

METACAML CONCOQTION (C-07)

USER GUIDE

Emir Pasalic Jeremy Siek Walid Taha

Document Information: \$Revision: 1.29 \$,
\$Date: 2007/01/05 23:00:38 \$

Copyright Rice University, 2006.

CONTENTS

1	Introduction	4
2	Getting Started	5
2.1	Installation	5
2.2	Starting	5
2.2.1	Interactive Loop	5
2.2.2	Standalone Bytecode Compiler	5
3	Indexed Types	6
3.1	Type Indices and Type Representations	6
3.2	Kinds	6
3.3	First-class Polymorphism	7
3.3.1	Type Abstraction	7
3.3.2	Type Application	8
3.3.3	Syntactic Sugar	8
3.4	Prooflets	8
3.5	Datatypes	9
3.5.1	Example: Singleton Booleans	10
3.5.2	Example: A Function on Singleton Booleans	10
3.5.3	Example: Sized Lists	10
3.5.4	Data Constructors	11
3.5.5	Pattern Matching	11
A	OCamlType Constants	13
B	Software License	14
C	Credits	15
D	Software README file	16

1 INTRODUCTION

MetaOCaml Concoq version C-07, or simply Concoq for short, is an extension to MetaOCaml with indexed types. Indexed types are a powerful tool that allows the programmer to express nested polymorphism as well as complex functional dependencies at the level of types. Only a small number of extensions to the term and type language are needed to give the user full access to the powerful Coq proof checker at the level of types. With these extensions, datatypes can be defined that reflect the size of a list or a vector in its type. An append function on sized lists can be given a type that reflects the way it acts on the size of lists. Using MetaOCaml staging constructs, it is possible to express tagless staged interpreters in Concoq. Because Coq is one of the most expressive checkers available, any fact that can be proved in Coq can be used to give more refined types to programs. Because Coq propositions can be viewed as types, Concoq provides a natural way to incorporate formal verification techniques into programming practice.

The user guide is intended as an introduction to the constructs introduced by this extension. By design, the manual is brief. It is assumed that the reader is familiar with OCaml. If that is not the case, we recommend consulting standard OCaml tutorial materials. To fully take advantage of the capabilities of Concoq, the user may want to consult standard Coq tutorial materials as well.

This user guide is current for Concoq version C-07. This version of MetaOCaml Concoq is based on Objective Caml version 3.08.0 and Coq version 8.01.

Future announcements and discussions (including user questions and answers) will be posted on `metaocaml-users@cs.rice.edu`. Please send comments, suggestions, and bug reports to the MetaOCaml developers mailing list `metaocaml-hackers@cs.rice.edu`.

To learn more about the design of Concoq and about related efforts, the reader is invited to read the research paper “Concoq: Indexed Types Now!” appearing in the proceedings of Partial Evaluation and Program Manipulation (PEPM), 2007.

2 GETTING STARTED

2.1 Installation

The current release is freely available online, and should be easy to install on unix-like systems.

1. Download and untar the .tar.gz file containing the source distribution from <http://www.metaocaml.org/concoqtion>.
2. We will use the name `$TOP` to refer to the root directory produced by extracting the archive.
3. Change the current directory to `$TOP/ometaconc`.

4. Type

```
./configure -prefix `pwd`
```

Note: The current version supports installation in the local directory only.

5. After successful configuration (which may take several minutes), type

```
make concoqtion
```

6. This compiles and installs Concoqtion. The binaries (`ocamlc`, `conc`,...) are installed into the `$TOP/ometaconc/bin` directory.

2.2 Starting

The current release supports both the interactive bytecode compiler and the stand alone bytecode compiler.

2.2.1 Interactive Loop

To run the toplevel interactive loop type

```
$TOP/bin/conc
```

The command `conc` takes the same command line arguments as the OCaml interactive shell `ocaml`.

2.2.2 Standalone Bytecode Compiler

The stand-alone bytecode compiler is `$TOP/bin/ocamlc` and takes the same command line arguments as the OCaml bytecode compiler.

This release does not support the native code compiler.

3 INDEXED TYPES

3.1 Type Indices and Type Representations

A type index is written as `' (c)` where `c` is a Coq term. Type indices can occur anywhere an OCaml type can. To allow the programmer to express functions on types, we provide a set of built-in Coq constants that represent basic OCaml types and type constructors: for any OCaml type named `t` we provide a Coq constant `OCaml_t` which can be used to construct the equivalent index type expression. For example, the Coq constant `OCaml_int` represents the OCaml type `int`. The type of the constant corresponds to the arity of the OCaml type constructor. For example, the constant `OCaml_list` takes one argument representing the argument of the OCaml `list` type constructor. So, writing

```
let x : int = 4
let y : int list = [2]
```

is equivalent to:

```
let x : ' (OCaml_int) = 4
let y : ' (OCaml_list OCaml_int) = [2]
```

In both cases, the interactive loop responds as with the following:

```
val x : int = 4
val y : int list = [2]
#
```

For user-defined types, the corresponding `OCaml_` constants are defined automatically:

```
type 'a lst = Nil | Cons of 'a * 'a lst

let y : ' (OCaml_lst OCaml_int) = Cons (2,Nil)
```

Moreover, the `OCaml_` constants obey the module naming discipline of the OCaml module system. For example:

```
let x : ' (Buffer.OCaml_t) = Buffer.create 4;;
val x : Buffer.t = <abstr>
#
```

3.2 Kinds

Concoction extends the OCaml type system with an explicit notion of *kinds*. Kinds are either omitted, or written using the same syntax as index type expressions: `' (c)`. All, standard OCaml types that classify values have the

kind ' (OCamlType). Type constructors have an arrow kind: for example, the OCaml type constructor `list` has the kind `OCamlType -> OