:

```
type point3d = {x : real, y : real, z : real}
```

A 3D polygon could in its simplest form just be a list of such points - i.e. a value of type `point3d list`. However, this is usually far from enough in a computer graphics application. The points of a polygon are usually called the *vertices* of the polygon. Besides the 3D point, a vertex can contain such things as color and a normal vector (for instance for representing a "fake" orientation of the polygon at that vertex to give a smooth look when visualizing adjacent polygons). However there are a number of other data one could imagine - such as texture coordinates for an arbitrary number of textures on the polygon (a texture could be an image that the polygon is "painted" with). In fact the ideal would be not to put any restrictions on what data one can put in such a vertex! In CeXL we can handle this by using polymorphic extensible records:

```
type ['a : ~{point}] vertex = {point : point3d, ... : 'a}
```

To describe what we have defined here we can say that "a vertex is a record containing the field `point` of type `point3d` and possibly other fields". The "possibly other fields" part is represented by the `, ... : 'a` part of the record type. However this is not enough to define such a type in CeXL. We need to explicitly say that the type variable `'a` may only be instantiated with a record which is not allowed to contain the field `point`. This is what the notation `['a : ~{point}]` means. The reason that `'a` may not contain the field `point` is that `'a` is used in a record already containing the field `point`.

Let's define a function which takes a record as parameter, removes the field `point` from that record and returns the rest of the record without the field `point`:

```
fun removePoint {point, ... = rest} =
      rest
```

As in Standard ML we don't need to put any explicit types to make this work - this is handled by the type inference.

What happens here is that we do a record pattern-match where the notation `, ... = rest` captures "the rest of the fields" in the parameter record. This

record is bound to the variable `rest`, where `rest` is just a normal identifier. We then return the record `rest` from the function.

This function gets the value description:

```
val ['a : ~{point}] removePoint : {point : 'b, ... : 'a} -> 'a
```

As in the type we declared before, `'a` must be a record which is not allowed to contain the field `point`.

Now let's try to define a function which updates the value of the `point` in a `vertex` without touching any of the other fields in the record:

```
fun updatePoint newPoint {point, ... = more} =
      {point = newPoint, ... = more}
```

Again we do a record pattern-match where the notation `, ... = more` captures "the rest of the fields" in the record into the variable `more`. When forming the new record in the result there is a similar notation for the record expressions which "puts the fields of the record `more` into the record in addition to the field `point`". It can also be read backwards as creating a record with the fields from `more` and adding the extra field `point` - sort of like adding an element to a list in ML.

The notation with the 3 dots is inherited from Standard ML. In ML one can write:

```
fun getPoint {point, ...} =
      point
```

This means "ignore the rest of the fields of the record" in the pattern-match. However, in ML this will require an explicit type constraint to say which fields the record really has. We don't need this type constraint in CeXL since the function `getPoint` becomes well-typed as it is in CeXL.

3.2 Restrictions on Type Variables in CeXL

Above when defining the type `vertex` we used the notation `['a : ~{point}]` in the type parameter list. The part `~{point}` is called a *restriction* on the type variable `'a`. It restricts `'a` to be instantiated only with records - even only those which do not contain the field `point`. In general in CeXL - whenever a type variable is explicitly mentioned it must be given exactly the constraints it has - if it has any. So writing a type variable without mentioning its constraints means that it is *intended* that the variable not have any restrictions.

Consider the `updatePoint` function from before. If we wanted to explicitly type the argument record of the function to be a vertex we could do it like this:

```
fun ['a : ~{point}] updatePoint newPoint ({point, ... = more} : 'a vertex) =
      {point = newPoint, ... = more}
```

Notice that the constraints are listed after the keyword `fun`. They can also be put after the keyword `val` for val bindings. This is similar to the notation for scoping type variables in ML and in fact, ML's notation is supported in CeXL for type variables without restrictions. However the purpose in CeXL is only to specify restrictions and no scoping necessarily occurs at these points in CeXL since scoping of type variables is done implicitly.

3.3 Partial Type Instantiation

In CeXL it is actually enough to write the function from before as:

```
fun updatePoint newPoint ({point, ... = more} : vertex) =
      {point = newPoint, ... = more}
```

This is due to the feature *Partial Type Instantiation* of CeXL. This is actually a small enhancement of the type inference algorithm compared to that of Standard ML. The declaration of `vertex` knows what type parameters the type has and if there are no instantiations of those, then the type construtor arguments can be omitted. The declaration of `vertex` also knows what restrictions the type variables must have so there is no need to repeat those either.

This example shows that we can avoid repeating the restriction on the parameter for `vertex`, while still making the explicit type constraint `vertex`.

3.4 Partial Type Instantiation Enhances Encapsulation

In the example above we were able to specify the type of the parameter of a function while having to specify neither which type parameters the type takes nor which restrictions these have. This actually enhances the encapsulation of the type `vertex`. If we were to specify all type parameters and their restrictions every time we wish to place a type constraint we would also have to modify all type constraints throughout the code every time we modify the type parameters or the restrictions of the type `vertex`.

It is particularly in combination with polymorphic extensible records that this is important that one does not have to repeat the constraints everywhere. If for example our `vertex` type is extended to always include a `color` field in addition to the `point` field, we would have to update the restriction on `'a` everywhere it is mentioned in the code, from being `~{point}` as it were before, to `~{point, color}`.

3.5 Named Type Parameters

To have a look at the *Named Type Parameters* feature, let's declare a parameterized type with several type parameters:

```
type ['a : ~{point, color}, 'b : absent | present]
     colvertex = {point : point3d, 'b color : color, ... : 'a}
```

We don't care how the type `color` is defined here. The `'b` with the restriction `absent | present` is actually an optional field. We will describe this later. This type has 2 type parameters `'a` and `'b`. It can be instantiated for instance as:

```
({}, absent) colvertex
```

However we also support instantiation with this notation:

```
['a = {}, 'b = absent] colvertex
```

This is because in CeXL it is not the *order* of the type parameter which matters - as in ML. It is the *names* of the parameters. In the examples above the first instantiation is actually equivalent to the second one because the notation

`({}, absent) colvertex` means that the type parameters are assumed to have the names `'a` and `'b` - and in that order. So this notation is dependent on both the order of the given arguments and the names of the type parameters.

We could redeclare the type to:

```
type ['vertexData : ~{point, color}, 'hasColor : absent | present]
     colvertex = {point : point3d, 'hasColor color : color, ... : 'vertexData}
```

This *cannot* be instantiated by `({}, absent) colvertex` and we would have to use the notation `['vertexData = {}, 'hasColor = absent] colvertex`. As can be seen, this is not compatible with ML! The worst example of this incompatibility would be to declare `colvertex` as this:

```
type ['b : ~{point, color}, 'a : absent | present]
     colvertex = {point : point3d, 'b color : color, ... : 'a}
```

Now the instantiation `({}, absent) colvertex` will be illegal because `'b` will be instantiated with `absent` and `'a` will be instantiated with `{}`! However using `(absent, {}) colvertex` would work as intended. What really happens is that types like: `(t1, t2, t3) t` are transformed into: `['a = t1, 'b = t2, 'c = t3] t`. The order of the arguments thus represent the use of a predefined sequence of type variables: `'a`, `'b`, `'c` etc. This sequence is defined in the appendices, in section 22.

To appreciate why this feature is added it can be mentioned that even a fairly simple extensible 3D polygon mesh data structure (which is a very fundamental data structure in a 3D graphics application) in CeXL is likely to be parameterized with 10-20 type variables. More elaborate data structures are worse so it is *not* desirable to use parameter names `'a`, `'b`, `'c` etc. nor to do "unnamed" instantiation of such parameterized types.

3.6 Named Type Parameters Benefit Partial Type Instantiation

Standard ML does not have the *Named Type Parameters* feature and uses the order of type parameters to distinguish them. So consider this type declaration in Standard ML and the function following:

```
type ('a, 'b) t = 'a list * int * 'b

fun f ((l, i, v) : ('a, real) t) = ()
```

The function explicitly types the function to take an argument of type `('a, real) t`. Notice that `'a` is only mentioned once so there is no need to state it's name. In CeXL we can omit the first type argument here by using the *Named Type Parameters* feature as follows:

```
fun f ((l, i, v) : ['b = real] t) = ()
```

If we were to support *Partial Type Instantiation* in Standard ML (i.e. without the *Named Type Parameters* feature) we would be forced to either include all type arguments or none of them - because consider:

```
fun f ((l, i, v) : real t) = ()
```

In CeXL the type argument `real` will refer to the first type parameter of `t` - which is called `'a`. However what we needed was for the parameter `'b` to be instantated with `real`.

The conclusion is that the feature *Named Type Parameters* benefits *Partial Type Instantiation* and that these 2 features go well together.

3.7 Declaration of Restrictions

In the examples until now we have specified restrictions directly where we need them. However it is also possible to declare restrictions and bind them to identifiers like this:

```
res rVertexData = ~{point}
```

Having declared this we can declare the type `vertex` from earlier as:

```
type ['vertexData : res rVertexData]
     vertex = {point : point3d, ... : 'vertexData}
```

As shown in the description of the features for CeXL, the keyword `res` is also for using a declared restriction. Declared restrictions also have their own separate name space.

Declaring restrictions is a way of enhancing encapsulation. If at some point it is necessary to specify that some type variable must have the same restriction as the restriction for one of the type parameters for an encapsulated data structure, then it is necessary to have this restriction of this data structure declared as a variable. This makes it possible to extend the restrictions of the data structure without having to alter all code *using* the data structure where it has specified the restrictions explicitly.

3.8 Combinable Restrictions

It is possible to combine restrictions which is done using the operator `+` between restrictions. An example where this benefits encapsulation of data structures would be:

```
res rUncoloredVertexData = res rVertexData + ~{color}

fun ['vertexData : rUncoloredVertexData]
    addColorToVertex color (vertex : ['vertexData = 'vertexData] vertex) =
      {color = color, ... = vertex}
```

Here we extend the restriction `rVertexData` to include the restriction that it may not contain the record field `color`. This code is valid whether or not `rVertexData` already has this restriction.

3.9 Optional Fields And Fieldcase

To start the examples of optional fields in CeXL we will first consider this record expression:

```
val r = {name = "John", age ?= present 28}
```

The notation `age ?=` means that the field age is created as an optional field. The expression `present 28` creates the optional field as a present integer field. The above example is actually equivalent to:

```
val r = {name = "John", age = 28}
```

Both of the above examples give the usual value description:

```
val r : {name : string, age : int}
```

We can also create the field `age` as being an absent field by:

```
val r = {name = "John", age ?= absent}
```

This will give the value description:

```
val r : {name : string}
```

Notice that the field `age` has disappeared. It should be noted at this point that the present and absent variables are completely general. So writing:

```
val field = present 5
```

Will give the variable `field` a value of type `(present, int) ?`. So `?` is a type constructor for field types. Similarly writing:

```
val field = absent
```

Will give `field` a value of type `(absent, 'a) ?`. These field types just happen to be displayed much more compactly inside record types but record fields really always contain field values.

We can do pattern-matching of optional fields with a notation similar to the notation from before:

```
fun printPersonName {name, age ?= age} =
      print name
```

This function gets the more interesting value description:

```
val ['a : absent | present] printPersonName : {name : string, 'a age : int}
```

The record notation `'a age : int` denotes that age is an optional field. The type variable `'a` must be instantiated with either the type `present` or the type `absent` which is what the restriction `['a : absent | present]` means. The function can be called in one of these 2 ways:

```
val _ = printPersonName {name = "John"}

val _ = printPersonName {name = "John", age = 28}
```

This is a little interesting since we cannot do this in ML. We can make it even more interesting by using the fieldcase construct to print the age if it is present and not print it if it isn't present. This is done in the following function:

```
fun printPerson {name, age ?= age} =
      fieldcase age in 'a of
        absent =>
          print name
      | present years =>
          print (String.concat [name, " is ", Int.toString years, " years old"])
      type unit end
```

The notation `fieldcase age in 'a of` means that we case out the optional field `age` where the type variable `'a` is the variable to contain one of the types `absent` or `present` depending on whether the field `age` is absent or present. The two cases for `absent` and `present years` should be quite clear. The `type unit end` tells us that the fieldcase construct must return `unit`.

A slightly more interesting function would be one which returns the record and increments the age if it is present:

```
fun nextYear {name, age ?= age} =
      fieldcase age in 'a of
        absent =>
          {name = name}
      | present years =>
          {name = name, age = years + 1}
      type {name : string, 'a age : int} end
```

The interesting part is that the result type contains the type variable `'a`. This will have a different instantiation in each of the 2 clauses. It would be hard for the type inference to infer such a type, which is why we always have to include the result type of a fieldcase construct explicitly. This also implies that we cannot omit any type arguments which must be `'a` by using the feature *Partial Type Instantiation*. As an example, the following will not work:

```
type 'a T = {name : string, 'a age : int}

fun nextYear {name, age ?= age} =
      fieldcase age in 'a of
        absent =>
          {name = name}
      | present years =>
          {name = name, age = years + 1}
      type T end
```

So all occurences of `'a` of the result of the fieldcase must be mentioned explicitly for the program to be valid.

3.10 Implementing Free Record Extension in CeXL

Record extension in CeXL is strict. That is, we may only add fields to a record which are not already present in the record. To do free extension of records in CeXL we can implement this explicitly. A polymorphic free extension operation which adds the field `f` to any record can be implemented in CeXL as follows:

```
fun freeextf value {f ?= _, ... = r} =
      {f = value, ... = r}
```

It works by matching out the field `f` as being optional and creating a new record with the rest of the fields `r` from the record and adding the field `f` to the new record. It would get the value description:

```
val ['a : ~{f}, 'c : absent | present] freeextf :
    'b -> {'c f : 'd, ... : 'a} -> {f : 'b, ... : 'a}
```

This shows that we have the power of free extension of records in CeXL. In section 7.1 there is a claim justifying that given our design criteria for CeXL, we have to represent record extension as strict extension at the type level. It would be possible to add free extension as syntactic sugar for the language but it has been decided to keep the supported syntax both at the value level and the type level to strict extension of records.

3.11 Why Absent and Present are 2 Different Types

The types `absent` and `present` are 2 different types in CeXL. That is, if the programmer was allowed to redeclare the types `absent` and `present`, they could have been declared as:

```
datatype absent = absent

datatype present = present
```

However this is not quite the way they are declared. The constructors `absent` and `present` are not visible in CeXL. Instead, two *variables* called `absent` and `present` are visible. These are really a field value and a function respectively. These variables have the following types:

```
val absent : (absent, 'a) ?

val present : 'a -> (present, 'a) ?
```

We saw how this was used in section 3.9 above. When they occur in the patterns of the fieldcase construct as constructors, it is only because they are treated specially here.

We need these 2 types to be two separate types to be able to make optional fields statically typed. We will see an example of this in the next section. In the mean time it should be realized that any concrete field value existing at runtime must have one of the types `(absent, 'a) ?` or `(present, 'a) ?`. No concrete value can exist with a type like `('a, 'b) ?`.

3.12 Why Optional Fields and Extensible Records Should be Strongly Statically Typed

Consider a long list or long array of vertices, say 1.000.000 vertices - which is not unrealistic for a 3D graphics application. If it is statically guaranteed that all vertices have the same type and binary layout at runtime it improves performance considerably to traverse them and while doing an operation on all vertices. As an example, consider this function:

```
type ['a : ~{point, color}, 'b : absent | present]
     vertex = {point : point3d, 'b color : color, ... : 'a}

fun ['a : ~{point, color}, 'b : absent | present]
    makeVertexGreen {point, color ?= color, ... = rest} =
      fieldcase color in 'a of
        absent =>
          {point = point, ... = rest}
      | present _ =>
          {point = point, color = greenColor, ... = rest}
      type ('a, 'b) vertex end
```

We can now write this function:

```
fun makeManyVerticesGreen vertices =
      map makeVertexGreen vertices
```

This function will get the value description:

```
val ['a : ~{point, color}, 'b : absent | present] makeManyVerticesGreen :
    {point : point3d, 'b color : color, ... : 'a} list ->
    {point : point3d, 'b color : color, ... : 'a} list
```

Whenever this function is called with a concrete list of vertices, the `'b` must be instantiated to a concete type. This `'b` is always the *same* `'b` for all elements in the list. Since `'b` can only be instantiated with either `present` or `absent` and since these are 2 different types as described above, the field `color` will either be present in *all* elements of the list or be absent in *all* elements of the list. So in the function `makeGreen`, the *same* clause of the fieldcase is guaranteed be executed for all elements in the list which is an obvious chance for optimizing the fieldcase away in the inner function - e.g. by compiler optimizations like code-hoisting [Appe98] or specialization [Jone93]. Even type instantiation would be enough.

This also implies that we cannot write code like the following:

```
val zero = {x = 0.0, y = 0.0, z = 0.0}
val a = Array.tabulate (1000000, fn _ => {point = zero})
val _ = Array.update(a, 500000, {point = zero, color = greenColor})
```

This is seen in comparison to for instance using a more dynamic data structure of named values for each vertex - or even just a more dynamic type system. The above examples is a very good reason to do the hard work of developing the optional fields in the language to become completely statically typed.

Exactly the same arguments hold about the statically typed extensible records that CeXL supports. We can give exactly the same guarantee as for the optional fields that all records in the lists above will be the same for all list elements.

3.13 We Only Remove Absent Fields at Val Bindings

To make the feature *Optional Fields* work, we need to keep track of which fields have been optional in a record during type inference even if they are instantiated with `absent` because the field is not there. For instance we can write this code:

```
fun f {optional ?= v, ... = r} =
      {optional ?= v, ... = r}
```

If at some point, say in a function `g`, we call the function `f` and we want to be sure that the field `optional` is not present we could write the following:

```
fun ['b : ~{optional}] g (r : {- optional : 'a, ... : 'b}) =
      (print "g called"; f r)
```

The `-` in the record type here signifies that the field `optional` must be absent.

If we pass an empty record through many functions which all support various optional fields, the resulting record will end up having lots of absent fields which are explicitly represented if we don't do something about this. This could potentially give hassle since record extension in CeXL is strict. Therefore, in CeXL it has been decided that all absent fields are removed from the types of non-function and non-reference values at each point of closure in the semantics - which means at each val binding. This also avoids some weird behaviour in the language. Consider these 2 functions:

```
fun ['a : ~{dummy}, 'b : absent | present]
    unitId1 (a : {'b dummy : int, ... : 'a}) = a
fun unitId2 (a : unit) = a
```

The functions would get these value descriptions:

```
val ['a : ~{dummy}, 'b : absent | present]
    unitId1 : {'b dummy : int, ... : 'a} -> {'b dummy : int, ... : 'a}
val unitId2 : unit -> unit
```

We will just note here that `unit` is equivalent to both `()` and `{}` as in ML. We can call both of these functions with no fields:

```
val u1 = unitId1 ()
val u2 = unitId2 ()
```

If we didn't remove all absent fields at each val binding the value descriptions would have been:

```
val u1 : {- dummy : int}
val u2 : unit
```

The weird behaviour here is that `u1` and `u2` get different types depending on how the function they are returned from is typed even though the implementation is the same and the given arguments are the same.

However in CeXL we *do* remove absent fields at the val bindings so the value descriptions become:

```
val u1 : unit
val u2 : unit
```

So we don't have any weird behaviour and it should give less hassle with strict record extension.

3.14 Use Case of Mutual Forward and Backwards Compatibility

In this section we will go through a complete use case where both optional fields and extensible records are needed. An important goal for CeXL is to be able to keep 2 pieces of code mutually forward and backwards compatible, even though they are maintained separately. We will refer to such pieces of code as a "plugin" and a "scene". A reader who is familiar with 3D graphics applications might know why.

Plugins are usually written by one person - a "plugin writer". Preferably the plugin writer should be a highly disciplined and skilled CeXL programmer. Scenes are usually created independently by another person - a "user". Actually such a user typically doesn't write the code by hand. It is usually generated automatically by a 3D graphics application - but it could also be partly hand-written by expert users.

- Day 0: Some plugin writer writes a CeXL plugin called "Plugin Version 1" containing the function definition:

```
fun fancyPlugin {data1, ... = more} =
      {data1 = (* some code *),
       result1 = (* some more code *),
       ... = more}
```

  This function thus gets the value description:

```
val ['a : ~{data1, result1}] fancyPlugin :
    {data1 : t1, ... : 'a} ->
    {data1 : t1, result1 : rt1, ... : 'a}
```

- Day 1: User A gets "Plugin Version 1" and creates a fancy scene with it. User A saves this scene as "Scene1". "Scene1" is thus a CeXL program[3] and it contains the code:

```
val v1 = fancyPlugin {data1 = v0}
```

- Day 2: The plugin writer from before updates the plugin to "Plugin Version 2", now with the code:

```
fun fancyPlugin {data1, data2 ?= data2, ... = more} =
      {data1 = (* the old code *),
       data2 ?= (* some new code producing an optional field based on data2 *),
       result1 = (* some enhanced code using data2 if it is present *),
       ... = more}
```

  The function gets this new value description:

```
val ['a : ~{data1, data2, result1}, 'b : absent | present] fancyPlugin :
      {data1 : t1, 'b data2 : t2, ... : 'a} ->
      {data1 : t1, 'b data2 : t2, result1 : rt1, ... : 'a}
```

[3] Actually it will be what is called an *ICeXL* program - but since we don't define ICeXL here, assume that it is a CeXL program

- Day 3: User A upgrades to "Plugin Version 2" and *must still be able to load "Scene1"*! This requires optional fields - in this case the field data2.
- Day 4: Now User A modifies "Scene1" such that it now contains the code:

  ```
  val v2 = fancyPlugin {data1 = v0, data2 = v1}
  ```

  This scene is saved as "Scene2".
- Day 5: User A publishes "Scene2" on the Internet and user B downloads it. User B has only got "Plugin Version 1" - but *must still be able to load "Scene2"*! This requires polymorphic extensible records - in this case the row type variable `'a`.

Hopefully this gives an impression of *why* we need both optional fields and extensible records for the domain specific purpose that CeXL is designed for. It will be a future project to describe this domain specific behaviour in detail.

3.15 Examples Comparing CeXL and ML

1. ```
   (* ML would assume this function to operate on int *)
   fun line a b x =
         a * x + b

   (* CeXL value description: *)
   val ['a : word | int | real] line : 'a -> 'a -> 'a -> 'a
   ```

2. ```
   (* ML would complain that all fields of the record have to be known *)
   fun personToString {name, age, ...} =
         name ^ ", " ^ (Int.toString age) ^ " years old"

   (* CeXL value description: *)
   val ['a : ~{name, age}] personToString :
         {name : string, age : int, ... : 'a} -> string

   (* We can call personToString with larger records *)
   val s1 = personToString {name = "John Doe", age = 30, email = "john@doe.com"}
   ```

3. ```
   (* ML would complain that all fields of the record have to be known *)
   val getX = #x

   (* CeXL value description: *)
   val ['a : ~{x}] getX : {x : 'b, ... : 'a} -> 'b
   ```

4. ```
   (* ML does not have extensible records *)
   fun addName (name : string) r =
         {name = name, ... = r}

   (* CeXL value description: *)
   val ['a : ~{name}] addName : string -> 'a -> {name : string, ... : 'a}
   ```

5. ```
   (* ML does not support removing fields from records *)
   fun removeField {field, ... = r} =
         r

   (* CeXL value description: *)
   val ['a : ~{field}] removeField : {field : 'b, ... : 'a} -> 'a
   ```

6. ```
   (* ML cannot do functional updates of record fields *)
   fun updateField newValue {field, ... = r} =
         {field = newValue, ... = r}

   (* CeXL value description: *)
   val ['a : ~{field}] updateField : 'b -> {field : 'b, ... : 'a} -> {field : 'b, ... : 'a}
   ```

7.
```
(* ML does not have optional record fields *)
fun transformVertex matrix {point, normal ?= normal, ... = more} =
      {point = transformPoint matrix point,
       normal ?= fieldcase normal in 'b of
                   absent    => absent
                 | present n => present (transformNormal matrix n)
                 type ('b, Normal3D.t) ? end,
       ... = more}

(* CeXL value description: *)
val ['a : ~{point, normal}, 'b : absent | present] transformVertex :
      Matrix4x4.t ->
        {point : Point3D.t, 'b normal : Normal3D.t, ... : 'a} ->
        {point : Point3D.t, 'b normal : Normal3D.t, ... : 'a}

(* It can be called with or without normal, with or without other fields.
   The result has normal if and only if the argument has. *)
val v1 = transformVertex m {point = p1}
val v2 = transformVertex m {point = p2, normal = n2}
val v3 = transformVertex m {point = p3, color = c3}
val v4 = transformVertex m {point = p4, normal = n4, color = c4}
```

8.
```
(* CeXL optional fields can have shared polymorphism -
   unlike many other calculi for extensible records. *)
fun updateOptField newValue {field ?= value} =
      fieldcase value in 'a of
        absent =>      {field ?= value}
      | present old => {field = newValue, updatedBy = "updateOptField"}
      type {'a field : 'b, 'a updatedBy : string} end

(* CeXL value description (both 'a and 'b occur separately multiple times): *)
val ['a : absent | present] updateOptField :
      'b -> {'a field : 'b} -> {'a field : 'b, 'a updatedBy : string}
```

3.16 More Examples In Section 25

More examples of CeXL code can be found in section 25 which contains a few regression test programs for some of the more subtle issues of the semantics.

4 Previous Work

In this section we will first go through earlier research work and afterwards compare the semantics of CeXL with such work. In particular the focus will be on record calculi since this is the main difference between CeXL and Standard ML.

4.1 Record Calculi

[Card88] introduces structural subtyping and power types which is very similar to the polymorphism in object oriented languages with inheritance. It typically allows a larger record to be used as an argument for a function taking a smaller record as argument; but see [Cast94] for the full story of covariance versus contravariance.

[Ohor92] develops a calculus for polymorphic field selection of records as in ML. An implementation calculus is also developed where field offsets in records are statically computed. A translation between the calculi is made along with proofs of soundness. Field offsets are calculated by polymorphic field abstraction. They also argue why type systems with subtyping can never get statically computed field offsets: Because records are not "exactly typed".

[Wand88] reviews the basic operations on records and defines a syntactic protocol for objects with multiple inheritance by using record operations and λ-calculus. Type inference for records with extension and concatenation is presented, but concatenation requires some tricky semantics involving constraint equations. Infinite label sets are represented with finite semantic objects, but it seems there is a restriction that extension variables for records must have exactly the same set of explicit labels (i.e. the labels which are forbidden in the extension) as the record they extend.

[Wand87] defines a record calculus with extensible records and variants. It includes a complete type inference algorithm (unfortunately it had an error in the original paper; it was subsequently fixed by the same author). Detailed proofs and reduction from the semantics to unification is given. It is also argued how records and variants can be used to represent simple objects with structural inheritance.

[Remy89] defines a semantics for primitive record and variant operations upon which other record and variant operations can be defined as macro-syntax as desired. Record fields consist of both a type and flag denoting presence or absence of the field. It has type inference, is sound and complete, uses the basic unification algorithm and can handle recursive types as well. The semantics assumes a finite set of labels though and would have to be slightly rewritten to use an infinite set of labels.

[Remy92a] covers how to encode record concatenation with record extension and this result is applied to a natural extension of ML. Concatenation requires records to be abstracted into functions though, which (as mentioned in the paper) requires investigation before it can be considered free at runtime. Both symmetric and asymmetric concatenation is explored and free and strict extension and field removal. The work is compared to other calculi and encoding multiple inheritance using records with concatenation is explained.

Projective ML [Remy92b] defines record extension and polymorphic records by introducing general elevation and projection constructs. This gives a fairly

simple semantics preserving many of the properties of λ-Calculus such as subject reduction and the Church-Rosser property. They introduce records with default values. They also support an operation taking all fields from one record except for one field which is taken from another record - whether that field is present or not. However there seems to be a problem with nested records in this semantics.

4.2 How ξ-Calculus Relates to Other Work

The base calculus of CeXL is called ξ-Calculus and we will briefly go through how this relates to other languages and calculi.

4.2.1 Extensible Records and Field Absence and Presence

[Remy89] is probably the calculus which resembles ξ-Calculus most in terms of what it can do with records and how this works. However, to represent infinite sets of labels with finite semantic objects it actually turns out that ξ-Calculus implements extensible records much as the semantics in [Wand88]. It handles record extension and treatment of infinite sets of labels in much the same way. ξ-Calculus does not require the equivalent of "extension variables" (to use the terminology of that paper) to have the same set of explicit labels as the record they extend though. As an example, [Wand88] would not be adequate to type the following (using CeXL syntax):

```
fun f {a, ... = r} =
      {a = a, b = "Hello", ... = r}
```

To handle this, we keep track of which labels are forbidden for the extension variables - somewhat in the style of [Remy92a] except that it only works in [Remy92a] because records are really abstracted as functions. If the encoding of [Remy92a] is used without function abstractions, it would not be adequate to type the above example either.

ξ-Calculus records are not always polymorphically extensible. Only when the programmer explicitly types them as such or uses them as such are they polymorphic and do they incur the potential overhead of extension polymorphism. So where needed by the programmer, we can type our records exactly which is advantageous according to [Ohor92]. This is in contrast to subtyping [Card88], where one can typically always pass a larger record to e.g. a function taking a record as argument. The ξ-Calculus record polymorphism also differs from subtyping in that we don't merely ignore additional fields - we can actually capture them in function parameters and we are able to pass them on separately.

ξ-Calculus does not support concatenation of records, since it seems to either add complexity to the language as in [Wand88] or to require that records are treated as function abstractions as in [Remy92a]. Record concatenation is considered more thoroughly in section 7.5.

The semantics for extensible records in ξ-Calculus is not given in terms of primitive record operations as in most other research. Instead it is incorporated in record pattern-matching and record expressions, since we need the semantics of a complete language which incorporates these features.

To be able to write programs which depend on the presence or absence of record fields, we have a fieldcase construct. This in turn is constructed using

parametrized types with restrictions. This is possibly related to Oligotypes [Heng87], refinement types [Free91] and intersection types [Bare83], [Pier91] but I have not looked into any of this. The parametrized types with restrictions also come in handy when we are defining the semantics for extensible records, most importantly to prevent an extension variable of a record to be instantiated with records containing forbidden fields. It also ensures that extension variables are instantiated with records in the first place, and not any other types. We would not be able to use a record calculus such as [Wand87] since it does not explicitly represent fields as present or absent and thus cannot explicitly assign types to optional fields.

Record extension in CeXL is strict whereas [Wand88] and [Remy89] uses free extension. We get the power of free extension though since we can case out optional fields and build a new record where we can be sure to either exclude or include an optional field as we desire.

Finally it should be noted, that we can explicitly specify optional fields with a specific type in ξ-Calculus - such as an optional field of type `'a list`. The semantics in [Remy89] can do this too, whereas the semantics in [Wand87], [Wand88], [Remy92a], [Remy92b] and [Ohor92] cannot do this and neither do they seem to mention this as a flaw anywhere. Subtyping cannot do this either.

4.2.2 The Rest of ξ-Calculus

The rest of ξ-Calculus is more or less like the core of the semantics of Standard ML '97 [SML97]. However we do not handle declaration of types or datatypes in ξ-Calculus. As we shall see, this is handled elsewhere in the language, but it is still basicly done as in Standard ML.

5 Walk-through of the CeXL Definition

This section is an easy walk-though of the Definition of CeXL. The actual definition with all the technical details is given in the appendices.

5.1 The Languages and Calculi Presented

The final language we wish to present is called CeXL. This in turn is reduced *purely syntactically* to Naked CeXL, which is the essential part of the language. Naked CeXL is implemented on top of a more fundamental calculus called ξ-Calculus. The languages will be presented in a bottom-up fashion, by starting with all the hard work on ξ-Calculus.

All the semantics for ξ-Calculus is presented first. Then we present Naked CeXL and a translation from Naked CeXL to ξ-Calculus. Finally, the real syntax of CeXL is presented along with how to reduce this into Naked CeXL. Everything is put together at the end, by a description of the initial environments for the semantics and for the translation into ξ-Calculus, and a description of how to apply everything to type-check and execute a CeXL program.

5.2 About the Separate ξ-Calculus

A goal of the calculus is to be able to translate Naked CeXL into ξ-Calculus and afterwards do type inference directly in ξ-Calculus. So there is no type inference on the source programs - not even on the reduced Naked CeXL programs. This turns the calculus into a kind of base programming language for implementing Naked CeXL and thus also CeXL.

One feature of Naked CeXL that we cannot represent in ξ-Calculus, is the datatype definitions and the exception declarations. The types and constructors generated from such declarations are inserted directly into the ξ-Calculus code without separate declarations. However, we have to insert declaration of exception constructors to ensure correct scoping of type variables and correct static semantics of exception declarations.

There are clearly disadvantages of defining ξ-Calculus separately. It complicates the whole semantics of the language specification, because the semantic behaviour of the language is spread out into both ξ-Calculus and Naked CeXL. This would definitely complicate proving properties about the language and reasoning about the semantics. It also makes the task of writing the specification bigger.

There are also advantages though. One can consider the runtime behaviour and the essential type system by looking only at ξ-Calculus. This is not the main motivation though. The main motivation is to make it easier to implement the language more directly from the specification, in a way that keeps the implementation more manageable and tractable. One fact is that it minimizes both the type inference implementation and the implementation of an interpreter - at least for a simple interpreter which is as close to the semantics as possible. For a more decent implementation such as a compiler, it also saves one from having to think of more than ξ-Calculus after type inference is done.

An exception to this though is, if one wishes to make the implementation interoperate with other languages or through dynimcally loaded libraries. Here

one would have to worry about Naked CeXL to support language constructs like datatype declarations.

It is the hope that the use of a separate ξ-Calculus and the quite straightforward semantics that it has been given, will lower the abstraction level required to read this language specification. There are hopefully not as many subtle and quite hidden features in this specification, as is the case with the Definition of Standard ML '97 [SML97].

5.3 Presenting Implicitly Typed ξ-Calculus

On the following page, we will see the type system, the semantic objects and a syntax for the expression language of ξ-Calculus in all it's glory from the specification. This is the most essential part of the specification and it fits on a single page, which makes a very solid point of reference for how ξ-Calculs looks. You will probably have to read the semantics of the specification to fully understand everything, but the 2 pages following have some comments with important facts about the type system.

5.4 Implicitly Typed ξ-Calculus

The following is the type system and the expression language of ξ-Calculus. This is the *most important page* of the specification. The comments on the next 2 pages give a slight idea of its meaning.

5.5 Syntax and Semantic Objects for the Type System

		Syntax	**Name**		**Semantic Objects**
		lab	Lab	=	"Record labels"
		x	VId	=	"Variable identifiers"
		α	TyVar	=	"Type variables"
		β	MetaVar	=	"Meta variables"
		a	PName	=	"Type parameter names" for constructor types
		d	TyName	=	"Type names"
		c	CName	=	"Constructor names"
ω	::=	$\{lab_1, \ldots, lab_n\}$	ExclLabs	=	Fin(Lab)
ψ	::=	$[a_1 = \psi_1, \ldots, a_n = \psi_n]d\{c_1, \ldots, c_k\}$	TyPat		$\cup_{p \geq 0}$ (PName $\times$ TyPat)^p $\times$ TyName $\times$ Fin(CName)
$\psi^{/m}$	::=	$\psi_1 \ / \cdots / \ \psi_m$	TyPats	=	$\cup_{p \geq 1}$ TyPat^p
ξ	::=	$\circ \mid \omega \mid \psi^{/m}$	Restrict	=	$\emptyset \cup$ ExclLabs $\cup$ TyPats
γ	::=	$\alpha :: \xi \mid \beta :: \xi$	Vars	=	TyVar $\times$ Restrict $\cup$ MetaVar $\times$ Restrict
ρ	::=	$\{lab_1 : \tau_1, \ldots, lab_m : \tau_m; \tau\} \mid \{\}$	Row	=	(Lab $\stackrel{fin}{\rightarrow}$ Type) $\times$ Type $\cup \emptyset$
ϕ	::=	τ **?** τ'	Field	=	Type $\times$ Type
κ	::=	$[a_1 = \tau_1, \ldots, a_n = \tau_n]d\{c_1, \ldots, c_k\}$	ConsType	=	$\cup_{p \geq 0}$ (PName $\times$ Type)^p $\times$ TyName $\times$ Fin(CName)
		$\tau \rightarrow \tau'$	Fun	=	Type $\times$ Type
τ	::=	$\gamma \mid \tau \rightarrow \tau' \mid \rho \mid \phi \mid \kappa$	Type	=	Vars $\cup$ Fun $\cup$ Row $\cup$ Field $\cup$ ConsType
σ	::=	$\forall[\alpha_1 :: \xi_1, \ldots, \alpha_n :: \xi_n].\tau$	TyScheme	=	$\cup_{p \geq 0}$ (TyVar $\times$ Restrict)^p $\times$ Type
r	::=	$\{lab_1 : \tau_1, \ldots, lab_n : \tau_n\}$	OrderRow	=	Lab $\stackrel{fin}{\rightarrow}$ Type

5.6 A Syntax for ξ-Calculus Expressions

(*expression*) e ::= $\lambda x.e \mid e_1\ e_2 \mid x \mid e : \tau \mid c : \sigma \mid c$ **ex** $\tau \mid scon \mid$
$\{\ \} \mid \{lab_1 = e_1, \ldots, lab_m = e_m, e\} \mid e_1$ **?** $e_2 \mid$
let $p_1 = e_1$ **;** $\cdots$ **;** $p_n = e_n$ **in** $e \mid$
letrec $p_1 = e_1$ **;** $\cdots$ **;** $p_m = e_m$ **in** $e \mid$
letex $c_1 : \tau_1, \ldots, c_m : \tau_m$ **in** $e \mid$
e **handle** $p_1 \rightrightarrows e_1 \parallel \cdots \parallel p_m \rightrightarrows e_m \mid$ **raise** e
case e **of** $p_1 \rightrightarrows e_1 \parallel \cdots \parallel p_m \rightrightarrows e_m \mid$
fieldcase e **in** α **of absent** $\rightrightarrows e_1 \parallel$ **present** $p \rightrightarrows e_2$ **type** $\tau \mid$

(*pattern*) p ::= $x \mid p : \tau \mid x$ **as** $p \mid c : \sigma \mid c(p) : \sigma \mid c$ **ex** $\mid c(p)$ **ex** $\tau \mid scon \mid$ _ $\mid$
$\{\ \} \mid \{lab_1 = p_1, \ldots, lab_m = p_m, p\} \mid p_1$ **?** p_2

5.7 Comments on the Type System

The following gives a brief idea of the type system of ξ-Calculus:

- The ξ represents restrictions on the allowed instantiations of type variables and meta variables.
- The restriction $\circ$ means "no restrictions" - so $\alpha :: \circ$ is like a "normal" type variable in Standard ML.
- The restriction ω represents a record excluding the labels ω.
- The restriction $\psi^{/m}$ is a list of allowed (mutually disjoint) constructor types. Each such type in the list is a ψ.
- The empty record $\{\}$ in ρ really represents: "All as-of-yet undefined fields are absent". This is actually what is seen as the type `unit` in CeXL. All concrete record values in CeXL consist of some or no extensible records extending one another and finally being extended by $\{\}$ at the end. Each such extensible record is written as $\{lab_1 : \tau_1, \ldots, lab_m : \tau_m; \tau\}$. It is only in functions, type declarations and the likes that it makes sense to have extensible record types which are not ending with $\{\}$, in which case they would end with a meta variable or a type variable.
- $\beta :: \omega$ may *not* be instantiated with a record *without fields* with new $\beta' :: \omega'$, i.e.: $\beta :: \omega \mapsto \{;\beta' :: \omega'\}$. Instead it should be instantiated with $\beta :: \omega \mapsto \beta' :: \omega'$. If we did not forbid this we would have problems unifying the types $\beta :: \omega$ and $\{;\beta :: \omega\}$ with each other. We ensure this in the inference rules of record expressions and record patterns for ξ-Calculus. The reason why $\beta :: \omega$ and $\{;\beta :: \omega\}$ would have to unify with each other is that they both denote an extensible record excluding the fields ω - i.e. they denote exactly the same thing.
- Consider the a_i used in the $a_i = \tau_i$ in the constructor parameters in ConsType and the $a_i = \psi_i$ in the parameters for the type pattern constructor in TyPat. These a_i are to be considered parameter names for constructor types. They will correspond to the type variable names of the closed type scheme of constructors. However they may not be considered type variables - which is why they are denoted a_i and not α_i. They are for allowing type parameters to be identified based on names rather than the order in which they appear. This is for giving CeXL the feature *Named Type Parameters*.
- The ConsType contains a finite set of constructor names (the Fin(CName)). These are the names of the constructors that the ConsType has declared. This is only used by an implementation to ensure correct implementation of pattern-matches and the checks that pattern-matches are exhaustive where this is required. This also means that they are not used at runtime, so an implementation can remove them after type inference.

  An example of a constructor type in CeXL is the type `bool` which would be represented by: $[]bool\{true, false\}$. The type `int` is represented by: $[]int\{\cdots\}$ to signify that it has many constructors (i.e. all the supported integer constants). The type `int list` becomes: $[a = []int\{\cdots\}]list\{nil, ::\}$.

- ConsType is also used for exceptions. Here only one predeclared constructor is used: $[]exn\{\}$. It is treated specially, since it gets its constructors added separately through exception declarations. Therefore we use an empty list of constructor names for it. The constructor $[]exn\{\}$ may only be used with the **ex** constructs and in type specifications in ξ-Calculus. Restrictions on the number k of constructor names allowed where ConsType occurs in will ensure this. When $k \geq 1$ it cannot be $[]exn\{\}$.

- The class Field of the form τ **?** τ' denotes the type of a record field. τ may only be the type $[]present\{present\}$, the type $[]absent\{absent\}$ or a meta variable or type variable properly restricted to such instantiations. τ' is the type of the value in the record field.

- σ is used in the value environment and for representing the closed types of constructors. Constructors will have σ of one of the following 2 forms:
  $\forall[\alpha_1 :: \xi_1, \ldots, \alpha_n :: \xi_n].[a_1 = \alpha_1 :: \xi_1, \ldots, a_n = \alpha_n :: \xi_n]d\{c_1, \ldots, c_k\}$ or
  $\forall[\alpha_1 :: \xi_1, \ldots, \alpha_n :: \xi_n].\tau' \rightarrow [a_1 = \alpha_1 :: \xi_1, \ldots, a_n = \alpha_n :: \xi_n]d\{c_1, \ldots, c_k\}$.

  As we shall see later these type schemes must generalize some type τ, which is written $\sigma \succ \tau$.

  During type inference the names a_i must be chosen so that they are the same as the α_i to avoid confusing the programmer and to avoid problems in identifying parameters in type patterns.

- r is only used during unification, where we will need to store a finite set of labels mapped to types. This is for transforming a record into a semantic representation where records are not treated as a recursive list of smaller records extending each other - as they usually are when represented by ρ.

5.8 Static Semantics of ξ-Calculus - Unification

Section 14 gives explicit inference rules for the unification between 2 given types. We need this to be able to implement a unification algorithm for the semantics of ξ-Calculus. The unification relation between 2 given types can be briefly stated as follows:

It is the standard unification relation with the following changes:

- All polymorphic variables and meta variables have restrictions on them which limit what they may be instantiated with. These restrictions are maintained by the unification
- Equality of records during unification is modulo reordering of fields. This is achieved by converting any records encountered to a semantic representation (denoted r in the syntax for the type system) which is independent of the order of the fields of the record. Unification of records is done using this representation because record types in ξ-Calculus may be composed of several smaller record types extending each other as a sequential list of records. For example, if we did not do this we would have problems unifying $\{lab_1 : \tau_1; \{lab_2 : \tau_2; \tau\}\}$ and $\{lab_2 : \tau_2; \{lab_1 : \tau_1; \tau\}\}$ with each other. Such two record types must unify, since they both denote the type of an extensible record with two fields, lab_1 and lab_2 of types τ_1 and τ_2 respectively, where both record types are extended with τ. So the two types denote exactly the same record.
- The unification relation deals with maintaining the property that extensible record types may never redefine the same field twice.
- Record fields must have field types of the form $\tau \ ? \ \tau'$ where τ is either $[]absent\{absent\}$ or $[]present\{present\}$ or a type variable or meta variable which is properly restricted to these instantiations
- Non-extensible records are represented modulo equality of $\{\}$ at the end of the record with records of absent fields and new $\{\}$ types - e.g. equalities like:
  $\{\} = \{lab : []absent\{absent\} \ ? \ \tau; \{\}\}$

Ordered records r and meta variables β from our semantic objects are only used for the unification when implementing the semantics. r is for transforming a record into the semantic representation mentioned above and β are the place holders for unknown types during unification. One exception though is that we explicitly have to handle meta variables in the rules for explicit type specifications in ξ-Calculus. This is necessary to support the feature *Partial Type Instantiation*.

5.8.1 Formal View and Implementation of Meta Variables with Restrictions

Restrictions on meta variables must always be defined at the time the meta variable is created - and they may never change. To restrict an existing meta variable further, it has to be bound to a new meta variable with a tighter but compatiple restriction. A meta variable may also only be bound to another type or meta variable once, and never be rebound. Whenever a meta variable is bound to a type (or a type variable or a meta variable), it must be checked that the type respects the restriction of the meta variable. We can only write a correct unification relation if we follow those guidelines.

When dealing with meta variables with restrictions, we adopt the following view:

- Meta variables will have restrictions at creation time which may never change. Meta variables with restrictions are denoted $\beta :: \xi$, where β is the variable and ξ its restriction.
- We will assume that we have an environment *env* which binds meta variables to types.

When implementing the unification relation there are 2 simple ways of representing meta variables with restrictions:

- We can emulate having just 1 environment, by keeping an updatable reference cell with each meta variable. The reference is either a restriction or a bound type. Restrictions may be updated to bind a compatiple type, which in turn may just be another meta variable with a tighter but compatiple restriction. However types may not be updated once they have been bound the first time.
- It could also be that one prefers to have a data structure where restrictions and bound types are stored separately, which can be done by always keeping the restriction (which may never change) with each meta variable and an updatable reference cell containing an optional bound type.

5.8.2 Unification of Restrictions

In all the inference rules presented where two restrictions have to be unified, it is always enough to check that they are simply equal. So we never do a general unification on restrictions. In section 7.2 we will consider some of the problems which arise if we were doing unification of restrictions.

All the special issues of restrictions should be handled appropriately by the inference rules of the unification relation which will be presented.

5.9 Static Semantics of ξ-Calculus

All the static semantics of ξ-Calculus with inference rules is presented in sections 15 and 16, and they assume that the unification relation between two types from section 14 is used.

Some of the most important facts about this semantics are the following:

- Scoping of type variables is implicit. Complete inference rules for determining this scoping are given. The scoping occur at the let and letrec constructs. The sections 15.4 and 15.5 in the appendix give a detailed description of how the scoping occurs. Section 16.3 gives even more detail.

  If it ever becomes desirable to add the explicit scoping of type variables found in Standard ML, this would amount to storing these explicit type variables in ξ-Calculus during translation from Naked CeXL, some changes to the inference rules for the implicit scoping, and a slight change in the closure operation.

- We do some tidying up of types when doing the closure operation at let and letrec bindings (which correspond to val bindings in CeXL according to the translation into ξ-Calculus). In particular we remove all absent fields from record types in the types of non-function and non-reference values, to avoid the weird behaviour described in section 3.13. Doing this for function values or reference values may be unsound and has been suppressed in the language specification from version 0.9.3.

- The closure operation of a type for quantifying types as type schemes is done at let and letrec bindings, which is similar to what is done in Standard ML [SML97]. The type variables which are implicitly scoped at an enclosing let or letrec binding is excluded from this quantification, as is also the case in Standard ML. One slight difference from Standard ML is that we require principal typings at top-level and in structures at each let and letrec binding in xi-Calculus (i.e. each val binding in CeXL) and we do not allow taking the next or previous let or letrec into consideration. As an example, the following is legal both in CeXL and in Standard ML:

  ```
  val _ = let
            val x = ref nil
            val _ = x := [5]
          in x end
  ```

  On the contratry, the following is prohibited in CeXL but a Standard ML implementation may choose to accept it:

  ```
  val x = ref nil
  val _ = x := [5]
  ```

  In CeXL, we would have to explicitly type the `ref nil` with `int list ref`, but one may have to do the same for some Standard ML implementations.

  The sections 15.3, 15.4 and 15.5 in the appendix give a detailed description of the value-restriction and the closure operation in CeXL. The inference rules (57) and (58) and the comments for these are also essential. It is recommended to read this to understand the issues involved.

- The inference rules are actually quite similar to what is done in Standard ML. However it has been avoided to make the semantics recursive at letrec bindings. This is a simplification compared to ML. We also have less restrictions on how type names may escape local scopes and pattern-matches, which also simplifies things. This is possible because in the translation from Naked CeXL to ξ-Calculus, we generate constructor names and type names for datatypes which are unique across the entire program.

- There are no special inference rules to deal with extensible records, the fieldcase construct and restrictions on type variables. Most of the complexity of extensible records and type variables with restrictions is handled in the unification relation of section 14.

- We have to deal with meta variables in the inference for explicit type constraints in ξ-Calculus. This is actually part of how a unification algorithm for ξ-Calculus is implemented, and it can be considered quite a "hack" in the semantics. It is necessary to support the feature *Partial Type Instantiation*. An alternative would be to move the semantics of declared types into ξ-Calculus, but that would take away the simplicity of ξ-Calculus. Declaration and instantiation of types is handled already during the translation from Naked CeXL to ξ-Calculus.

  A more clean way to handle explicit type constraints, might be to handle the well-formedness of these during the Naked CeXL to ξ-Calculus translation. We already handle the semantics of type function applications and declarations at that point, so it would in fact be a natural place.

- In the type specification for the result type of the fieldcase construct, all occurrences of the type variable which is "cased upon" in the fieldcase must be explicitly mentioned. So these cannot be left out by the CeXL programmer using the CeXL feature *Partial Type Instantiation*. This is due to the way that the rule for the fieldcase works in the static semantics. It is potentially a hard problem to change the semantics to releave the programmer from this burden, but fortunately it is a small burden.

5.10 Dynamic Semantics of ξ-Calculus

The dynamic semantics of ξ-Calculus is presented in section 17. It is quite similar to that of Standard ML and is quite straight-forward. Since CeXL is reduced entirely to ξ-Calculus, this actually gives the complete runtime behaviour of CeXL programs. We shall not go through this semantics any further.

5.11 Grammar of Naked CeXL

The grammar for Naked CeXL is given in section 18. It is a BNF grammar, which is quite similar to that of the bare language in the Definition of Standard ML [SML97]. The following comments apply to the syntax. This mostly relates to where the syntax differs from Standard ML:

- The notation `{a, b, ... = c}` for records is trying to be backwards compatiple with Standard ML '97 [SML97]. [Remy89] uses the notation `{a, b, c...}` and the small difference is mostly to try to prevent programmers from making typographic mistakes. I claim that `{a, b, c...}` is too close to `{a, b, c,...}` which means something else in pattern-matches. The use of the extra `=` is an attempt to make the notation more consistent with the notation for matching record fields into variables and the notation for putting values into record fields in record expressions.

- The grammar for optional fields tries to be as minimal as possible, by not showing anything in front of ordinary present fields and showing only a `-` in front of absent fields.

- The restriction of records with forbidden fields are represented by "negated" curly braces `~{lab1, lab2}`. This is to symbolize a negation of the record braces. The negated braces are used to keep the possibility open for allowing restrictions to include records in the future if necessary.

  I have considered using `}lab1, lab2{` but this soon becomes *very* disturbing to read - especially when it is in the vicinity of the normal record notation. Using `{{lab1, lab2}}` gives conflicts in the lexical analysis when making `}}` a reserved symbol, so neither is this good enough. Consider the following expression to realize what the conflict is:
  `{loc = {pos = 5, line = 2}}`.

- The fieldcase construct looks something like
  `fieldcase v in 'a of absent => e1 | present p => e2 type t end`
  which can be read as "we case v in the variable 'a of the cases ... all having the type t". The use of `in`, `of`, `type` and `end` tries not to introduce new keywords unnecessarily. The words `absent` and `present` in the *fieldmatch* rule are not keywords, but rather identifiers for the predeclared variables, which have to be handled specially in the fieldcase construct.

- The syntax of type declarations and datatype declarations introduce a different syntax for the type parameters. We have to introduce *some* kind of new syntax to accommodate the specification of restrictions. I have taken this opportunity to interpret the type parameters sort of like a "record" where the type variable *names* matter, rather than as a "tuple" as in Standard ML where only the *order* of the parameters matters. The usual

"tuple" notation for type variables found in Standard ML, is added as syntactic sugar in the full CeXL grammar. The grammar `['a : int | real]` is inspired by what is mentioned in [ML2000]. One reason to use `[` and `]` around the parameters rather than for instance `{` and `}` is to avoid confusion with the curly braces used in record types and restrictions.

I conjecture that the "record" notation for type parameters can make programs more tractable when using many type parameters on a type - and given that optional fields and extensible records require type parameters, this will definitely come in handy. Furthermore, some of the domain specific programs that CeXL is designed for really *do* use *many* type variables.

We also use this notation for placing restrictions on type variables in front of val bindings or fun bindings. However, as was already argued for the semantics, we do not really quantify with these variables as in Standard ML '97. Semantically we scope type variables at these points - but some variables may be scoped further in or further out in the semantics of ξ-Calculus.

5.12 The Naked CeXL to ξ-Calculus Translation

In section 19, the rules for translating from Naked CeXL to ξ-Calculus are given. This translation works by operating on Naked CeXL syntax and translating this into ξ-Calculus syntax. The translation is presented as translation functions taking Naked CeXL syntax, and usually some environment, as argument and producing ξ-Calculus code from this. There are also translation functions which only manipulate environments.

This translation is mostly syntactic. However some semantics is also used. In particular, declarations of restrictions, types and datatypes must be checked according to the static semantics of ξ-Calculus. A closure operation is also done on datatypes, to quantify them properly and allow them to be recursive. During the translation, no types or datatypes are declared in the resulting ξ-Calculus code. However, exceptions are declared, to ensure correct scoping of type variables. This also ensures that the types of declared exceptions are checked according to the ξ-Calculus static semantics.

All declared types (datatypes do not count here) only exist in this translation. They are instantiated immediately during the translation. This is done by application of type functions in a way which supports *Partial Type Instantiation*. The details of this are described in section 19.3.

It is also during this translation, that unique constructor names and unique names for datatypes are generated. The simple structures which make up a simplified version of the module system found in Standard ML, are also eliminated during the translation. This is done by appropriate prefixing of identifiers. A set of translation functions prefix the identifiers declared at top-level in a structure appropriately.

All of this translation is quite tedious and looks daunting. It should however be quite straight-forward to implement.

5.13 The Full Syntax for CeXL

The complete lexical syntax and grammar for CeXL is presented in the sections 20 and 21. The syntax and the way it is presented is quite similar to that of Standard ML, except for what is already described for Naked CeXL, so there is not much to say about it.

5.14 CeXL To Naked CeXL Translation

Section 22 presents the rules for transforming CeXL into Naked CeXL. So this part works sort of like a strip-show :-) It is quite similar to how the full grammar for Standard ML is reduced to the bare ML language in the Defintion of Standard ML, so there is no reason to describe it further.

5.15 Putting It All Together

The sections 23 and 24 complete the language specification, by putting all the big parts together. All initial environments used for the semantics and the translation from Naked CeXL to ξ-Calculus are presented in section 23 . In section 24, it is described how a CeXL program is translated into ξ-Calculus and type-checked in the appropriate environments, and how it is subsequently executed. This completes the walk-through of the Definition of CeXL.

6 Notes on Implementation

In this section there will be a few hints on compiler implementation. In particular, it will be justified that CeXL programs can be compiled into efficient code when doing whole-program compilation - without even requiring very fancy compiler technology. We will also go through the implementation supplied for this language specification.

6.1 Removing "Weak Typing"

Overloaded operators are typed as parametrized types with restrictions, similar to `real | int | word`. As mentioned in section 2.1, this could be considered a kind of weak typing. It is possible to specialize this away at compile time - except if the types are to be exported through a foreign function interface or other kinds of binary program interfaces. For the domain specific purpose that CeXL is designed for it is actually intended that all source code be available, to allow global program optimizations. This allows us to do whole-program compilation. In this case, we can *always* specialize the above types away.

6.2 Instantiating All Polymorphism During Compilation

If we wish to do whole-program compilation of ξ-Calculus, it might be desirable to instantiate all polymorphic types during compilation (i.e. to monomorphorize programs).

The fieldcase and all other constructs introduced in CeXL will not give any problems or any huge blowup in the size of the generated code when instantiating all polymorphism (except maybe in very unrealistic programs). In fact, this can eliminate fieldcase entirely during compilation, as well as the need to represent the absent and present constructors at runtime at all. This can also eliminate record polymorphism, thus reducing the extensible polymorphic records to extensible monomorphic records, which again can be reduced to ordinary static records as found in ML (by e.g. inserting appropriate record pattern-matches and record expressions).

This is an easy way of demonstrating that at least for whole-program compilation, the extensible records of CeXL can be implemented without giving any significant runtime overhead (extending monomorphic records *might* give some copying occasionally). However, none of this will be proved here.

6.3 A Compiler Will Be Another Project

Writing a compiler for CeXL will be a future project of mine, and I will not describe this any further here.

6.4 The Implementation Provided

Alongside this language specification, some essential parts of a CeXL implementation are provided. This implementation demonstrates that the language can be implemented quite easily. The following code is provided online:

- The abstract syntax for the ξ-Calculus type system and expression language. This is what is presented in section 13.
- An interpreter for ξ-Calculus is included. It implements the dynamic semantics of section 17.

  There is a very neat trick in this implementation which deserves mentioning. A function-value in the interpreter is implemented as a function taking a ξ-Calculus-value as argument and returning a ξ-Calculus-value as result. That is, by this datatype clause: `datatype t = Fun of t -> t` where `t` also includes the constructors for all other ξ-Calculus-values. This way we can also use precompiled functions as function values. Whenever a function is declared in ξ-Calculus, the interpreter "packs" the interpretation of the closure into such a function. So when calling a function we actually don't care if it is interpreted or not - we can just execute it and get the result.
- A simple pretty-printer for showing ξ-Calculus types using CeXL syntax.
- The unification relation for ξ-Calculus of section 14. This consitutes a large part of the entire unification algorithm.

  Most of this implementation of the unification relation deals with the aspects required to support extensible records, optional fields and type variables with restrictions, so this is significantly different than for the usual unification relation of ML.
- The static type inference for ξ-Calculus of sections 15 and 16 is implemented using a unification algorithm. This is the last part of the unification algorithm.
- The abstract syntax for Naked CeXL along with functions for reducing most of the Full CeXL syntax into Naked CeXL syntax. This is what is presented in sections 18, 21 and 22.
- The translation from Naked CeXL to ξ-Calculus of section 19.

A binary version of all the above including a parser is also provided, along with a fairly large set of CeXL programs for regression-testing. Everyting is available online at:

- `http://www.CeX3D.net/cexl/`

7 Results of Other Language Features Which Have Been Considered

During the design of CeXL, other language features than those in the final language has been explored. I will mention some important conclusions here. These could be considered research results.

7.1 Relating Free Extension to Exactly Typed Records and Optional Fields with Fieldcase

In this section we will justify that when requiring records to be *exactly typed* while supporting *optional fields* in a way *usable by a fieldcase construct* we really get all the power of *free extension* of records - but in a way which must use *strict extension at the type level* in order to work. This also means that exactly typed extensible records with optional fields usable for a fieldcase construct prevents us from supporting truly free record extension without representing it as strict extension at the type level. The use of the term *exactly typed* is as explained in [Ohor92]. Let's state all this as one claim:

> **Claim**: *Exactly typed records* supporting *optional fields* in a way *usable by a fieldcase construct* in a language with ML's polymorphism implies that *record extension must be strict* at the *type level*.

Justification: We will do the justification by trying to give a type for a function doing free extension with a field `f` for a record. We will get to the conlusion that either it must be typed with strict extension or else we get conflicts no matter what we try.

The types one could consider for a function adding the field `f` to a record are only the following when we want *exactly typed* records:

```
(* free extension type 1 - gives conflict with unification *)
'a -> {f : int, ... : 'b}

(* free extension type 2 - gives conflicts with optional fields for fieldcase *)
'a -> {f : int, ... : 'a}

(* free extension type 3 - the types really use strict extension *)
{'b f : 'c, ... : 'a} -> {f : int, ... : 'a}
```

The comments indicate the conclusions we will get to during this justification.

So consider the first type:

```
'a -> {f : int, ... : 'b}
```

We can easily make this work at first sight because the `'a` and `'b` are 2 independent types. We can just say that `'a` may or may not contain the field `f` - we don't care. The record `{f : int, ... : 'b}` contains the field `f` and we can easily define that `'b` may not contain `f`. The problem here is that we have no way of relating `'a` and `'b` to each other. So for instance we could instantiate `'a` with `{}` and `'b` with `{dummy : real}`. But if this is supposed to be the type

of a function which only adds the field `f` to a record it is surely a problem that it unifies with a function which also adds a field `dummy`! We will have problems implementing such a function which chooses to add or remove fields correctly at runtime only depending on the instantiation of its type where it is used. So this type can be ruled out.

Consider the second type suggested:

```
'a -> {f : int, ... : 'a}
```

If we want *exactly typed* records we must accept that `'a` is a record type which either contains the field `f` or does not contain the field `f`.

If we consider when `'a` does not contain `f` then everything is fine. This is also the only way that CeXL will accept to interpret such a type by disallowing `'a` to contain the field `f` in the above type.

If on the other hand we assume that `'a` contains the field `f` then this type starts to be doubtful because the record `{f : int, ... : 'a}` would contain `f` twice then. This could be solved by defining that the `f` in `'a` is replaced by the explicitly mentioned `f` - which means that `'a` inside the record does really not contain `f`. This seems to be a conflict at first since the 2 occurences of `'a` would not unify then. We could solve this by making a distinction of when `'a` occurs alone and when it occurs inside another record type.

But a problem arises when we want to support optional fields. If the field `f` is optional as in this type:

```
'a -> {'b f : int, ... : 'a}
```

Then if `f` is already present in `'a`, the record `{'b f : int, ... : 'a}` would contain an `f` potentially of a different type depending on whether the explicitly mentioned `f` is present or absent. And then the fact that `f` is supposed to be absent is not even true! We could decide that an explicit absent `f` removes the field `f` from the type `{'b f : int, ... : 'a}` - i.e. `f` is removed if `'b` is instantiated to `absent`. This would be the only consistent behaviour here.

The problem with this comes when we actually want to extract the field `f` from a record of type `{'b f : int, ... : 'a}`. If the explicitly mentioned `f` overrides any `f` present in `'a` then consider this program:

```
fun g ({f ?= f, ... = r} : {'b f : int, ... : 'a}) r2 =
      if true then r else r2
```

If we were to allow such types as above then this program would take out an optional `f` of the first record given to `g`. The result type of the function must be compatiple with both `r` and `r2`. According to the type `{'b f : int, ... : 'a}`, `r` could contain a field `f` of another type than int! Thus the following call to `f` would typecheck but be unsound:

```
val v = g {f = 5} {f = "Unsound"}
```

The following sure wouldn't be any better, since we have `f` twice in the first record:

```
val v = g {f = 5, f = "Unsound"} {f = "Unsound"}
```

So clearly we must either disallow the `'a` to contain `f` or assign a different type than `'a` to `r2`. If we choose a completly unrelated type variable `'c` we have a problem similar to the problem in the type `'a -> {f : int, ... : 'b}` that we considered in the very first example for a type for a function adding the field `f` to a record. That is, the problem that `'c` is completely unrelated to `'a`.

So we have to accept that `r2` must be given a type which prevents it from containing the field `f` - but it must also unify with `'a`! This means that a type like `{f : int, ... : 'a}` must always prevent `'a` from containing the field `f` when we have to support optional fields suitable for a fieldcase construct.

This leaves us with the final type for adding the field `f` freely to a record:

```
{'b f : 'c, ... : 'a} -> {f : int, ... : 'a}
```

And this justifies the implication of the claim - that we must type record extension as strict extension. □

A polymorphic free extension operation was implemented in the example in section 3.10.

For comparison, the semantics of [Remy89] makes optional fields and free extension of records coexist. It may be hard to do a direct comparison with this article - but the point where this article fails in the above claim is that it uses a kind of subtyping. This in turn means that records are not *exactly typed*. As mentioned in section 4, all the articles [Wand87], [Wand88], [Remy92a], [Remy92b] and [Ohor92] fail in not supporting optional fields in a way suitable for allowing a fieldcase construct.

So, since the programs that CeXL is intended to be used for require both extensible records, optional fields suitable for a fieldcase construct and exact typing of records we can conclude that the design of CeXL with respect to extensible records most likely can't be done any better when using ML's polymorphism. A few differences in the design of these records might be possible, but not many significant differences.

7.2 Considering Polymorphic Restrictions

When developing this semantics, some investigation has been done on supporting restrictions with polymorphism - i.e. such that restrictions can contain type variables. An example would be: `['a : real | 'b list]`.

The conclusion of the investigation is that when supporting restrictions on type variables in the first place, type variables used in restrictions must also support restrictions themselves. This leads to a recursive notion of restrictions on type variables. In the example above `'b` does not have any restrictions but during unification a restriction for `'b` must be maintained.

The problem in the semantics, which was developed for this at an earlier stage, was that every time a meta variable was instantiated during unification, *all* types (up to the next closure operation) which at some point during the type inference had been required to respect a restriction had to be rechecked that they would still respect all restrictions after the instantiation. This would most likely give a very high time complexity of the type inference - not to mention that the semantics was quite complex. So this problem is not trivial - in case anyone should endeavour the quest of looking into this.

It is possible that constraint semantics can handle this, but I have not looked into this. To my knowledge, constraint semantics is also more complex if exposed to the programmer - and it is certaintly not very ML-like.

7.3 Considering the Semantics of a General Typecase

During the development of this semantics it was initially decided to go with a more general typecase construct instead of the fieldcase construct used now. One point where it failed was when requiring polymorphism in restrictions on type variables - as just described above.

It is not a problem to implement a typecase construct if we restrict ourselves to restrictions on variables which are monomorphic. It doesn't give any huge benefits though - with the exception of being able to actually implement the overloaded operators of Standard ML such as `+` and `*` directly in ξ-Calculus (assuming we have the mono-typed operators available).

It should also be noted that mixing value-based case and typecase in one language construct is not trivial either. This might give the programmer some convenience but probably also some confusion.

A typecase using monomorphic restrictions would be *almost* enough for ξ-Calculus to support the statically typed exceptions which we will briefly consider now.

7.4 Considering Staticly Typed Exceptions for ξ-Calculus

Staticly typed exceptions would be interesting. If we exploited restrictions ξ of the form $\psi/^m$ on type variables $\alpha :: \xi$, we could let the exception handler clauses work as a typecase on exceptions. The exceptions being caught would then be considered to have type α in an exception handler. Each declared exception would have to generate a new exception type name which is unique across the whole program. We could thus avoid introducing exceptions as a dynamic type, which is what is done in ξ-Calculus and Standard ML. However, we would have to prevent exceptions from containing type variables, to allow exception types to be generated statically. An example of this can be found in the regression test in section 25.4.

Supporting statically typed restrictions complicates the semantics of exception handlers though. In particular, we need to require that each clause has a type disjoint from the other clauses, just as we would have to do in a general typecase construct. A simple way is to just require an exception type in each clause and that these be different in each clause. We still have to be able to have a last clause for exception handling though, which is like a wild-card catching all types of exceptions. This would require a special-case in the semantics regarding disjoint restrictions. As stated already, introducing general polymorphic restrictions for handling this is not trivial. So a simpler special-case for this would be the closest one could get to a "simple" way of handling statically typed exceptions.

As described for the general typecase, it is also very difficult to mix the notion of a value-based case and a typecase in one language construct. Being restricted to only casing at types will make it more cumbersome to case out different values in an exception handler - even though this is possibly done only very rarely.

To sum up, statically typed exceptions avoids having to introduce a dynamic type `exn`. However, it would also complicate the semantics quite a bit and probably give some inconvenience for the programmer occasionally. So there are not many things to count in favour of having exceptions statically typed. This is why it has been ruled out as a feature for ξ-Calculus.

7.5 Considering the Semantics of Record Concatenation

A good way of being able to concatenate two arbitrary records is described in [Remy92a]. However this approach requires records to be abstracted as functions.

It is easy to concatenate two records if one of the records has a known type of a non-extensible record. We will describe this below. However, if both records are either unknown or extensible (which also means that they are far from completely known) it is... indeed a challenge! It means that we will end up having a type variable for the extension part of each record. These two type variables will be dependent on one another in some way. For instance, the restrictions on one of the type variables could depend on the other type variable.

If we call the concatenation operator `##`, one attempt at giving it a type would be:

```
val ['a : ~{}, 'b : exclrec 'a] ## : 'a * 'b -> ???
```

The idea is that the restriction `exclrec 'a` should exclude the fields of the record `'a` in the record `'b`. One problem is what the result type should be. Another problem is that now we have a type variable in a restriction! And according to the earlier arguments about polymorphic restrictions, this can easily complicate things considerably.

If we restrict the concatenation to work only when one of the operands (say the left operand) has a non-extensible record type, where all fields are known, things become much simpler. It could for instance be added to CeXL by syntactical rewriting, like the following:

```
val ['a : ~{lab_1, ,,, , lab_n}] ## :
      {lab_1 : 'a_1, ,,, , lab_n : 'a_n} * 'a ->
      {lab_1 : 'a_1, ,,, , lab_n : 'a_n, ... : 'a}
```

Here we use `,,,` to denote repetition, to avoid confusion with the reserved symbol `....` Adding such a feature will have the limitation that it will sometimes be required to add an explicit type specification for the left operand. This is similar to when using `...` in a record-pattern in Standard ML.

This kind of concatenation has not been added to the language. At the time of this writing, I have actually only found one example where record concatenation would be useful for a 3D graphics application. I will not go through the example, but it requires that both of the 2 records are extensible to be of any benefit - which means that it is the hard way. However even in this example that I found, a little extra manual coding in CeXL and extra work in maintaining the code is an acceptable solution.

8 How Hard it was to Create This Language

I have developed the CeXL programming language out of a need for a completely general and extensible architecture for a 3D graphics application.

8.1 Early Investigation of 3D Architectures

In the beginning, it was not even realized that this should in fact be a programming language. Hard investigation of exsisting architectures for 3D graphics applications has been a large initial part of the quest, which I began already around 1997-1998. The plugin architecture of the 3D program Maya [Maya] has been a very prominent source of inspiration for this part. Discussions with certain people about programming languages, OpenGL, 3D graphics architectures, semantics and whatnot has also been a source of inspiration.

8.2 An Old Version of CeXL Has Been Used Until Now

When it was realized that it was in fact a programming language which was needed, some experiments and implementations of extensible records were first created. In fact, for a long time there has been an early implementation and formal grammar description of CeXL. This version of the language never had a complete semantics and the implementation had problems handling the combination of optional fields and extensible records in certain cases. At the time of this writing, this old language is used in the commercially available program *CeX3D Converter* [CeXC], the interactive showreel available from Hardcore Processing [HcPReel] and an in-house 3D graphics application at Hardcore Processing called *CeX3D SM* [CeX3DSM] - for subdivision surface modelling.

The reader can download the free demo versions of *CeX3D Converter* [CeXC] to verify some of these facts. The old download directory
`http://www.CeX3D.net/Download/` (yes, that must be written with an uppercase 'D') contains older versions of CeX3D Converter. The version 3.5 (which was released 2000-10-29) and later versions use the old CeXL language. Try loading the binary file of such a version into a text editor and search for some text which looks like record types :-) These types are string constants in the program (which is why you can see them in the binary file as text) and are used for type-checking the CeXL representation of 3D objects loaded by CeX3D Converter.

Indeed, the problems mentioned above with the combination of optional fields and extensible records can be provoked by a known bug in earlier versions of CeX3D Converter. When converting 3D objects with certian properties from LightWave [LightWave] format, the program would give a type error message from the type-check!

8.3 Development of CeXL for This Specification

Both developing the old version of CeXL and continuing the development up to the current version has been *really* hard work. During the initial work on the old CeXL language, mostly optional fields and extensible records were investigated and gave many challenges. Every time one tries solving one problem, some new problem turns up and requires the whole thing to be thought through all the way from the beginning.

The same has continued to happen during the development up to the present version of CeXL. Everytime a small change is made or the next feature is to be added, the whole thing needs to be redesigned - usually several times before one gets it right. Writing a semantics enables one to literally work on the *whole* language at the same time without loosing the overview. However it really *is* required that one has the *whole* language in one's head at the same time to make everything work. It can be said that designing a programming language is not a deterministic process.

One thing that makes the semantics of CeXL very hard is that it is statically typed. There is a *substantial* amount of work just in getting the type inference and the whole static semantics to work - just for checking that the programs are valid. On the other hand, writing the dynamic semantics for this is quite easy. However, it is also this static typing which gives the language many benefits. In particular with respect to efficient implementation and prevention of programmer mistakes.

It is conceivable that using *Action Semantics* [Action] (instead of Operational Semantics) would make the design process simpler and more modular. However I have not looked into this.

8.4 The Ease of Implementation From Semantics

When the semantics of the language is complete, creating a simple interpreted implementation of the language is a relatively small task. Writing the interpreter for the ξ-Calculus language was *trivial*. Implementing the type inference for ξ-Calculus requires understanding of the much more difficult static semantics - but it was not very hard for me to implement this, since I wrote the semantics. Implementing the Naked CeXL to ξ-Calculus translation may be a little tedious but is easy and it reduces the language considerably. The syntactical reduction from the full CeXL grammar to the Naked CeXL grammar is easily done on the fly during parsing of the grammar.

8.5 Conclusion And Warnings About Language Design

Let's state it again - designing this language was *hard*. So, be thankful that you only have to read and understand this final language specification (even though it may be hard reading) and not all the problems that I had to go through to get to this final specification. Many problems are already solved in Standard ML, but changing some of the fundamentals and putting it all together into a new language is hard.

If you stray aside from this semantics even just a little bit without knowing what you're doing - you are likely to run into heavy problems. So, adding new features to this language is not something that you can always *just* do.

Some syntac sugar is easy to add of course - like adding classes and objects with structural inheritance as in an object oriented language. This is because the fundamental semantics with its extensible records can handle this already (see the future work section).

However, altering the more fundamental behaviour of the language, such as tangling too much with the restrictions on type variables, changing the behaviour of extensible records or optional fields or the likes will lead down very dark and cloudy path in your life. This is true in particular if you haven't got experiences similar to those that I gained while designing this language.

9 Grand Conclusion

The following sums up some of the most important conclusions and results of this specification for the CeXL language:

- A language specification for a complete programming language for real-life use has been defined. The intended use is domain specific for certain 3D graphics applications.
- The language specification has formal grammar and semantics for all aspects of the language.
- It is likely that the soundness property for this language holds - even though it has not been proved. The worst case scenario on this issue should be that a few details might have to be corrected or modified.
- The language is compatiple with Standard ML '97 to a high degree.
- The language has a substantial feature set relating to extensible records, optional record fields and type variables with restrictions, none of which are present in Standard ML '97. These features all complement one another in many ways to make the language as useful as possible for what the language has been designed for.
- Some justification that the most central part of the language could not have been done any better has been made, when taking the design requirements into consideration.
- Many alternative features for the language have been explored and it has been argued what challenges or problems exist regarding those features.
- An implementation for the most central parts of the language is supplied, including many regression test programs for a complete implementation. It is verified that the implementation works as expected and no problems are identified.
- An implementation and future versions of this specification will be maintained online at:

  `http://www.CeX3D.net/cexl/`

10 Directions for Future Work

I would like (did I have the time) to extend the language in the following ways:

- Extend it to the full Standard ML [SML97] - possibly stripping off a few obsolete features and making minor ajustments to other features of SML'97. This includes adding a full module system and either removing equality types entirely or representing their polymorphism using restrictions. [ML2000] has good suggestions for what could be removed.
- An alternative to the SML Module System could be considered. I have some specific ideas in this direction but they will not be mentioned here.
- It might be worth considering to restructure the semantics so that it becomes more similar to the FLINT language [Shao97].
- We can currently, without any changes to the semantics, add classes and objects with structural inheritance to the language as syntactic sugar as described in [Wand88]. Since we don't have record concatenation, we can not handle multiple inheritance though. It would require something like a general typecase construct and more general polymorphic restrictions on type variables to do statically typed downcasts. As mentioned, the polymorphic restrictions would most likely increase the complexity of the semantics considerably - if not also the time complexity of the type inference. However these ideas are interesting.

10.1 Closing Remarks About the Future

The CeXL language of this specification will be incorporated in *CeX3D Converter* [CeXC] and *CeX3D SM* [CeX3DSM] as soon as possible. *CeX3D SM* is still an in-house program at Hardcore Processing though and is still very slow to work with, since the implementation is still only an interpreter. So it is not known when this will be relased to the public. This will also require a revised design of another language, *ICeXL*, which is used by *CeX3D SM* - a language which has values in the same type system as CeXL.

Other computer graphics related applications which are based on CeXL might also soon see the light of day.

Most importantly, all these applications will be easy to extend from now on, which is the whole point of the CeXL language. Writing a compiler for CeXL has a high priority now, so that for instance *CeX3D SM* will become a very fast computer graphics application - which is also one of the points of the CeXL language.

In short, the completion and release of this specification is quite an important event at Hardcore Processing. It is hopefully soon to become important for the entire computer graphics industry as well :-)

- An implementation and future versions of this specification will be maintained online at:

  `http://www.CeX3D.net/cexl/`

11 Acknowledgements

Fritz Henglein has given many helpful hints and references for relevant material, has explained useful things about semantics, has given helpful input for the presentation of the document and has been supervisor for the project. Stephen Weeks and Matthew Fluet of the *MLton* [MLton] compiler project have given helpful clarifications about the Definition of Standard ML '97. Discussions with Stuart Croy, Scott A. Crosby and Peter Lund (a.k.a. firefly) about programming languages, OpenGL, 3D graphics architectures, semantics and whatnot has also been an early source of inspiration. In addition to Fritz Henglein, the following people have given feedback on this document: Peter Lund, Torben Mogensen, Dave Berry and Brent Fulgham.

References

The Most Important Reference

[SML97] Robin Milner, Mads Tofte, Robert Harper, David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

References for Record Caculi

[Wand88] Mitchell Wand. *Type Inference for Record Concatenation and Multiple Inheritance*. College of Computer Science North Eastern University in Boston, October 1988.

[Remy89] Rémy Didier. Typing Records and Variants in a Natural Extension of ML. *ACM Symposium on Principles of Programming Languages*, 1989.

[Remy92a] Rémy Didier. *Typing Record Concatenation for Free*. Institut National de Recherche en Informatique et en Automatique, August 1992.

[Remy92b] Rémy Didier. *Projective ML*. INRIA-Rocquencourt. Apr 10, 1992.

[Ohor92] Atsushi Ohori. *A Compilation Method for ML-Style Polymorphic Record Calculi*. Oki Electric Industry, Kansai Laboratory, 1992.

[Remy01] Rémy Didier. *Efficient Representation of Extensible Records*. INRIA-Rocquencourt. June 12, 2001.

References for Subtyping and Object Calculi

[Card88] Luca Cardelli. *Structural Subtyping and the Notion of Power Type*. Digital Equipment Corporation, Systems Research Center, 1988.

[Card87] Luca Cardelli. *Typechecking Dependent Types and Subtypes*. Digital Equipment Corporation, Systems Research Center, 1987.

[Wand87] Mitchell Wand. Complete Type Inference for Simple Objects. *IEEE Proceedings of Symposium on Logic in Computer Science* / College of Computer Science, Northeastern University, Boston, 1987.

[Guis92] Guiseppe Castagna, Giorgio Ghelli, Guiseppe Longo. A Calculus for Overloaded Functions with Subtyping. *ACM Conference on LISP and Functional Programming 1992*, 1992.

[Cast94] Giuseppe Castagna. *Covariance and Contravariance : Conflict without a Cause*. Laboratoire d'Informatique, Department de Mathématiques et d'Infomatique, Ecole Normale Supérieure, 1994.

Various References of Inspiration

[Kahr93] Stefan Kahrs. *Mistakes and Ambiguities in the Definition of Standard ML*. Unviversity of Edinburgh, April 1993. URL and Updated listing:
`http://www.cs.ukc.ac.uk/pubs/1993/569/`
`ftp://ftp.dcs.ed.ac.uk/pub/smk/SML/errors-new.ps.Z`

[ML2000] The ML2000 Working Group. *Principles and a Preliminary Design for ML2000*. March 1999.

[Shao97] Zhong Shao. *Typed Common Intermediate Format*. Yale University 1997.

[Cart91] Robert Cartwright, Mike Figan. Soft Typing. *Proceedings of ACM SIGPLAN 1991* / Department of Computer Science, Rice University, 1991.

[Shie01] Mark Shields, Erik Meijer. Type-Indexed Rows. *ACM SIGPLAN 2001, p. 17-19*, January 2001.

[Heng87] Fritz Henglein. *A Polymorphic Type Model for {SETL}*. New York University, 1987.

[Free91] T. Freeman, F. Pfenning. Refinement Types for ML (Extended Abstract). *Proceedings of ACM SIGPLAN 1991*, June 1991.

[Bare83] H. Barendregt, M. Coppo, M. Dezani-Ciancaglini. A Filter Lambda Model and the Completeness of Type Assignment. J. Symbolic Logic 48(4) pages 931-940, 1983.

[Pier91] *Programming with Intersection Types and Bounded Polymorphism*. Ph.D. thesis, Carnegie Mellon University, 1991. Available as School of Computer Science technical report CMU-CS-91-205.

[Appe98] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

[Jone93] Niel Jones, Carsten K. Gomard, Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International (UK) Ltd, 1993.

[Action] Brics. *Action Semantics*.
`http://www.brics.dk/Projects/AS/`

[MLton] Stephen Weeks et. al. *MLton*.
`http://www.mlton.org/`

[Maya] Alias. *Maya*.
`http://www.alias.com`

[LightWave] NewTek. *LightWave 3D*.
`http://www.newtek.com`

[Houdini] Side Effects Software. *Houdini*.
`http://www.sidefx.com`

Early Uses of CeXL

[CeXC] Hardcore Processing. *CeX3D Converter*, 2000.
`http://www.CeX3D.net/converter/`

[CeX3DSM] Hardcore Processing. *CeX3D SM*, to appear for general the public.
`http://www.CeX3D.net/sm/`

[HcPReel] Hardcore Processing. *Hardcore Processing Interactive Showreel*, 2001.
`http://www.HardcoreProcessing.com/company/showreel/`

12 Appendices

The last sections of this document are the appendices containing the semantics and the actual language specification of CeXL. We will start the appendices by giving a short overview of the language specification.

12.1 The Languages and Calculi Presented

The final language we wish to present is called CeXL. This in turn is reduced *purely syntactically* to Naked CeXL, which is the essential part of the language. Naked CeXL is implemented on top of a more fundamental calculus called ξ-Calculus. The languages will be presented in a bottom-up fashion, by starting with all the hard work on ξ-Calculus.

All the semantics for ξ-Calculus is presented first. Then we present Naked CeXL and a translation from Naked CeXL to ξ-Calculus. Finally the real syntax of CeXL is presented along with how to reduce this into Naked CeXL. Everything is put together at the end by a description of the initial environments for the semantics and for the translation into ξ-Calculus and a description of how to apply everything to type-check and execute a CeXL program.

12.2 About the Separate ξ-Calculus

A goal of the calculus is to be able to translate Naked CeXL into ξ-Calculus and afterwards do type inference directly in ξ-Calculus. So there is no type inference on the source programs - not even on the reduced Naked CeXL programs. This turns the calculus into a kind of base programming language for implementing Naked CeXL and thus also CeXL.

One feature of Naked CeXL that we cannot represent in ξ-calculus is the datatype definitions and the exception declarations. The types and constructors generated from such declarations are inserted directly into the ξ-Calculus code without separate declarations. However we have to insert exception constructors to ensure correct scoping of type variables and correct static semantics of exception declarations

13 Implicitly Typed ξ-Calculus

The following is the type system and the expression language of ξ-Calculus. This is the *most important page* of this specification. The comments on the next 2 pages give a slight idea of its meaning.

13.1 Syntax and Semantic Objects for the Type System

		Syntax	**Name**		**Semantic Objects**
		lab	Lab	=	"Record labels"
		x	VId	=	"Variable identifiers"
		α	TyVar	=	"Type variables"
		β	MetaVar	=	"Meta variables"
		a	PName	=	"Type parameter names" for constructor types
		d	TyName	=	"Type names"
		c	CName	=	"Constructor names"
ω	::=	$\{lab_1, \ldots, lab_n\}$	ExclLabs	=	Fin(Lab)
ψ	::=	$[a_1 = \psi_1, \ldots, a_n = \psi_n]d\{c_1, \ldots, c_k\}$	TyPat		$\cup_{p \geq 0}$ (PName $\times$ TyPat)^p $\times$ TyName $\times$ Fin(CName)
$\psi^{/m}$	::=	$\psi_1 \ / \cdots / \ \psi_m$	TyPats	=	$\cup_{p \geq 1}$ TyPat^p
ξ	::=	$\circ \mid \omega \mid \psi^{/m}$	Restrict	=	$\emptyset \cup$ ExclLabs $\cup$ TyPats
γ	::=	$\alpha :: \xi \mid \beta :: \xi$	Vars	=	TyVar $\times$ Restrict $\cup$ MetaVar $\times$ Restrict
ρ	::=	$\{lab_1 : \tau_1, \ldots, lab_m : \tau_m; \tau\} \mid \{\}$	Row	=	(Lab $\overset{fin}{\rightarrow}$ Type) $\times$ Type $\cup \emptyset$
ϕ	::=	τ **?** τ'	Field	=	Type $\times$ Type
κ	::=	$[a_1 = \tau_1, \ldots, a_n = \tau_n]d\{c_1, \ldots, c_k\}$	ConsType	=	$\cup_{p \geq 0}$ (PName $\times$ Type)^p $\times$ TyName $\times$ Fin(CName)
		$\tau \rightarrow \tau'$	Fun	=	Type $\times$ Type
τ	::=	$\gamma \mid \tau \rightarrow \tau' \mid \rho \mid \phi \mid \kappa$	Type	=	Vars $\cup$ Fun $\cup$ Row $\cup$ Field $\cup$ ConsType
σ	::=	$\forall[\alpha_1 :: \xi_1, \ldots, \alpha_n :: \xi_n].\tau$	TyScheme	=	$\cup_{p \geq 0}$ (TyVar $\times$ Restrict)^p $\times$ Type
r	::=	$\{lab_1 : \tau_1, \ldots, lab_n : \tau_n\}$	OrderRow	=	Lab $\overset{fin}{\rightarrow}$ Type

13.2 A Syntax for ξ-Calculus Expressions

(*expression*) e ::= $\lambda x.e \mid e_1\ e_2 \mid x \mid e : \tau \mid c : \sigma \mid c$ **ex** $\tau \mid scon \mid$
$\{\ \} \mid \{lab_1 = e_1, \ldots, lab_m = e_m, e\} \mid e_1$ **?** $e_2 \mid$
let $p_1 = e_1$; $\cdots$; $p_n = e_n$ **in** $e \mid$
letrec $p_1 = e_1$; $\cdots$; $p_m = e_m$ **in** $e \mid$
letex $c_1 : \tau_1, \ldots, c_m : \tau_m$ **in** $e \mid$
e **handle** $p_1 \rightrightarrows e_1 \parallel \cdots \parallel p_m \rightrightarrows e_m \mid$ **raise** e
case e **of** $p_1 \rightrightarrows e_1 \parallel \cdots \parallel p_m \rightrightarrows e_m \mid$
fieldcase e **in** α **of absent** $\rightrightarrows e_1 \parallel$ **present** $p \rightrightarrows e_2$ **type** $\tau \mid$

(*pattern*) p ::= $x \mid p : \tau \mid x$ **as** $p \mid c : \sigma \mid c(p) : \sigma \mid c$ **ex** $\mid c(p)$ **ex** $\tau \mid scon \mid _ \mid$
$\{\ \} \mid \{lab_1 = p_1, \ldots, lab_m = p_m, p\} \mid p_1$ **?** p_2

13.3 Comments on the Type System

The following gives a breif idea of the type system - but the details follow later:

- The ξ represents restrictions on the allowed instantiations of type variables and meta variables.
- The restriction $\circ$ means "no restrictions" - so $\alpha :: \circ$ is like a "normal" type variable in Standard ML.
- The restriction ω represents a record excluding the labels ω.
- The restriction $\psi^{/m}$ is a list of allowed (mutually disjoint) constructor types. Each such type in the list is a ψ.
- The empty record $\{\}$ in ρ really represents: "All as-of-yet undefined fields are absent". This is actually what is seen as the type `unit` in CeXL. All concrete record values in CeXL consist of some or no extensible records extending one another and finally being extended by $\{\}$ at the end. Each such extensible record is written as $\{lab_1 : \tau_1, \ldots, lab_m : \tau_m; \tau\}$. It is only in functions, type declarations and the likes that it makes sense to have extensible record types which are not ending with $\{\}$, in which case they would end with a meta variable or a type variable.
- $\beta :: \omega$ may *not* be instantiated with a record *without fields* with new $\beta' :: \omega'$, i.e.: $\beta :: \omega \mapsto \{;\beta' :: \omega'\}$. Instead it should be instantiated with $\beta :: \omega \mapsto \beta' :: \omega'$. If we did not forbid this we would have problems unifying the types $\beta :: \omega$ and $\{;\beta :: \omega\}$ with each other. We ensure this in the inference rules of record expressions and record patterns for ξ-Calculus. The reason why $\beta :: \omega$ and $\{;\beta :: \omega\}$ would have to unify with each other is that they both denote an extensible record excluding the fields ω - i.e. they denote exactly the same thing.
- Consider the a_i used in the $a_i = \tau_i$ in the constructor parameters in ConsType and the $a_i = \psi_i$ in the parameters for the type pattern constructor in TyPat. These a_i are to be considered parameter names for constructor types. They will correspond to the type variable names of the closed type scheme of constructors. However they may not be considered type variables - which is why they are denoted a_i and not α_i. They are for allowing type parameters to be identified based on names rather than the order in which they appear. This is for giving CeXL the feature *Named Type Parameters*.
- The ConsType contains a finite set of constructor names (the Fin(CName)). These are the names of the constructors that the ConsType has declared. This is only used by an implementation to ensure correct implementation of pattern-matches and the checks that pattern-matches are exhaustive where this is required. This also means that they are not used at runtime, so an implementation can remove them after type inference.

  An example of a constructor type in CeXL is the type `bool` which would be represented by: $[]bool\{true, false\}$. The type `int` is represented by: $[]int\{\cdots\}$ to signify that it has many constructors (i.e. all the supported integer constants). The type `int list` becomes: $[a = []int\{\cdots\}]list\{nil, ::\}$.

- ConsType is also used for exceptions. Here only one predeclared constructor is used: $[]exn\{\}$. It is treated specially, since it gets its constructors added separately through exception declarations. Therefore we use an empty list of constructor names for it. The constructor $[]exn\{\}$ may only be used with the **ex** constructs and in type specifications in ξ-Calculus. Restrictions on the number k of constructor names allowed where ConsType occurs in will ensure this. When $k \geq 1$ it cannot be $[]exn\{\}$.

- The class Field of the form τ **?** τ' denotes the type of a record field. τ may only be the type $[]present\{present\}$, the type $[]absent\{absent\}$ or a meta variable or type variable properly restricted to such instantiations. τ' is the type of the value in the record field.

- σ is used in the value environment and for representing the closed types of constructors. Constructors will have σ of one of the following 2 forms:
  $\forall[\alpha_1 :: \xi_1, \ldots, \alpha_n :: \xi_n].[a_1 = \alpha_1 :: \xi_1, \ldots, a_n = \alpha_n :: \xi_n]d\{c_1, \ldots, c_k\}$ or
  $\forall[\alpha_1 :: \xi_1, \ldots, \alpha_n :: \xi_n].\tau' \rightarrow [a_1 = \alpha_1 :: \xi_1, \ldots, a_n = \alpha_n :: \xi_n]d\{c_1, \ldots, c_k\}$.

  As we shall see later these type schemes must generalize some type τ, which is written $\sigma \succ \tau$.

  During type inference the names a_i must be chosen so that they are the same as the α_i to avoid confusing the programmer and to avoid problems in identifying parameters in type patterns.

- r is only used during unification, where we will need to store a finite set of labels mapped to types. This is for transforming a record into a semantic representation where records are not treated as a recursive list of smaller records extending each other - as they usually are when represented by ρ.

14 Static Semantics of ξ-Calculus - Unification

This section gives explicit inference rules for the unification between 2 given types. We need this to be able to implement a unification algorithm for the semantics of ξ-Calculus. The unification relation between 2 given types can be briefly stated as follows:

It is the standard unification relation with the following changes:

- All polymorphic variables and meta variables have restrictions on them which limit what they may be instantiated with. These restrictions are maintained by the unification
- Equality of records during unification is modulo reordering of fields. This is achieved by converting any records encountered to a semantic representation (denoted r in the syntax for the type system) which is independent of the order of the fields of the record. Unification of records is done using this representation because record types in ξ-Calculus may be composed of several smaller record types extending each other as a sequential list of records. For example, if we did not do this we would have problems unifying $\{lab_1 : \tau_1; \{lab_2 : \tau_2; \tau\}\}$ and $\{lab_2 : \tau_2; \{lab_1 : \tau_1; \tau\}\}$ with each other. Such two record types must unify, since they both denote the type of an extensible record with two fields, lab_1 and lab_2 of types τ_1 and τ_2 respectively, where both record types are extended with τ. So the two types denote exactly the same record.
- The unification relation deals with maintaining the property that extensible record types may never redefine the same field twice.
- Record fields must have field types of the form $\tau \ ? \ \tau'$ where τ is either $[]absent\{absent\}$ or $[]present\{present\}$ or a type variable or meta variable which is properly restricted to these instantiations
- Non-extensible records are represented modulo equality of $\{\}$ at the end of the record with records of absent fields and new $\{\}$ types - e.g. equalities like:
  $\{\} = \{lab : []absent\{absent\} \ ? \ \tau; \{\}\}$

Ordered records r and meta variables β from our semantic objects are only used for the unification when implementing the semantics. r is for transforming a record into the semantic representation mentioned above and β are the place holders for unknown types during unification. One exception though is that we explicitly have to handle meta variables in the rules for explicit type specifications in ξ-Calculus. This is necessary to support the feature *Partial Type Instantiation.*

14.1 Formal View and Implementation of Meta Variables with Restrictions

Restrictions on meta variables must always be defined at the time the meta variable is created - and they may never change. To restrict an existing meta variable further, it has to be bound to a new meta variable with a tighter but compatiple restriction. A meta variable may also only be bound to another type or meta variable once, and never be rebound. Whenever a meta variable is bound to a type (or a type variable or a meta variable), it must be checked that the type respects the restriction of the meta variable. We can only write a correct unification relation if we follow those guidelines.

When dealing with meta variables with restrictions, we adopt the following view:

- Meta variables will have restrictions at creation time which may never change. Meta variables with restrictions are denoted $\beta :: \xi$, where β is the variable and ξ its restriction.
- We will assume that we have an environment *env* which binds meta variables to types.

When implementing the unification relation there are 2 simple ways of representing meta variables with restrictions:

- We can emulate having just 1 environment, by keeping an updatable reference cell with each meta variable. The reference is either a restriction or a bound type. Restrictions may be updated to bind a compatiple type, which in turn may just be another meta variable with a tighter but compatiple restriction. However types may not be updated once they have been bound the first time.
- It could also be that one prefers to have a data structure where restrictions and bound types are stored separately, which can be done by always keeping the restriction (which may never change) with each meta variable and an updatable reference cell containing an optional bound type.

14.2 Unification of Restrictions

In all the inference rules presented where two restrictions have to be unified, it is always enough to check that they are simply equal. So we never do a general unification on restrictions.

All the special issues of restrictions should be handled appropriately by the inference rules of the unification relation which will be presented.

14.3 Restrictions on Types: $\tau|_\xi$

We need to be able to impose restrictions on types in the semantics of ξ-calculus in the rules for extensible records, record fields and fieldcase. This is done during unification by introducing a fresh meta variable with a restriction. To write a semantics for ξ-calculus which does not involve meta variables we introduce the notation $\tau|_\xi$ to denote that τ is a type restricted by ξ. During unification this just means introducing a new meta variable with restriction ξ by setting $\tau = \beta :: \xi$ for a fresh β.

14.4 Judgement Forms for the ξ Unification Relation

Unifying Judgements

$\tau \leftrightarrow \tau'$	Unify τ and τ'.
$r; \tau \leftrightarrow^{emptyrec}$	Unify record r which ends in τ with the empty record.
$r; \tau \leftrightarrow^{rec} r'; \tau'$	Unify records r and r' where r ends with τ and r' ends with τ'.
$\tau; \omega \vdash^{ord} r; \tau'$	Convert record τ to representation r which is independent of the order of fields and which ends in τ'. τ is prevented from containing any of the fields in ω.

Checking Judgements

$\tau \gg \xi$	Check that τ respects the restriction ξ. This will *not* do any unification (i.e. no β will be bound).
$\tau \gg \psi$	Check that τ respects the restriction ψ. This will *not* do any unification (i.e. no β will be bound).
$\psi \gg \xi$	Check that ψ respects the restriction ξ. This will *not* do any unification (i.e. no β will be bound).
$\xi \gg \xi'$	Check that ξ is more restrictive than ξ'. It can also be understood as ξ being "less polymorphic" than ξ'. This will *not* do any unification (i.e. no β will be bound).

14.5 Inference Rules for the ξ Unification Relation

14.5.1 Unifying Types Different From Records and Meta Variables

$\boxed{\tau \leftrightarrow \tau'}$

$$\frac{}{\alpha :: \xi \leftrightarrow \alpha :: \xi} \quad (1)$$

$$\frac{\tau \leftrightarrow \tau'' \qquad \tau' \leftrightarrow \tau'''}{\tau \rightarrow \tau' \leftrightarrow \tau'' \rightarrow \tau'''} \quad (2)$$

$$\frac{\tau \leftrightarrow \tau'' \qquad \tau' \leftrightarrow \tau'''}{\tau \ \mathbf{?} \ \tau' \leftrightarrow \tau'' \ \mathbf{?} \ \tau'''} \quad (3)$$

$$\frac{\forall i \in \{1, \ldots, n\} : \tau_i \leftrightarrow \tau'_i \qquad n, k \geq 0}{[a_1 = \tau_1, \ldots, a_n = \tau_n]d\{c_1, \ldots, c_k\} \leftrightarrow [a_1 = \tau'_1, \ldots, a_n = \tau'_n]d\{c_1, \ldots, c_k\}} \quad (4)$$

Comments:

- Rule (4): We must support unification of $[]exn\{\}$ so $k \geq 0$.

14.5.2 Unifying Meta Variables

$\boxed{\tau \leftrightarrow \tau'}$

$$\frac{\beta \in Dom\ env \quad env(\beta) \leftrightarrow \tau}{\beta :: \xi \leftrightarrow \tau} \tag{5}$$

$$\frac{\beta \neq \beta' \quad \beta \notin Dom\ env \quad \beta' \in Dom\ env \quad \beta :: \xi \leftrightarrow env(\beta')}{\beta :: \xi \leftrightarrow \beta' :: \xi'} \tag{6}$$

$$\frac{\beta \notin Dom\ env}{\beta :: \xi \leftrightarrow \beta :: \xi} \tag{7}$$

$$\frac{\beta \notin Dom\ env \quad m \geq 1 \quad \{lab_1 : \tau_1, \ldots, lab_m : \tau_m; \tau\}, \{\} \vdash^{ord} r; \tau' \quad \{\}; \beta :: \omega \leftrightarrow^{rec} r; \tau'}{\beta :: \omega \leftrightarrow \{lab_1 : \tau_1, \ldots, lab_m : \tau_m; \tau\}} \tag{8}$$

$$\frac{\beta \notin Dom\ env \quad m \geq 1 \quad \beta'\ fresh \quad \omega = \{\} \quad \{lab_1 : \tau_1, \ldots, lab_m : \tau_m; \tau\}, \{\} \vdash^{ord} r; \tau' \quad \{\}; \beta' :: \omega \leftrightarrow^{rec} r; \tau' \quad \beta\ does\ not\ occur\ in\ any\ of\ \tau_1, \ldots, \tau_m, \tau}{\begin{array}{c}\beta :: \circ \leftrightarrow \{lab_1 : \tau_1, \ldots, lab_m : \tau_m; \tau\} \\ bind(env, \beta \mapsto \beta' :: \omega)\end{array}} \tag{9}$$

$$\frac{\beta \notin Dom\ env \quad \beta\ does\ not\ occur\ in\ \tau \quad \tau \neq \{lab_1 : \tau_1, \ldots, lab_n : \tau_n; \tau'\} \quad \tau \neq \beta' :: \xi' \quad \tau \gg \xi}{\begin{array}{c}\beta :: \xi \leftrightarrow \tau \\ bind(env, \beta \mapsto \tau)\end{array}} \tag{10}$$

$$\frac{\beta \neq \beta' \quad \beta, \beta' \notin Dom\ env \quad \xi' \gg \xi}{\begin{array}{c}\beta :: \xi \leftrightarrow \beta' :: \xi' \\ bind(env, \beta \mapsto \beta' :: \xi')\end{array}} \tag{11}$$

$$\frac{\beta \neq \beta' \quad \beta, \beta' \notin Dom\ env \quad \xi \gg \xi' \quad \xi' \neq \xi}{\begin{array}{c}\beta :: \xi \leftrightarrow \beta' :: \xi' \\ bind(env, \beta' \mapsto \beta :: \xi)\end{array}} \tag{12}$$

$$\frac{\tau \neq \beta' :: \xi' \quad \beta :: \xi \leftrightarrow \tau}{\tau \leftrightarrow \beta :: \xi} \tag{13}$$

Comments:

- Rules (8), (9): Unification with an extensible record must be handled specially.
- Rule (10): The premise $\tau \gg \xi$ ensures that τ really respects the restriction ξ before τ is bound to β.
- Rules (11), (12): The *least restrictive* should have the *most restrictive* bound to it. Rule (11) also covers the case where the restrictions are the same, where we don't care which variable is bound to which.
- Rule (13): The unification is symmetric.

14.5.3 Unifying Records

$\boxed{\tau \leftrightarrow \tau'}$

$$\frac{\begin{array}{c} m, n \geq 1 \\ \{lab_1 : \tau_1, \ldots, lab_n : \tau_n; \tau\}, \{\} \vdash^{ord} r; \tau'' \\ \{lab'_1 : \tau'_1, \ldots, lab'_m : \tau'_m; \tau'\}, \{\} \vdash^{ord} r'; \tau''' \\ r; \tau'' \leftrightarrow^{rec} r'; \tau''' \end{array}}{\{lab_1 : \tau_1, \ldots, lab_n : \tau_n; \tau\} \leftrightarrow \{lab'_1 : \tau'_1, \ldots, lab'_m : \tau'_m; \tau'\}} \tag{14}$$

$$\frac{}{\{\} \leftrightarrow \{\}} \tag{15}$$

$$\frac{\begin{array}{c} m \geq 1 \\ \{lab_1 : \tau_1, \ldots, lab_m : \tau_m; \tau\}; \{\} \vdash^{ord} r; \tau' \\ r; \tau' \leftrightarrow^{emptyrec} \end{array}}{\{\} \leftrightarrow \{lab_1 : \tau_1, \ldots, lab_m : \tau_m; \tau\}} \tag{16}$$

$$\frac{\begin{array}{c} m \geq 1 \\ \{lab_1 : \tau_1, \ldots, lab_m : \tau_m; \tau\}; \{\} \vdash^{ord} r; \tau' \\ r; \tau' \leftrightarrow^{emptyrec} \end{array}}{\{lab_1 : \tau_1, \ldots, lab_m : \tau_m; \tau\} \leftrightarrow \{\}} \tag{17}$$

14.5.4 Unifying With the Empty Record

$\boxed{r; \tau \leftrightarrow^{emptyrec}}$

$$\frac{\forall lab_i \in Dom\ r : r(lab_i) \leftrightarrow [] absent\{absent\}\ ?\ \tau_i}{r; \{\} \leftrightarrow^{emptyrec}} \tag{18}$$

$$\frac{\forall lab_i \in Dom\ r : r(lab_i) \leftrightarrow [] absent\{absent\}\ ?\ \tau_i}{\begin{array}{c} r; \beta :: \omega \leftrightarrow^{emptyrec} \\ bind(env, \beta \mapsto \{\}) \end{array}} \tag{19}$$

Comments:

- Rule (19): The $\beta :: \omega$ here comes from the relation $\tau; \omega \vdash^{ord} r; \tau'$ which guarantees that $\beta \notin Dom\ env$.

14.5.5 Unifying Order Independent Records

$$\boxed{r; \tau \leftrightarrow^{rec} r'; \tau'}$$

$$\frac{\begin{array}{c}\forall lab_i \in Dom\ r \backslash Dom\ r' : r(lab_i) \leftrightarrow [] absent\{absent\}\ ?\ \tau_i \\ \forall lab_i \in Dom\ r' \backslash Dom\ r : r'(lab_i) \leftrightarrow [] absent\{absent\}\ ?\ \tau'_i \\ \forall lab_i \in Dom\ r \cap Dom\ r' : r(lab_i) \leftrightarrow r'(lab_i)\end{array}}{r; \{\} \leftrightarrow^{rec} r'; \{\}} \tag{20}$$

$$\frac{\begin{array}{c}\forall lab_i \in Dom\ r \backslash Dom\ r' : r(lab_i) \leftrightarrow [] absent\{absent\}\ ?\ \tau_i \\ \forall lab_i \in Dom\ r' \backslash Dom\ r : r'(lab_i) \leftrightarrow [] absent\{absent\}\ ?\ \tau'_i \\ \forall lab_i \in Dom\ r \cap Dom\ r' : r(lab_i) \leftrightarrow r'(lab_i)\end{array}}{r; \alpha :: \omega \leftrightarrow^{rec} r'; \alpha :: \omega} \tag{21}$$

$$\frac{\begin{array}{c}\beta''\ fresh \quad m, n \geq 0 \\ \forall lab_i \in Dom\ r \cap Dom\ r' : r(lab_i) \leftrightarrow r'(lab_i) \\ \{lab_1, \ldots, lab_n\} = Dom\ r \backslash Dom\ r' \quad \{lab_1, \ldots, lab_n\} \cap \omega' = \emptyset \\ \{lab'_1, \ldots, lab'_m\} = Dom\ r' \backslash Dom\ r \quad \{lab'_1, \ldots, lab'_m\} \cap \omega = \emptyset \\ \beta'\ does\ not\ occur\ in\ any\ of\ r(lab_1), \ldots, r(lab_n) \\ \beta\ does\ not\ occur\ in\ any\ of\ r'(lab'_1), \ldots, r'(lab'_m)\end{array}}{\begin{array}{c}r; \beta :: \omega \leftrightarrow^{rec} r'; \beta' :: \omega' \\ if\ n \geq 1 : bind(env, \beta' \mapsto \{lab_1 : r(lab_1), \ldots, lab_n : r(lab_n); \beta'' :: \omega \cup \omega'\}) \\ if\ n = 0 : bind(env, \beta' \mapsto \beta'' :: \omega \cup \omega') \\ if\ m \geq 1 : bind(env, \beta \mapsto \{lab'_1 : r'(lab'_1), \ldots, lab'_m : r'(lab'_m); \beta'' :: \omega \cup \omega'\}) \\ if\ m = 0 : bind(env, \beta \mapsto \beta'' :: \omega \cup \omega')\end{array}} \tag{22}$$

$$\frac{\begin{array}{c}\omega \subseteq \omega' \quad m \geq 0 \\ \{lab_1, \ldots, lab_m\} = Dom\ r' \backslash Dom\ r \quad \{lab_1, \ldots, lab_m\} \cap \omega = \emptyset \\ Dom\ r \backslash Dom\ r' \cap \omega' = \emptyset \\ \forall lab_i \in Dom\ r \backslash Dom\ r' : r(lab_i) \leftrightarrow [] absent\{absent\}\ ?\ \tau_i \\ \forall lab_i \in Dom\ r \cap Dom\ r' : r(lab_i) \leftrightarrow r'(lab_i) \\ \beta\ does\ not\ occur\ in\ any\ of\ r'(lab_1), \ldots, r'(lab_m)\end{array}}{\begin{array}{c}r; \beta :: \omega \leftrightarrow^{rec} r'; \alpha :: \omega' \\ if\ m \geq 1 : bind(env, \beta \mapsto \{lab_1 : r'(lab_1), \ldots, lab_m : r'(lab_m); \alpha :: \omega'\}) \\ if\ m = 0 : bind(env, \beta \mapsto \alpha :: \omega')\end{array}} \tag{23}$$

$$\frac{\begin{array}{c}m \geq 0 \\ \{lab_1, \ldots, lab_m\} = Dom\ r' \backslash Dom\ r \quad \{lab_1, \ldots, lab_m\} \cap \omega = \emptyset \\ \forall lab_i \in Dom\ r \backslash Dom\ r' : r(lab_i) \leftrightarrow [] absent\{absent\}\ ?\ \tau_i \\ \forall lab_i \in Dom\ r \cap Dom\ r' : r(lab_i) \leftrightarrow r'(lab_i) \\ \beta\ does\ not\ occur\ in\ any\ of\ r'(lab_1), \ldots, r'(lab_m)\end{array}}{\begin{array}{c}r; \beta :: \omega \leftrightarrow^{rec} r'; \{\} \\ if\ m \geq 1 : bind(env, \beta \mapsto \{lab_1 : r'(lab_1), \ldots, lab_m : r'(lab_m); \{\}\}) \\ if\ m = 0 : bind(env, \beta \mapsto \{\})\end{array}} \tag{24}$$

$$\frac{r'; \beta :: \omega' \leftrightarrow^{rec} r; \alpha :: \omega}{r; \alpha :: \omega \leftrightarrow^{rec} r'; \beta :: \omega'} \tag{25}$$

$$\frac{r'; \beta :: \omega \leftrightarrow^{rec} r; \{\}}{r; \{\} \leftrightarrow^{rec} r'; \beta :: \omega} \tag{26}$$

Comments:

- Rule (21): The $\alpha :: \omega$ comes from the τ in the relation $\tau'; \omega' \vdash^{ord} r; \tau$ so it is already ensured that $Dom\ r \backslash Dom\ r' \cap \omega = \emptyset$ and that $Dom\ r' \backslash Dom\ r \cap \omega = \emptyset$.
- Rules (22)-(26): All the $\beta :: \omega$ here comes from the relation $\tau; \omega \vdash^{ord} r; \tau'$ or the unification with meta variables, so it is already guaranteed that $\beta \notin Dom\ env$.
- Rule (23): The premise $\{lab_1, \ldots, lab_m\} \cap \omega = \emptyset$ is actually not necessary, because is already ensured by the premise $\omega \subseteq \omega'$ and because the $\alpha :: \omega'$ comes from the τ' in the relation $\tau; \omega \vdash^{ord} r; \tau'$, where it is already ensured that $\{lab_1, \ldots, lab_m\} \cap \omega' = \emptyset$.
- Rules (22), (23) and (24): We only bind meta variables to an extensible record if there is at least one field. Otherwise we just bind each meta variable directly to either another meta variable, a type variable or the empty record, respectively.
- Rules (25) and (26): The unification is symmetric.

14.5.6 Making Record Fields Independent of Order

$\boxed{\tau; \omega \vdash^{ord} r; \tau'}$

$$\frac{\{lab_1, \ldots, lab_m\} \cap \omega = \emptyset \quad m \geq 1 \quad \omega' = \omega \cup \{lab_1, \ldots, lab_m\} \quad \tau; \omega' \vdash^{ord} r; \tau'}{\{lab_1 : \tau_1, \ldots, lab_m : \tau_m; \tau\}; \omega \vdash^{ord} r \cup \{lab_1 : \tau_1, \ldots, lab_m : \tau_m\}; \tau'} \tag{27}$$

$$\frac{}{\{\}; \omega \vdash^{ord} \{\}; \{\}} \tag{28}$$

$$\frac{\beta \notin Dom\ env \quad \omega' \subseteq \omega}{\beta :: \omega; \omega' \vdash^{ord} \{\}; \beta :: \omega} \tag{29}$$

$$\frac{\beta \notin Dom\ env \quad \beta'\ fresh \quad \omega' \nsubseteq \omega}{\beta :: \omega; \omega' \vdash^{ord} \{\}; \beta' :: \omega \cup \omega' \quad bind(env, \beta \mapsto \beta' :: \omega \cup \omega')} \tag{30}$$

$$\frac{\beta \notin Dom\ env \quad \beta'\ fresh}{\beta :: \circ; \omega \vdash^{ord} \{\}; \beta' :: \omega \quad bind(env, \beta \mapsto \beta' :: \omega)} \tag{31}$$

$$\frac{\beta \in Dom\ env \quad env(\beta); \omega \vdash^{ord} r; \tau}{\beta :: \xi; \omega \vdash^{ord} r; \tau} \tag{32}$$

$$\frac{\omega' \subseteq \omega}{\alpha :: \omega; \omega' \vdash^{ord} \{\}; \alpha :: \omega} \tag{33}$$

Comments:

- Rule (29): The premise $\beta \notin Dom\ env$ here means that the variable has not been bound to the environment before. And since β is given as "input" to the clause it really means that β is an already allocated but still free variable. In this rule the meta variable $\beta :: \omega$ is just kept as it is and returned from the clause, since ω' is a subset of or equal to ω.
- Rule (30): In this rule β' is just a fresh variable with an new name, which could also have been expressed as $\beta' \notin Dom\ env$ but we use the notation *fresh* to indicate the purpose. The $bind(env, \beta \mapsto \beta')$ in the conclusion means that we bind the previously free variable β to the new variable β'.

14.5.7 Type Respects Restriction

$\boxed{\tau \gg \xi}$

$$\frac{\xi \gg \xi'}{\alpha :: \xi \gg \xi'} \tag{34}$$

$$\frac{\tau \neq \alpha :: \xi \quad \tau \neq \beta :: \xi'}{\tau \gg \circ} \tag{35}$$

$$\frac{\begin{array}{c} m \geq 1 \\ \tau \neq \alpha :: \xi \quad \tau \neq \beta :: \xi' \\ \exists i \in \{1, \ldots, m\} : \tau \gg \psi_i \end{array}}{\tau \gg \psi_1 \ / \ \cdots \ / \ \psi_m} \tag{36}$$

$$\frac{}{\{\} \gg \omega} \tag{37}$$

$\boxed{\tau \gg \psi}$

$$\frac{\beta \in Dom\ env \quad env(\beta) \gg \psi}{\beta :: \xi \gg \psi} \tag{38}$$

$$\frac{\xi \gg \psi}{\alpha :: \xi \gg \psi} \tag{39}$$

$$\frac{\forall i \in \{1, \ldots, n\} : \tau_i \gg \psi_i \quad n, k \geq 0}{[a_1 = \tau_1, \ldots, a_n = \tau_n]d\{c_1, \ldots, c_k\} \gg [a_1 = \psi_1, \ldots, a_n = \psi_n]d\{c_1, \ldots, c_k\}} \tag{40}$$

Comments:

- Notice that neither the relation $\tau \gg \xi$ nor the relation $\tau \gg \psi$ does any unification.
- Rule (36): We can safely use $\exists$ here since $\tau \gg \psi_i$ does not unify anything.
- Notice that there is no rule for $\beta :: \xi$ where $\beta \notin Dom\ env$ in the relation $\tau \gg \psi$. Nested $[a_1 = \tau_1, \ldots, a_n = \tau_n]d\{c_1, \ldots, c_k\}$ may contain $\beta :: \xi$ and in a perfect world this would really require *unification*. So ξ-Calculus and thus also CeXL has the limitation that the programmer may have to supply an explicit type constraint somewhere to avoid this situation.
- Rule (38): The premise $env(\beta) \gg \psi$ is checked with the relation $\tau \gg \psi$.
- Rule (39): The ψ in the premise $\xi \gg \psi$ here is really a ξ consisting of a single ψ_i in a $\psi_1 \ / \ \cdots \ / \ \psi_m$. So this premise is checked with the relation $\xi \gg \xi'$ which does not do any unification.
- Rule (40): We must be able to unify $[]exn\{\}$ so $k \geq 0$.

14.5.8 Type Pattern Respects Restriction

$\boxed{\psi \gg \xi}$

$$\frac{}{\psi \gg \circ} \tag{41}$$

$$\frac{\psi \in \{\psi_1, \ldots, \psi_m\}}{\psi \gg \psi_1 \ / \ \cdots \ / \ \psi_m} \tag{42}$$

14.5.9 ξ More Restrictive Than ξ'

$\boxed{\xi \gg \xi'}$

$$\frac{}{\xi \gg \circ} \tag{43}$$

$$\frac{\omega \supseteq \omega'}{\omega \gg \omega'} \tag{44}$$

$$\frac{1 \leq n \leq m \quad \{\psi_1, \ldots, \psi_n\} \subseteq \{\psi'_1, \ldots, \psi'_m\}}{\psi_1 \ / \ \cdots \ / \ \psi_n \gg \psi'_1 \ / \ \cdots \ / \ \psi'_m} \tag{45}$$

Comments:

- Rules (41) and (42): These rules are only used in the translation from Naked CeXL to ξ-Calculus.
- Rule (45): This only works as intended because all ψ_i are mutually disjoint and similarly for all ψ'_j. The intuition is that for the left hand side to be more restrictive, it must have less type patterns and each type pattern must exist on the right hand side.

15 Static Semantics of ξ-Calculus - Inference

This section describes the general rules which are used when doing type inference for the static semantics of ξ-Calculus.

15.1 Environments

We will use the following environments for the static semantics of ξ-Calculus:

Γ	VId $\overset{fin}{\rightarrow}$ TyScheme	Environment of variables bound to type schemes
Δ	TyVar $\overset{fin}{\rightarrow}$ Restrict	Environment of type variables bound to restrictions

15.2 Generalizing by Type Schemes: $\sigma \succ \tau$

At certain points during type inference, type schemes must generalize some type being inferred. This is written $\sigma \succ \tau$. In practice, for an implementation based on the unification algorithm, this means that the variables $\alpha_1 :: \xi_1, \ldots, \alpha_n :: \xi_n$ must be instantiated with meta variables $\beta_1 :: \xi_1, \ldots, \beta_n :: \xi_n$ with the same restrictions. Thus, the $\forall$ quantifier disappears and the type resulting from the instantiation is ready to be unified, since it contains appropriate meta variables.

To spell it out, if σ has the form

$$\forall[\alpha_1 :: \xi_1, \ldots, \alpha_n :: \xi_n].[a_1 = \alpha_1 :: \xi_1, \ldots, a_n = \alpha_n :: \xi_n]d\{c_1, \ldots, c_k\}$$

then we can set

$$\tau = [a_1 = \beta_1 :: \xi_1, \ldots, a_n = \beta_n :: \xi_n]d\{c_1, \ldots, c_k\}$$

where τ is now the type ready for unification. If instead σ has the form

$$\forall[\alpha_1 :: \xi_1, \ldots, \alpha_n :: \xi_n].\tau' \rightarrow [a_1 = \alpha_1 :: \xi_1, \ldots, a_n = \alpha_n :: \xi_n]d\{c_1, \ldots, c_k\}$$

we set (using the typical notation for substitution in τ' inside the $\lceil \ \rceil$)

$$\begin{aligned}\tau = \tau'\lceil\beta_1 :: \xi_1/\alpha_1 :: \xi_1, \ldots, \beta_n :: \xi_n/\alpha_n :: \xi_n\rceil \rightarrow \\ [a_1 = \beta_1 :: \xi_1, \ldots, a_n = \beta_n :: \xi_n]d\{c_1, \ldots, c_k\}\end{aligned}$$

As mentioned earlier, σ will always have one of these 2 forms for constructors. If σ is an arbitrary type scheme it will have the general form

$$\forall[\alpha_1 :: \xi_1, \ldots, \alpha_n :: \xi_n].\tau'$$

in which case we can set (again using the notation for substitution)

$$\tau = \tau'\lceil\beta_1 :: \xi_1/\alpha_1 :: \xi_1, \ldots, \beta_n :: \xi_n/\alpha_n :: \xi_n\rceil$$

For all the substitutions in this section, it should be noted that any constructors (i.e. ConsType) occuring nested within τ' may not have its type parameters substituted. It is because that these are just for naming the parameters of the constructors. This is also why they are called a rather than α. We didn't substitute them in the above descriptions of constructor types either. They are used to give CeXL the feature *Named Type Parameters*.

15.3 Non-expansive Expressions

In order to treat polymorphic references and exceptions, the class of ξ-Calculus expressions is partitioned in to two classes, the *expansive* and the *non-expansive* expressions. An expression is *non-expansive* if it can be generated by the following grammar from the non-terminal *ne*:

$$\begin{array}{llll} (\textit{non-expansive}) & ne & ::= & \lambda x.e \mid ce\ ne \mid x \mid ne : \tau \mid c : \sigma \mid c\ \mathbf{ex}\ \tau \mid scon \mid \\ & & & \{\,\} \mid \{lab_1 = ne_1, \ldots, lab_m = ne_m, ne\} \mid \\ & & & ne_1\ ?\ ne_2 \\ \\ (\textit{constructor exp}) & ce & ::= & ce : \tau \mid c : \sigma \mid c\ \mathbf{ex}\ \tau \end{array}$$

Restriction: Within a *ce* we require that $c \neq ref$.
Notice that e in the body of the λ construct may be any expression.

All other expressions are said to be *expansive*. The idea is that the dynamic evaluation of a non-expansive expression will not extend the domain of the memory, while the evaluation of an expansive expression might.

15.4 Scope of Explicit Type Variables

In CeXL, a type or datatype binding can explicitly introduce type variables whose scope is that binding. The `val` bindings also bind type variables. The binding of type variables in `val` bindings is handled by the semantics of ξ-Calculus. In ξ-Calculus consider the part $p_1 = e_1\ ;\ \cdots\ ;\ p_n = e_n$ of a **let** or a **letrec** expression. We call such a part a *value binding* since it comes from a `val` binding in CeXL. It is for these value bindings that type variables are bound in ξ-Calculus.

The binding of type variables in value bindings happen by the occurrences of explicit free type variables in the "$: \tau$" of a typed expression or pattern or in the "τ" of an **ex** or a **letex** construct or in the "α" or the "τ" of a **fieldcase** construct. The type parameters $a_1, \ldots, a_n$ found in ConsType and TyPat do not count here - since they are really just to be considered as parameter names for constructor types. For the rest of this section we consider only the free occurrences of type variables in the places just stated.

Every occurrence of a **let** or **letrec** expression is said to *scope* a set of explicit type variables determined as follows.

First, a free occurrence of α in a value binding $p_1 = e_1\ ;\ \cdots\ ;\ p_n = e_n$ is said to be *unguarded* if the occurrence is not part of a smaller value binding $p'_1 = e'_1\ ;\ \cdots\ ;\ p'_m = e'_m$ of a **let** or **letrec** expression within the expressions e_1, ..., e_n. In this case we say that α occurs *unguarded* in the value binding $p_1 = e_1\ ;\ \cdots\ ;\ p_n = e_n$.

Then we say that α is *implicitly scoped* at a particular value binding in a program if (1) α occurs unguarded in this value binding, and (2) α does not occur unguarded in any larger value binding containing the given one.

In section 16.3 we will see the rules for traversing a ξ-Calculus program to infer the implicitly scoped type variables for a value binding. They are used in the inference rules for expressions in the let and letrec constructs. All occurences of type variables must have the *same* restrictions everywhere in the scope of their

bindings. This is also ensured by the inference rules presented in section 16.3. Thus for example in the two declarations:

$$\begin{array}{l}\textbf{let } v = \textbf{let } id : \alpha :: \circ \to \alpha :: \circ = \lambda x.x \textbf{ in } (\lambda y.\{\})(id\ id) \\ \textbf{in } \{\}\end{array}$$

$$\begin{array}{l}\textbf{let } v = (\lambda y.\ (\lambda x.(x : \alpha :: \circ))) \\ \qquad (\ \textbf{let } id : \alpha :: \circ \to \alpha :: \circ = \lambda x.x \textbf{ in } (\lambda y.\{\})(id\ id)\ ) \\ \textbf{in } \{\}\end{array}$$

the type variable α is scoped differently. In the first example α is implicitly scoped at the binding of *id*. In the second example α is implicitly scoped at the outer binding of *v*.

Then, according to the inference rules the first example can be elaborated, but the second cannot since α is bound at the outer declaration leaving no possibility of two different instantiations of the type of *id* in the application *id id*. Section 25.2 of Appendix A has more example programs of legal and illegal scoping of variables.

15.5 Closure

Let τ be a type and A a semantic object. Then $Clos_A(\tau)$, the *closure* of τ with respect to A, is the type scheme $\sigma = \forall[\alpha_1 :: \xi_1, \ldots, \alpha_n :: \xi_n].\tau$ where $\alpha_1 :: \xi_1, \ldots, \alpha_n :: \xi_n = tyvars\ \tau \backslash tyvars\ A$. Commonly A will be an environment Δ of type variables. We abbreviate the *total* closure $Clos_{\{\}}(\tau)$ to $Clos(\tau)$. If the range of a value environment Γ (as defined for the inference rules in section 16) contains only types (rather than arbitrary type schemes) we set

$$Clos_A(\Gamma) = \{x \mapsto Clos_A(\tau) \mid \Gamma(x) = \tau\}$$

Closing a value environment Γ that stems from the elaboration of a value binding $p_1 = e_1\ ;\ \cdots\ ;\ p_n = e_n$ requires extra care to ensure type safety of references and exceptions, correct scoping of explicit type variables and correct choice of implicitly inferred type variables. Assume that Δ is the environment of type variables which have been implicitly scoped by outer value bindings. The closure is taken with respect to Δ. Assume that Δ' is the inferred environment of type variables which are implicitly scoped at the value binding. Assume that Γ' is the variable environment from outside the value binding. The value binding is not allowed to bind the same variable twice, which is ensured by syntactical constraints. Thus for each $x \in \mathrm{Dom}\ \Gamma$ there is a unique $p_i = e_i$ in the value binding which binds x. If $\Gamma(x) = \tau$, then let

$$Clos_{\Delta,\Delta',\Gamma',p_1=e_1;\cdots;p_n=e_n}\Gamma(x) = \forall[\alpha_1 :: \xi_1, \ldots, \alpha_n :: \xi_n].\tau, \text{ where}$$

$$[\alpha_1 :: \xi_1, \ldots, \alpha_n :: \xi_n] = \begin{cases} [tyvars\ \tau \backslash\ tyvars\ \Delta] & \text{, if } e \text{ is non-expansive} \\ [\,] & \text{, if } e \text{ is expansive} \end{cases}$$

During closure, some type variables are implicitly inferred. These are the type variables which result from free types, which are represented as unbound meta variables in the type system. Such type variables for τ must be chosen to be fresh variables outside the set

$$tyvars\ p_1 \cup tyvars\ e_1 \cup \cdots \cup tyvars\ p_n \cup tyvars\ e_n \cup tyvars\ \Delta.$$

Then we have

$$tyvars\ \tau \cap ((tyvars\ p_i \cup tyvars\ e_i) \backslash tyvars\ \Delta') = \emptyset$$

after closure. This way of chosing type variables will avoid capturing type variables which occur unguarded in any value bindings of nested let or letrec expressions. The program "Legal Test 2" in section 25.2 of Appendix A is an example of a program which must elaborate without problems.

We always require principal types at top-level[4] value bindings and to be sure to find a valid typing of programs if one exists, we must always find the most general unifier of all types during closure. This is done by instantiating unbound meta variables to type variables according to these rules:

- No meta variables are instantiated with type variables in the types of bound variables for expansive expressions.
- If the closure is taken in a top-level let or letrec, all unbound meta variables must be instantiated with type variables.
- As many meta variables as possible must be instantiated with type variables in the types of bound variables for non-expansive expressions. This will always exclude the meta variables occuring in the sorrounding environment Γ'.

Note that the first 2 of the 3 rules taken together imply that declaration of polymorphic variables resulting from expansive expressions will fail at top-level. The programs "Illegal Test 1" and "Illegal Test 2" in section 25.3 of Appendix A are examples of programs which must be rejected.

15.6 Tidying Up During Closure

When we take the closure, we need to tidy up the type being closed to avoid ending up with silly types. It is also described here what really happens with the meta variables used in the unification algorithm. What we have to do is the following:

- Replace unbound meta variables $\beta :: \xi$, where the restriction ξ represents a *single concrete type*, by the type that the ξ represents. For instance, $\beta :: []int\{\cdots\}$ should be instantiated to $[]int\{\cdots\}$. This happens in restrictions of the form $\psi/^m$ whenever there is only one ψ. Converting this to a type amounts to recursively traversing the constructor types of ψ and replacing them with types consisting of the same constructor types.
- It is also possible for all record types occurring in the inferred type to be flattened into a single record type, rather than a list of smaller records - but it is not necessary. Whether this would give more efficiency or not will most likely depend on the implementation. The only requirement is, that records must be displayed to the user as flattened records, so the easiest would definitely be to do it at this point.

[4] What we call top-level here is the outermost sequence of let, letrec and letex expressions in ξ-Calculus

- In all record types, all record fields which are bound to types of the form $[]absent\{absent\}\ ?\ \tau$ must be *removed* from the record types after the closure operation in types for non-function and non-reference values. This avoids some vacant limitations on extensions of record types, since we are doing strict extension of records. It actually also avoids some weird behaviour in the type system of the language, as we saw in the example in section 3.13.

  The reason for not just removing absent fields immediately during inference can be seen in the justification for the claim in section 7.1. Also, removing absent fields for function values or reference values may be unsound

- Unbound meta variables are bound to fresh type variables according to the rules of the previous section and all bound meta variables are replaced with what they are bound to.

15.7 Predefined Constructors and Type Schemes

In the semantics of ξ-Calculus, we rely on the following constructor types (ConsType) to be predefined:

$[]absent\{absent\}$	signifies an absent record field
$[]present\{present\}$	signifies a present record field
$[a = \cdot]ref\{ref\}$	special constructor for supporting references
$[]exn\{\}$	special constructor only for exceptions

We also rely on the following type schemes (TyScheme) to be predefined:

σ_{abs}	=	$\forall[].[]absent\{absent\}$	for constructor for absent record field
σ_{pre}	=	$\forall[].[]present\{present\}$	for constructor for present record field
σ_{ref}	=	$\forall[\alpha :: \circ].\alpha :: \circ \rightarrow [a = \alpha :: \circ]ref\{ref\}$	for constructor for references

16 Type Inference for ξ-Calculus

16.1 Judgement Forms

Essential Judgements

$\Gamma; \Delta \vdash e \Rightarrow \tau$	Infer type of e in environment Γ
$\Gamma; \Delta \vdash p \Rightarrow \tau; \Gamma'$	Infer type of p in Γ and produce new Γ'

Well-formedness of Restrictions and Type Specifications

$\vdash \xi$	Check that the restriction ξ is well-formed
$! \ \psi_1 \nleftrightarrow \cdots \nleftrightarrow \psi_m \ !$	Check that all type patterns ψ_i are mutually disjoint
$\psi_1 \nleftrightarrow \psi_2$	Check that a pair of type patterns ψ_1 and ψ_2 are disjoint
$\Delta \vdash^{tspec} \tau$	Verify that τ is a valid type specification with restrictions on type variables as given in Δ assuming that the restrictions in Δ are already well-formed
$\Delta; \omega \vdash^{tspecrec} \tau$	Verify that τ is a valid record type specification without any of the fields in ω and with restrictions on type variables as given in Δ assuming that the restrictions in Δ are already well-formed
$\Delta \vdash^{tspecopt} \tau$	Verify that τ is a valid type specification for an optional record field and with restrictions on type variables as given in Δ assuming that the restrictions in Δ are already well-formed

Rules for Implicitly Scoped Type Variables

$\Delta \blacktriangleright p_1 = e_1 \ ; \ \cdots \ ; \ p_n = e_n \Rightarrow \Delta'$	Infer the implicitly scoped type variables from the *value binding* part of a let or a letrec expression where Δ is already scoped at that value binding
$\Delta \blacktriangleright e \Rightarrow \Delta'$	Infer the implicitly scoped type variables from e where Δ is scoped in the enclosing value binding
$\Delta \blacktriangleright p \Rightarrow \Delta'$	Infer the implicitly scoped type variables from p where Δ is scoped in the enclosing value binding
$\Delta \blacktriangleright^{tspec} \tau \Rightarrow \Delta'$	Infer the implicitly scoped type variables of τ where Δ is scoped in the enclosing value binding

16.2 Inference Rules

Expressions

$$\boxed{\Gamma;\Delta \vdash e \Rightarrow \tau}$$

$$\frac{\Gamma + \{x \mapsto \tau\};\Delta \vdash e \Rightarrow \tau'}{\Gamma;\Delta \vdash \lambda x.e \Rightarrow \tau \to \tau'} \tag{46}$$

$$\frac{\Gamma;\Delta \vdash e_1 \Rightarrow \tau \to \tau' \quad \Gamma;\Delta \vdash e_2 \Rightarrow \tau}{\Gamma;\Delta \vdash e_1 e_2 \Rightarrow \tau'} \tag{47}$$

$$\frac{x \in Dom\ \Gamma \quad \sigma = \Gamma(x) \quad \sigma \succ \tau}{\Gamma;\Delta \vdash x \Rightarrow \tau} \tag{48}$$

$$\frac{\Gamma;\Delta \vdash e \Rightarrow \tau \quad \Delta \vdash^{tspec} \tau}{\Gamma;\Delta \vdash e : \tau \Rightarrow \tau} \tag{49}$$

$$\frac{\sigma \succ \tau}{\Gamma;\Delta \vdash c : \sigma \Rightarrow \tau} \tag{50}$$

$$\frac{}{\Gamma;\Delta \vdash c\ \mathbf{ex}\ \tau \Rightarrow \tau} \tag{51}$$

$$\frac{}{\Gamma;\Delta \vdash scon \Rightarrow type(scon)} \tag{52}$$

$$\frac{}{\Gamma;\Delta \vdash \mathbf{\{\}} \Rightarrow \{\}} \tag{53}$$

$$\frac{\begin{array}{c} lab_1,\ldots,lab_m \text{ are all distinct} \quad m \geq 1 \\ \forall i \in \{1,\ldots,m\} : \Gamma;\Delta \vdash e_i \Rightarrow \tau_i\ ?\ \tau_i' \\ \forall i \in \{1,\ldots,m\} : \xi_i = []absent\{absent\}\ /\ []present\{present\} \\ \forall i \in \{1,\ldots,m\} : \tau_i|_{\xi_i} \\ \omega = \{lab_1,\ldots,lab_m\} \quad \tau|_\omega \quad \Gamma;\Delta \vdash e \Rightarrow \tau \end{array}}{\Gamma;\Delta \vdash \mathbf{\{}lab_1\mathbf{=}e_1\mathbf{,}\ldots\mathbf{,}lab_m\mathbf{=}e_m\mathbf{,}e\mathbf{\}} \Rightarrow \{lab_1 : \tau_1\ ?\ \tau_1', \ldots, lab_m : \tau_m\ ?\ \tau_m'; \tau\}} \tag{54}$$

$$\frac{\begin{array}{c} \Gamma;\Delta \vdash e_1 \Rightarrow \tau_1 \quad \Gamma;\Delta \vdash e_2 \Rightarrow \tau_2 \\ \xi = []absent\{absent\}\ /\ []present\{present\} \quad \tau_1|_\xi \end{array}}{\Gamma;\Delta \vdash e_1\ \mathbf{?}\ e_2 \Rightarrow \tau_1\ ?\ \tau_2} \tag{55}$$

$$\frac{\forall i \in \{1,\ldots,m\} : \Delta \vdash^{tspec} \tau_i \quad \Gamma;\Delta \vdash e \Rightarrow \tau' \quad m \geq 1}{\Gamma;\Delta \vdash \mathbf{letex}\ c_1 : \tau_1, \ldots,\ c_m : \tau_m\ \mathbf{in}\ e \Rightarrow \tau'} \tag{56}$$

$$\frac{\begin{array}{c}\Delta \blacktriangleright p_1 = e_1 \;;\; \cdots \;;\; p_n = e_n \Rightarrow \Delta' \quad n \geq 0 \\ \{\}; \Delta + \Delta' \vdash p_1 \Rightarrow \tau_1; \Gamma_1 \quad \Gamma; \Delta + \Delta' \vdash e_1 \Rightarrow \tau_1 \\ \Gamma_1; \Delta + \Delta' \vdash p_2 \Rightarrow \tau_2; \Gamma_2 \quad \Gamma; \Delta + \Delta' \vdash e_2 \Rightarrow \tau_2 \\ \vdots \\ \Gamma_{n-1}; \Delta + \Delta' \vdash p_n \Rightarrow \tau_n; \Gamma_n \quad \Gamma; \Delta + \Delta' \vdash e_n \Rightarrow \tau_n \\ p_1, \ldots, p_n \textit{ together may not bind the same variable } x \textit{ more than once} \\ \forall i \in \{1, \ldots, n\} : p_i \textit{ is exhaustive on } \tau_i \\ \Gamma'_n \;=\; Clos_{\Delta, \Delta', \Gamma, p_1 = e_1; \cdots; p_n = e_n}\ \Gamma_n \\ \Gamma + \Gamma'_n; \Delta \vdash e \Rightarrow \tau \end{array}}{\Gamma; \Delta \vdash \textbf{let } p_1 = e_1 \;;\; \cdots \;;\; p_n = e_n \textbf{ in } e \Rightarrow \tau} \tag{57}$$

$$\frac{\begin{array}{c}\Delta \blacktriangleright p_1 = e_1 \;;\; \cdots \;;\; p_m = e_m \Rightarrow \Delta' \quad m \geq 1 \\ \{\}; \Delta + \Delta' \vdash p_1 \Rightarrow \tau_1; \Gamma_1 \quad \Gamma + \Gamma_m; \Delta + \Delta' \vdash e_1 \Rightarrow \tau_1 \\ \Gamma_1; \Delta + \Delta' \vdash p_2 \Rightarrow \tau_2; \Gamma_2 \quad \Gamma + \Gamma_m; \Delta + \Delta' \vdash e_2 \Rightarrow \tau_2 \\ \vdots \\ \Gamma_{m-1}; \Delta + \Delta' \vdash p_m \Rightarrow \tau_m; \Gamma_m \quad \Gamma + \Gamma_m; \Delta + \Delta' \vdash e_m \Rightarrow \tau_m \\ p_1, \ldots, p_m \textit{ together may not bind the same variable } x \textit{ more than once} \\ \forall i \in \{1, \ldots, m\} : p_i \textit{ is exhaustive on } \tau_i \\ \Gamma'_m \;=\; Clos_{\Delta, \Delta', \Gamma, p_1 = e_1; \cdots; p_m = e_m}\ \Gamma_m \\ \Gamma + \Gamma'_m; \Delta \vdash e \Rightarrow \tau \end{array}}{\Gamma; \Delta \vdash \textbf{letrec } p_1 = e_1 \;;\; \cdots \;;\; p_m = e_m \textbf{ in } e \Rightarrow \tau} \tag{58}$$

$$\frac{\begin{array}{c}\Gamma; \Delta \vdash e \Rightarrow \tau \quad m \geq 1 \\ \Gamma; \Delta \vdash p_1 \Rightarrow \tau; \Gamma_1 \quad \Gamma_1; \Delta \vdash e_1 \Rightarrow \tau' \\ \vdots \\ \Gamma; \Delta \vdash p_m \Rightarrow \tau; \Gamma_m \quad \Gamma_m; \Delta \vdash e_m \Rightarrow \tau' \\ \forall i \in \{1, \ldots, m\} : \textit{each variable } x \textit{ in } p_i \textit{ occurs only once} \\ \textit{A warning must be issued if } p_1, \ldots, p_m \textit{ are not exhaustive on } \tau \\ p_1, \ldots, p_m \textit{ may not be redundant on } \tau \end{array}}{\Gamma; \Delta \vdash \textbf{case } e \textbf{ of } p_1 \rightrightarrows e_1 \parallel \cdots \parallel p_m \rightrightarrows e_m \Rightarrow \tau'} \tag{59}$$

$$\frac{\begin{array}{c}\alpha \in Dom\ \Delta \quad \xi = \Delta(\alpha) \\ \xi = [] \textit{absent}\{\textit{absent}\} \;/\; [] \textit{present}\{\textit{present}\} \\ \Gamma; \Delta \vdash e \Rightarrow \alpha :: \xi \;?\; \tau' \\ \Gamma; \Delta \vdash p \Rightarrow \tau'; \Gamma' \quad \Delta \vdash^{tspec} \tau \\ \textit{each variable } x \textit{ in } p \textit{ occurs only once} \quad p \textit{ is exhaustive on } \tau' \\ \Gamma; \Delta \vdash e_1 \Rightarrow \tau \lceil\ [] \textit{absent}\{\textit{absent}\} \;/\; \alpha :: \xi\ \rceil \\ \Gamma'; \Delta \vdash e_2 \Rightarrow \tau \lceil\ [] \textit{present}\{\textit{present}\} \;/\; \alpha :: \xi\ \rceil \end{array}}{\Gamma; \Delta \vdash \textbf{fieldcase } e \textbf{ in } \alpha \textbf{ of absent} \rightrightarrows e_1 \parallel p \rightrightarrows e_2 \textbf{ type } \tau \Rightarrow \tau} \tag{60}$$

$$\frac{\begin{array}{c} m \geq 1 \quad \Gamma;\Delta \vdash e \Rightarrow \tau \\ \Gamma;\Delta \vdash p_1 \Rightarrow [\,]exn\{\};\Gamma_1 \quad \Gamma_1;\Delta \vdash e_1 \Rightarrow \tau \\ \vdots \\ \Gamma;\Delta \vdash p_m \Rightarrow [\,]exn\{\};\Gamma_m \quad \Gamma_m;\Delta \vdash e_m \Rightarrow \tau \\ \forall i \in \{1,\ldots,m\} : \textit{each variable } x \textit{ in } p_i \textit{ occurs only once} \\ p_1,\ldots,p_m \textit{ may not be redundant on } [\,]exn\{\} \end{array}}{\Gamma;\Delta \vdash e \textbf{ handle } p_1 \rightrightarrows e_1 \parallel \cdots \parallel p_m \rightrightarrows e_m \Rightarrow \tau} \tag{61}$$

$$\frac{\Gamma;\Delta \vdash e \Rightarrow [\,]exn\{\}}{\Gamma;\Delta \vdash \textbf{raise } e \Rightarrow \tau} \tag{62}$$

Comments on Static Semantics of Expressions

- Rules (48) and (50): The instantiation of type schemes allows different occurrences of a single variable x or constructor c to assume different types.
- Rule (49) and (60): τ must indicate the degrees of freedom exactly by using type variables. The restrictions on these type variables are given by the environment Δ.
- Rule (51): τ will have one of the 2 forms $[\,]exn\{\}$ or $\tau' \rightarrow [\,]exn\{\}$ which is ensured by the Naked CeXL to ξ-Calculus translation.
- Rule (53): Giving the empty record the type $\{\}$ means that it will unify only with record types containing absent fields and ending in $\{\}$.
- Rule (54): Notice that m can *not* be zero - thus the empty extensible record cannot be expressed as an expression, but only as type restrictions. The premise $\tau|_\omega$ is part of what asserts that all record field names in any record type being inferred are distinct.
- Rule (56): For soundness we need exception declarations both to ensure correct scoping of type variables and to be able to allocate exception names dynamically. When type variables are allowed in exception types we would get problems if exception names were not allocated dynamically. See the regression test in section 25.4 for an example of this. We could have avoided exception declarations in ξ-Calculus and dynamic exception name allocation by disallowing type variables in exceptions.
- Rules (57) and (58): The premise $tyvars\ \Gamma'_m \setminus tyvars\ \Delta = \emptyset$ ensures that type variables occuring free in the range of Γ_m are bound by the closure operation, unless they are already scoped by an outer let or letrec binding. The type variables scoped by an outer let or letrec are those in Δ.
- Rules (58): From the inference of all the p_i we see that any type scheme occuring in Γ_m will have to be a type so any use of a recursive function in its own body must be assigned the same type.
- Rule (60): The notation inside $\lceil\ \rceil$ is the usual notation for substitution (i.e. β-conversion).
- Rule (62): Note that τ does not occur in the premise, so a raise expression has "arbitrary" type.

Patterns

$$\boxed{\Gamma;\Delta \vdash p \Rightarrow \tau;\Gamma'}$$

$$\frac{}{\Gamma;\Delta \vdash x \Rightarrow \tau;\Gamma + \{x \mapsto \tau\}} \tag{63}$$

$$\frac{\Gamma;\Delta \vdash p \Rightarrow \tau;\Gamma'}{\Gamma;\Delta \vdash x\ \mathbf{as}\ p \Rightarrow \tau;\Gamma' + \{x \mapsto \tau\}} \tag{64}$$

$$\frac{\Delta \vdash^{tspec} \tau \qquad \Gamma;\Delta \vdash p \Rightarrow \tau;\Gamma'}{\Gamma;\Delta \vdash p : \tau \Rightarrow \tau;\Gamma'} \tag{65}$$

$$\frac{\sigma \succ \tau}{\Gamma;\Delta \vdash c : \sigma \Rightarrow \tau;\Gamma} \tag{66}$$

$$\frac{\sigma \succ \tau' \to \tau \qquad \Gamma;\Delta \vdash p \Rightarrow \tau';\Gamma'}{\Gamma;\Delta \vdash c(p) : \sigma \Rightarrow \tau;\Gamma'} \tag{67}$$

$$\frac{}{\Gamma;\Delta \vdash c\ \mathbf{ex} \Rightarrow []exn\{\};\Gamma} \tag{68}$$

$$\frac{\Gamma;\Delta \vdash p \Rightarrow \tau;\Gamma'}{\Gamma;\Delta \vdash c(p)\ \mathbf{ex}\ \tau \Rightarrow []exn\{\};\Gamma'} \tag{69}$$

$$\frac{}{\Gamma;\Delta \vdash scon \Rightarrow type(scon);\Gamma} \tag{70}$$

$$\frac{}{\Gamma;\Delta \vdash _ \Rightarrow \tau;\Gamma} \tag{71}$$

$$\frac{}{\Gamma;\Delta \vdash \mathbf{\{\}} \Rightarrow \{\};\Gamma} \tag{72}$$

$$\frac{\begin{array}{c} lab_1,\ldots,lab_m \text{ are all distinct} \qquad m \geq 1 \\ \forall i \in \{1,\ldots,m\} : \Gamma_i;\Delta \vdash p_i \Rightarrow \tau_i\ ?\ \tau_i';\Gamma_{i+1} \\ \forall i \in \{1,\ldots,m\} : \xi_i = []absent\{absent\}\ /\ []present\{present\} \\ \forall i \in \{1,\ldots,m\} : \tau_i|_{\xi_i} \\ \omega = \{lab_1,\ldots,lab_m\} \qquad \tau|_\omega \qquad \Gamma_{m+1};\Delta \vdash p \Rightarrow \tau;\Gamma_{m+2} \end{array}}{\Gamma_1;\Delta \vdash \mathbf{\{}lab_1\mathbf{=}e_1\mathbf{,}\ldots\mathbf{,}lab_m\mathbf{=}e_m\mathbf{,e\}} \Rightarrow \{lab_1 : \tau_1\ ?\ \tau_1', \ldots, lab_m : \tau_m\ ?\ \tau_m';\tau\};\Gamma_{m+2}} \tag{73}$$

$$\frac{\Gamma;\Delta \vdash p \Rightarrow \tau;\Gamma'}{\Gamma;\Delta \vdash present : \sigma_{pre}\ \mathbf{?}\ p \Rightarrow []present\{present\}\ ?\ \tau;\Gamma'} \tag{74}$$

Comments on Static Semantics of Patterns

- Rule (65): τ must indicate the degrees of freedom exactly by using type variables. The restrictions on these type variables are given by the environment Δ.
- Rule (66): σ must have the form $\forall[\alpha_1 :: \xi_1, \ldots, \alpha_n :: \xi_n].[a_1 = \alpha_1 :: \xi_1, \ldots, a_n = \alpha_n :: \xi_n]d\{c_1, \ldots, c_k\}$ which is esured during Naked CeXL to ξ-calculus translation.
- Rule (67): σ must have the form $\forall[\alpha_1 :: \xi_1, \ldots, \alpha_n :: \xi_n].\tau' \rightarrow [a_1 = \alpha_1 :: \xi_1, \ldots, a_n = \alpha_n :: \xi_n]d\{c_1, \ldots, c_k\}$ which is esured during Naked CeXL to ξ-calculus translation.
- Rules (66) and (67): The instantiation of type schemes allows different occurrences of a single constructor c to assume different types.
- Rule (74): The pattern-matching construct p_1 **?** p_2 can only be used with the constructor *present* of type $[]present\{present\}$. This constructor type has the predefined type scheme σ_{pre} which was described in section 15.7.

Well-formed Restrictions

$\boxed{\vdash \xi}$

$$\frac{}{\vdash \circ} \tag{75}$$

$$\frac{}{\vdash \omega} \tag{76}$$

$$\frac{m \geq 1 \qquad ! \ \psi_1 \nleftrightarrow \cdots \nleftrightarrow \psi_m \ !}{\psi_1 \ / \ \cdots \ / \ \psi_m} \tag{77}$$

Mutually Disjoint Type Patterns

$\boxed{! \ \psi_1 \nleftrightarrow \cdots \nleftrightarrow \psi_m \ !}$

$$\frac{\forall i, j \in \{1, \ldots, m\}, i \neq j : \psi_i \nleftrightarrow \psi_j \qquad m \geq 1}{! \ \psi_1 \nleftrightarrow \cdots \nleftrightarrow \psi_m \ !} \tag{78}$$

Disjoint Type Patterns

$\boxed{\psi_1 \nleftrightarrow \psi_2}$

$$\frac{d \neq d' \qquad n, k, q, z \geq 0 \qquad d, d' \neq exn}{[a_1 = \psi_1, \ldots, a_n = \psi_n] d\{c_1, \ldots, c_q\} \nleftrightarrow [a'_1 = \psi'_1, \ldots, a'_k = \psi'_k] d'\{c'_1, \ldots, c'_z\}} \tag{79}$$

$$\frac{\exists i \in \{1, \ldots, n\} : \psi_i \nleftrightarrow \psi'_i \qquad n, q \geq 0 \qquad d \neq exn}{[a_1 = \psi_1, \ldots, a_n = \psi_n] d\{c_1, \ldots, c_q\} \nleftrightarrow [a_1 = \psi'_1, \ldots, a_n = \psi'_n] d\{c_1, \ldots, c_q\}} \tag{80}$$

Comments

- Rule (80): We can use the $\exists$ quantor to compare the pairs of ψ_i with ψ'_i without problems here because the relation $\nleftrightarrow$ does not do any unification. It basically just checks for equality (actually non-equality).

Type Specification

$$\boxed{\Delta \vdash^{tspec} \tau}$$

$$\frac{\beta \in Dom\ env \quad \tau = env(\beta) \quad \Delta \vdash^{tspec} \tau}{\Delta \vdash^{tspec} \beta :: \xi} \tag{81}$$

$$\frac{\beta \notin Dom\ env}{\Delta \vdash^{tspec} \beta :: \xi} \tag{82}$$

$$\frac{\alpha \in Dom\ \Delta \quad \xi = \Delta(\alpha)}{\Delta \vdash^{tspec} \alpha :: \xi} \tag{83}$$

$$\frac{\Delta \vdash^{tspec} \tau \quad \Delta \vdash^{tspec} \tau'}{\Delta \vdash^{tspec} \tau \to \tau'} \tag{84}$$

$$\frac{\forall i \in \{1, \ldots, n\} : \Delta \vdash^{tspec} \tau_i \qquad n, k \geq 0}{\Delta \vdash^{tspec} [a_1 = \tau_1, \ldots, a_n = \tau_n] d\{c_1, \ldots, c_k\}} \tag{85}$$

$$\frac{\Delta \vdash^{tspecopt} \tau \quad \Delta \vdash^{tspec} \tau'}{\Delta \vdash^{tspec} \tau\ ?\ \tau'} \tag{86}$$

$$\frac{}{\Delta \vdash^{tspec} \{\}} \tag{87}$$

$$\frac{\begin{array}{c} lab_1, \ldots, lab_m\ distinct \quad m \geq 1 \\ \forall i \in \{1, \ldots, m\} : \Delta \vdash^{tspecopt} \tau_i \\ \forall i \in \{1, \ldots, m\} : \Delta \vdash^{tspec} \tau_i' \\ \Delta; \{lab_1, \ldots, lab_m\} \vdash^{tspecendrec} \tau \end{array}}{\Delta \vdash^{tspec} \{lab_1\ :\ \tau_1\ ?\ \tau_1', \ldots, lab_m\ :\ \tau_m\ ?\ \tau_m'; \tau\}} \tag{88}$$

Record End Type Specifications

$$\boxed{\Delta; \omega \vdash^{tspecrec} \tau}$$

$$\frac{\beta \in Dom\ env \quad \tau = env(\beta) \quad \Delta; \omega \vdash^{tspecrec} \tau}{\Delta; \omega \vdash^{tspecrec} \beta :: \xi} \tag{89}$$

$$\frac{\beta \notin Dom\ env \quad \xi \gg \omega}{\Delta; \omega \vdash^{tspecrec} \beta :: \xi} \tag{90}$$

$$\frac{\begin{array}{c} \beta \notin Dom\ env \quad \beta' fresh \\ \omega \gg \xi \quad \xi \neq \omega \end{array}}{\begin{array}{c} \Delta; \omega \vdash^{tspecrec} \beta :: \xi \\ bind(env, \beta \mapsto \beta' :: \omega) \end{array}} \tag{91}$$

$$\frac{\alpha \in Dom\ \Delta \quad \xi = \Delta(\alpha) \quad \xi \gg \omega}{\Delta; \omega \vdash^{tspecrec} \alpha :: \xi} \tag{92}$$

$$\frac{}{\Delta; \omega \vdash^{tspecrec} \{\}} \tag{93}$$

$$\frac{\begin{array}{c} lab_1, \ldots, lab_m \; distinct \quad m \geq 1 \\ \omega \cap \{lab_1, \ldots, lab_m\} = \emptyset \quad \omega' = \omega \cup \{lab_1, \ldots, lab_m\} \\ \forall i \in \{1, \ldots, m\} : \Delta \vdash^{tspecopt} \tau_i \\ \forall i \in \{1, \ldots, m\} : \Delta \vdash^{tspec} \tau'_i \\ \Delta; \omega' \vdash^{tspecendrec} \tau \end{array}}{\Delta; \omega \vdash^{tspec} \{lab_1 \; : \; \tau_1 \; ? \; \tau'_1, \ldots, lab_m \; : \; \tau_m \; ? \; \tau'_m; \tau\}} \tag{94}$$

Optional Field Type Specifications

$$\boxed{\Delta \vdash^{tspecopt} \tau}$$

$$\frac{\beta \in Dom \; env \quad \tau = env(\beta) \quad \Delta \vdash^{tspecopt} \tau}{\Delta \vdash^{tspecopt} \beta :: \xi} \tag{95}$$

$$\frac{\begin{array}{c} \beta \notin Dom \; env \\ \xi \gg []absent\{absent\} \; / \; []present\{present\} \end{array}}{\Delta \vdash^{tspecopt} \beta :: \xi} \tag{96}$$

$$\frac{\begin{array}{c} \beta \notin Dom \; env \quad \beta' fresh \\ []absent\{absent\} \; / \; []present\{present\} \gg \xi \\ []absent\{absent\} \; / \; []present\{present\} \neq \xi \end{array}}{\begin{array}{c} \Delta \vdash^{tspecopt} \beta :: \xi \\ bind(env, \beta \mapsto \beta' :: []absent\{absent\} \; / \; []present\{present\}) \end{array}} \tag{97}$$

$$\frac{\begin{array}{c} \alpha \in Dom \; \Delta \quad \xi = \Delta(\alpha) \\ \xi \gg []absent\{absent\} \; / \; []present\{present\} \end{array}}{\Delta \vdash^{tspecopt} \alpha :: \xi} \tag{98}$$

$$\frac{}{\Delta \vdash^{tspecopt} []absent\{absent\}} \tag{99}$$

$$\frac{}{\Delta \vdash^{tspecopt} []present\{present\}} \tag{100}$$

Comments on Type Specifications

- Rules (81)-(100): These rules check that record types in type specifications are well-formed and that all type variables occuring in the type have the same restrictions as specified in the environment of type variables. All type variables must have their restrictions specified explicitly. An occurrence of a type variable also explicitly indicates the degree of freedom in a type, since a type variable can never be bound to a type.

- Rules (81), (82), (89), (90), (91), (95), (96) and (97): Meta variables are explicitly handled in type specifications. This is actually part of how a unification algorithm for ξ-Calculus is implemented and it can be considered quite a "hack" in the semantics. It is necessary to support the feature *Partial Type Instantiation*. An alternative would be to move the semantics of declared types into ξ-Calculus, but that would take away the simplicity of ξ-Calculus. Another alternative is to check the well-formedness of types during the translation from Naked CeXL, which might be a prettier way than this, since we already need to check declarations at that time.

In these rules, *env* refers to the environment of meta variables used in the unification relation of section 14. Relations like $\xi \gg \omega$ are also from that section.

- Rule (85): The constructor $[]exn\{\}$ is allowed in type specifications which is why we have $k \geq 0$ rather than $k \geq 1$.
- Rule (94): This rule will not be used if the type specifications in the rules (49), (56), (60) and (65) are checked before any other unification is done in those 3 rules. This is because, that the translation from Naked CeXL to ξ-Calculus will never generate type specifications of this form directly.

16.3 Rules for Implicitly Scoped Type Variables

The following are the inference rules for implicitly scoped type variables. For these inference rules, we will define an operator $\uplus$ for combining two environments Δ and Δ' of type variables in a certain way.

16.3.1 The $\uplus$ Operator

We define $\uplus$ as follows. Using the $\uplus$ operator on two environments Δ and Δ' of type variables, will make a new environment Δ'', where Dom Δ'' = Dom $\Delta \cup$ Dom Δ'. However, the $\uplus$ operation must fail, if for some α we have $\xi = \Delta(\alpha)$, $\xi' = \Delta'(\alpha)$ and $\xi \neq \xi'$. If $\uplus$ fails, it means that the ξ-Calculus program (and thus also the CeXL program it comes from) is invalid due to conflicting restrictions on the same type variable, and this must be reported to the user as an error. If $\uplus$ succeeds, the resulting Δ'' is given by $\Delta'' = \Delta \cup \Delta'$, which is equivalent to when we write $\Delta'' = \Delta + \Delta'$.

Inference Rules

$$\boxed{\Delta \blacktriangleright p_1 = e_1 \; ; \; \cdots \; ; \; p_n = e_n \Rightarrow \Delta'}$$

$$\frac{\begin{array}{c} n \geq 0 \\ \forall i \in \{1, \ldots, n\} : \Delta \blacktriangleright p_i \Rightarrow \Delta_i \\ \forall i \in \{1, \ldots, n\} : \Delta \blacktriangleright e_i \Rightarrow \Delta'_i \\ \Delta' = \Delta_1 \uplus \Delta'_1 \uplus \Delta_2 \uplus \Delta'_2 \uplus \cdots \uplus \Delta_n \uplus \Delta'_n \end{array}}{\Delta \blacktriangleright p_1 = e_1 \; ; \; \cdots \; ; \; p_n = e_n \Rightarrow \Delta'} \tag{101}$$

$$\boxed{\Delta \blacktriangleright e \Rightarrow \Delta'}$$

$$\frac{\Delta \blacktriangleright e \Rightarrow \Delta'}{\Delta \blacktriangleright \lambda x.e \Rightarrow \Delta'} \tag{102}$$

$$\frac{\Delta \blacktriangleright e_1 \Rightarrow \Delta_1 \quad \Delta \blacktriangleright e_2 \Rightarrow \Delta_2}{\Delta \blacktriangleright e_1 e_2 \Rightarrow \Delta_1 \uplus \Delta_2} \tag{103}$$

$$\frac{}{\Delta \blacktriangleright x \Rightarrow \{\}} \tag{104}$$

$$\frac{\Delta \blacktriangleright e \Rightarrow \Delta_1 \quad \Delta \blacktriangleright^{tspec} \tau \Rightarrow \Delta_2}{\Delta \blacktriangleright e : \tau \Rightarrow \Delta_1 \uplus \Delta_2} \tag{105}$$

$$\frac{}{\Delta \blacktriangleright c : \sigma \Rightarrow \{\}} \tag{106}$$

$$\frac{\Delta \blacktriangleright^{tspec} \tau \Rightarrow \Delta'}{\Delta \blacktriangleright c \; \mathbf{ex} \; \tau \Rightarrow \Delta'} \tag{107}$$

$$\frac{}{\Delta \blacktriangleright scon \Rightarrow \{\}} \tag{108}$$

$$\frac{}{\Delta \blacktriangleright \mathbf{\{\}} \Rightarrow \{\}} \tag{109}$$

$$\frac{\forall i \in \{1,\ldots,m\} : \Delta \blacktriangleright e_i \Rightarrow \Delta_i \quad m \geq 1 \quad \Delta \blacktriangleright e \Rightarrow \Delta'}{\Delta \blacktriangleright \{lab_1{=}e_1,\ldots,lab_m{=}e_m,e\} \Rightarrow \Delta_1 \uplus \cdots \uplus \Delta_m \uplus \Delta'} \tag{110}$$

$$\frac{\Delta \blacktriangleright e_1 \Rightarrow \Delta_1 \quad \Delta \blacktriangleright e_2 \Rightarrow \Delta_2}{\Delta \blacktriangleright e_1 \ \mathbf{?}\ e_2 \Rightarrow \Delta_1 \uplus \Delta_2} \tag{111}$$

$$\frac{\Delta \blacktriangleright e \Rightarrow \Delta' \quad n \geq 0}{\Delta \blacktriangleright \mathbf{let}\ p_1 = e_1\ \mathbf{;}\ \cdots\ \mathbf{;}\ p_n = e_n\ \mathbf{in}\ e \Rightarrow \Delta'} \tag{112}$$

$$\frac{\Delta \blacktriangleright e \Rightarrow \Delta' \quad m \geq 1}{\Delta \blacktriangleright \mathbf{letrec}\ p_1 = e_1\ \mathbf{;}\ \cdots\ \mathbf{;}\ p_m = e_m\ \mathbf{in}\ e \Rightarrow \Delta'} \tag{113}$$

$$\frac{\begin{array}{c}\forall i \in \{1,\ldots,m\} : \Delta \blacktriangleright p_i \Rightarrow \Delta_i \quad m \geq 1 \\ \forall i \in \{1,\ldots,m\} : \Delta \blacktriangleright e_i \Rightarrow \Delta'_i \quad \Delta \blacktriangleright e \Rightarrow \Delta' \\ \Delta'' = \Delta_1 \uplus \Delta'_1 \uplus \Delta_2 \uplus \Delta'_2 \uplus \cdots \uplus \Delta_m \uplus \Delta'_m \uplus \Delta'\end{array}}{\Delta \blacktriangleright \mathbf{case}\ e\ \mathbf{of}\ p_1 \rightrightarrows e_1 \parallel \cdots \parallel p_m \rightrightarrows e_m \Rightarrow \Delta''} \tag{114}$$

$$\frac{\begin{array}{c}\alpha \in Dom\ \Delta \quad \xi = \Delta(\alpha) \\ \xi = []absent\{absent\}\ /\ []present\{present\} \\ \Delta \blacktriangleright e \Rightarrow \Delta_1 \quad \Delta \blacktriangleright e_1 \Rightarrow \Delta_2 \\ \Delta \blacktriangleright p \Rightarrow \Delta_3 \quad \Delta \blacktriangleright e_2 \Rightarrow \Delta_4 \\ \Delta \blacktriangleright^{tspec} \tau \Rightarrow \Delta_5 \\ \Delta' = \Delta_1 \uplus \Delta_2 \uplus \Delta_3 \uplus \Delta_4 \uplus \Delta_5\end{array}}{\Delta \blacktriangleright \mathbf{fieldcase}\ e\ \mathbf{in}\ \alpha\ \mathbf{of\ absent} \rightrightarrows e_1 \parallel p \rightrightarrows e_2\ \mathbf{type}\ \tau \Rightarrow \Delta'} \tag{115}$$

$$\frac{\begin{array}{c}\alpha \notin Dom\ \Delta \\ \xi = []absent\{absent\}\ /\ []present\{present\} \\ \Delta \blacktriangleright e \Rightarrow \Delta_1 \quad \Delta \blacktriangleright e_1 \Rightarrow \Delta_2 \\ \Delta \blacktriangleright p \Rightarrow \Delta_3 \quad \Delta \blacktriangleright e_2 \Rightarrow \Delta_4 \\ \Delta \blacktriangleright^{tspec} \tau \Rightarrow \Delta_5 \\ \Delta' = \{\alpha \mapsto \xi\} \uplus \Delta_1 \uplus \Delta_2 \uplus \Delta_3 \uplus \Delta_4 \uplus \Delta_5\end{array}}{\Delta \blacktriangleright \mathbf{fieldcase}\ e\ \mathbf{in}\ \alpha\ \mathbf{of\ absent} \rightrightarrows e_1 \parallel p \rightrightarrows e_2\ \mathbf{type}\ \tau \Rightarrow \Delta'} \tag{116}$$

$$\frac{\forall i \in \{1,\ldots,m\} : \Delta \blacktriangleright^{tspec} \tau_i \Rightarrow \Delta_i \quad \Delta \blacktriangleright e \Rightarrow \Delta' \quad m \geq 1}{\Delta \blacktriangleright \mathbf{letex}\ c_1 : \tau_1\ ,\ \ldots,\ c_m : \tau_m\ \mathbf{in}\ e \Rightarrow \Delta_1 \uplus \Delta_2 \uplus \cdots \uplus \Delta_m \uplus \Delta'} \tag{117}$$

$$\frac{\begin{array}{c}\forall i \in \{1,\ldots,m\} : \Delta \blacktriangleright p_i \Rightarrow \Delta_i \quad m \geq 1 \\ \forall i \in \{1,\ldots,m\} : \Delta \blacktriangleright e_i \Rightarrow \Delta'_i \quad \Delta \blacktriangleright e \Rightarrow \Delta' \\ \Delta'' = \Delta_1 \uplus \Delta'_1 \uplus \Delta_2 \uplus \Delta'_2 \uplus \cdots \uplus \Delta_m \uplus \Delta'_m \uplus \Delta'\end{array}}{\Delta \blacktriangleright e\ \mathbf{handle}\ p_1 \rightrightarrows e_1 \parallel \cdots \parallel p_m \rightrightarrows e_m \Rightarrow \Delta''} \tag{118}$$

$$\frac{\Delta \blacktriangleright e \Rightarrow \Delta'}{\Delta \blacktriangleright \mathbf{raise}\ e \Rightarrow \Delta'} \tag{119}$$

$$\boxed{\Delta \blacktriangleright p \Rightarrow \Delta'}$$

$$\frac{}{\Delta \blacktriangleright x \Rightarrow \{\}} \tag{120}$$

$$\frac{\Delta \blacktriangleright p \Rightarrow \Delta'}{\Delta \blacktriangleright x \ \mathbf{as}\ p \Rightarrow \Delta} \tag{121}$$

$$\frac{\Delta \blacktriangleright p \Rightarrow \Delta_1 \quad \Delta \blacktriangleright^{tspec} \tau \Rightarrow \Delta_2}{\Delta \blacktriangleright p : \tau \Rightarrow \Delta_1 \uplus \Delta_2} \tag{122}$$

$$\frac{}{\Delta \blacktriangleright c : \sigma \Rightarrow \{\}} \tag{123}$$

$$\frac{\Delta \blacktriangleright p \Rightarrow \Delta'}{\Delta \blacktriangleright c(p) : \sigma \Rightarrow \Delta'} \tag{124}$$

$$\frac{}{\Delta \blacktriangleright c\ \mathbf{ex} \Rightarrow \Delta} \tag{125}$$

$$\frac{\Delta \blacktriangleright p \Rightarrow \Delta_1 \quad \Delta \blacktriangleright^{tspec} \tau \Rightarrow \Delta_2}{\Delta \blacktriangleright c(p)\ \mathbf{ex}\ \tau \Rightarrow \Delta_1 \uplus \Delta_2} \tag{126}$$

$$\frac{}{\Delta \blacktriangleright scon \Rightarrow \{\}} \tag{127}$$

$$\frac{}{\Delta \blacktriangleright _ \Rightarrow \{\}} \tag{128}$$

$$\frac{}{\Delta \blacktriangleright \mathbf{\{\}} \Rightarrow \{\}} \tag{129}$$

$$\frac{\forall i \in \{1,\ldots,m\} : \Delta \blacktriangleright p_i \Rightarrow \Delta_i \quad m \geq 1 \quad \Delta \blacktriangleright p \Rightarrow \Delta'}{\Delta \blacktriangleright \mathbf{\{}lab_1\mathbf{=}p_1\mathbf{,}\ldots\mathbf{,}lab_m\mathbf{=}p_m\mathbf{,}p\mathbf{\}} \Rightarrow \Delta_1 \uplus \cdots \uplus \Delta_m \uplus \Delta'} \tag{130}$$

$$\frac{\Delta \blacktriangleright p \Rightarrow \Delta'}{\Delta \blacktriangleright present : \sigma_{pre}\ \mathbf{?}\ p \Rightarrow \Delta'} \tag{131}$$

$\boxed{\Delta \blacktriangleright^{tspec} \tau \Rightarrow \Delta'}$

$$\frac{\beta \notin Dom\ env}{\Delta \blacktriangleright^{tspec} \beta :: \xi \Rightarrow \{\}} \tag{132}$$

$$\frac{\beta \in Dom\ env \quad \tau = env(\beta) \quad \Delta \blacktriangleright^{tspec} \tau \Rightarrow \Delta'}{\Delta \blacktriangleright^{tspec} \beta :: \xi \Rightarrow \Delta'} \tag{133}$$

$$\frac{\alpha \in Dom\ \Delta \quad \xi = \Delta(\alpha)}{\Delta \blacktriangleright^{tspec} \alpha :: \xi \Rightarrow \{\}} \tag{134}$$

$$\frac{\alpha \notin Dom\ \Delta}{\Delta \blacktriangleright^{tspec} \alpha :: \xi \Rightarrow \{\alpha \mapsto \xi\}} \tag{135}$$

$$\frac{\Delta \blacktriangleright^{tspec} \tau \Rightarrow \Delta_1 \quad \Delta \blacktriangleright^{tspec} \tau' \Rightarrow \Delta_2}{\Delta \blacktriangleright^{tspec} \tau \rightarrow \tau' \Rightarrow \Delta_1 \uplus \Delta_2} \tag{136}$$

$$\frac{\Delta \blacktriangleright^{tspec} \tau \Rightarrow \Delta_1 \quad \Delta \blacktriangleright^{tspec} \tau' \Rightarrow \Delta_2}{\Delta \blacktriangleright^{tspec} \tau\ \mathbf{?}\ \tau' \Rightarrow \Delta_1 \uplus \Delta_2} \tag{137}$$

$$\frac{\forall i \in \{1, \ldots, n\} : \Delta \blacktriangleright^{tspec} \tau_i \Rightarrow \Delta_i \quad n, k \geq 0}{\Delta \blacktriangleright^{tspec} [a_1 = \tau_1, \ldots, a_n = \tau_n] d\{c_1, \ldots, c_k\} \Rightarrow \Delta_1 \uplus \cdots \uplus \Delta_n} \tag{138}$$

$$\frac{}{\Delta \blacktriangleright^{tspec} \{\} \Rightarrow []} \tag{139}$$

$$\frac{\forall i \in \{1, \ldots, m\} : \Delta \blacktriangleright^{tspec} \tau_i \Rightarrow \Delta_i \quad m \geq 1 \quad \Delta \blacktriangleright^{tspec} \tau \Rightarrow \Delta'}{\Delta \blacktriangleright^{tspec} \{lab_1\ :\ \tau_1\ ?\ \tau'_1, \ldots, lab_m\ :\ \tau_m\ ?\ \tau'_m; \tau\} \Rightarrow \Delta_1 \uplus \cdots \uplus \Delta_n \uplus \Delta'} \tag{140}$$

Comments on Implicitly Scoped Type Variables

- Rule (101): This rule handles the *value binding* part of a let or letrec expression. This is where the rules for scoped type variables are initiated from.
- Rules (106), (123), (124) and (131): Constructor type schemes do not scope any type variables because the constructors are always quantified as closed types without any free type variables. This is ensured by the closure operation of datatypes in the Naked CeXL to ξ-Calculus translation.
- Rules (112) and (113): Notice that it is exactly at the nested let and letrec expressions that the scoping stops.
- Rules (115) and (116): Notice that there is no rule for when α is present in Δ with a different restriction than $[]absent\{absent\}$ / $[]present\{present\}$ as this would be an error.

  The way we handle the α here is similar to how α is handled in rules (134) and (135) except that ξ must be $[]absent\{absent\}$ / $[]present\{present\}$.
- Rule (134): Notice that both α and ξ must be the same.
- Rule (135): The α must not occur at all in Δ.
- Rules (134) and (135): Notice that there is no rule for when α is present in Δ with a different restriction than the specified one as this would be an error.
- Rule (138): Notice that the variables $a_1, \ldots, a_n$ are not scoped. As mentioned, these are not type variables but only parameter names for the constructor type - even though they denote the same names as type variables do.

  We allow the constructor $[]exn\{\}$ here, so $k \geq 0$.

17 Dynamic Semantics

This is the dynamic semantics which gives the runtime computational behaviour of ξ-Calculus.

17.1 Semantic Objects for Values

Syntax	**Name**		**Semantic Objects**
	$\{FAIL\}$		"Failure"
$addr$	Addr		"Addresses"
sv	SVal		"Special values"
bv	BasVal		"Basic values"
c	ConsId		"Constructor identifiers"
en	ExName		"Exception Names"
v	Val	=	$\{:=\} \cup$ SVal $\cup$ BasVal $\cup$ ConsVal $\cup$ ExVal $\cup$ FieldVal $\cup$ RecordVal $\cup$ Addr $\cup$ FcnClosure
k	ConsVal	=	ConsId $\cup$ (ConsId $\times$ Val)
ex	ExVal	=	ExName $\cup$ (ExName $\times$ Val)
$[ex]$ or p	Pack	=	ExVal
f	FieldVal	=	ConsId $\times$ Val
r	RecordVal	=	Lab $\xrightarrow{fin}$ Val
(x, e, Γ, Γ')	FcnClosure	=	VId $\times$ Expression $\times$ ValEnv $\times$ ValEnv
mem	Mem	=	Addr $\xrightarrow{fin}$ Val
ens	ExNameSet	=	Fin(ExName)
s	State	=	Mem $\times$ ExNameSet
Γ	ValEnv	=	VId $\xrightarrow{fin}$ Val

17.2 Notes About the Semantic Objects

Exceptions are given names (ExName) dynamically at runtime. See the regression test in section 25.4 for an example where this is necessary. The class Pack is used for exceptions which have been raised, so they have a different semantic significance than ExVal.

17.3 Primitive Functions: BasVal and $APPLY$

BasVal represents primitive functions in ξ-Calculus. We represent execution of such a function with the semantic function:

$$APPLY : BasVal \times Val \rightarrow Val \cup Pack$$

17.4 Predefined Semantic Objects

The following constructor identifiers (ConsId) must be predeclared:

```
ref  absent  present
```

The following exception names (ExName) must be predeclared:

```
Match  Bind
```

We don't use Bind but we reserve it, to allow scaling CeXL up to the full Standard ML '97 feature set.

17.5 Function Closures

The informal understanding of a *function closure* (x, e, Γ, Γ') is as follows: When the function closure is applied to a value v, e will be evaluated in the environment Γ modified in a special sense by Γ' and with the binding $x \mapsto v$ added. The domain Dom Γ' contains those identifiers to be treated recursively in the evaluation. To achieve this effect, the evaluation of e will take place not in $\Gamma + \Gamma' + \{x \mapsto v\}$ but in $\Gamma + Rec\ \Gamma' + \{x \mapsto v\}$ where Rec : ValEnv $\rightarrow$ ValEnv is defined as follows:

- Dom(Rec Γ) = Dom(Γ)
- If $\Gamma(x) \notin$ FcnClosure, then (Rec $\Gamma(x)$) = $\Gamma(x)$
- If $\Gamma(x)$ = (x', e, Γ', Γ''), then (Rec $\Gamma(x)$) = (x', e, Γ', Γ)

The effect is that, before application of (x, e, Γ, Γ') to v, the function closures in Ran Γ are "unrolled" once, to prepare for their possible recursive application during the evaluation of e.

This device is adopted to ensure that all semantic objects are finite (by controlling the unrolling of recursion). The operator Rec is invoked in just two places in the semantic rules: In the rule for letrec and in the rule for evaluating an application expression *exp atexp* in the case that *exp* evaluates to a function closure.

17.6 Conventions for the Inferences Rules

The semantic rules allow sentences of the form

$$s; A \vdash phrase \Rightarrow A'; s'$$

to be inferred, where A is usually an environment, A' is some semantic object and s, s' are the states before and after the evaluation represented by

the sentence. Some hypotheses in rules are not of this form; they are called *side-conditions*.

In most rules the states s and s' are omitted from the sentences; they are only included for those rules which are directly concerned with the state - either referring to its contents or changing it. When omitted, the convention for restoring them is as follows. If the rule is presented in the form

$$\frac{A_1 \vdash phrase_1 \Rightarrow A_1' \quad A_2 \vdash phrase_2 \Rightarrow A_2' \quad \cdots \quad \cdots \quad A_n \vdash phrase_n \Rightarrow A_n'}{A \vdash phrase \Rightarrow A'}$$

then the full form is intended to be

$$\frac{s_0; A_1 \vdash phrase_1 \Rightarrow A_1'; s_1 \quad s_1; A_2 \vdash phrase_2 \Rightarrow A_2'; s_2 \quad \cdots \quad \cdots \quad s_{n-1}; A_n \vdash phrase_n \Rightarrow A_n'; s_n}{s_0; A \vdash phrase \Rightarrow A'; s_n}$$

(Any side-conditions are left unaltered). Thus the left-to-right order of the hypotheses indicate the order of evaluation. Note that in the case $n = 0$, when there are no hypotheses (except possibly side-conditions), we have $s_0 = s_n$; this implies that the rule causes no side effect. The convention is called the *state-convention*, and must be applied to each version of a rule obtained by inclusion or omission of its options.

A second convention, the *exception convention*, is adopted to deal with the propagation of exception packets p. For each rule whose full form (ignoring side conditions) is

$$\frac{s_1; A_1 \vdash phrase_1 \Rightarrow A_1'; s_1' \quad \cdots \quad s_n; A_n \vdash phrase_n \Rightarrow A_n'; s_n'}{s_1; A_1 \vdash phrase \Rightarrow A'; s'}$$

and for each k where $1 \leq k \leq n$, for which the result A_k' is not a packet p, an extra rule is added of the form

$$\frac{s_1; A_1 \vdash phrase_1 \Rightarrow A_1'; s_1' \quad \cdots \quad s_k; A_k \vdash phrase_k \Rightarrow p'; s'}{s_1; A_1 \vdash phrase \Rightarrow p'; s'}$$

where p' does not occur in the original rule. There is one exception to the exception convention though. No extra rule is added for rule (163) which deals with exception handling, since an exception handler is the only means by which propagation of an exception can be arrested.

The exception convention indicates that evaluation of phrases in the hypothesis terminates with the first phrase whose result is a packet (other than one already treated in the rule), and this packet is the result of the phrase in the conclusion.

A third convention is that we allow compound variables (variables built from the variables used to represent semantic objects and the symbol "/") to range over unions of semantic objects. For instance the compound variable v/p ranges over Val $\cup$ Pack. We also allow $x/FAIL$ to range over $X \cup \{FAIL\}$ where x ranges over X.

17.7 Judgement Forms

$s; \Gamma \vdash e \Rightarrow v; s'$	Evaluate value v of expression e in environments Γ
$s; \Gamma; v \vdash p \Rightarrow \Gamma'/FAIL; s'$	Match pattern p against the value v and produce environment Γ' or return $FAIL$
$s; r \vdash^{\#} lab \Rightarrow v; r'; s'$	Extract value v at label lab from record r and produce record r' with the field lab removed

17.8 Inference Rules

Expressions

$$\boxed{\Gamma \vdash e \Rightarrow v}$$

$$\frac{}{\Gamma \vdash \lambda x.e \Rightarrow (x, e, \Gamma, \{\})} \tag{141}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow (x, e, \Gamma', \Gamma'') \quad \Gamma \vdash e_2 \Rightarrow v \quad \Gamma' + Rec\ \Gamma'' + \{x \mapsto v\} \vdash e \Rightarrow v'}{\Gamma \vdash e_1\ e_2 \Rightarrow v'} \tag{142}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow c \quad \Gamma \vdash e_2 \Rightarrow v \quad c \neq ref}{\Gamma \vdash e_1\ e_2 \Rightarrow (c, v)} \tag{143}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow en \quad \Gamma \vdash e_2 \Rightarrow v}{\Gamma \vdash e_1\ e_2 \Rightarrow (en, v)} \tag{144}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow bv \quad \Gamma \vdash e_2 \Rightarrow v \quad \text{APPLY}(bv,\ v) = v'/p}{\Gamma \vdash e_1\ e_2 \Rightarrow v'/p} \tag{145}$$

$$\frac{s; \Gamma \vdash e_1 \Rightarrow ref; s' \quad s'; \Gamma \vdash e_2 \Rightarrow v; s'' \quad s'' = (mem, ens) \quad addr \notin Dom\ mem}{s; \Gamma \vdash e_1\ e_2 \Rightarrow addr; (mem + \{addr \mapsto v\}, ens)} \tag{146}$$

$$\frac{s; \Gamma \vdash e_1 \Rightarrow :=; s' \quad s'; \Gamma \vdash e_2 \Rightarrow \{1 \mapsto addr, 2 \mapsto v\}; s'' \quad s'' = (mem, ens)}{s; \Gamma \vdash e_1\ e_2 \Rightarrow \{\}; (mem + \{addr \mapsto v\}, ens)} \tag{147}$$

$$\frac{x \in Dom\ \Gamma \quad v = \Gamma(x)}{\Gamma \vdash x \Rightarrow v} \tag{148}$$

$$\frac{\Gamma \vdash e \Rightarrow v}{\Gamma \vdash e : \tau \Rightarrow v} \tag{149}$$

$$\frac{}{\Gamma \vdash c : \sigma \Rightarrow c} \tag{150}$$

$$\frac{\Gamma(c) = en}{\Gamma \vdash c\ \mathbf{ex}\ \tau \Rightarrow en} \tag{151}$$

$$\frac{}{\Gamma \vdash scon \Rightarrow val(scon)} \tag{152}$$

$$\frac{}{\Gamma \vdash \mathbf{\{\}} \Rightarrow \{\}} \tag{153}$$

$$\frac{\forall i \in \{1,\ldots,m\} : \Gamma \vdash e_i \Rightarrow v_i \quad \Gamma \vdash e \Rightarrow r \quad \{v'_{i_1},\ldots,v'_{i_n}\} = \{v'_i \mid v_i = (present, v'_i)\}}{\Gamma \vdash \{lab_1 = e_1,\ \ldots,\ lab_m = e_m,\ e\} \Rightarrow \{lab_{i_1} \mapsto v'_{i_1},\ldots,lab_{i_n} \mapsto v'_{i_n}\} + r} \tag{154}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow c \quad \Gamma \vdash e_2 \Rightarrow v \quad f = (c, v)}{\Gamma \vdash e_1 \ ?\ e_2 \Rightarrow f} \tag{155}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 \Rightarrow v_1 \quad \Gamma; v_1 \vdash p_1 \Rightarrow \Gamma_1 \\ \Gamma \vdash e_2 \Rightarrow v_2 \quad \Gamma; v_2 \vdash p_2 \Rightarrow \Gamma_2 \\ \vdots \\ \Gamma \vdash e_n \Rightarrow v_n \quad \Gamma; v_n \vdash p_n \Rightarrow \Gamma_n \\ \Gamma + \Gamma_1 + \cdots + \Gamma_n \vdash e \Rightarrow v\end{array}}{\Gamma \vdash \textbf{let}\ p_1 = e_1\ ;\ \cdots\ ;\ p_n = e_n\ \textbf{in}\ e \Rightarrow v} \tag{156}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 \Rightarrow v_1 \quad \Gamma; v_1 \vdash p_1 \Rightarrow \Gamma_1 \\ \Gamma \vdash e_2 \Rightarrow v_2 \quad \Gamma; v_2 \vdash p_2 \Rightarrow \Gamma_2 \\ \vdots \\ \Gamma \vdash e_n \Rightarrow v_n \quad \Gamma; v_n \vdash p_n \Rightarrow \Gamma_n \\ \Gamma + Rec(\Gamma_1 + \cdots + \Gamma_n) \vdash e \Rightarrow v\end{array}}{\Gamma \vdash \textbf{letrec}\ p_1 = e_1\ ;\ \cdots\ ;\ p_n = e_n\ \textbf{in}\ e \Rightarrow v} \tag{157}$$

$$\frac{\begin{array}{c}\Gamma \vdash e \Rightarrow v \\ i \in \{1,\ldots,m\} \\ \Gamma; v \vdash p_1 \Rightarrow FAIL \\ \vdots \\ \Gamma; v \vdash p_{i-1} \Rightarrow FAIL \\ \Gamma; v \vdash p_i \Rightarrow \Gamma' \quad \Gamma + \Gamma' \vdash e_i \Rightarrow v'\end{array}}{\Gamma \vdash \textbf{case}\ e\ \textbf{of}\ p_1 \rightrightarrows e_1 \parallel \cdots \parallel p_m \rightrightarrows e_m \Rightarrow v'} \tag{158}$$

$$\frac{\Gamma \vdash e \Rightarrow v \quad \forall i \in \{1,\ldots,m\} : \Gamma; v \vdash p_i \Rightarrow FAIL}{\Gamma \vdash \textbf{case}\ e\ \textbf{of}\ p_1 \rightrightarrows e_1 \parallel \cdots \parallel p_m \rightrightarrows e_m \Rightarrow [Match]} \tag{159}$$

$$\frac{\Gamma \vdash e \Rightarrow f \quad f = (absent, v) \quad \Gamma \vdash e_1 \Rightarrow v'}{\Gamma \vdash \textbf{fieldcase}\ e\ \textbf{in}\ \alpha\ \textbf{of absent} \rightrightarrows e_1 \parallel \cdots \parallel \textbf{present}\ p \rightrightarrows e_2\ \textbf{type}\ \tau \Rightarrow v'} \tag{160}$$

$$\frac{\Gamma \vdash e \Rightarrow f \quad f = (present, v) \quad \Gamma; v \vdash p \Rightarrow \Gamma' \quad \Gamma + \Gamma' \vdash e_2 \Rightarrow v'}{\Gamma \vdash \textbf{fieldcase}\ e\ \textbf{in}\ \alpha\ \textbf{of absent} \rightrightarrows e_1 \parallel \cdots \parallel \textbf{present}\ p \rightrightarrows e_2\ \textbf{type}\ \tau \Rightarrow v'} \tag{161}$$

$$\frac{s = (mem, ens) \quad \forall i \in \{1,\ldots,m\} : en_i \notin Dom\ ens \quad \forall i,j \in \{1,\ldots,m\}, i \neq j : en_i \neq en_j \quad m \geq 1 \quad s' = (mem, ens + \{en_1,\ldots,en_m\}) \quad s'; \Gamma + \{c_1 \mapsto en_1,\ldots,c_m \mapsto en_m\} \vdash e \Rightarrow v; s''}{s; \Gamma \vdash \textbf{letex}\ c_1 : \tau_1,\ \ldots,\ c_m : \tau_m \textbf{in}\ e \Rightarrow v; s''} \tag{162}$$

$$\frac{\Gamma \vdash e \Rightarrow v}{\Gamma \vdash e \ \mathbf{handle}\ p_1 \rightrightarrows\ e_1 \parallel \cdots \parallel\ p_m \rightrightarrows\ e_m \Rightarrow v} \tag{163}$$

$$\frac{\begin{array}{c}\Gamma \vdash e \Rightarrow [ex] \\ i \in \{1, \ldots, m\} \\ \Gamma; ex \vdash p_1 \Rightarrow FAIL \\ \vdots \\ \Gamma; ex \vdash p_{i-1} \Rightarrow FAIL \\ \Gamma; ex \vdash p_i \Rightarrow \Gamma' \quad \Gamma + \Gamma' \vdash e_i \Rightarrow v\end{array}}{\Gamma \vdash e \ \mathbf{handle}\ p_1 \rightrightarrows\ e_1 \parallel \cdots \parallel\ p_m \rightrightarrows\ e_m \Rightarrow v} \tag{164}$$

$$\frac{\begin{array}{c}\Gamma \vdash e \Rightarrow [ex] \\ \forall i \in \{1, \ldots, m\} : \Gamma; ex \vdash p_i \Rightarrow FAIL\end{array}}{\Gamma \vdash e \ \mathbf{handle}\ p_1 \rightrightarrows\ e_1 \parallel \cdots \parallel\ p_m \rightrightarrows\ e_m \Rightarrow [ex]} \tag{165}$$

$$\frac{\Gamma \vdash e \Rightarrow ex}{\Gamma \vdash \mathbf{raise}\ e \Rightarrow [ex]} \tag{166}$$

Patterns

$\boxed{\Gamma; v \vdash p \Rightarrow \Gamma' \ / \ \text{FAIL}}$

$$\frac{}{\Gamma; v \vdash x \Rightarrow \{x \mapsto v\}} \tag{167}$$

$$\frac{\Gamma; v \vdash p \Rightarrow \Gamma'/FAIL}{\Gamma; v \vdash p : \tau \Rightarrow \Gamma'/FAIL} \tag{168}$$

$$\frac{\Gamma; v \vdash p \Rightarrow \Gamma'/FAIL}{\Gamma; v \vdash x \textbf{ as } p \Rightarrow \{x \mapsto v\} + \Gamma'/FAIL} \tag{169}$$

$$\frac{v = c}{\Gamma; v \vdash c : \sigma \Rightarrow \{\}} \tag{170}$$

$$\frac{v \neq c}{\Gamma; v \vdash c : \sigma \Rightarrow FAIL} \tag{171}$$

$$\frac{c \neq ref \quad v = (c, v') \quad \Gamma; v' \vdash p \Rightarrow \Gamma'/FAIL}{\Gamma; v \vdash c(p) : \sigma \Rightarrow \Gamma'/FAIL} \tag{172}$$

$$\frac{c \neq ref \quad v \neq (c, v')}{\Gamma; v \vdash c(p) : \sigma \Rightarrow FAIL} \tag{173}$$

$$\frac{s = (mem, ens) \quad mem(addr) = v \quad s; \Gamma; v \vdash p \Rightarrow \Gamma'/FAIL; s}{s; \Gamma; addr \vdash ref(p) : \sigma \Rightarrow \Gamma'/FAIL; s} \tag{174}$$

$$\frac{\Gamma(c) = en \quad v = en}{\Gamma; v \vdash c \textbf{ ex} \Rightarrow \{\}} \tag{175}$$

$$\frac{\Gamma(c) = en \quad v \neq en}{\Gamma; v \vdash c \textbf{ ex} \Rightarrow FAIL} \tag{176}$$

$$\frac{\Gamma(c) = en \quad v = (en, v') \quad v' \vdash \Gamma; p \Rightarrow \Gamma'/FAIL}{\Gamma; v \vdash c(p) \textbf{ ex } \tau \Rightarrow \Gamma'/FAIL} \tag{177}$$

$$\frac{\Gamma(c) = en \quad v \neq (en, v')}{\Gamma; v \vdash c(p) \textbf{ ex } \tau \Rightarrow FAIL} \tag{178}$$

$$\frac{v = val(scon)}{\Gamma; v \vdash scon \Rightarrow \{\}} \tag{179}$$

$$\frac{v \neq val(scon)}{\Gamma; v \vdash scon \Rightarrow FAIL} \tag{180}$$

$$\frac{}{\Gamma; v \vdash _ \Rightarrow \{\}} \tag{181}$$

$$\frac{}{\Gamma; r \vdash \textbf{\{\}} \Rightarrow \{\}} \tag{182}$$

$$\frac{\begin{array}{c} r_1 \vdash^{\#} lab_1 \Rightarrow v_1; r_2 \quad \Gamma; v_1 \vdash p_1 \Rightarrow \Gamma_1 \\ \vdots \\ r_m \vdash^{\#} lab_m \Rightarrow v_m; r_{m+1} \quad \Gamma; v_m \vdash p_m \Rightarrow \Gamma_m \\ \Gamma; r_{m+1} \vdash p \Rightarrow \Gamma_{m+1} \end{array}}{\Gamma; r_1 \vdash \{lab_1 = p_1, \ldots, lab_m = p_m,\ p\} \Rightarrow \Gamma_1 + \cdots + \Gamma_m + \Gamma_{m+1}} \tag{183}$$

$$\frac{\begin{array}{c} i \in \{1, \ldots, m\} \\ r_1 \vdash^{\#} lab_1 \Rightarrow v_1; r_2 \quad \Gamma; v_1 \vdash p_1 \Rightarrow \Gamma_1 \\ \vdots \\ r_{i-1} \vdash^{\#} lab_{i-1} \Rightarrow v_{i-1}; r_i \quad \Gamma; v_{i-1} \vdash p_{i-1} \Rightarrow \Gamma_{i-1} \\ r_i \vdash^{\#} lab_i \Rightarrow v_i; r_{i+1} \quad \Gamma; v_i \vdash p_i \Rightarrow FAIL \end{array}}{\Gamma; r_1 \vdash \{lab_1 = p_1, \ldots, lab_m = p_m,\ p\} \Rightarrow FAIL} \tag{184}$$

$$\frac{\begin{array}{c} r_1 \vdash^{\#} lab_1 \Rightarrow v_1; r_2 \quad \Gamma; v_1 \vdash p_1 \Rightarrow \Gamma_1 \\ \vdots \\ r_m \vdash^{\#} lab_m \Rightarrow v_m; r_{m+1} \quad \Gamma; v_m \vdash p_m \Rightarrow \Gamma_m \\ \Gamma; r_{m+1} \vdash p \Rightarrow FAIL \end{array}}{\Gamma; r_1 \vdash \{lab_1 = p_1, \ldots, lab_m = p_m,\ p\} \Rightarrow FAIL} \tag{185}$$

$$\frac{v = f = (present, v') \quad \Gamma; v' \vdash p \Rightarrow \Gamma'/FAIL}{\Gamma; v \vdash present : \sigma_{pre}\ ?\ p \Rightarrow \Gamma'/FAIL} \tag{186}$$

Record Field Extraction and Removal

$\boxed{r \vdash^{\#} lab \Rightarrow v; r'}$

$$\frac{\begin{array}{c} lab \in Dom\ r \quad r' = r \backslash \{lab \mapsto \cdot\} \\ v = r(lab) \quad f = (present, v) \end{array}}{r \vdash^{\#} lab \Rightarrow f; r'} \tag{187}$$

$$\frac{lab \notin Dom\ r \quad f = (absent, \{\})}{r \vdash^{\#} lab \Rightarrow f; r} \tag{188}$$

17.9 Comments on Dynamic Semantics

- Rule (146): $addr \notin$ Dom mem indicates that $addr$ is a fresh memory address.
- Rules (151), (162), (175), (176), (177) and (178): The Naked CeXL to ξ-Calculus translation ensures that storing the map from constructor names to exception names in the value environment does not give any conflicts.
- Rule (154): Claimed theorem: e always evaluates to a record value r not containing any of the labels $lab_{i_1} \cdots lab_{i_n}$. Also notice that these are the only fields we add - i.e. those with present field values.
- Rule (162): $en_i \notin$ Dom ens indicates that en_i are fresh exception names. $en_i \neq en_j$ indicates that they are also all different.

- Rule (186): The static semantics will ensure that v actually is a present field value at this point.
- Rule (188): We just put an arbitrary value (here $\{\}$) in the absent field value. It's never going to be used.

18 Grammar of Naked CeXL

This section presents the grammar of Naked CeXL - the language with the full power of the CeXL language but without syntactic sugar. The lexical details of how identifiers and program constants look are given later, when we present the full syntax.

18.1 Notational Conventions

The following conventions are used:

- The brackets $\langle\ \rangle$ encose optional phrases.
- For any class X (over which x ranges) we define the syntax class Xseq (over which xseq ranges) as follows:

xseq	::=	$(x_1, \cdots, x_n)$	(sequence, n $\geq$ 1)
			(empty sequence)
		x	(singleton sequence)

  (Note that the "$\cdots$" used here, a meta-symbol indicating syntactic repetition, must not be confused with "..." which is a reserved word of the language.)

- Alternative forms for each phrase class are in order of decreasing precedence. This precedence resolves ambiguity in parsing as explained in section 21 which contains the full CeXL grammar.
- L (respectively R) means left (respectively right) association.
- The syntax of types binds more tightly than that of expressions.
- The syntax of restrictions binds more tightly than that of types.
- Each iterated construct (e.g. *match* ...) extends as far right as possible; thus, parentheses may be needed around an expression which terminates with a match, e.g. "fn *match*", if this occurs within a larger match.

18.2 Grammar Productions

Patterns

atpat	::=	_	wildcard
		scon	special constant
		longvid	value identifier
		{ ⟨ *patrow* ⟩ **}**	record
		(*pat* **)**	
pat	::=	*atpat*	atomic
		longvid atpat	constructed pattern
		pat **:** *ty*	typed
		vid ⟨ **:** *ty* ⟩ **as** *pat*	layered
patrow	::=	*lab* **=** *pat* ⟨ **,** *patrow* ⟩	present field
		lab `?=` *pat* ⟨ **,** *patrow* ⟩	optional field
		... ⟨ **=** *pat* ⟩	optional at end of row

Expressions and Matches

atexp	::=	*scon*	special constant
		longvid	value identifier
		{ ⟨ *exprow* ⟩ **}**	record
		let ⟨ *decs* ⟩ **in** *exp* **end**	local declaration
		fieldcase *exp* **in** *tyvar* **of** *fieldmatch* **type** *ty* **end**	case on optional field
		(*exp* **)**	
exp	::=	*atexp*	atomic
		exp atexp	application (L)
		exp **:** *ty*	typed (L)
		exp **handle** *match*	handle exception
		raise *exp*	raise exception
		case *exp* **of** *match*	case
		fn *match*	function
exprow	::=	*lab* **=** *exp* ⟨ **,** *exprow* ⟩	present field
		lab `?=` *exp* ⟨ **,** *exprow* ⟩	optional field
		... **=** *exp*	optional at end of row
fieldmatch	::=	`absent =>` exp_1 `| present` *atpat* `=>` exp_2	match optional field
match	::=	*mrule* ⟨ `|` *match* ⟩	
mrule	::=	*pat* `=>` *exp*	

Declarations

decs	::=	*dec* ⟨ **;** ⟩ ⟨ *decs* ⟩	sequential declaration
dec	::=	**val** *typms valbind*	value declaration
		val *typms* **rec** *valbind*	recursive value declaration
		res *vid* ⟨ **=** *crestr* ⟩	restriction declaration
		type *typbind*	type declaration
		datatype *datbind*	datatype declaration
		datatype *tycon* **=** **datatype** *longtycon*	datatype replication
		exception *exbind*	exception declaration
valbind	::=	*pat* **=** *exp* ⟨ **and** *valbind* ⟩	
typbind	::=	*typms tycon* **=** *ty* ⟨ **and** *typbind* ⟩	
datbind	::=	*typms tycon* **=** *conbind* ⟨ **and** *datbind* ⟩	
conbind	::=	*vid* ⟨ **of** *ty* ⟩ ⟨ **|** *conbind* ⟩	
exbind	::=	*vid* ⟨ **of** *ty* ⟩ ⟨ **and** *exbind* ⟩	
		vid **=** *longvid* ⟨ **and** *exbind* ⟩	

Simple Nested Structures

program	::=	*strdecs*	a CeXL program
strdecs	::=	*strdec* ⟨ **;** ⟩ ⟨ *strdecs* ⟩	sequential structure declaration
strdec	::=	*dec*	declaration
		structure *strbind*	structure
strbind	::=	*strid* **=** **struct** ⟨ *strdecs* ⟩ **end**	structure binding
		strid **=** *longstrid*	structure replication

Type Parameters

typms	::=	⟨ [*typarams*] ⟩	possibly restricted type parameters
typarams	::=	*tyvars* ⟨ **:** *crestr* ⟩ ⟨ **,** *typarams* ⟩	type parameters
tyvars	::=	*tyvar* ⟨ **;** *tyvars* ⟩	type variables

Types

tyargs	::=	*tyvar* **=** *ty* ⟨ **,** *tyargs* ⟩	type arguments
fieldstatus	::=	**-**	absent field
		tyvar	optional field
			present field
tyrow	::=	*fieldstatus lab* **:** *ty* ⟨ **,** *tyrow* ⟩	row type
		... **:** *tyvar*	optional at end of row
ty	::=	*tyvar*	type variable
		{ ⟨ *tyrow* ⟩ **}**	record type
		⟨ [*tyargs*] ⟩ *longtycon*	type construction
		ty `->` *ty'*	function type expression (R)
		(*ty* **)**	

Restrictions

crestr	::=	*restr* ⟨ `+` *crestr* ⟩	combination of restrictions
restr	::=	∼**{** ⟨ *labels* ⟩ **}**	forbidden fields
		typats	type patterns
		res *longvid*	use declared restriction
labels	::=	*lab* ⟨ **,** *labels* ⟩	set of labels
typats	::=	*typat* ⟨ `|` *typats* ⟩	type patterns
typat	::=	⟨ [*typatargs*] ⟩ *longtycon*	type constructor pattern
typatargs	::=	*tyvar* **=** *typat* ⟨ **,** *typatargs* ⟩	type pattern arguments

18.3 Syntactic Limitations

- No expression row, pattern row or type-expression row may bind the same *lab* twice.
- No restriction *labels* may contain the same *lab* twice.
- No binding *valbind*, *typbind*, *datbind* or *exbind* may bind the same identifier twice; this also applies to value identifiers within a *datbind*.
- No *tyvarseq* may contain the same *tyvar* twice.
- No *typarams*, *tyargs* or *typatargs* may bind the same *tyvar* twice at the same nesting level. That is, a *tyvar* occuring nested is unrelated to any other *tyvar*, so the limitation only applies to *tyvar*s immediately in the same *typarams*, *tyargs* or *typatargs*.
- For each value binding *pat* = *exp* within **val rec**, *exp* must be of the form **fn** *match*. The derived form of function-value binding given in section 22 necessarily obeys this restriction.
- No *datbind*, *valbind* or *exbind* may bind `true`, `false`, `nil`, `ref`, `div`, `mod`, `o`, `absent` or `present`. No *datbind* or *exbind* may bind `it`.
- No *datbind* or *typbind* may bind the type constructor `?`.
- No *exbind* may bind `Match` or `Bind`.
- No *exbind* which occur directly at top-level or in a structure may contain type variables. Hence, only *exbind* within the *decs* part of a let-expression may contain type variables.
- No *datbind* or *typbind* may refer to type variables other than those mentioned in the parameter list *typarams* in the beginning of the *datbind* or *typbind*.
- No real constant may occur in a pattern.
- The infix identifier `=` may not occur in a pattern.
- Any *tyrow* in a record type containing **...** **:** *tyvar* at the end must also contain at least one field.
- Any *patrow* in a record pattern containing **...** ⟨ = *pat* ⟩ at the end must also contain at least one field.
- Any *exprow* in a record expression containing **...** = *exp* at the end must also contain at least one field.

18.4 Comments on the Grammar

Some of the reasons for why the grammar looks as it does were already given in section 5.11.

19 Naked CeXL To ξ-Calculus Translation

This translation is mostly syntactial. However, some semantic operations are also done, e.g. taking the closure in datatypes.

Additional Syntax and Semantic Objects for Translation

		Syntax	**Name**		**Semantic Objects**
θ	::=	$\Lambda[\alpha_1 :: \xi_1, \ldots, \alpha_n :: \xi_n].\tau$	TyFun	=	$\cup_{p \geq 0}$ (TyVar $\times$ Restrict) $^p \times$ Type
ζ	::=	$\{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}$	TyArgs	=	TyVar $\xrightarrow{fin}$ Type
Δ	::=	$\{\alpha_1 \mapsto \xi_1, \ldots, \alpha_n \mapsto \xi_n\}$	TyVarRes	=	TyVar $\xrightarrow{fin}$ Restrict
is	::=	$\mathbf{e} \mid \mathbf{c} \mid \mathbf{v}$	IdStatus	=	$\emptyset \cup \emptyset \cup \emptyset$
VE	::=	$\{longvid_1 \mapsto (\sigma_1, is_1, vid_1), \ldots,$ $longvid_n \mapsto (\sigma_n, is_n, vid_n)\}$	ValEnv	=	LongVId $\xrightarrow{fin}$ TyScheme $\times$ IdStatus $\times$ VId
RE	::=	$\{longvid_1 \mapsto \xi_1, \ldots,$ $longvid_n \mapsto \xi_n\}$	ResEnv	=	LongVId $\xrightarrow{fin}$ Restrict
TE	::=	$\{longtycon_1 \mapsto \theta_1, \ldots,$ $longtycon_n \mapsto \theta_n\}$	TyEnv	=	LongTyCon $\xrightarrow{fin}$ TyFun
N	::=	$\{vid_1, \ldots, vid_n\}$	CNames	=	Fin(VId)
NN	::=	$\{vid_1 \mapsto vid'_1, \ldots, vid_n \mapsto vid'_n\}$	NameMap	=	VId $\xrightarrow{fin}$ VId
ν	::=	$\{d_1, \ldots, d_n\}$	TyNames	=	Fin(TyName)
E	::=	(VE, RE, TE, Δ)	Env	=	ValEnv $\times$ ResEnv $\times$ TyEnv $\times$ TyVarRes
		$(N, \nu, prefix)$	Global	=	CNames $\times$ TyNames $\times$ LongVId

19.1 About the Translation

In the translation rules we place parentheses as needed in the ξ-Calculus code for syntactical grouping. We write side conditions at the side in a separate column. This translation will not declare any types or datatypes in the translated code. Types and datatypes will instead be placed in the resulting ξ-Calculus program as part of the syntax as they are needed. Exceptions are delcared in ξ-Calculus though, but they are still also placed explicitly in the ξ-Calculus program where they are used.

We also maintain a global state during the translation. This is the semantic object Global. It is used for generating new unique names for datatype constructors and unique names for the types of datatypes. We do this by updating the global state as needed during the translation. For this we use $fresh(N) = vid$ for choosing a $vid \notin Dom\ N$ and subsequently adding $\{vid\}$ to N to prevent it from being chosen again. We do the same with $fresh(\nu) = vid$.

19.2 The Prefix Variable

The global state also contains the variable *prefix*. During translation, *prefix* contains the name of the (possibly nested) structure which is currently being translated. We use this for prefixing the exception names and variable names to be used in the generated ξ-Calculus code, so as to avoid name clashes. We add this prefix to a variable name *vid* by writing *prefix* + *vid*. We also use similar notation in other places for concatenating variable names.

The *prefix* variable is explicitly passed along in the translation functions for translating structure declarations. This shows how the variable is maintained during translation.

When packing structures into the environment, we need to be able to remove the outer structure identifier from the variable *prefix*. We do this by defining a the following function *inn* accodring to the following rules:

prefix		*inn(prefix)*	
$strid_1.$	$\mapsto$		single identifier becomes nothing
$strid_1.strid_2.\cdots.strid_n.$	$\mapsto$	$strid_2.\cdots.strid_n.$	sequence of identifiers, $n \geq 2$ (notice the extra period after $strid_n$)

19.3 Application of Type Functions: $\zeta\theta$

We write application of a type function θ to an argument ζ of types as $\zeta\theta$. If $\theta = \Lambda[\alpha_1 :: \xi_1, \ldots, \alpha_n :: \xi_n].\tau$ and $\zeta = \{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}$ we set

$$\zeta\theta = \tau\lceil\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n\rceil$$

understood as the usual rules for substitution (β-conversion). As usual, the variables in ConsType and TyPat may not be substituted, since they are just parameter names for the constructors.

The order of the arguments in ζ is irrelevant here, but the names of the type variables must match between the type function and the arguments. This is what gives CeXL the feature *Named Type Parameters*. Furthermore, the types $\tau_1, \ldots, \tau_n$ must respect the restrictions $\xi_1, \ldots, \xi_n$ respectively. This can be done by unifying them with a fresh free type respecting this restriction. E.g. for τ_i we can create the fresh free type τ_i' and restrict it with $\tau_i'|_{\xi_i}$ and unify τ_i with τ_i'.

During this substitution, we also allow some of the arguments to be missing. This is what gives CeXL the feature *Optional Type Arguments*. I.e. we only require Dom $\zeta \subseteq$ Dom θ (where Dom θ is understood as the TyVars in the parameter list). For all missing arguments, we create fresh types with the restriction of their parameters in θ. These types are used for further unification, so it will really be fresh meta variables. So as an example, assume that $\theta = \Lambda[\alpha_1 :: \xi_1, \ldots, \alpha_n :: \xi_n].\tau$ and $\zeta = \{\alpha_1 \mapsto \tau_1, \ldots, \alpha_i \mapsto \tau_i\}$ with $i \leq n$. For the type function application $\zeta\theta$, we create fresh $\tau_{i+1}, \ldots, \tau_n$ and restrict them with $\tau_{i+1}|_{\xi_{i+1}}, \ldots, \tau_n|_{\xi_n}$. These $\tau_{i+1}, \ldots, \tau_n$ are thus free for further unification.

$\zeta\theta$ still becomes:

$$\zeta\theta = \tau\lceil\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n\rceil$$

19.4 No Free Types: $\rhd\tau\lhd$

When translating declarations of types, datatypes and exceptions, we need to make sure that a declared type τ do not contain any free types. We represent this condition by the notation: $\rhd\tau\lhd$. In an implemenation based on the unification algorithm, it corresponds to checking that τ do not cantain any meta variables. This condition does not prevent τ from containing type variables though. This check is required because of the CeXL feature *Partial Type Instantiation*.

19.5 Combination of Restrictions: $\bowtie$

When translating combinations of restrictions *crestr*, we need to be able to combine restrictions. We do this by defining the operator $\bowtie$ on restrictions by these rules:

$$
\begin{array}{lcll}
\xi & \bowtie & \circ & \Rightarrow \quad \xi \\
\circ & \bowtie & \xi & \Rightarrow \quad \xi \\
\omega & \bowtie & \omega' & \Rightarrow \quad \omega \cup \omega' \\
\psi_1/\cdots/\psi_m & \bowtie & \psi'_1/\cdots/\psi'_k & \Rightarrow \quad \psi''_1/\cdots/\psi''_q \text{ where } q \geq 1 \text{ and} \\
 & & & \quad \{\psi''_1,\ldots,\psi''_q\} = \{\psi_1,\ldots,\psi_m\} \cap \{\psi'_1,\ldots,\psi'_k\}
\end{array}
$$

19.6 Predefined Semantic Objects

We need the predefined semantic objects that we used in ξ-Calculus. We also have a few additional predefined objects.

We need the following constructor types (ConsType) to be predefined:

$[]absent\{absent\}$	signifies an absent record field
$[]present\{present\}$	signifies a present record field
$[a = \cdot]ref\{ref\}$	special constructor for supporting references
$[]exn\{\}$	special constructor only for exceptions

We also need the following type schemes (TyScheme) to be predefined:

σ_{unit}	$=$	$\forall[].\{\}$	for the type unit
σ_{abs}	$=$	$\forall[].[]absent\{absent\}$	for constructor for absent record field
σ_{pre}	$=$	$\forall[].[]present\{present\}$	for constructor for present record field
σ_{ref}	$=$	$\forall[\alpha :: \circ].\alpha :: \circ \rightarrow [a = \alpha :: \circ]ref\{ref\}$	for constructor for references

σ_{unit} is used in the translation as a dummy type scheme for variables stored in the value environment VE. It is because we don't need the types of variables in this translation. We only need the types of constructors and exceptions.

We need to prevent certain constructor names from being redeclared. This is done by putting them in the global N before starting the translation. So when starting the translation we define:

$$N = \{true, false, nil, ::, ref, absent, present\}.$$

Similarly we need to prevent certain type names from being redeclared. When starting the translation we thus define:

$$\nu = \{bool, int, real, word, string, char, list, ref, absent, present, exn, \texttt{?}\}.$$

Finally, when the infix operator `:=` occurs it must be represented as such for the semantics in ξ-Calculus to work as intended.

19.7 Translation Functions

The translation is defined as translation functions. They translate syntactically from Naked CeXL syntax to ξ-Calculus syntax. We use the translation functions below. The [and] around the ξ-Calculus syntax emphasizes that the resulting ξ-Calculus syntax may not always form complete valid syntactical ξ-Calculus entities. This happens where the Naked CeXL grammar divides its syntax into a finer granularity of entities than the ξ-Calculus grammar. The ϵ-functions differ in that they usually return some kind of environment. The $\xi\epsilon$-functions return a pair of both ξ-Calculus code and an environment.

$$
\begin{array}{lcl}
\xi_{atpat} & : & Env \times NakedCeXL_{atpat} \rightarrow [\xi_{pattern}] \\
\xi_{pat} & : & Env \times NakedCeXL_{pat} \rightarrow [\xi_{pattern}] \\
\xi_{patrow} & : & Env \times NakedCeXL_{patrow} \rightarrow [\xi_{pattern}] \\
 & & \\
\xi_{atexp} & : & Env \times NakedCeXL_{atexp} \rightarrow [\xi_{expression}] \\
\xi_{exp} & : & Env \times NakedCeXL_{exp} \rightarrow [\xi_{expression}] \\
\xi_{exprow} & : & Env \times NakedCeXL_{exprow} \rightarrow [\xi_{expression}] \\
\xi_{fmatch} & : & Env \times NakedCeXL_{fieldmatch} \rightarrow [\xi_{expression}] \\
\xi_{match} & : & Env \times NakedCeXL_{match} \rightarrow [\xi_{expression}] \\
\xi_{mrule} & : & Env \times NakedCeXL_{mrule} \rightarrow [\xi_{expression}] \\
\xi_{decs} & : & Env \times NakedCeXL_{decs} \times NakedCeXL_{exp} \rightarrow [\xi_{expression}] \\
 & & \\
\xi_{dec} & : & Env \times Env \times NakedCeXL_{dec} \rightarrow [\xi_{expression}] \\
\xi_{valbind} & : & Env \times Env \times NakedCeXL_{valbind} \rightarrow [\xi_{expression}] \\
\xi_{exbind} & : & Env \times NakedCeXL_{exbind} \rightarrow [\xi_{expression}] \\
 & & \\
\xi_{tyvar} & : & Env \times NakedCeXL_{tyvar} \rightarrow \xi_{TyVar \times Restrict} \\
\xi_{fldstat} & : & Env \times NakedCeXL_{fieldstatus} \rightarrow \xi_{Type} \\
\xi_{tyrow} & : & Env \times NakedCeXL_{tyrow} \rightarrow [\xi_{Row}] \\
\xi_{ty} & : & Env \times NakedCeXL_{ty} \rightarrow \xi_{Type} \\
 & & \\
\xi_{crestr} & : & Env \times NakedCeXL_{crestr} \rightarrow \xi_{Restrict} \\
\xi_{restr} & : & Env \times NakedCeXL_{restr} \rightarrow \xi_{Restrict} \\
\xi_{labels} & : & NakedCeXL_{labels} \rightarrow [\xi_{ExclLabs}] \\
\xi_{typats} & : & Env \times NakedCeXL_{typats} \rightarrow \xi_{TyPats} \\
\xi_{typat} & : & Env \times NakedCeXL_{typat} \rightarrow \xi_{TyPat} \\
\xi_{typatargs} & : & Env \times NakedCeXL_{typatargs} \rightarrow [\xi_{TyPat}] \\
 & & \\
\xi_{program} & : & Env \times LongVid \times NakedCeXL_{program} \rightarrow \xi_{expression} \\
\xi\epsilon_{strdecs} & : & Env \times LongVid \times NakedCeXL_{strdecs} \rightarrow [\xi_{expression}] \times Env \\
\xi\epsilon_{strdec} & : & Env \times LongVid \times NakedCeXL_{strdec} \rightarrow [\xi_{expression}] \times Env \\
\xi\epsilon_{strbind} & : & Env \times LongVid \times NakedCeXL_{strbind} \rightarrow [\xi_{expression}] \times Env \\
 & & \\
\epsilon_{atpat} & : & Env \times Flag \times NakedCeXL_{atpat} \rightarrow Env \\
\epsilon_{pat} & : & Env \times Flag \times NakedCeXL_{pat} \rightarrow Env \\
\epsilon_{patrow} & : & Env \times Flag \times NakedCeXL_{patrow} \rightarrow Env \\
\epsilon_{dec} & : & Env \times NakedCeXL_{dec} \rightarrow Env \\
\epsilon_{valbind} & : & Env \times Flag \times NakedCeXL_{valbind} \rightarrow Env \\
\epsilon_{typbind} & : & Env \times NakedCeXL_{typbind} \rightarrow Env \\
\epsilon_{datbind} & : & Env \times NakedCeXL_{datbind} \rightarrow Env \\
\epsilon_{connames} & : & NakedCeXL_{conbind} \rightarrow NameMap \\
\epsilon_{conbind} & : & Env \times NameMap \times \xi_{TyScheme} \times NakedCeXL_{conbind} \rightarrow ValEnv \\
\epsilon_{exbind} & : & Env \times NakedCeXL_{exbind} \rightarrow Env \\
\epsilon_{typarams} & : & Env \times NakedCeXL_{typarams} \rightarrow TyVarRes \\
\epsilon_{typms} & : & Env \times NakedCeXL_{typms} \rightarrow TyVarRes \\
\epsilon_{tyargs} & : & Env \times NakedCeXL_{tyargs} \rightarrow TyArgs \\
 & & \\
\epsilon_{packstruct} & : & Env \times Env \times LongVid \times NakedCeXL_{strdecs} \rightarrow Env \\
\epsilon_{packstrdec} & : & Env \times Env \times LongVid \times NakedCeXL_{strdec} \rightarrow Env \\
\epsilon_{packdec} & : & Env \times Env \times LongVid \times NakedCeXL_{dec} \rightarrow Env \\
\epsilon_{packstrbind} & : & Env \times Env \times LongVid \times NakedCeXL_{strbind} \rightarrow Env \\
\epsilon_{packvalbind} & : & Env \times Env \times LongVid \times NakedCeXL_{valbind} \rightarrow Env \\
\epsilon_{packtypbind} & : & Env \times Env \times LongVid \times NakedCeXL_{typbind} \rightarrow Env \\
\epsilon_{packexbind} & : & Env \times Env \times LongVid \times NakedCeXL_{exbind} \rightarrow Env \\
\epsilon_{packdatbind} & : & Env \times Env \times LongVid \times NakedCeXL_{datbind} \rightarrow Env \\
\epsilon_{packconbind} & : & ValEnv \times LongVid \times NakedCeXL_{conbind} \rightarrow ValEnv \\
\epsilon_{packatpat} & : & Env \times LongVid \times NakedCeXL_{atpat} \rightarrow ValEnv \\
\epsilon_{packpat} & : & Env \times LongVid \times NakedCeXL_{pat} \rightarrow ValEnv \\
\epsilon_{packpatrow} & : & Env \times LongVid \times NakedCeXL_{patrow} \rightarrow ValEnv
\end{array}
$$

19.8 Translation Rules

In addition to the syntax to be translated, many of the translation functions take an environment consisting of a VE, an RE, a TE and a Δ parameter. We only write these environments explicitly in those rules where they are not just transferred unaltered to nested translation functions. We typically write them as (VE, RE, TE, Δ) or just (E).

If a program being translated cannot be handled by these rules, it is to be reported as an error in the Naked CeXL program, and thus in the CeXL program from whence it was stripped.

Env × Naked CeXL	ξ-Calculus	Conditions
ξ_{atpat}[| _ |]	_	
ξ_{atpat}[| *scon* |]	*scon*	
$\xi_{atpat}(VE, RE, TE, \Delta)$[| *vid* |]	*vid'*	$VE(vid) = (\sigma_{unit}, \mathbf{v}, vid')$
$\xi_{atpat}(VE, RE, TE, \Delta)$[| *longvid* |]	*vid'* **:** σ	$VE(longvid) = (\sigma, \mathbf{c}, vid')$
$\xi_{atpat}(VE, RE, TE, \Delta)$[| *longvid* |]	*vid'* **ex**	$VE(longvid) = (\forall[].[]exn\{\}, \mathbf{e}, vid')$
ξ_{atpat}[| **{** *patrow* **}** |]	**{** ξ_{patrow}[| *patrow* |] **}**	
ξ_{atpat}[| **{}** |]	**{}**	
ξ_{atpat}[| **(** *pat* **)** |]	ξ_{pat}[| *pat* |]	
ξ_{pat}[| *atpat* |]	ξ_{atpat}[| *atpat* |]	
$\xi_{pat}(VE, RE, TE, \Delta)$ [| *longvid atpat* |]	*vid'* **(** $\xi_{atpat}(VE, RE, TE, \Delta)$[| *atpat* |] **) :** σ	$VE(longvid) = (\sigma, \mathbf{c}, vid')$
$\xi_{pat}(VE, RE, TE, \Delta)$ [| *longvid atpat* |]	*vid'* **(** $\xi_{atpat}(VE, RE, TE, \Delta)$[| *atpat* |] **) ex** τ	$VE(longvid) = (\forall[].\tau \to []exn\{\}, \mathbf{e}, vid')$
ξ_{pat}[| *pat* **:** *ty* |]	ξ_{pat}[| *pat* |] **:** ξ_{ty}[| *ty* |]	
ξ_{pat}[| *vid* **:** *ty* **as** *pat* |]	*vid'* **as (** ξ_{pat}[| *pat* |] **:** ξ_{ty}[| *ty* |] **)**	$VE(vid) = (\sigma_{unit}, \mathbf{v}, vid')$
ξ_{pat}[| *vid* **as** *pat* |]	*vid'* **as** ξ_{pat}[| *pat* |]	$VE(vid) = (\sigma_{unit}, \mathbf{v}, vid')$
ξ_{patrow}[| *lab* **?=** *pat* |]	*lab* = ξ_{pat}[| *pat* |] **, {}**	
ξ_{patrow}[| *lab* **?=** *pat* **,** *patrow* |]	*lab* = ξ_{pat}[| *pat* |] **,** ξ_{patrow}[| *patrow* |]	
ξ_{patrow}[| *lab* = *pat* |]	*lab* = **(** *present* **:** σ_{pre}**) ?** **(** ξ_{pat}[| *pat* |] **) , {}**	
ξ_{patrow}[| *lab* = *pat* **,** *patrow* |]	*lab* = **(** *present* **:** σ_{pre}**) ?** **(** ξ_{pat}[| *pat* |] **) ,** ξ_{patrow}[| *patrow* |]	
ξ_{patrow}[| **...** = *pat* |]	ξ_{pat}[| *pat* |]	
ξ_{patrow}[| **...** |]	_	

Env × Flag × Naked CeXL	Env	Conditions
$\epsilon_{atpat}(E, rec)$[| **_** |]	E	
$\epsilon_{atpat}(E, rec)$[| *scon* |]	E	
$\epsilon_{atpat}((VE, RE, TE, \Delta), false)$[| *vid* |]	$(VE + \{vid \mapsto (\sigma_{unit}, \mathbf{v}, prefix + vid)\}, RE, TE, \Delta)$	$VE(vid) = (\sigma_{unit}, \mathbf{v}, vid')$ or $vid \notin$ Dom VE
$\epsilon_{atpat}((VE, RE, TE, \Delta), false)$[| *longvid* |]	(VE, RE, TE, Δ)	$VE(longvid) = (\sigma, \mathbf{c}, vid')$
$\epsilon_{atpat}((VE, RE, TE, \Delta), false)$[| *longvid* |]	(VE, RE, TE, Δ)	$VE(longvid) = (\forall[].[]exn\{\}, \mathbf{e}, vid')$
$\epsilon_{atpat}((VE, RE, TE, \Delta), true)$[| *vid* |]	$(VE + \{vid \mapsto (\sigma_{unit}, \mathbf{v}, prefix + vid)\}, RE, TE, \Delta)$	$VE(vid) = (\sigma, is, vid')$ or $vid \notin$ Dom VE
$\epsilon_{atpat}(E, rec)$[| **{** *patrow* **}** |]	$\epsilon_{patrow}(E, rec)$[| *patrow* |]	
$\epsilon_{atpat}(E, rec)$[| **{}** |]	E	
$\epsilon_{atpat}(E, rec)$[| **(** *pat* **)** |]	$\epsilon_{pat}(E, rec)$[| *pat* |]	
$\epsilon_{pat}(E, rec)$[| *atpat* |]	$\epsilon_{atpat}(E, rec)$[|*atpat* |]	
$\epsilon_{pat}((VE, RE, TE, \Delta), rec)$[| *longvid atpat* |]	$\epsilon_{atpat}((VE, RE, TE, \Delta), rec)$[| *atpat* |]	$VE(longvid) = (\sigma, \mathbf{c}, vid')$
$\epsilon_{pat}((VE, RE, TE, \Delta), rec)$[| *longvid atpat* |]	$\epsilon_{atpat}((VE, RE, TE, \Delta), rec)$[| *atpat* |]	$VE(longvid) = (\forall[].\tau \rightarrow []exn\{\}, \mathbf{e}, vid')$
$\epsilon_{pat}(E, rec)$[| *pat* **:** *ty* |]	$\epsilon_{pat}(E, rec)$[| *pat* |]	
$\epsilon_{pat}(E, false)$[| *vid* ⟨ **:** *ty* ⟩ **as** *pat* |]	$(VE + \{vid \mapsto (\sigma_{unit}, \mathbf{v}, prefix + vid)\}, RE, TE, \Delta)$ where $\epsilon_{pat}(E, rec)$[| *pat* |] $= (VE, RE, TE, \Delta)$	$VE(vid) = (\sigma_{unit}, \mathbf{v}, vid')$ or $vid \notin$ Dom VE
$\epsilon_{pat}(E, true)$[| *vid* ⟨ **:** *ty* ⟩ **as** *pat* |]	$(VE + \{vid \mapsto (\sigma_{unit}, \mathbf{v}, prefix + vid)\}, RE, TE, \Delta)$ where $\epsilon_{pat}(E, rec)$[| *pat* |] $= (VE, RE, TE, \Delta)$	$VE(vid) = (\sigma, is, vid')$ or $vid \notin$ Dom VE
$\epsilon_{patrow}(E, rec)$[| *lab* **=** *pat* |]	$\epsilon_{pat}(E, rec)$[| *pat* |]	
$\epsilon_{patrow}(E, rec)$[| *lab* **?=** *pat* |]	$\epsilon_{pat}(E, rec)$[| *pat* |]	
$\epsilon_{patrow}(E, rec)$[| *lab* **=** *pat* **,** *patrow* |]	$\epsilon_{patrow}(E', rec)$[| *patrow* |] where $\epsilon_{pat}(E, rec)$[| *pat* |] $= E'$	
$\epsilon_{patrow}(E, rec)$[| *lab* **?=** *pat* **,** *patrow* |]	$\epsilon_{patrow}(E', rec)$[| *patrow* |] where $\epsilon_{pat}(E, rec)$[| *pat* |] $= E'$	
$\epsilon_{patrow}(E, rec)$[| **...** **=** *pat* |]	$\epsilon_{pat}(E, rec)$[| *pat* |]	
$\epsilon_{patrow}(E, rec)$[| **...** |]	E	

Comments:

- The flag parameter *rec* is used to control the translation for val bindings in declarations, as will be seen later in the translation function $\epsilon_{valbind}$.

Env × Naked CeXL	ξ-Calculus	Conditions
ξ_{atexp}[| *scon* |]	*scon*	
$\xi_{atexp}(VE, RE, TE, \Delta)$[| *longvid* |]	*vid'*	$VE(longvid) = (\sigma_{unit}, \mathbf{v}, vid')$
$\xi_{atexp}(VE, RE, TE, \Delta)$[| *longvid* |]	*vid'* **:** σ	$VE(longvid) = (\sigma, \mathbf{c}, vid')$
$\xi_{atexp}(VE, RE, TE, \Delta)$[| *longvid* |]	*vid'* **ex** τ	$VE(longvid) = (\forall[].\tau, \mathbf{e}, vid')$
ξ_{atexp}[| **{** *exprow* **}** |]	**{** ξ_{exprow}[| *exprow* |] **}**	
ξ_{atexp}[| **{}** |]	**{}**	
ξ_{atexp}[| **let in** *exp* **end** |]	ξ_{exp}[| *exp* |]	
ξ_{atexp}[| **let** *decs* **in** *exp* **end** |]	ξ_{decs}[| *decs* |][| *exp* |]	
$\xi_{atexp}(E)$[| **fieldcase** *exp* **in** *tyvar* **of** *fieldmatch* **type** *ty* **end** |]	**fieldcase** $\xi_{exp}(E)$[| *exp* |] **in** *tyvar* **of** $\xi_{fmatch}(E)$[| *fieldmatch* |] **type** $\xi_{ty}(E)$[| *ty* |]	$(E) = (VE, RE, TE, \Delta)$ $\Delta(tyvar) =$ $[]absent\{absent\}$ / $[]present\{present\}$ or $tyvar \notin$ Dom Δ
ξ_{atexp}[| **(** *exp* **)** |]	ξ_{exp}[| *exp* |]	
ξ_{exp}[| *atexp* |]	ξ_{atexp}[| *atexp* |]	
ξ_{exp}[| *exp atexp* |]	ξ_{exp}[| *exp* |] ξ_{atexp}[| *atexp* |]	
ξ_{exp}[| *exp* **:** *ty* |]	ξ_{exp}[| *exp* |] **:** ξ_{ty}[| *ty* |]	
ξ_{exp}[| **raise** *exp* |]	**raise** ξ_{exp}[| *exp* |]	
ξ_{exp}[| *exp* **handle** *match* |]	ξ_{exp}[| *exp* |] **handle** ξ_{match}[| *match* |]	
ξ_{exp}[| **case** *exp* **of** *match* |]	**case** ξ_{exp}[| *exp* |] **of** ξ_{match}[| *match* |]	
ξ_{exp}[| **fn** *match* |]	λx.**case** x **of** ξ_{match}[| *match* |]	fresh x
ξ_{exprow}[| *lab* `?=` *exp* |]	*lab* **=** ξ_{exp}[| *exp* |] **, {}**	
ξ_{exprow}[| *lab* `?=` *exp* **,** *exprow* |]	*lab* **=** ξ_{exp}[| *exp* |] **,** ξ_{exprow}[| *exprow* |]	
ξ_{exprow}[| *lab* **=** *exp* |]	*lab* **=** (*present* **:** σ_{pre}) **?** (ξ_{exp}[| *exp* |]) **, {}**	
ξ_{exprow}[| *lab* **=** *exp* **,** *exprow* |]	*lab* **=** (*present* **:** σ_{pre}) **?** (ξ_{exp}[| *exp* |]) **,** ξ_{exprow}[| *exprow* |]	
ξ_{exprow}[| **...** **=** *exp* |]	ξ_{exp}[| *exp* |]	

Env × Naked CeXL	ξ-Calculus	Conditions
$\xi_{fmatch}(E)$[| `absent =>` exp_1 `|` `present` *atpat* `=>` exp_2 |]	**absent** $\Rightarrow$ $\xi_{exp}(E)$[| exp_1 |] $\|\|$ **present** $\xi_{atpat}(E')$[| *atpat* |] $\Rightarrow$ $\xi_{exp}(E')$[| exp_2 |] where $\epsilon_{atpat}(E, false)$[| *atpat* |] $= E'$	
$\xi_{match}(E)$[| *mrule* |]	$\xi_{mrule}(E)$[| *mrule* |]	
$\xi_{match}(E)$[| *mrule* `|` *match* |]	$\xi_{mrule}(E)$[| *mrule* |] $\|$ $\xi_{match}(E)$[| *match* |]	
$\xi_{mrule}(E)$[| *pat* `=>` *exp* |]	$\xi_{pat}(E')$[| *pat* |] $\Rightarrow$ $\xi_{exp}(E')$[| *exp* |] where $\epsilon_{pat}(E, false)$[| *pat* |] $= E'$	
$\xi_{decs}(E)$[| *dec* ⟨ **;** ⟩ |][| *exp* |]	$\xi_{dec}(E, E')$[| *dec* |] $\xi_{exp}(E')$[| *exp* |] where $\epsilon_{dec}(E)$[| *dec* |] $= E'$	
$\xi_{decs}(E)$[| *dec* ⟨ **;** ⟩ *decs* |][| *exp* |]	$\xi_{dec}(E, E')$[| *dec* |] $\xi_{decs}(E')$[| *decs* |][| *exp* |] where $\epsilon_{dec}(E)$[| *dec* |] $= E'$	

Comments:

- The ξ_{dec} translation function is given 2 environments. The first environment is the unmodified environment before the declaration *dec* takes place. The second environment is the modified environment where the declarations of *dec* have been added. This allows us to handle the val bindings correctly in other rules.

Env × Env × Naked CeXL	ξ-Calculus	Conditions
$\xi_{dec}((VE, RE, TE, \Delta_1), (VE', RE', TE', \Delta_1'))$ [| **val** *typms valbind* |]	**let** $\xi_{valbind}((VE', RE', TE', \Delta_2'), (VE, RE, TE, \Delta_2))$ [| *valbind* |] **in** where $\epsilon_{typms}(VE, RE, TE, \Delta_1)$[| *typms* |] $= \Delta_2$ and $\epsilon_{typms}(VE', RE', TE', \Delta_1')$[| *typms* |] $= \Delta_2'$	
$\xi_{dec}(E, (VE', RE', TE', \Delta_1'))$ [| **val** *typms* **rec** *valbind* |]	**letrec** $\xi_{valbind}((VE', RE', TE', \Delta_2'), (VE', RE', TE', \Delta_2'))$ [| *valbind* |] **in** where $\epsilon_{typms}(VE', RE', TE', \Delta_1')$[| *typms* |] $= \Delta_2'$	
$\xi_{dec}(E, E')$[| **exception** *exbind* |]	**letex** $\xi_{exbind}(E')$[| *exbind* |] **in**	
$\xi_{dec}(E, E')$[| **res** *vid* ⟨ = *crestr* ⟩ |]		
$\xi_{dec}(E, E')$[| **type** *typbind* |]		
$\xi_{dec}(E, E')$[| **datatype** *datbind* |]		
$\xi_{dec}(E, E')$[| **datatype** *tycon* = **datatype** *longtycon* |]		
$\xi_{valbind}(E_1, E_2)$[| *pat* = *exp* |]	$\xi_{pat}(E_1)$[| *pat* |] $= \xi_{exp}(E_2)$[| *exp* |]	
$\xi_{valbind}(E_1, E_2)$[| *pat* = *exp* **and** *valbind* |]	$\xi_{pat}(E_1)$[| *pat* |] $= \xi_{exp}(E_2)$[| *exp* |] ; $\xi_{valbind}(E_1, E_2)$[| *valbind* |]	

Env × Naked CeXL	ξ-Calculus	Conditions
$\xi_{exbind}(VE, RE, TE, \Delta)$[| *vid* ⟨ **of** *ty* ⟩ |]	*vid*' : τ	$VE(vid) = (\forall[].\tau, \mathbf{e}, vid')$
$\xi_{exbind}(VE, RE, TE, \Delta)$[| *vid* ⟨ **of** *ty* ⟩ **and** *exbind* |]	*vid*' : τ , $\xi_{exbind}(VE, RE, TE, \Delta)$[| *exbind* |]	$VE(vid) = (\forall[].\tau, \mathbf{e}, vid')$
$\xi_{exbind}(VE, RE, TE, \Delta)$[| *vid* = *longvid* |]	*vid*' : τ	$VE(vid) = (\forall[].\tau, \mathbf{e}, vid')$
$\xi_{exbind}(VE, RE, TE, \Delta)$[| *vid* = *longvid* **and** *exbind* |]	*vid*' : τ , $\xi_{exbind}(VE, RE, TE, \Delta)$[| *exbind* |]	$VE(vid) = (\forall[].\tau, \mathbf{e}, vid')$

Comments:

- $\xi_{valbind}$ takes two environments as argument. The first is to be used for the patterns in the bindings and the second is to be used for the expressions in the bindings.
- In the rule for val in ξ_{dec} we pass the environments (E', E) to $\xi_{valbind}$ because the patterns must include the new bindings (contained in E') whereas in the expressions these bindings are not seen at this point (i.e. they are not in E).
- On the other hand we pass (E', E') to $\xi_{valbind}$ in the val rec rule to make the recursive binding work.
- For the *typms* in the val and val rec bindings we update the Δ_1 and Δ_1' respectively to Δ_2 and Δ_2' respectively, to get the specified restrictions in the value bindings.
- In the rule for exception in ξ_{dec} pass E' to ξ_{exbind} because we need to use the newly declared exception names when generating the ξ-Calculus code.

Env × Naked CeXL	Env	Conditions
$\epsilon_{dec}(VE, RE, TE, \Delta)$ [| **val** *typms valbind* |]	(VE', RE', TE', Δ) where $\epsilon_{valbind}((VE, RE, TE, \Delta'), false)[\| \ valbind \ \|] = (VE', RE', TE', \Delta'')$ and $\epsilon_{typms}(VE, RE, TE, \Delta)[\| \ typms \ \|] = \Delta'$	
$\epsilon_{dec}(VE, RE, TE, \Delta)$ [| **val** *typms* **rec** *valbind* |]	(VE', RE', TE', Δ) where $\epsilon_{valbind}((VE, RE, TE, \Delta'), true)[\| \ valbind \ \|] = (VE', RE', TE', \Delta'')$ and $\epsilon_{typms}(VE, RE, TE, \Delta)[\| \ typms \ \|] = \Delta'$	

Comments

- In the val rec binding we pass *true* to the *rec* parameter for ϵ_{pat}. This means that we ignore all constructors and exceptions in patterns here. This allows new identifiers to be declared even though the same identifiers have been declared earlier as constructors.[5]
- Notice that neither of the environments Δ' and Δ'' are propagated out from ϵ_{dec}.

[5] This corresponds to rule (26) in The Definition of Standard ML '97 where identifier status is overwritten in the recursive binding

Env × Naked CeXL	Env	Conditions
$\epsilon_{dec}(VE, RE, TE, \Delta)$[| **res** *vid* |]	$(VE, RE + \{vid \mapsto \circ\}, TE, \Delta)$	
$\epsilon_{dec}(VE, RE, TE, \Delta)$[| **res** *vid* = *crestr* |]	$(VE, RE + \{vid \mapsto \xi\}, TE, \Delta)$ where $\xi_{crestr}(VE, RE, TE, \Delta)$[| *crestr* |] $= \xi$	
$\epsilon_{dec}(VE, RE, TE, \Delta)$[| **type** *typbind* |]	$\epsilon_{typbind}(VE, RE, TE, \Delta)$[| *typbind* |]	
$\epsilon_{dec}(VE, RE, TE, \Delta)$[| **datatype** *datbind* |]	$(VE + Clos\ VE', RE', TE', \Delta')$ where $\epsilon_{datbind}(\{\}, RE, TE + TE', \Delta)$[| *datbind* |] $= (VE', RE', TE', \Delta')$	The rules below apply to $Clos\ VE'$
$\epsilon_{dec}(VE, RE, TE, \Delta)$[| **datatype** *tycon* = **datatype** *longtycon* |]	(VE, RE, TE + $\{tycon \mapsto \theta\}, \Delta$)	TE($longtycon$) $= \theta$ $\theta = \Lambda[\alpha_1 :: \xi_1, \ldots, \alpha_n :: \xi_n].\kappa$
$\epsilon_{dec}(VE, RE, TE, \Delta)$[| **exception** *exbind* |]	$\epsilon_{exbind}(VE, RE, TE, \Delta)$[| *exbind* |]	

Rules for Constructor Quantification:

- Notice that the binding for TE is recursive in the translation of datatypes. This allows the types to be used recursively in the declaration.
- When taking the closure of the value environment VE of constructors resulting from translation with $\xi_{datbind}$, the following rule apply: The choice of quantification of the resulting type scheme σ must be such that each constructor is quantified with the same variables as the ConsType κ. So assume that

  $$\kappa = [a_1 = \alpha_1 :: \xi_1, \cdots, a_n = \alpha_n :: \xi_n]d\{c_1, \ldots, c_k\}$$

  with $a_1 \equiv \alpha_1, \ldots, a_n \equiv \alpha_n$ (i.e. the a_i and α_i are the same names). Then the resulting type scheme σ must be quantified with

  $$\forall[\alpha_1 :: \xi_1, \cdots, \alpha_n :: \xi_n]$$

  This is the same quantification as the one explicitly given in the translation function $\epsilon_{conbind}$, so the easiest for an implementation is to just keep this quantification.

  This quantification means that if the constructor has type of the form κ (the same κ as before) then

  $$\sigma = \forall[\alpha_1 :: \xi_1, \cdots, \alpha_n :: \xi_n].[a_1 = \alpha_1 :: \xi_1, \cdots, a_n = \alpha_n :: \xi_n]d\{c_1, \ldots, c_k\}$$

  and if the constructor has type of the form $\tau \to \kappa$ then

  $$\sigma = \forall[\alpha_1 :: \xi_1, \cdots, \alpha_n :: \xi_n].\tau \to [a_1 = \alpha_1 :: \xi_1, \cdots, a_n = \alpha_n :: \xi_n]d\{c_1, \ldots, c_k\}$$

Comments

- The right-hand side of datatype replications is only required to be a ConsType, which means that also the primitive types like int, real, string etc. are allowed.
- None of $\epsilon_{datbind}$ or $\epsilon_{typbind}$ change the environment Δ, so it is safe to return the Δ' that they return.

Env × Flag × Naked CeXL	Env	Conditions
$\epsilon_{valbind}(E, rec)[\mid\ pat = exp\ \mid]$ $\epsilon_{valbind}(E, rec)[\mid\ pat = exp$ **and** $valbind\ \mid]$	$\epsilon_{pat}(E, rec)[\mid\ pat\ \mid]$ $\epsilon_{valbind}(\ \epsilon_{pat}(E, rec)[\mid\ pat\ \mid]\ , rec)[\mid\ valbind\ \mid]$	

Env × Naked CeXL	Env	Conditions
$\epsilon_{typbind}(VE, RE, TE, \Delta)[\mid\ typms$ $tycon = ty\ \mid]$	(VE, RE, TE + $\{tycon \mapsto$ $\Lambda[\alpha_1 :: \xi_1, \ldots, \alpha_n :: \xi_n].\tau\}, \Delta)$ where $\epsilon_{typms}(VE, RE, TE, \{\})[\mid\ typms\ \mid] = \Delta' =$ $\{\alpha_1 \mapsto \xi_1, \cdots, \alpha_n \mapsto \xi_n\}$ and $\xi_{ty}(VE, RE, TE, \Delta')[\mid\ ty\ \mid] = \tau$	$\triangleright\tau\triangleleft$ $\Delta' \vdash^{tspec} \tau$
$\epsilon_{typbind}(VE, RE, TE, \Delta)[\mid\ typms$ $tycon = ty$ **and** $typbind\ \mid]$	$\epsilon_{typbind}(VE, RE, TE + \{tycon \mapsto$ $\Lambda[\alpha_1 :: \xi_1, \ldots, \alpha_n :: \xi_n].\tau\}, \Delta)$ $[\mid\ typbind\ \mid]$ where $\epsilon_{typms}(VE, RE, TE, \{\})[\mid\ typms\ \mid] = \Delta' =$ $\{\alpha_1 \mapsto \xi_1, \cdots, \alpha_n \mapsto \xi_n\}$ and $\xi_{ty}(VE, RE, TE, \Delta')[\mid\ ty\ \mid] = \tau$	$\triangleright\tau\triangleleft$ $\Delta' \vdash^{tspec} \tau$

Comments:

- In the side conditions we use $\triangleright\tau\triangleleft$ to ensure that τ doesn't contain any free type variables. We use $\Delta \vdash^{tspec} \tau$ from the static semantics of ξ-Calculus to make sure that the type τ is well-formed.
- We pass an empty environment of type variables bound to restrictions when translating type parameters for declarations. This is because only the type variables present in the parameter lists are allowed when we declare types and datatypes. The syntactic restrictions ensure this.

Env × Naked CeXL	Env	Conditions
$\epsilon_{datbind}(VE, RE, TE, \Delta)$[| *typms* *tycon* = *conbind* |]	(VE + $\epsilon_{conbind}((VE, RE, TE, \Delta'), NN'$, $\forall[\alpha_1 :: \xi_1, \cdots, \alpha_n :: \xi_n].\kappa)$[| *conbind* |], RE, TE + $\{tycon \mapsto \Lambda[\alpha_1 :: \xi_1, \cdots, \alpha_n :: \xi_n].\kappa\}$, $\Delta\}$ where $\epsilon_{connames}$[| *conbind* |] = NN' = $\{vid_1 \mapsto vid'_1, \ldots, vid_n \mapsto vid'_n\}$ and $\epsilon_{typms}(VE, RE, TE, \{\})$[| *typms* |] = Δ' = $\{\alpha_1 \mapsto \xi_1, \cdots, \alpha_n \mapsto \xi_n\}$ and $\kappa = [a_1 = \alpha_1 :: \xi_1, \cdots, a_n = \alpha_n :: \xi_n]vid\{vid'_1, \ldots, vid'_n\}$ and $a_1 \equiv \alpha_1, \ldots, a_n \equiv \alpha_n$	fresh(ν) = *vid*
$\epsilon_{datbind}(VE, RE, TE, \Delta)$[| *typms* *tycon* = *conbind* **and** *datbind* |]	$\epsilon_{datbind}$(VE + $\epsilon_{conbind}((VE, RE, TE, \Delta'), NN'$, $\forall[\alpha_1 :: \xi_1, \cdots, \alpha_n :: \xi_n].\kappa)$[| *conbind* |], RE, TE + $\{tycon \mapsto \Lambda[\alpha_1 :: \xi_1, \cdots, \alpha_n :: \xi_n].\kappa\}$, $\Delta\}$ [| *datbind* |] where $\epsilon_{connames}$[| *conbind* |] = NN' = $\{vid_1 \mapsto vid'_1, \ldots, vid_n \mapsto vid'_n\}$ and $\epsilon_{typms}(VE, RE, TE, \{\})$[| *typms* |] = Δ' = $\{\alpha_1 \mapsto \xi_1, \cdots, \alpha_n \mapsto \xi_n\}$ and $\kappa = [a_1 = \alpha_1 :: \xi_1, \cdots, a_n = \alpha_n :: \xi_n]vid\{vid'_1, \ldots, vid'_n\}$ and $a_1 \equiv \alpha_1, \ldots, a_n \equiv \alpha_n$	fresh(ν) = *vid*

Env × NameMap × TyScheme × Naked CeXL	ValEnv	Conditions
$\epsilon_{conbind}(E, NN, \forall[\alpha_1 :: \xi_1, \cdots, \alpha_n :: \xi_n].\kappa)$[| *vid* |]	$\{vid \mapsto (\forall[\alpha_1 :: \xi_1, \cdots, \alpha_n :: \xi_n].\kappa, \mathbf{c}, vid')\}$	$NN(vid) = vid'$
$\epsilon_{conbind}((VE, RE, TE, \Delta), NN$, $\forall[\alpha_1 :: \xi_1, \cdots, \alpha_n :: \xi_n].\kappa)$ [| *vid* **of** *ty* |]	$\{vid \mapsto (\forall[\alpha_1 :: \xi_1, \cdots, \alpha_n :: \xi_n].\tau \to \kappa, \mathbf{c}, vid')\}$ where $\xi_{ty}(VE, RE, TE, \Delta)$[| *ty* |] = τ	$NN(vid) = vid'$ $\triangleright\tau\triangleleft$ $\Delta \vdash^{tspec} \tau$
$\epsilon_{conbind}(E, NN, \forall[\alpha_1 :: \xi_1, \cdots, \alpha_n :: \xi_n].\kappa)$ [| *vid* | *conbind* |]	$\{vid \mapsto (\forall[\alpha_1 :: \xi_1, \cdots, \alpha_n :: \xi_n].\kappa, \mathbf{c}, vid')\}$ + $\epsilon_{conbind}(E, NN, \kappa)$[| *conbind* |]	$NN(vid) = vid'$
$\epsilon_{conbind}((VE, RE, TE, \Delta), NN$, $\forall[\alpha_1 :: \xi_1, \cdots, \alpha_n :: \xi_n].\kappa)$ [| *vid* **of** *ty* | *conbind* |]	$\{vid \mapsto (\forall[\alpha_1 :: \xi_1, \cdots, \alpha_n :: \xi_n].\tau \to \kappa, \mathbf{c}, vid')\}$ + $\epsilon_{conbind}((VE, RE, TE, \Delta), NN, \kappa)$[| *conbind* |] where $\xi_{ty}(VE, RE, TE, \Delta)$[| *ty* |] = τ	$NN(vid) = vid'$ $\triangleright\tau\triangleleft$ $\Delta \vdash^{tspec} \tau$

Naked CeXL	NameMap	Conditions
$\epsilon_{connames}$[| *vid* ⟨ **of** *ty* ⟩ |]	$\{vid \mapsto vid'\}$	fresh(N) = *vid*'
$\epsilon_{connames}$[| *vid* ⟨ **of** *ty* ⟩ | *conbind* |]	$\{vid \mapsto vid'\}$ + $\epsilon_{connames}$[| *conbind* |]	fresh(N) = *vid*'

Comments:

- The syntactical restrictions ensure that a datatype declaration does not declare the same constructor twice within the same datatype.

  The translation rules map the constructor names to the names in the map NN.

- $a_1 \equiv \alpha_1, \ldots, a_n \equiv \alpha_n$ means that the a_i and the α_i are the same variable names.

Env × Naked CeXL	Env	Conditions
$\epsilon_{exbind}(VE, RE, TE, \Delta)[\mid vid \mid]$	$(VE + \{vid \mapsto (\sigma_{exn}, \mathbf{e}, prefix + vid)\},$ $RE, TE, \Delta)$	
$\epsilon_{exbind}(VE, RE, TE, \Delta)[\mid vid$ **of** $ty \mid]$	$(VE + \{vid \mapsto (\forall[].\tau \to \kappa_{exn}, \mathbf{e}, prefix + vid)\},$ $RE, TE, \Delta)$ where $\xi_{ty}(VE, RE, TE, \Delta)[\mid ty \mid] = \tau$	$\triangleright\tau\triangleleft$ if *exbind* occurs directly at top-level or in a structure, we require: $tyvars\ \tau = \emptyset$
$\epsilon_{exbind}(VE, RE, TE, \Delta)[\mid vid$ **and** $exbind \mid]$	$\epsilon_{exbind}(VE + \{vid \mapsto (\sigma_{exn}, \mathbf{e}, prefix + vid)\},$ $RE, TE, \Delta)[\mid exbind \mid]$	
$\epsilon_{exbind}(VE, RE, TE, \Delta)[\mid vid$ **of** ty **and** $exbind \mid]$	$\epsilon_{exbind}(VE + \{vid \mapsto$ $(\forall[].\tau \to \kappa_{exn}, \mathbf{e}, prefix + vid)\},$ $RE, TE, \Delta)[\mid exbind \mid]$ where $\xi_{ty}(VE, RE, TE, \Delta)[\mid ty \mid] = \tau$	$\triangleright\tau\triangleleft$ if *exbind* occurs directly at top-level or in a structure, we require: $tyvars\ \tau = \emptyset$
$\epsilon_{exbind}(VE, RE, TE, \Delta)[\mid vid = longvid \mid]$	$(VE + \{vid \mapsto (\forall[].\tau, \mathbf{e}, vid')\},$ $RE, TE, \Delta)$	$VE(longvid) = (\forall[].\tau, \mathbf{e}, vid')$
$\epsilon_{exbind}(VE, RE, TE, \Delta)[\mid vid = longvid$ **and** $exbind \mid]$	$\epsilon_{exbind}(VE + \{vid \mapsto (\forall[].\tau, \mathbf{e}, vid')\},$ $RE, TE, \Delta)[\mid exbind \mid]$	$VE(longvid) = (\forall[].\tau, \mathbf{e}, vid')$

Comments:

- For exception declarations we prefix the internal constructor names with the structure they are declared in - just as we do for variables.
- In the syntactic restrictions of CeXL and Naked CeXL, we require that exceptions declared directly at top-level or in a structure do not contain any type variables. However, in the presence of the feature *Partial Type Instantiation* this syntactic restriction is not enough. Therefore we require that it still holds here in the translation for the τ which have been translated from Naked CeXL.
- We do not check here that τ is well-formed, since this is also done in ξ-Calculus at the letex bindings. The environment Δ does not include the implicitly bound type variables here, so the check for well-formedness can only be done correctly in ξ-Calculus.

Env × Naked CeXL	TyVarRes	Conditions
$\epsilon_{typms}(VE, RE, TE, \Delta)$[| |]	Δ	
$\epsilon_{typms}(E)$[| **[** *typarams* **]** |]	$\epsilon_{typarams}(E)$[| *typarams* |]	
$\epsilon_{typarams}(E)$[| *tyvar*$_1$**;** $\cdots$ **;** *tyvar*$_m$ |]	$\{tyvar_1 \mapsto \circ, \ldots, tyvar_m \mapsto \circ\} + \Delta$ where $E = (VE, RE, TE, \Delta)$	$m \geq 1$
$\epsilon_{typarams}(E)$[| *tyvar*$_1$**;** $\cdots$ **;** *tyvar*$_m$ **:** *crestr* |]	$\{tyvar_1 \mapsto \xi, \ldots, tyvar_m \mapsto \xi\} + \Delta$ where $\xi_{crestr}(E)$[| *crestr* |] $= \xi$ and $E = (VE, RE, TE, \Delta)$	$m \geq 1$
$\epsilon_{typarams}(E)$[| *tyvar*$_1$**;** $\cdots$ **;** *tyvar*$_m$ **,** *typarams* |]	$\{tyvar_1 \mapsto \circ, \ldots, tyvar_m \mapsto \circ\}+$ $\epsilon_{typarams}(E)$[| *typarams* |]	$m \geq 1$
$\epsilon_{typarams}(E)$[| *tyvar*$_1$**;** $\cdots$ **;** *tyvar*$_m$ **:** *crestr* **,** *typarams* |]	$\{tyvar_1 \mapsto \xi, \ldots, tyvar_m \mapsto \xi\}+$ $\epsilon_{typarams}(E)$[| *typarams* |] where $\xi_{crestr}(E)$[| *crestr* |] $= \xi$	$m \geq 1$

Env × Naked CeXL	TyArgs	Conditions
$\epsilon_{tyargs}(E)$[| *tyvar* **=** *ty* |]	$\{tyvar \mapsto \xi_{ty}(E)$[| *ty* |]$\}$	
$\epsilon_{tyargs}(E)$[| *tyvar* **=** *ty* **,** *tyargs* |]	$\{tyvar \mapsto \xi_{ty}(E)$[| *ty* |]$\} + \epsilon_{tyargs}(E)$[| *tyargs* |]	

Env × Naked CeXL	TyVar × Restrict	Conditions
$\xi_{tyvar}(VE, RE, TE, \Delta)$[| *tyvar* |]	*tyvar* :: $\circ$	*tyvar* $\notin Dom\ \Delta$
$\xi_{tyvar}(VE, RE, TE, \Delta)$[| *tyvar* |]	*tyvar* :: ξ	Δ(*tyvar*) $= \xi$

Env × Naked CeXL	ξ-Calculus	Conditions
$\xi_{fldstat}(E)$[| **-** |]	$[]absent\{absent\}$	
$\xi_{fldstat}(E)$[| *tyvar* |]	$\xi_{tyvar}(E)$[| *tyvar* |]	
$\xi_{fldstat}(E)$[| |]	$[]present\{present\}$	
$\xi_{tyrow}(E)$[| *fieldstatus lab* **:** *ty* **,** *tyrow* |]	*lab* : $\xi_{fldstat}(E)$[| *fieldstatus* |] ? $\xi_{ty}(E)$[| *ty* |] , $\xi_{tyrow}(E)$[| *tyrow* |]	*tyrow* $\neq$ **...** **:** *tyvar*
$\xi_{tyrow}(E)$[| *fieldstatus lab* **:** *ty* **,** **...** **:** *tyvar* |]	*lab* : $\xi_{fldstat}(E)$[| *fieldstatus* |] ? $\xi_{ty}(E)$[| *ty* |] ; $\xi_{tyvar}(E)$[| *tyvar* |]	
$\xi_{tyrow}(E)$[| *fieldstatus lab* **:** *ty* |]	*lab* : $\xi_{fldstat}(E)$[| *fieldstatus* |] ? $\xi_{ty}(E)$[| *ty* |] ; {}	
$\xi_{ty}(E)$[| *tyvar* |]	$\xi_{tyvar}(E)$[| *tyvar* |]	
$\xi_{ty}(E)$[| **{}** |]	{}	
$\xi_{ty}(E)$[| **{** *tyrow* **}** |]	{ $\xi_{tyrow}(E)$[| *tyrow* |] }	
$\xi_{ty}(VE, RE, TE, \Delta)$[| *longtycon* |]	$\zeta\theta$ type function application subject to the restrictions in section 19.3, where $\zeta = \{\}$	*longtycon* $\neq$ **?** TE(*longtycon*) $= \theta$
$\xi_{ty}(VE, RE, TE, \Delta)$ [| **[** *tyargs* **]** *longtycon* |]	$\zeta\theta$ type function application subject to the restrictions in section 19.3, where $\zeta = \epsilon_{tyargs}(VE, RE, TE, \Delta)$[| *tyargs* |]	*longtycon* $\neq$ **?** TE(*longtycon*) $= \theta$
$\xi_{ty}(E)$[| *ty* **->** *ty'* |]	$\xi_{ty}(E)$[| *ty* |] $\rightarrow \xi_{ty}(E)$[| *ty'* |]	
$\xi_{ty}(E)$[| **(** *ty* **)** |]	$\xi_{ty}(E)$[| *ty* |]	
$\xi_{ty}(E)$[| **[** **'a** $=$ *ty*$_1$**,** **'b** $=$ *ty*$_2$ **]** **?** |]	$\xi_{ty}(E)$[| *ty*$_1$ |] ? $\xi_{ty}(E)$[| *ty*$_2$ |]	
$\xi_{ty}(E)$[| **[** **'b** $=$ *ty*$_2$**,** **'a** $=$ *ty*$_1$ **]** **?** |]	$\xi_{ty}(E)$[| *ty*$_1$ |] ? $\xi_{ty}(E)$[| *ty*$_2$ |]	
$\xi_{ty}(E)$[| **[** **'a** $=$ *ty* **]** **?** |]	$\xi_{ty}(E)$[| *ty* |] ? τ	fresh τ
$\xi_{ty}(E)$[| **[** **'b** $=$ *ty* **]** **?** |]	τ ? $\xi_{ty}(E)$[| *ty* |]	fresh τ
$\xi_{ty}(E)$[| **?** |]	τ_1 ? τ_2	fresh τ_1, fresh τ_2

Env × Naked CeXL	ξ-Calculus	Conditions
$\xi_{crestr}(E)$[| *restr* |]	ξ where $\xi_{restr}(E)$[| *restr* |] = ξ	$\vdash \xi$
$\xi_{crestr}(E)$[| *restr* **+** *crestr* |]	$\xi \bowtie \xi_{crestr}(E)$[| *crestr* |] where $\xi_{restr}(E)$[| *restr* |] = ξ	$\vdash \xi$
$\xi_{restr}(E)$[| ∼**{ }** |] $\xi_{restr}(E)$[| ∼**{** *labels* **}** |] $\xi_{restr}(E)$[| *typats* |] $\xi_{restr}(VE, RE, TE, \Delta)$[| **res** *longvid* |]	{} { ξ_{labels}[| *labels* |] } $\xi_{typats}(E)$[| *typats* |] ξ	 $RE(longvid) = \xi$
ξ_{labels}[| *lab* |] ξ_{labels}[| *lab* **,** *labels* |]	*lab* *lab* , ξ_{labels}[| *labels* |]	
$\xi_{typats}(E)$[| *typat* |] $\xi_{typats}(E)$[| *typat* **|** *typats* |]	$\xi_{typat}(E)$[| *typat* |] $\xi_{typat}(E)$[| *typat* |] / $\xi_{typats}(E)$[| *typats* |]	
$\xi_{typat}(VE, RE, TE, \Delta)$[| *longtycon* |]	$[]d\{c_1, \ldots, c_k\}$	$TE(longtycon) =$ $\Lambda[].[]d\{c_1, \ldots, c_k\}$ $k \geq 0,\ d \neq exn$
$\xi_{typat}(VE, RE, TE, \Delta)$[| **[** *typatargs* **]** *longtycon* |]	$[a_1 = \psi_1, \ldots, a_n = \psi_n]d\{c_1, \ldots, c_k\}$ where $\xi_{typatargs}(VE, RE, TE, \Delta)$[| *typatargs* |] = $a_1 = \psi_1, \ldots, a_n = \psi_n$	$TE(longtycon) =$ $\Lambda[\alpha_1 :: \xi_1, \ldots, \alpha_n :: \xi_n].$ $[a_1 = \alpha_1 :: \xi_1, \cdots, a_n = \alpha_n :: \xi_n]$ $d\{c_1, \ldots, c_k\}$ $k \geq 0,\ d \neq exn$ $\psi_1 \gg \xi_1,\ \ldots,\ \psi_n \gg \xi_n$
$\xi_{typatargs}(E)$[| *tyvar* **=** *typat* |] $\xi_{typatargs}(E)$[| *tyvar* **=** *typat* **,** *typatargs* |]	*tyvar* = $\xi_{typat}(E)$[| *typat* |] *tyvar* = $\xi_{typat}(E)$[| *typat* |] , $\xi_{typatargs}(E)$[| *typatargs* |]	

Comments:

- Notice that when translating type patterns, the number of parameters and the parameter names specified in the Naked CeXL syntax must be the same as declared for the constructors used in the type pattern.

  It is expected that type patterns will not be used very much in the language except for specifying `absent | present`, since we don't support any general typecase construct.

- We don't allow the type name $[]exn\{\}$ in *typat* - even though it probably wouldn't cause any problems. This is enforced by $d \neq exn$.

Env × LongVId × Naked CeXL	ξ-Calculus	Conditions
$\xi\epsilon_{program}(E, prefix)[\mid\ strdecs\ \mid]$	ξ_{code} **{}** where $\quad \xi\epsilon_{strdecs}(E, prefix)[\mid\ strdecs\ \mid] = (\xi_{code}, E')$	

Env × LongVId × Naked CeXL	ξ-Calculus × Env	Conditions
$\xi\epsilon_{strdecs}(E, prefix)[\mid\ strdec\ \langle\ \mathbf{;}\ \rangle\ \mid]$	$\xi\epsilon_{strdec}(E, prefix)[\mid\ strdec\ \mid]$	
$\xi\epsilon_{strdecs}(E, prefix)[\mid\ strdec\ \langle\ \mathbf{;}\ \rangle\ strdecs\ \mid]$	$(\xi_{code}\ \xi'_{code}, E'')$ where $\quad \xi\epsilon_{strdec}(E, prefix)[\mid\ strdec\ \mid] = (\xi_{code}, E')$ and $\quad \xi\epsilon_{strdecs}(E', prefix)[\mid\ strdecs\ \mid] = (\xi'_{code}, E'')$	
$\xi\epsilon_{strdec}(E, prefix)[\mid\ dec\ \mid]$	$(\xi_{dec}(E, E')[\mid\ dec\ \mid], E')$ where $\quad \epsilon_{dec}(E)[\mid\ dec\ \mid] = E'$	
$\xi\epsilon_{strdec}(E, prefix)[\mid\ \mathbf{structure}\ strbind\ \mid]$	$\xi\epsilon_{strbind}(E, prefix)[\mid\ strbind\ \mid]$	
$\xi\epsilon_{strbind}(E, prefix)[\mid\ strid\ \mathbf{=}\ \mathbf{struct}\ strdecs\ \mathbf{end}\ \mid]$	(ξ_{code}, E'') where $\quad \xi\epsilon_{strdecs}(E, prefix + strid + ".")[\mid\ strdecs\ \mid] = (\xi_{code}, E')$ and $\quad \epsilon_{packstruct}(E, E', strid + ".")[\mid\ strdecs\ \mid] = E''$	
$\xi\epsilon_{strbind}((VE, RE, TE, \Delta), prefix)$ $\quad [\mid\ strid = longstrid\ \mid]$	$($ $,(VE + \{strid + "." + longvid \mapsto (\sigma, is, vid) \mid$ $\quad VE(longstrid + "." + longvid) = (\sigma, is, vid)\},$ $\quad RE + \{strid + "." + longvid \mapsto \xi \mid$ $\quad RE(longstrid + "." + longvid) = \xi\},$ $\quad TE + \{strid + "." + longtycon \mapsto \theta \mid$ $\quad TE(longstrid + "." + longtycon) = \theta\},$ $\quad \Delta))$	

Comments:

- The translation functions $\xi\epsilon_{strdecs}$, $\xi\epsilon_{strdec}$ and $\xi\epsilon_{strbind}$ return a pair of both ξ-Calculus program code and an environment. We use variables like ξ_{code} to represent generated ξ-Calculus code in these rules.
- The translation of *dec*, *strdec* and *strdecs* results in both a ξ-Calculus program which is a (possibly empty) sequence of nested let, letrec and letex expressions. This means that the sequence of *strdecs* needs an expression at the end of the resulting program. This expression for the last let, letrec or letex will always be **{}** and it is supplied in the translation of *program*.
- During translation of simple nested structures, we explicitly pass the variable *prefix* along with the translation functions. We consider this the global variable *prefix* in all the other translation rules. This shows how the variable is maintained. The prefix variable is not updated anywhere else during translation so this does not give any ambiguity.
- In the rule for structure replication in $\xi\epsilon_{strbind}$ we lookup all values in the environment having the prefix $longstrid + "."$ and add them to the environment while giving them the new prefix $strid + "."$. This rule does not produce any ξ-Calculus code, which explains the empty space before the first comma.

Env × Env × LongVId × Naked CeXL	Env	Conditions
$\epsilon_{packstruct}(E, E', longvid)$[| *strdec* ⟨ **;** ⟩ |] $\epsilon_{packstruct}(E, E', longvid)$[| *strdec* ⟨ **;** ⟩ *strdecs* |]	$\epsilon_{packstrdec}(E, E', longvid)$[| *strdec* |] $\epsilon_{packstruct}(E'', E', longvid)$[| *strdecs* |] where $\epsilon_{packstrdec}(E, E', longvid)$[| *strdec* |] = E''	
$\epsilon_{packstrdec}(E, E', longvid)$[| **structure** *strbind* |] $\epsilon_{packstrdec}(E, E', longvid)$[| *dec* |]	$\epsilon_{packstrbind}(E, E', longvid)$[| *strbind* |] $\epsilon_{packdec}(E, E', longvid)$[| *dec* |]	
$\epsilon_{packdec}(E, E', longvid)$ [| **val** *typms valbind* |]	$\epsilon_{packvalbind}(E, E', longvid)$[| *valbind* |]	
$\epsilon_{packdec}(E, E', longvid)$ [| **val** *typms* **rec** *valbind* |]	$\epsilon_{packvalbind}(E, E', longvid)$[| *valbind* |]	
$\epsilon_{packdec}(E, E', longvid)$[| **type** *typbind* |] $\epsilon_{packdec}(E, E', longvid)$[| **exception** *exbind* |] $\epsilon_{packdec}(E, E', longvid)$[| **datatype** *datbind* |]	$\epsilon_{packtypbind}(E, E', longvid)$[| *typbind* |] $\epsilon_{packexbind}(E, E', longvid)$[| *exbind* |] $\epsilon_{packdatbind}(E, E', longvid)$[| *datbind* |]	
$\epsilon_{packdec}((VE, TE, RE, \Delta), (VE', TE', RE', \Delta'),$ $longvid)$[| **datatype** *tycon* **=** **datatype** *longtycon* |]	$(VE, RE,$ $TE + \{longvid + vid \mapsto \theta\}, \Delta)$	$TE'(inn(longvid$ $+tycon) = \theta$
$\epsilon_{packdec}((VE, TE, RE, \Delta), (VE', TE', RE', \Delta'),$ $longvid)$[| **res** *vid* ⟨ **=** *crestr* ⟩ |]	$(VE, RE + \{longvid + vid \mapsto \xi\},$ $TE, \Delta)$	$RE'(inn(longvid$ $+vid) = \xi$
$\epsilon_{packstrbind}(E, E', longvid)$ [| *strid* **struct** *strdecs* **end** |]	$\epsilon_{packstruct}(E, E', longvid + strid + ".")$ [| *strdecs* |]	
$\epsilon_{packstrbind}((VE, RE, TE, \Delta),$ $(VE', RE', TE', \Delta'), longvid')$ [| *strid* **=** *longstrid* |]	$(VE + \{longvid' + strid + "." + longvid \mapsto (\sigma, is, vid)$ $\mid VE'(inn(longvid') + strid + "." + longvid) =$ $(\sigma, is, vid)\},$ $RE + \{longvid' + strid + "." + longvid \mapsto \xi \mid$ $RE'(inn(longvid') + strid + "." + longvid) = \xi\},$ $TE + \{longvid' + strid + "." + longtycon \mapsto \theta \mid$ $TE'(inn(longvid') + strid + "." + longtycon) =$ $\theta\},$ $\Delta)$	

Comments:

- The rules for $\epsilon_{packstruct}$ are given 2 environments. Those values declared in the second environment which are also declared in the structure are added to the first environment with the structure name as a prefix.
- In the rule for structure replication in $\epsilon_{packstrbind}$ we lookup all values in the second environment having the prefix $inn(longvid') + strid + "."$ and insert them in the first environment, giving them the new prefix $longvid' + strid + "."$.

Env × Env × LongVId × Naked CeXL	Env	Conditions
$\epsilon_{packvalbind}((VE, TE, RE, \Delta), E', longvid)$ [| pat = exp |]	$(VE + VE', RE, TE, \Delta)$ where $\epsilon_{packpat}(E', longvid)[\| pat \|] = VE'$	
$\epsilon_{packvalbind}((VE, TE, RE, \Delta), E', longvid)$ [| pat = exp **and** $valbind$ |]	$\epsilon_{packvalbind}((VE + VE', RE, TE, \Delta), E', longvid)[\| strdecs \|]$ where $\epsilon_{packpat}(E', longvid)[\| pat \|] = VE'$	
$\epsilon_{packtypbind}((VE, TE, RE, \Delta), (VE', TE', RE', \Delta'), longvid)[\| typms\ tycon = ty \|]$	$(VE, RE, TE + \{longvid + tycon \mapsto \theta\}, \Delta)$	$TE'(inn(longvid) + tycon) = \theta$
$\epsilon_{packtypbind}((VE, TE, RE, \Delta), (VE', TE', RE', \Delta'), longvid)[\| typms\ tycon = ty$ **and** $typbind \|]$	$\epsilon_{packtypbind}((VE, RE, TE + TE'', \Delta), (VE', TE', RE', \Delta'), longvid)[\| typbind \|]$ where $TE'' = \{longvid + tycon \mapsto \theta\}$	$TE'(inn(longvid) + tycon) = \theta$
$\epsilon_{packexbind}((VE, TE, RE, \Delta), (VE', TE', RE', \Delta'), longvid)[\| vid \langle$ **of** $ty \rangle \|]$	$(VE + \{longvid + vid \mapsto (\sigma, \mathbf{e}, vid')\}, RE, TE, \Delta)$	$VE'(inn(longvid) + vid) = (\sigma, \mathbf{e}, vid')$
$\epsilon_{packexbind}((VE, TE, RE, \Delta), (VE', TE', RE', \Delta'), longvid)[\| vid \langle$ **of** $ty \rangle \|]$	(VE, RE, TE, Δ)	$VE'(inn(longvid) + vid) \neq (\sigma, \mathbf{e}, vid')$
$\epsilon_{packexbind}((VE, TE, RE, \Delta), (VE', TE', RE', \Delta'), longvid')[\| vid = longvid \|]$	$(VE + \{longvid' + vid \mapsto (\sigma, \mathbf{e}, vid')\}, RE, TE, \Delta)$	$VE'(inn(longvid') + vid) = (\sigma, \mathbf{e}, vid')$
$\epsilon_{packexbind}((VE, TE, RE, \Delta), (VE', TE', RE', \Delta'), longvid')[\| vid = longvid \|]$	(VE, RE, TE, Δ)	$VE'(inn(longvid') + vid) \neq (\sigma, \mathbf{e}, vid')$
$\epsilon_{packexbind}((VE, TE, RE, \Delta), (VE', TE', RE', \Delta'), longvid)[\| vid \langle$ **of** $ty \rangle$ **and** $exbind \|]$	$\epsilon_{packexbind}((VE + VE'', RE, TE, \Delta), (VE', TE', RE', \Delta'), longvid)[\| exbind \|]$ where $VE'' = \{longvid + vid \mapsto (\sigma, \mathbf{e}, vid')\}$	$VE'(inn(longvid) + vid) = (\sigma, \mathbf{e}, vid')$
$\epsilon_{packexbind}((VE, TE, RE, \Delta), (VE', TE', RE', \Delta'), longvid)[\| vid \langle$ **of** $ty \rangle$ **and** $exbind \|]$	$\epsilon_{packexbind}((VE, RE, TE, \Delta), (VE', TE', RE', \Delta'), longvid)[\| exbind \|]$	$VE'(inn(longvid) + vid) \neq (\sigma, \mathbf{e}, vid')$
$\epsilon_{packexbind}((VE, TE, RE, \Delta), (VE', TE', RE', \Delta'), longvid')[\| vid = longvid$ **and** $exbind \|]$	$\epsilon_{packexbind}((VE + VE'', RE, TE, \Delta), (VE', TE', RE', \Delta'), longvid')[\| exbind \|]$ where $VE'' = \{longvid' + vid \mapsto (\sigma, \mathbf{e}, vid')\}$	$VE'(inn(longvid') + vid) = (\sigma, \mathbf{e}, vid')$
$\epsilon_{packexbind}((VE, TE, RE, \Delta), (VE', TE', RE', \Delta'), longvid')[\| vid = longvid$ **and** $exbind \|]$	$\epsilon_{packexbind}((VE, RE, TE, \Delta), (VE', TE', RE', \Delta'), longvid')[\| exbind \|]$	$VE'(inn(longvid') + vid) \neq (\sigma, \mathbf{e}, vid')$
$\epsilon_{packdatbind}((VE, TE, RE, \Delta), (VE', TE', RE', \Delta'), longvid)$ [| $typms\ tycon = conbind \|]$	$(VE + VE'', RE, TE + \{longvid + tycon \mapsto \theta\}, \Delta)$ where $\epsilon_{packconbind}(VE', longvid)[\| conbind \|] = VE''$	$TE'(inn(longvid) + tycon) = \theta$
$\epsilon_{packdatbind}((VE, TE, RE, \Delta), (VE', TE', RE', \Delta'), longvid)$ [| $typms\ tycon = conbind$ **and** $datbind \|]$	$\epsilon_{packdatbind}((VE + VE'', RE, TE + TE'', \Delta), E', longvid)[\| datbind \|]$ where $\epsilon_{packconbind}(VE', longvid)[\| conbind \|] = VE''$ and $TE'' = \{longvid + tycon \mapsto \theta\}$	$TE'(inn(longvid) + tycon) = \theta$
$\epsilon_{packconbind}(VE, longvid)[\| vid \langle$ **of** $ty \rangle \|]$	$\{longvid + vid \mapsto (\sigma, \mathbf{c}, vid')\}$	$VE(inn(longvid) + vid) = (\sigma, \mathbf{c}, vid')$
$\epsilon_{packconbind}(VE, longvid)[\| vid \langle$ **of** $ty \rangle \|]$	$\{\}$	$VE(inn(longvid) + vid) \neq (\sigma, \mathbf{c}, vid')$
$\epsilon_{packconbind}(VE, longvid)$ [| $vid \langle$ **of** $ty \rangle$ | $conbind \|]$	$\{longvid + vid \mapsto (\sigma, \mathbf{c}, vid')\} + \epsilon_{packconbind}(VE, longvid)[\| conbind \|]$	$VE(inn(longvid) + vid) = (\sigma, \mathbf{c}, vid')$
$\epsilon_{packconbind}(VE, longvid)$ [| $vid \langle$ **of** $ty \rangle$ | $conbind \|]$	$\epsilon_{packconbind}(VE, longvid)[\| conbind \|]$	$VE(inn(longvid) + vid) \neq (\sigma, \mathbf{c}, vid')$

Env × Env × LongVId × Naked CeXL	Env	Conditions
$\epsilon_{packatpat}(E, longvid)$[| _ |]	{}	
$\epsilon_{packatpat}(E, longvid)$[| *scon* |]	{}	
$\epsilon_{packatpat}((VE, RE, TE, \Delta), longvid)$[| *vid* |]	$\{longvid + vid \mapsto (\sigma, \mathbf{v}, vid')\}$	$VE(inn(longvid) + vid) = (\sigma, \mathbf{v}, vid')$
$\epsilon_{packatpat}((VE, RE, TE, \Delta), longvid)$[| *vid* |]	{}	$VE(inn(longvid) + vid) \neq (\sigma, \mathbf{v}, vid')$
$\epsilon_{packatpat}(E, longvid)$[| **{** *patrow* **}** |]	$\epsilon_{packpatrow}(E, longvid)$[| *patrow* |]	
$\epsilon_{packatpat}(E, longvid)$[| **{}** |]	{}	
$\epsilon_{packatpat}(E, longvid)$[| **(** *pat* **)** |]	$\epsilon_{packpat}(E, longvid)$[| *pat* |]	
$\epsilon_{packpat}(E, longvid)$[| *atpat* |]	$\epsilon_{packatpat}(E, longvid)$[|*atpat* |]	
$\epsilon_{packpat}(E, longvid')$[| *longvid atpat* |]	$\epsilon_{packatpat}(E, longvid')$[| *atpat* |]	
$\epsilon_{packpat}(E, longvid)$[| *pat* **:** *ty* |]	$\epsilon_{packpat}(E, longvid)$[| *pat* |]	
$\epsilon_{packpat}((VE, RE, TE, \Delta), longvid)$ [| *vid* ⟨ **:** *ty* ⟩ **as** *pat* |]	$\{longvid + vid \mapsto (\sigma, \mathbf{v}, vid')\}+$ $\epsilon_{packpat}((VE, RE, TE, \Delta), longvid)$[| *pat* |]	$VE(inn(longvid) + vid) = (\sigma, \mathbf{v}, vid')$
$\epsilon_{packpat}((VE, RE, TE, \Delta), longvid)$ [| *vid* ⟨ **:** *ty* ⟩ **as** *pat* |]	$\epsilon_{packpat}((VE, RE, TE, \Delta), longvid)$[| *pat* |]	$VE(inn(longvid) + vid) \neq (\sigma, \mathbf{v}, vid')$
$\epsilon_{packpatrow}(E, longvid)$[| *lab* = *pat* |]	$\epsilon_{packpat}(E, longvid)$[| *pat* |]	
$\epsilon_{packpatrow}(E, longvid)$[| *lab* ?= *pat* |]	$\epsilon_{packpat}(E, longvid)$[| *pat* |]	
$\epsilon_{packpatrow}(E, longvid)$ [| *lab* = *pat* **,** *patrow* |]	$\epsilon_{packpat}(E, longvid)$[| *pat* |]+ $\epsilon_{packpatrow}(E, longvid)$[| *patrow* |]	
$\epsilon_{packpatrow}(E, longvid)$ [| *lab* ?= *pat* **,** *patrow* |]	$\epsilon_{packpat}(E, longvid)$[| *pat* |]+ $\epsilon_{packpatrow}(E, longvid)$[| *patrow* |]	
$\epsilon_{packpatrow}(E, longvid)$[| ... = *pat* |]	$\epsilon_{packpat}(E)$[| *pat* |]	

Comments:

- Notice that we only pack variables if they occur in the environment VE marked as variables. If they are not marked as variables it means that they have been redeclared later in the structure as a constructor or an exception. In that case it will be packed appropriately by a later declaration in the structure.

  We do similarly for constructors and exceptions.

20 CeXL Lexical Syntax

This section describes the lexical part of the CeXL syntax.

20.1 Reserved Words and Symbols

The following is the list of reserved words in CeXL. They may not (except for `div`, `mod` and `o`) be used as identifiers:

```
and  as  datatype  do  op  exception  raise  handle
andalso  case  else  end  fun  fn  if  in  let  of
orelse  then  val  rec  while  fwhile  with  withtype
open  local  nonfix  infix  infixr  abstype
structure  struct  type  res  fieldcase  div  mod  o
```

The following is a list of reserved symbols in CeXL. They may not (except for the infix operators) be used as identifiers:

```
(  )  [  ]  {  }  ~{  ,  :  ;  ...  _  |  =  <>  :=
=>  ->  #  ::  @  ?  +  -  <  >  <=  >=  *  /  ^  ?=
```

20.2 Special Constants

We assume an underlying alphabet of N characters (N $\geq$ 256), numbered 0 to $N-1$, which agrees with the ASCII character set on the characters numbered 0 to 127. The interval $[0, N-1]$ is called the *ordinal range* of the alphabet. A *string constant* is a sequence, between quotes ("), of zero or more printable characters (i.e. numbered $33-126$), spaces or escape sequences. Each escape sequence starts with the escape character \, and stands for a character sequence. The escape sequences are:

`\a`	A single character interpreted by the system as alert (ASCII 7)
`\b`	Backspace (ASCII 8)
`\t`	Horizontal tab (ASCII 9)
`\n`	Linefeed, also known as newline (ASCII 10)
`\v`	Vertical tab (ASCII 11)
`\f`	Form feed (ASCII 12)
`\r`	Carriage return (ASCII 13)
`\` *ddd*	The single character with number *ddd* (3 decimal digits denoting an integer in the ordinal range of the alphabet).
`\u` *xxxx*	The single character with number *xxxx* (4 hexadecimal digits denoting an integer in the ordinal range of the alphabet).
`\"`	"
`\\`	\
\ $f \cdots f$ \	This sequence is ignored, where $f \cdots f$ stands for a sequence of one or more formatting characters.

The *formatting characters* (which are also referred to as *whitespace*) are the following subset of the non-printable characters: Space, tab, newline and formfeed. The last form allows long strings to be written on more than one line, by writing \ at the end of one line and at the start of the next.

A *character constant* is a sequence of the form `#`s, where s is a string constant denoting a string of size one character.

An *integer constant* (in *decimal notation*) is an optional negation symbol (`~`) followed by a non-empty sequence of decimal digits `0`,...,`9`. An *integer constant* (in *hexadecimal notation*) is an optional negation symbol (`~`) followed by `0x` followed by a non-empty sequence of hexadecimal digits `0`,...,`9` and `a`,...,`f`. (`A`,...,`F` may be used as alternatives for `a`,...,`f`.)

A *word constant* (in *decimal notation*) is `0w` followed by a non-empty sequence of decimal digits. A *word constant* (in *hexadecimal notation*) is `0wx` followed by a non-empty sequence of hexadecimal digits.

A *real constant* is an integer constant in decimal notation, possibly followed by a point (`.`) and one or more decimal digits, possibly followed by an exponent symbol (`e` or `E`) and an integer constant in decimal notation; at least one of the optional parts must occur, hence no integer constant is a real constant. Examples: `0.7`, `3.3E5` and `3e~7`. Non-examples: `23`, `.3`, `4.E5` and `1e2.0`.

Libraries may provide multiple numeric types and multiple string types. To each string type corresponds an alphabet with ordinal range $[0, N-1]$ for some $N \geq 256$; each alphabet must agree with the ASCII character set on the characters numbered 0 to 127. When multiple alphabets are supported, all characters of a given string constant are interpreted over the same alphabet. For each special constant, overloading resolution is used for determining the type of the constant. However we will not define overloading resolution in this version of the specification since it is currently not used.

We denote by SCon the class of *special constants*, i.e. the integer, real, word, character and string constants; we shall use *scon* to range over SCon.

20.3 Comments

A *comment* is any character sequence within comment brackets `(* *)` in which comment brackets are properly nested. No space is allowed between the two characters which make up a comment bracket `(*` or `*)`. An unmatched `(*` should be detected by the language implementation.

20.4 Identifiers

The classes of *identifiers* are the following:

InfVid	(infix identifiers)	
VId	(value identifiers)	Long
TyVar	(type variables)	
TyCon	(type constructors)	Long
Lab	(record labels)	
StrId	(structure identifiers)	Long

We use *vid*, *tycon*, *lab* etc. to range over VId, TyCon and Lab etc. For each class marked "Long" there is a class LongX of *long identifiers*; if x ranges over X then *longx* ranges over LongX.

The syntax of these long identifiers is given by the following:

longx	::=	x	identifier
		$strid_1.\cdots.strid_n.x$	qualified identifier ($n \geq 1$)

The qualified identifiers are for referring to identifiers declared within structures in CeXL. The class LongVId is extended to include identifiers of the form $strid_1.\cdots.strid_n.infvid$ ($n \geq 1$). This will allow identifiers such as `Word.+`, `Int.*` etc. to be used but not to be declared.

An *alphanumeric* identifier is either a letter or a prime (') followed by a sequence of letters, digits, primes or underscores (_). A *symbolic* identifier is any non-empty sequence of the following symbols:

```
!  %  &  $  #  +  -  /  :  <  =  >  ?  @  \  ~  `  ^  |  *
```

Identifiers are either *alphanumeric* or *symbolic* and in either case, reserved words and reserved symbols are excluded. This means that for example `#` and `|` are not identifiers, but `##` and `|=|` are identifiers. The only exception to this rule is that reserved words and symbols in the class InfVid of infix operators may be used as infix identifiers. These infix operators may also be used as part of a longvid identifier as already described. No infix operators may be rebound; this precludes any syntactic ambiguity.

The following are all infix operators with fixity and precedence specified:

infvid	Precedence	Fixity
`+`	6	Left
`-`	6	Left
`*`	7	Left
`/`	7	Left
`<`	4	Left
`>`	4	Left
`<=`	4	Left
`>=`	4	Left

infvid	Precedence	Fixity
`=`	4	Left
`<>`	4	Left
`::`	5	Right
`@`	5	Right
`^`	6	Left
`:=`	3	Left
`div`	7	Left
`mod`	7	Left
`o`	3	Left

All the infix operators are reserved words or reserved symbols as already described.

A type variable *tyvar* may be any alphanumeric identifier starting with a prime. The classes VId, TyCon and Lab are represented by identifiers not starting with a prime. None of these classes contain any of the infix operators. TyCon is extended to include the constructor `?` which is a reserved symbol. The class Lab is extended to include the numeric labels `1` `2` `3` $\cdots$, i.e. any numeral not starting with `0`. The identifier class StrId is represented by alphanumeric identifiers not starting with a prime.

TyVar is therefore disjoint from the other five classes and InfVid is disjoint from the other classes. Otherwise, the syntax class of an occurrence of identifier *id* in a CeXL phrase is determined by the rule that immediately before "." - i.e. in a long identifier, *id* is a structure identifier.

By means of the above rules and the given grammar an implementation can determine the class to which each identifier occurrence belongs. In this document we therefore assume that the classes are all disjoint.

20.5 Lexical Analysis

Each item of lexical analysis is either a reserved word, a reserved symbol, a special constant, an infix identifier, a (possibly long) identifier, a type variable, a (possibly long) type constructor, a label or a (possibly long) structure identifier. Comments and whitespace separate items (except within string constants) and are otherwise ignored. At each stage the longest next item is taken.

21 Full CeXL Grammar

This section presents the full grammar of the CeXL language with all its syntactical sugar.

21.1 Notational Conventions

The following conventions are used:

- The brackets $\langle\ \rangle$ encose optional phrases.
- For any class X (over which x ranges) we define the syntax class Xseq (over which xseq ranges) as follows:

xseq	::=	$(x_1, \cdots, x_n)$	(sequence, n $\geq$ 1)
			(empty sequence)
		x	(singleton sequence)

  (Note that the "$\cdots$" used here, a meta-symbol indicating syntactic repetition, must not be confused with "..." which is a reserved word of the language.)

- Alternative forms for each phrase class are in order of decreasing precedence. This precedence resolves ambiguity in parsing in the following way. Suppose that a phrase class - we take *exp* as an example - has two alternative forms F_1 and F_2, such that F_1 ends with an *exp* and F_2 starts with an *exp*. A specific case is

  F_1: **if** exp_1 **then** exp_2 **else** exp_3
  F_2: exp_1 **andalso** exp_2

  It will be enough to see how ambiguity is resolved in this specific case. Suppose that the lexical sequence

  $\cdots\ \cdots$ **if** $\cdots$ **then** $\cdots$ **else** *exp* **andalso** $\cdots\ \cdots$

  is to be parsed, where *exp* stands for a lexical sequence which is already determined as a subphrase (if necessary by applying the precedence rule). Then the higher precedence F_2 (in this case) dictates that *exp* associates to the right, i.e. that the correct parse takes the form

  $\cdots\ \cdots$ **if** $\cdots$ **then** $\cdots$ **else** (*exp* **andalso** $\cdots$) $\cdots$

  not the form

  $\cdots$ ($\cdots$ **if** $\cdots$ **then** $\cdots$ **else** *exp*) **andalso** $\cdots\ \cdots$

  Note particularly that the use of precedence does not decrease the class of admissible phrases; it merely rejects alternative ways of parsing certain phrases. In particular, the purpose is not to prevent a phrase, which is an instance of a form with higher precedence, having a constituent which is an instance of a form with lower precedence. Thus for example

$\cdots$ **andalso if** $\cdots$ **then** $\cdots$ **else** $\cdots$ $\cdots$

is quite permissible, and will be parsed as

$\cdots$ **andalso** (**if** $\cdots$ **then** $\cdots$ **else** $\cdots$) $\cdots$

- L (respectively R) means left (respectively right) association.
- The syntax of types binds more tightly than that of expressions.
- The syntax of restrictions binds more tightly than that of types.
- Each iterated construct (e.g. *match* ...) extends as far right as possible; thus, parentheses may be needed around an expression which terminates with a match, e.g. "fn *match*", if this occurs within a larger match.

21.2 The Grammar Productions

Patterns

atpat	::=	_	wildcard
		scon	special constant
		longvid	value identifier
		{ ⟨ *patrow* ⟩ **}**	record
		()	0-tuple
		(pat_1 **,** $\cdots$ **,** pat_n)	n-tuple, $n \geq 2$
		[pat_1 **,** $\cdots$ **,** pat_n]	list, $n \geq 0$
		(*pat*)	
pat	::=	*atpat*	atomic
		longvid atpat	constructed pattern
		pat_1 *infvid* pat_2	infixed value construction
		pat **:** *ty*	typed
		vid ⟨ **:** *ty* ⟩ **as** *pat*	layered
patrow	::=	*lab* **=** *pat* ⟨ **,** *patrow* ⟩	present field
		lab **?=** *pat* ⟨ **,** *patrow* ⟩	optional field
		vid ⟨ **:** *ty* ⟩ ⟨ **as** *pat* ⟩ ⟨ **,** *patrow* ⟩	label as variable
		... ⟨ **=** *pat* ⟩	optional at end of row

Expressions and Matches

atexp	::=	*scon*	special constant
		longvid	value identifier
		op *infvid*	infix value as identifier
		{ ⟨ *exprow* ⟩ **}**	record
		# *lab*	record field selector
		()	0-tuple
		(exp_1 **,** $\cdots$ **,** exp_n **)**	n-tuple, $n \geq 2$
		[exp_1 **,** $\cdots$ **,** exp_n **]**	list, $n \geq 0$
		(exp_1 **;** $\cdots$ **;** exp_n **)**	sequence, $n \geq 2$
		let ⟨ *decs* ⟩ **in** exp_1 **;** $\cdots$ **;** exp_n **end**	local declaration, $n \geq 1$
		fieldcase *exp* **in** *tyvar* **of** *fieldmatch* **type** *ty* **end**	case on optional field
		(*exp* **)**	
appexp	::=	*atexp*	
		appexp atexp	application expression (L)
infexp	::=	*appexp*	
		$infexp_1$ *infvid* $infexp_2$	infixed application
exp	::=	*infexp*	
		exp **:** *ty*	typed (L)
		exp **handle** *match*	handle exception
		exp_1 **andalso** exp_2	conjunction
		exp_1 **orelse** exp_2	disjunction
		raise *exp*	raise exception
		if exp_1 **then** exp_2 **else** exp_3	conditional
		fwhile *exp* **do** ⟨ *decs* ⟩ ⟨ **in** ⟨ *iterbind* ⟩ ⟩ **end**	functional iteration
		case *exp* **of** *match*	case analysis
		fn *match*	function
exprow	::=	*lab* **=** *exp* ⟨ **,** *exprow* ⟩	present field
		lab `?=` *exp* ⟨ **,** *exprow* ⟩	optional field
		... **=** *exp*	optional at end of row
iterbind	::=	*vid* **=** *exp* ⟨ **,** *iterbind* ⟩	bind for iteration
fieldmatch	::=	`absent =>` exp_1 `|` `present` *atpat* `=>` exp_2	match optional field
match	::=	*mrule* ⟨ `|` *match* ⟩	
mrule	::=	*pat* `=>` *exp*	

Declarations

decs	::=	*dec* ⟨ **;** ⟩ ⟨ *decs* ⟩	sequential declaration
dec	::=	**val** *typms valbind*	value declaration
		val *typms* **rec** *valbind*	recursive value declaration
		fun *typms fvalbind*	function declaration
		res *vid* ⟨ = *crestr* ⟩	restriction declaration
		type *typbind*	type declaration
		datatype *datbind*	datatype declaration
		datatype *tycon* = **datatype** *longtycon*	datatype replication
		exception *exbind*	exception declaration
valbind	::=	*pat* = *exp* ⟨ **and** *valbind* ⟩	
fvalbind	::=	$vid\ atpat_{11} \cdots atpat_{1n}$ ⟨ **:** *ty* ⟩ = exp_1	$m, n \geq 1$
		| $vid\ atpat_{21} \cdots atpat_{2n}$ ⟨ **:** *ty* ⟩ = exp_2	
		⋮ ⋮	
		| $vid\ atpat_{m1} \cdots atpat_{mn}$ ⟨ **:** *ty* ⟩ = exp_m	
		⟨ **and** *fvalbind* ⟩	
typbind	::=	*typms tycon* = *ty* ⟨ **and** *typbind* ⟩	
datbind	::=	*typms tycon* = *conbind* ⟨ **and** *datbind* ⟩	
conbind	::=	*vid* ⟨ **of** *ty* ⟩ ⟨ | *conbind* ⟩	
exbind	::=	*vid* ⟨ **of** *ty* ⟩ ⟨ **and** *exbind* ⟩	
		vid = *longvid* ⟨ **and** *exbind* ⟩	

Simple Nested Structures

program	::=	*strdecs*	a CeXL program
strdecs	::=	*strdec* ⟨ **;** ⟩ ⟨ *strdecs* ⟩	sequential structure declaration
strdec	::=	*dec*	declaration
		structure *strbind*	structure
strbind	::=	*strid* = **struct** ⟨ *strdecs* ⟩ **end**	structure binding
		strid = *longstrid*	structure replication

Type Parameters

typms	::=	[*typarams*]	possibly restricted type parameters
		tyvarseq	unrestricted type parameters
typarams	::=	*tyvars* ⟨ **:** *crestr* ⟩ ⟨ **,** *typarams* ⟩	type parameters
tyvars	::=	*tyvar* ⟨ **;** *tyvars* ⟩	type variables

Types

tyargs	::=	*tyvar* **=** *ty* ⟨ **,** *tyargs* ⟩	type arguments
fieldstatus	::=	**-**	absent field
		tyvar	optional field
			present field
tyrow	::=	*fieldstatus lab* **:** *ty* ⟨ **,** *tyrow* ⟩	row type
		... **:** *tyvar*	optional at end of row
ty	::=	*tyvar*	type variable
		{ ⟨ *tyrow* ⟩ **}**	record type
		[*tyargs*] *longtycon*	type construction
		tyseq longtycon	tupled construction
		ty_1 ***** $\cdots$ ***** ty_n	tuple type, $n \geq 2$
		ty `->` *ty'*	function type expression (R)
		(*ty* **)**	

Restrictions

crestr	::=	*restr* ⟨ `+` *crestr* ⟩	combination of restrictions
restr	::=	∼**{** ⟨ *labels* ⟩ **}**	forbidden fields
		typats	type patterns
		res *longvid*	use declared restriction
labels	::=	*lab* ⟨ **,** *labels* ⟩	set of labels
typats	::=	*typat* ⟨ `|` *typats* ⟩	type patterns
typat	::=	⟨ [*typatargs*] ⟩ *longtycon*	type constructor pattern
		($typat_1$ **,** $\cdots$ **,** $typat_n$ **)** *longtycon*	tupled constructor pattern, $n \geq 1$
typatargs	::=	*tyvar* **=** *typat* ⟨ **,** *typatargs* ⟩	type pattern arguments

Comments:

- Instead of making the type parameters [*typarams*] optional in the grammar production for *typms* (as was done in the Naked CeXL grammar), the case where there are no type parameters is now encompassed by the *tyvarseq* production of the grammar.

  The same happens in *ty* for [*tyargs*] *longtycon* and *tyseq longtycon*.

  The grammar for *typat* in the restrictions is different, in that it always requires the parentheses to be present when using the tupled notation - even when there is only one *typat* argument.

21.3 Syntactic Limitations

The syntactic limitations are as for Naked CeXL:

- No expression row, pattern row or type-expression row may bind the same *lab* twice.
- No restriction *labels* may contain the same *lab* twice.
- No binding *valbind*, *typbind*, *datbind* or *exbind* may bind the same identifier twice; this also applies to value identifiers within a *datbind*.
- No *tyvarseq* may contain the same *tyvar* twice.
- No *typarams*, *tyargs* or *typatargs* may bind the same *tyvar* twice at the same nesting level. That is, a *tyvar* occuring nested is unrelated to any other *tyvar*, so the limitation only applies to *tyvar*s immediately in the same *typarams*, *tyargs* or *typatargs*.
- For each value binding *pat* = *exp* within **val rec**, *exp* must be of the form **fn** *match*. The derived form of function-value binding given in section 22 necessarily obeys this restriction.
- No *datbind*, *valbind* or *exbind* may bind `true`, `false`, `nil`, `ref`, `div`, `mod`, `o`, `absent` or `present`. No *datbind* or *exbind* may bind `it`.
- No *datbind* or *typbind* may bind the type constructor `?`.
- No *exbind* may bind `Match` or `Bind`.
- No *exbind* which occur directly at top-level or in a structure may contain type variables. Hence, only *exbind* within the *decs* part of a let-expression may contain type variables.
- No *datbind* or *typbind* may refer to type variables other than those mentioned in the parameter list *typarams* in the beginning of the *datbind* or *typbind*.
- No real constant may occur in a pattern.
- The infix identifier `=` may not occur in a pattern.
- Any *tyrow* in a record type containing **...** `:` *tyvar* at the end must also contain at least one field.
- Any *patrow* in a record pattern containing **...** ⟨ = *pat* ⟩ at the end must also contain at least one field.
- Any *exprow* in a record expression containing **...** = *exp* at the end must also contain at least one field.

21.4 Displaying Types to the User

Types displayed to the user should generally:

- Have all records flattened. This is also the only syntax supported by the CeXL grammar
- Records which are really tuples should be displayed as such
- Type quantifications which are for unnamed instantiation (i.e. having type variables called `'a`, `'b`, `'c` etc.) without restrictions on the type variables should be displayed using the tupled construction notation.

  This will not always be compatible with Standard ML notation but it will be compatible those places where the conventions of using `'a`, `'b`, `'c` (and in that order!) for type parameter names are followed

22 CeXL To Naked CeXL Translation

Several derived forms are provided in CeXL. They are presented in the following tables. Each derived form is given with its equivalent form. Thus each row of the tables should be considered as a rewriting rule

CeXL $\Rightarrow$ Naked CeXL

and these rules must be applied repeatedly to a phrase until it is transformed into a phrase of Naked CeXL.

During the translation of the rules the translation is assumed to be done by maintaining the grammatical grouping of the parsed CeXL syntax. So for instance the translation of the following singleton list:

[`5 :: nil`]

is reduced to

(`5 :: nil`) `:: nil`

with the parentheses indicating syntactic grouping.

The following comments apply to specific translation rules:

- During the translation, infix identifiers *infvid* will be placed in Naked CeXL where usually only long identifiers *longvid* are allowed. They should be treated in Naked CeXL as if they were simply long identifiers *longvid*.
- In the derived forms for tuples, in terms of records, we use $\bar{n}$ to mean the CeXL numeral which stands for the natural number n.
- In the derived forms for type arguments expressed in tuple notation, we use $\hat{n}$ to mean the CeXL type variable which is the nth element in the lexicographical order of type variables:

  `'a,'b,`$\cdots$`,'z,'aa,'ab,`$\cdots$`,'az,'ba,'bb,'bc,`$\cdots$`,'zz,'aaa,'aab,`$\cdots$
- A new phrase class of function-value bindings is introduced, accompanied by a new declaration form **fun** *fvalbind*. The mixed forms **val rec** *fvalbind*, **val** *fvalbind* and **fun** *valbind* are not allowed - though the first form arises during the translation into Naked CeXL.

22.1 Atomic Expressions

CeXL *atexp*	Naked CeXL	Conditions
op *infvid*	*infvid*	
# *lab*	**fn** { *lab* = *vid* , **...** } `=>` *vid*	fresh *vid*
( )	{}	
(exp_1 , $\cdots$, exp_n)	{ `1` = exp_1 , $\cdots$, $\bar{n}$ = exp_n }	$n \geq 2$
[exp_1 , $\cdots$, exp_n]	exp_1 `::` $\cdots$ `::` exp_n `:: nil`	$n \geq 0$
(exp_1 ; $\cdots$; exp_n ; *exp*)	**case** exp_1 **of** (_) `=>`	$n \geq 1$
	$\vdots$	
	case exp_n **of** (_) `=>` *exp*	

22.2 Infix Expressions

CeXL *infexp*	Naked CeXL	Conditions
$infexp_1$ *infvid* $infexp_2$	*infvid* ($infexp_1$, $infexp_2$)	

22.3 Expressions

CeXL *exp*	Naked CeXL	Conditions
if exp_1 **then** exp_2 **else** exp_3	**case** exp_1 **of** `true =>` exp_2 `| false =>` exp_3	
exp_1 **orelse** exp_2	**if** exp_1 **then** `true` **else** exp_2	
exp_1 **andalso** exp_2	**if** exp_1 **then** exp_2 **else** `false`	
let ⟨ *decs* ⟩ **in** exp_1 ; ⋯ ; exp_n **end**	**let** ⟨ *decs* ⟩ **in** (exp_1 ; ⋯ ; exp_n) **end**	$n \geq 2$
fwhile *exp* **do** ⟨ *decs* ⟩ ⟨ **in** ⟩ **end**	**let val rec** *vid* = **fn** () `=>` **if** *exp* **then let** ⟨ *decs* ⟩ **in** *vid*() **end else** () **in** *vid*() **end**	fresh *vid*
fwhile *exp* **do** ⟨ *decs* ⟩ **in** $vid_1 = exp_1, \cdots, vid_n = exp_n$ **end**	**let val rec** *vid* = **fn** { $vid_1, \cdots, vid_n$ } `=>` **if** *exp* **then let** ⟨ *decs* ⟩ **in** *vid* { $vid_1 = exp_1, \cdots, vid_n = exp_n$ } **end** **else** { $vid_1 = vid_1, \cdots, vid_n = vid_n$ } **in** *vid* { $vid_1 = vid_1, \cdots, vid_n = vid_n$ } **end**	$n \geq 1$ fresh *vid*

22.4 Atomic Patterns

CeXL *atpat*	Naked CeXL	Conditions
( )	{}	
(pat_1 , ⋯ , pat_n)	{ `1` = pat_1 , ⋯ , $\bar{n}$ = pat_n }	$n \geq 2$
[pat_1 , ⋯ , pat_n]	pat_1 `::` ⋯ `::` pat_n `::` `nil`	$n \geq 0$

22.5 Patterns

CeXL *pat*	Naked CeXL	Conditions
$infpat_1$ *infvid* $infpat_2$	*infvid* ($infpat_1$, $infpat_2$)	

22.6 Pattern Rows

CeXL *patrow*	Naked CeXL	Conditions
vid ⟨ **:** *ty* ⟩ ⟨ **as** *pat* ⟩ ⟨ , *patrow* ⟩	*vid* = *vid* ⟨ **:** *ty* ⟩ ⟨ **as** *pat* ⟩ ⟨ , *patrow* ⟩	

22.7 Declarations

CeXL *dec*	Naked CeXL	Conditions
fun *typms fvalbind*	**val** *typms* **rec** *fvalbind*	

22.8 Type Parameters

CeXL *typms*	Naked CeXL	Conditions
tyvar	[*tyvar*]	
($tyvar_1$, $\cdots$, $tyvar_n$)	[$tyvar_1$, $\cdots$, $tyvar_n$]	$n \geq 1$

22.9 Function-value Bindings

CeXL *fvalbind*	Naked CeXL	Conditions
$vid\ atpat_{11} \cdots atpat_{1n}\ \langle$ `:` $ty\ \rangle = exp_1$ `|` $vid\ atpat_{21} \cdots atpat_{2n}\ \langle$ `:` $ty\ \rangle = exp_2$ $\vdots$ $\quad$ $\vdots$ `|` $vid\ atpat_{m1} \cdots atpat_{mn}\ \langle$`:`$ty\rangle = exp_m$ $\langle$ **and** *fvalbind* $\rangle$	vid = **fn** vid_1 `=>` $\cdots$ **fn** vid_n `=>` **case** (vid_1 , $\cdots$, vid_n) **of** $(atpat_{11},\cdots,atpat_{1n})$ `=>` $exp_1\ \langle$ `:` $ty\ \rangle$ `|` $(atpat_{21},\cdots,atpat_{2n})$ `=>` $exp_2\ \langle$ `:` $ty\ \rangle$ $\vdots$ $\quad$ $\vdots$ `|` $(atpat_{m1},\cdots,atpat_{mn})$ `=>` $exp_m\ \langle$`:`$ty\rangle$ $\langle$ **and** *fvalbind* $\rangle$	$m, n \geq 1$ fresh and distinct $vid_1, \cdots, vid_n$

22.10 Type Expressions

CeXL *ty*	Naked CeXL	Conditions
ty longtycon	[`'a` = *ty*] *longtycon*	
(ty_1 , $\cdots$, ty_n) *longtycon*	[`'a` = ty_1 , $\cdots$, $\hat{n}$ = ty_n] *longtycon*	$n \geq 1$
ty_1 `*` $\cdots$ `*` ty_n	`{ 1 :` ty_1 , $\cdots$, $\bar{n}$ `:` ty_n `}`	$n \geq 2$

22.11 Type Pattern in Restrictions

CeXL *typat*	Naked CeXL	Conditions
($typat_1$, $\cdots$, $typat_n$) *longtycon*	[`'a` = $typat_1$, $\cdots$, $\hat{n}$ = $typat_n$] *longtycon*	$n \geq 1$

23 Initial Environments

Certain semantic objects are expected to be present initially for CeXL programs. This includes predefined operators, contructors, exceptions and so forth. Such semantic objects are required both for the static semantics and for the dynamic semantics. We define most of the semantic objects either in ξ-Calculus or by description, since not all the semantic objects can be created directly using CeXL notation. All names for identifiers, type constructors and the likes will be as they are in CeXL though, since this is necessary.

23.1 Environment for Static Semantics of ξ-Calculus

The initial environment Δ of type variables bound to restrictions in the static semantics will be called Δ_{sta} and is defined to be empty: $\Delta_{sta} = \{\}$.

The initial environment Γ of variables bound to type schemes in the static semantics will be called Γ_{sta} and is defined to contain:

vid	$\mapsto$	σ
`!`	$\mapsto$	$\forall$['a:: $\circ$].['a='a:: $\circ$]*ref*{*ref*} $\rightarrow$'a:: $\circ$
`:=`	$\mapsto$	$\forall$['a:: $\circ$].{1 : ['a='a:: $\circ$]*ref*{*ref*}, 2 :'a:: $\circ$; {}} $\rightarrow$ {}
`@`	$\mapsto$	$\forall$['a:: $\circ$].{1 : ['a='a:: $\circ$]*list*{*nil,::*}, 2 : ['a='a:: $\circ$]*list*{*nil,::*}; {}} $\rightarrow$ ['a='a:: $\circ$]*list*{*nil,::*}
`absent`	$\mapsto$	$\forall$['a:: $\circ$].[]*absent*{*absent*} ? 'a:: $\circ$
`present`	$\mapsto$	$\forall$['a:: $\circ$].'a:: $\circ$ $\rightarrow$ ([]*present*{*present*} ? 'a:: $\circ$)
`=`	$\mapsto$	$\forall$['a:: ξ_{eq}].{1 :'a:: ξ_{eq}, 2 :'a:: ξ_{eq}; {}} $\rightarrow$ []*bool*{*true, false*}
`<>`	$\mapsto$	$\forall$['a:: ξ_{eq}].{1 :'a:: ξ_{eq}, 2 :'a:: ξ_{eq}; {}} $\rightarrow$ []*bool*{*true, false*}
`+`	$\mapsto$	$\forall$['a:: ξ_{num}].{1 :'a:: ξ_{num}, 2 :'a:: ξ_{num}; {}} $\rightarrow$ 'a:: ξ_{num}
`-`	$\mapsto$	$\forall$['a:: ξ_{num}].{1 :'a:: ξ_{num}, 2 :'a:: ξ_{num}; {}} $\rightarrow$ 'a:: ξ_{num}
`*`	$\mapsto$	$\forall$['a:: ξ_{num}].{1 :'a:: ξ_{num}, 2 :'a:: ξ_{num}; {}} $\rightarrow$ 'a:: ξ_{num}
`/`	$\mapsto$	$\forall$[].{1 : []*real*{$\cdots$}, 2 : []*real*{$\cdots$}; {}} $\rightarrow$ []*real*{$\cdots$}
`<`	$\mapsto$	$\forall$['a:: ξ_{numtxt}].{1 :'a:: ξ_{numtxt}, 2 :'a:: ξ_{numtxt}; {}} $\rightarrow$ []*bool*{*true, false*}
`>`	$\mapsto$	$\forall$['a:: ξ_{numtxt}].{1 :'a:: ξ_{numtxt}, 2 :'a:: ξ_{numtxt}; {}} $\rightarrow$ []*bool*{*true, false*}
`<=`	$\mapsto$	$\forall$['a:: ξ_{numtxt}].{1 :'a:: ξ_{numtxt}, 2 :'a:: ξ_{numtxt}; {}} $\rightarrow$ []*bool*{*true, false*}
`>=`	$\mapsto$	$\forall$['a:: ξ_{numtxt}].{1 :'a:: ξ_{numtxt}, 2 :'a:: ξ_{numtxt}; {}} $\rightarrow$ []*bool*{*true, false*}
`~`	$\mapsto$	$\forall$['a:: $\xi_{realint}$].'a:: $\xi_{realint}$ $\rightarrow$ 'a:: $\xi_{realint}$
`^`	$\mapsto$	$\forall$[].{1 : []*string*{$\cdots$}, 2 : []*string*{$\cdots$}; {}} $\rightarrow$ []*string*{$\cdots$}
`o`	$\mapsto$	$\forall$['a:: $\circ$,'b:: $\circ$,'c:: $\circ$].{1 :'a:: $\circ$ $\rightarrow$'b:: $\circ$, 2 :'c:: $\circ$ $\rightarrow$'a:: $\circ$; {}} $\rightarrow$'c:: $\circ$ $\rightarrow$'b:: $\circ$
`mod`	$\mapsto$	$\forall$['a:: $\xi_{wordint}$].{1 :'a:: $\xi_{wordint}$, 2 :'a:: $\xi_{wordint}$; {}} $\rightarrow$ 'a:: $\xi_{wordint}$
`div`	$\mapsto$	$\forall$['a:: $\xi_{wordint}$].{1 :'a:: $\xi_{wordint}$, 2 :'a:: $\xi_{wordint}$; {}} $\rightarrow$ 'a:: $\xi_{wordint}$
`abs`	$\mapsto$	$\forall$['a:: $\xi_{realint}$].'a:: $\xi_{realint}$ $\rightarrow$ 'a:: $\xi_{realint}$

Where we have used the following restrictions:

$\xi_{realint}$	=	[]*real*{$\cdots$} / []*int*{$\cdots$}
$\xi_{wordint}$	=	[]*word*{$\cdots$} / []*int*{$\cdots$}
ξ_{num}	=	[]*word*{$\cdots$} / []*real*{$\cdots$} / []*int*{$\cdots$}
ξ_{numtxt}	=	[]*word*{$\cdots$} / []*real*{$\cdots$} / []*int*{$\cdots$} / []*string*{$\cdots$} / []*char*{$\cdots$}
ξ_{eq}	=	[]*word*{$\cdots$} / []*int*{$\cdots$} / []*string*{$\cdots$} / []*char*{$\cdots$}

23.2 Environment for Dynamic Semantics of ξ-Calculus

The initial state s in the dynamic semantics is called s_{dyn} and is the same as (mem_{dyn}, ens_{dyn}). The exception names ens_{dyn} of this state are the exception names *Match* and *Bind*.

The initial environment Γ of variables bound to values in the dynamic semantics will be called Γ_{dyn} and is defined to contain:

vid	$\mapsto$	v	Class of v
`:=`	$\mapsto$	$:=$	$\{:=\}$
`absent`	$\mapsto$	$(absent, \{\})$	FieldVal
`present`	$\mapsto$	$(x, (present : \sigma_{pre})\ ?\ x, \{\}, \{\})$	FcnClosure
`Match`	$\mapsto$	*Match*	ExVal
`Bind`	$\mapsto$	*Bind*	ExVal

Here σ_{pre} is as defined earlier: $\sigma_{pre} = \forall[].[]present\{present\}$. Furthermore, Γ_{dyn} contains value bindings defining the following functions:

- The operators `=`, `<>`, `+`, `-`, `*`, `/`, `<`, `>`, `<=`, `>=` are defined as one would expect.
- `~` negates a real or an integer.
- `^` concatenates two strings.
- `abs` takes the absolute value of a real or an integer.
- `mod` takes modulus of an integer or a word.
- `div` makes integer division of an integer or a word.
- The operators `o`, `!` and `@` are defined by the equivalent of this CeXL code:

```
fun ! (ref v) = v
fun o (f, g) = fn x => f(g(x))
fun @ (x::xs, l) = x::(@ (xs, l))
  | @ (nil  , l) = l
```

We will not go through any more details here, since it is outside the scope of this definition. A basis library definition is appropriate for defining the operators in more detail.

23.3 Environment for Naked CeXL To ξ-Calculus Translation

The initial value environment VE of variables bound to values is called VE_{trans}. It is defined to contain:

longvid	$\mapsto$	(σ, is, vid)
`ref`	$\mapsto$	$(\forall[\texttt{'a}:: \circ].\texttt{'a} :: \circ \rightarrow [\texttt{'a='a}:: \circ]ref\{ref\}, \mathbf{c}, ref)$
`!`	$\mapsto$	$(\sigma_{unit}, \mathbf{v}, !)$
`:=`	$\mapsto$	$(\sigma_{unit}, \mathbf{v}, :=)$
`nil`	$\mapsto$	$(\forall[\texttt{'a}:: \circ].[\texttt{'a='a}:: \circ]list\{nil,::\}, \mathbf{c}, nil)$
`::`	$\mapsto$	$(\forall[\texttt{'a}:: \circ].\{\texttt{1} : \texttt{'a}:: \circ, \texttt{2} : [\texttt{'a='a}:: \circ]list\{nil,::\}; \{\}\} \rightarrow [\texttt{'a='a}:: \circ]list\{nil,::\}, \mathbf{c}, ::)$
`@`	$\mapsto$	$(\sigma_{unit}, \mathbf{v}, @)$
`true`	$\mapsto$	$(\forall[].[]bool\{true,false\}, \mathbf{c}, true)$
`false`	$\mapsto$	$(\forall[].[]bool\{true,false\}, \mathbf{c}, false)$
`absent`	$\mapsto$	$(\sigma_{unit}, \mathbf{v}, absent)$
`present`	$\mapsto$	$(\sigma_{unit}, \mathbf{v}, present)$
`Match`	$\mapsto$	$(\forall[].[]exn\{\}, \mathbf{e}, Match)$
`Bind`	$\mapsto$	$(\forall[].[]exn\{\}, \mathbf{e}, Bind)$
`=`	$\mapsto$	$(\sigma_{unit}, \mathbf{v}, =)$
`<>`	$\mapsto$	$(\sigma_{unit}, \mathbf{v}, \texttt{<>})$
`+`	$\mapsto$	$(\sigma_{unit}, \mathbf{v}, +)$
`-`	$\mapsto$	$(\sigma_{unit}, \mathbf{v}, -)$
`*`	$\mapsto$	$(\sigma_{unit}, \mathbf{v}, *)$
`/`	$\mapsto$	$(\sigma_{unit}, \mathbf{v}, /)$
`<`	$\mapsto$	$(\sigma_{unit}, \mathbf{v}, <)$
`>`	$\mapsto$	$(\sigma_{unit}, \mathbf{v}, >)$
`<=`	$\mapsto$	$(\sigma_{unit}, \mathbf{v}, \texttt{<=})$
`>=`	$\mapsto$	$(\sigma_{unit}, \mathbf{v}, \texttt{>=})$
`~`	$\mapsto$	$(\sigma_{unit}, \mathbf{v}, \sim)$
`^`	$\mapsto$	$(\sigma_{unit}, \mathbf{v}, \hat{})$
`o`	$\mapsto$	$(\sigma_{unit}, \mathbf{v}, o)$
`mod`	$\mapsto$	$(\sigma_{unit}, \mathbf{v}, mod)$
`div`	$\mapsto$	$(\sigma_{unit}, \mathbf{v}, div)$
`abs`	$\mapsto$	$(\sigma_{unit}, \mathbf{v}, abs)$

Here we have used the type scheme $\sigma_{unit} = \forall[].\{\}$ which is used as a dummy type scheme for variables. An important thing to notice here is that `absent` and `present` are variables - not constructors. They are for creating field values, not just constructor values.

The initial restriction environment RE of variables bound to restrictions is called RE_{trans}. The initial type variable restriction environment Δ of type variables bound to restrictions is called Δ_{trans}. They are both defined to be empty:

$$RE_{trans} = \{\}$$
$$\Delta_{trans} = \{\}.$$

The initial type environment TE of type constructors bound to type functions is called TE_{trans} and is defined to contain:

longtycon	$\mapsto$	θ
`unit`	$\mapsto$	$\Lambda[].\{\}$
`bool`	$\mapsto$	$\Lambda[].[]bool\{true,false\}$
`absent`	$\mapsto$	$\Lambda[].[]absent\{absent\}$
`present`	$\mapsto$	$\Lambda[].[]present\{present\}$
`int`	$\mapsto$	$\Lambda[].[]int\{\cdots\}$
`word`	$\mapsto$	$\Lambda[].[]word\{\cdots\}$
`real`	$\mapsto$	$\Lambda[].[]real\{\cdots\}$
`char`	$\mapsto$	$\Lambda[].[]char\{\cdots\}$
`string`	$\mapsto$	$\Lambda[].[]string\{\cdots\}$
`list`	$\mapsto$	$\Lambda[$`'a`$::\circ].[$`'a=''a`$::\circ]list\{nil,::\}$
`ref`	$\mapsto$	$\Lambda[$`'a`$::\circ].[$`'a=''a`$::\circ]ref\{ref\}$
`exn`	$\mapsto$	$\Lambda[].[]exn\{\}$

The initial environment E is called E_{trans} and is defined as:

$$E_{trans} = (VE_{trans}, RE_{trans}, TE_{trans}, \Delta_{trans})$$

The initial global set N of declared constructor names is called N_{trans} and we define:

$$N_{trans} = \{true, false, nil, ::, ref, absent, present\}.$$

The initial global set ν of declared type names is called ν_{trans} and we define:

$$\nu_{trans} = \{bool, int, real, word, string, char, list, ref, absent, present, exn, \texttt{?}\}.$$

23.4 Extending to a Complete Basis Library

Extending the initial environments to a complete basis library in the style of the Standard ML Basis Library is outside the scope of this definition. However, the following must be realized:

- Operators like `Word.+`, `Int.*` etc. cannot be declared with the CeXL language, so these would have to be predeclared somehow by an implementation.
- If structures like `Word32` and `Real64` are to be added they must extend the overloading of built-in operators like `+` and `mod` appropriately.
- In some of the examples of this document, we rely on the function `print` of type `string -> unit` to be defined. `print` is for printing the given string on the standard output stream.

24 Type-check and Execution of CeXL Programs

A CeXL program is represented by the phrase class *program* in the full CeXL grammar of section 21. Such a program is reduced to a Naked CeXL program using the rules of section 22.

If this succeeds without errors, the resulting Naked CeXL program is then translated to ξ-Calculus by the rules presented in section 19 using the initial environments of section 23.3. Assuming that the Naked CeXL program is called *program*, the translation corresponds to the following rule:

$$\xi_{program}(E_{trans},"")[|\ program\ |] = e \tag{189}$$

If the translation to ξ-Calculus is done without errors, e will be the resulting ξ-Calculus program. The type-check is done on ξ-Calculus using the rules presented in sections 14, 15 and 16 with the initial environment from section 23.1. This corresponds to the following rule:

$$\Gamma_{sta}; \Delta_{sta} \vdash e \Rightarrow \tau \tag{190}$$

If this type-inference completes without problems, we now have a valid well-typed ξ-Calculus program. τ will always be the type $\{\}$ and is not used. This ξ-Calculus program e may now be executed according to the dynamic semantics in section 17 using the initial environment from section 23.2. This corresponds to the rule:

$$\Gamma_{dyn} \vdash e \Rightarrow v \tag{191}$$

The v is always the value of the empty record and is not used. This completes the type-check and execution of CeXL programs and thus also this language specification.

25 A Few Regression Tests

This section shows some suitable regression tests for testing a CeXL implementation. It is far from enough for being a complete test and a more complete set of tests is included in the implementation made available online. However, at least some of the more tricky issues are tested here - including issues that common programs might not trigger.

There are both illegal programs which are supposed to trigger a type error and legal programs which are supposed to pass the type checker and run correctly.

The small example programs already mentioned various places in this document should also be included in a regression test.

25.1 Demonstrating Value Restriction

Legal Test 1

```
(* A legal program which demonstrates the value restriction *)
fun f a b = b

(* This is illegal and does not give problems with the value restriction *)
val g : string -> string = f 5
```

Illegal Test 1

```
(* An illegal program which demonstrates the value restriction *)
fun f a b = b

(* This is illegal due to the value restriction *)
val g = f 5
```

25.2 Demonstrating Scoping of Type Variables

Legal Test 1

```
(* A legal program showing the scope of type variables *)
fun f (a : 'a) =
      let
        (* The fact that 'b is only mentioned in a nested let
           means that 'a and 'b are allowed to be
           instantiated independently. *)
        fun id (b : 'b) = b

        (* So this is legal *)
        val v1 = id a
      in
        (a, id)
      end
(* And this is legal
   (we need the type constraints because of the value restriction) *)
val (v3, f1 : string -> string) = (f 11)
val v4 = f1 "Different Type"

val (v5, f2 : int -> int) = (f "Different Type")
val v6 = f2 13
```

Illegal Test 1a

```
(* An illegal program showing the scope of type variables *)
fun f (a : 'a) =
      (nil : 'b list;
       let
         (* The fact that 'b is mentioned directly under f
            means that 'a and 'b may not be instantiated
            independently. *)
         fun id (b : 'b) = b

         (* So this is NOT legal *)
         val v1 = id a
       in
         (a, id)
       end
      )
```

Legal Test 2

```
(* A legal program showing the scope of type variables *)
fun f (a : 'a) x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 =
      let
        (* The fact that 'b is only mentioned in a nested let
           means that 'a and 'b are allowed to be
           instantiated independently. *)
        (* And the parameters x1 x2 etc. should not have used 'b already! *)
        fun id (b : 'b) = b

        (* So this is legal *)
        val v1 = id a
      in
        (a, id)
      end

(* And this is legal and we must be able to give different types to
   'a and 'b and to all the parameters x1 x2 etc.
   (we need the type constraints because of the value restriction) *)
val (v3, f1 : string -> string) = (f 11 1.0 () 3.0 () 5.0 () 7.0 () 9.0 ())
val v4 = f1 "Different Type"

val (v5, f2 : int -> int) = (f "String" 1.0 () 3.0 () 5.0 () 7.0 () 9.0 ())
val v6 = f2 13
```

25.3 Demonstrating Inference of Most General Unifier

Illegal Test 1

```
(* This is an illegal CeXL program, which worked with
   versions 0.9.0 and 0.9.1 of the Definition of CeXL. *)
(* This is also an illegal Standard ML '97 program. *)
fun freeMeta a =
      let
        val (b, c) = a
      in
        c
      end

(* If the type of freeMeta is inferred correctly, this should be illegal, *)
(* but it succeeded in my implementation of CeXL before version 0.9.2 *)
val str = (freeMeta ("Hello", 5)) ^ " string concatenated with integer 5"
val _ = print str
```

Illegal Test 2

```
(* This is an illegal CeXL program, which worked with
   versions 0.9.0 and 0.9.1 of the Definition of CeXL. *)
(* This is also an illegal Standard ML '97 program. *)
fun metaScope x a b =
      let
        val aList = [x, a]
        val bList = [x, b]
      in
        (aList, bList)
      end

(* If the type of freeMeta is inferred correctly, this should be illegal, *)
(* but it succeeded in my implementation of CeXL before version 0.9.2 *)
val (is : int list, strs : string list) = metaScope 2 3 5
val _ = case strs of
          str::_ => print str
        | nil    => print "Error: Empty list returned!"
```

25.4 Demonstrating Requirement for Dynamic Allocation of Exception Names

Legal Test 1

```
(* Example which demonstrates why it is necessary to allocate
   exception names dynamically when type variables are allowed in
   exceptions.
   This test is valid both for Standard ML'97 and CeXL. *)
fun makeFuns (a : 'a) =
      let
        exception E of 'a

        fun raiseF () =
              (raise E a; ())

        fun handleF (f : unit -> unit) =
              (f ();
               print ("Error: Function did not raise exception " ^
                      "as expected!");
               a)
                handle E x => x
      in
        (raiseF, handleF)
      end

val (raiseInt, handleInt) = makeFuns 5
val (raiseString, handleString) = makeFuns "Crash test"

(* If exception names were not allocated dynamically,
   this would give us two examples of unsafe type-casts! *)
val _ = (case handleInt raiseString of
           i =>
             print ("Error: Succeeded in making an unsafe type-cast " ^
                    "from string to int")
        )
           handle _ => print "OK 1/2\n"
val _ = (case handleString raiseInt of
           s =>
             print ("Error: Succeeded in making an unsafe type-cast " ^
                    "from int to string giving the ill-defined value: " ^ s)
        )
           handle _ => print "OK 2/2\n"
```

25.5 Tests for Extensible Records

Legal Test 1

```
(* Example which checks that the type inference of extensible records
   works for record expressions across variable bindings *)
fun extend3Times r =
      let
        val r1 = {a1 = 2, a2 = "Hello", a3 = 3.0, ... = r}
        val r2 = {b1 = "World", b2 = 5, b3 = 11.0, ... = r1}
        val r3 = {c1 = 13.0, c2 = 17, c3 = "Extend", ... = r2}
      in
        r3
      end

(* Check that these patterns are valid *)
val {a1, ...} = extend3Times ()
val {b1, b2, b3, ...} = extend3Times ()
val {c1, c3, ...} = extend3Times ()
val {a1, a2, a3, b1, b2, b3, c1, c2, c3, ...} = extend3Times ()
```

Legal Test 2

```
(* Example which checks that the type inference of extensible records
   works for record expressions across various expressions *)
fun extend3Times r =
      {c1 = 13.0, c2 = 17, c3 = "Extend", ... =
       case {b1 = "World", b2 = 5, b3 = 11.0, ... =
             let
             in
               {a1 = 2, a2 = "Hello", a3 = 3.0, ... =
                (fn x => x) r}
             end
            } of
         x => x
      }

(* Check that these patterns are valid *)
val {a1, ...} = extend3Times ()
val {b1, b2, b3, ...} = extend3Times ()
val {c1, c3, ...} = extend3Times ()
val {a1, a2, a3, b1, b2, b3, c1, c2, c3, ...} = extend3Times ()
```

Legal Test 3

```
(* Example which checks that the type inference of extensible records
   works for record patterns across variable bindings *)
fun extract3Times {a1, a2, a3, ... = r1} =
      let
        val {b1, b2, b3, ... = r2} = r1
      in
        let
          val {c1, c2, c3, ... = r3} = r2
        in
          r3
        end
      end

(* Check that these calls are also valid *)
val r1 = extract3Times {a1 = 2, a2 = "Hello", a3 = 3.0,
                        b1 = "World", b2 = 5, b3 = 11.0,
                        c1 = 13.0, c2 = 17, c3 = "Extend"}

val r2 = extract3Times {a1 = 2, a2 = "Hello", a3 = 3.0, a4 = 23,
                        b1 = "World", b2 = 5, b3 = 11.0, b4 = 27.0,
                        c1 = 13.0, c2 = 17, c3 = "Extend", c4 = "Extra"}
```

Legal Test 4

```
(* Example which checks that the type inference of extensible records
   works for record patterns across variable bindings *)
fun extract3Times {a1, a2, a3, ... = r1} =
      let
        val {b1, b2, b3, ... = r2} = r1
      in
        let
          val {c1, c2, c3, ... = r3} = r2
        in
          r3
        end
      end

(* Check that these calls are also valid *)
val r1 = extract3Times {a1 = 2, a2 = "Hello", a3 = 3.0,
                        b1 = "World", b2 = 5, b3 = 11.0,
                        c1 = 13.0, c2 = 17, c3 = "Extend"}

val r2 = extract3Times {a1 = 2, a2 = "Hello", a3 = 3.0, a4 = 23,
                        b1 = "World", b2 = 5, b3 = 11.0, b4 = 27.0,
                        c1 = 13.0, c2 = 17, c3 = "Extend", c4 = "Extra"}
```

Illegal Test 1a

```
(* Example which checks that the type inference of extensible records
   prevents duplicate labels for record expressions across variable
   bindings *)
fun extend3Times r =
      let
        val r1 = {a1 = 2, a2 = "Hello", a3 = 3.0, ... = r}
        val r2 = {b1 = "World", b2 = 5, b3 = 11.0, a1 = 19, ... = r1}
        val r3 = {c1 = 13.0, c2 = 17, c3 = "Extend", ... = r2}
      in
        r3
      end
```

Illegal Test 1b

```
(* Example which checks that the type inference of extensible records
   prevents duplicate labels for record expressions across variable
   bindings *)
fun extend3Times r =
      let
        val r1 = {a1 = 2, a2 = "Hello", a3 = 3.0, ... = r}
        val r2 = {b1 = "World", b2 = 5, b3 = 11.0, ... = r1}
        val r3 = {a2 = "Goodbye", c1 = 13.0, c2 = 17, c3 = "Extend", ... = r2}
      in
        r3
      end
```

Illegal Test 1c

```
(* Example which checks that the type inference of extensible records
   prevents duplicate labels for record expressions across variable
   bindings *)
fun extend3Times r =
      let
        val r1 = {a1 = 2, c2 = 23, a2 = "Hello", a3 = 3.0, ... = r}
        val r2 = {b1 = "World", b2 = 5, b3 = 11.0, ... = r1}
        val r3 = {c1 = 13.0, c2 = 17, c3 = "Extend", ... = r2}
      in
        r3
      end
```

Illegal Test 2a

```
(* Example which checks that the type inference of extensible records
   prevents duplicate labels for record expressions across variable
   bindings *)
fun extend3Times r =
      {c1 = 13.0, c2 = 17, c3 = "Extend", ... =
       case {b1 = "World", b2 = 5, b3 = 11.0, a2 = "Goodbye", ... =
             let
             in
               {a1 = 2, a2 = "Hello", a3 = 3.0, ... =
                (fn x => x) r}
             end
            } of
         x => x
      }
```

Illegal Test 2b

```
(* Example which checks that the type inference of extensible records
   prevents duplicate labels for record expressions across variable
   bindings *)
fun extend3Times r =
      {a3 = 23.0, c1 = 13.0, c2 = 17, c3 = "Extend", ... =
       case {b1 = "World", b2 = 5, b3 = 11.0, ... =
             let
             in
               {a1 = 2, a2 = "Hello", a3 = 3.0, ... =
                (fn x => x) r}
             end
            } of
         x => x
      }
```

Illegal Test 2c

```
(* Example which checks that the type inference of extensible records
   prevents duplicate labels for record expressions across variable
   bindings *)
fun extend3Times r =
      {c1 = 13.0, c2 = 17, c3 = "Extend", ... =
       case {b1 = "World", b2 = 5, b3 = 11.0, ... =
             let
             in
               {a1 = 2, a2 = "Hello", c3 = "Goodbye", a3 = 3.0, ... =
                (fn x => x) r}
             end
            } of
         x => x
      }
```

Illegal Test 3a

```
(* Example which checks that the type inference of extensible records
   works for record patterns across variable bindings *)
fun extract3Times {a1, a2, a3, b1, ... = r1} =
      let
        val {b1, b2, b3, ... = r2} = r1
      in
        let
          val {c1, c2, c3, ... = r3} = r2
        in
          r3
        end
      end
```

Illegal Test 3b

```
(* Example which checks that the type inference of extensible records
   works for record patterns across variable bindings *)
fun extract3Times {a1, a2, a3, ... = r1} =
      let
        val {b1, b2, a3, b3, ... = r2} = r1
      in
        let
          val {c1, c2, c3, ... = r3} = r2
        in
          r3
        end
      end
```

Illegal Test 3c

```
(* Example which checks that the type inference of extensible records
   works for record patterns across variable bindings *)
fun extract3Times {a1, a2, a3, ... = r1} =
      let
        val {b1, b2, b3, ... = r2} = r1
      in
        let
          val {b3, c1, c2, c3, ... = r3} = r2
        in
          r3
        end
      end
```

Illegal Test 4a

```
(* Example which checks that the type inference of extensible records
   works for record patterns across various expressions *)
fun extract3Times {a1, a2, a3, b2, ... = r1} =
      case r1 of
        {b1, b2, b3, ... = r2} =>
          (fn {c1, c2, c3, ... = r3} => r3) r2
```

Illegal Test 4b

```
(* Example which checks that the type inference of extensible records
   works for record patterns across various expressions *)
fun extract3Times {a1, a2, a3, ... = r1} =
      case r1 of
        {b1, b2, b3, ... = r2} =>
          (fn {a1, c1, c2, c3, ... = r3} => r3) r2
```

Illegal Test 4c

```
(* Example which checks that the type inference of extensible records
   works for record patterns across various expressions *)
fun extract3Times {a1, a2, a3, ... = r1} =
      case r1 of
        {b1, b2, a2, b3, ... = r2} =>
          (fn {c1, c2, c3, ... = r3} => r3) r2
```

25.6 Examples of Type Inference in CeXL

The following are a few contrived examples to demonstrate some of the behahiour of the type inference - in particular for polymorphic extensible records and optional fields.

25.6.1 Legal Programs and Their Types

1. ```
   val f = fn z => (fn x => ({c = 5, x}, x)) {b = "", z}

   (* CeXL type: *)
   val ['a : ~{a, b}] f : 'a  -> ({c : int, b : string, ... : 'a} * {b : string, ... : 'a})
   ```

2. ```
   val f = fn y => (fn x => ({b = 0, x}, {c = 0, x})) ({a = 5, y})

   (* CeXL type: *)
   val ['a : ~{a, b, c}] f :  'a  -> ({b : int, a : int, ... : 'a} * {c : int, a : int, ... : 'a})
   ```

3. ```
   fun f {a, b, ... = r} = f {a, b, ... = r}

   (* CeXL type: *)
   val ['c : ~{a, b}] f :  {a : 'a, b : 'b, ... : 'c} -> 'd
   ```

25.6.2 Illegal Programs

1. ```
   fun f {a, b, ... = r} = f {a = a, b = b, c = 2, ... = r}
   ```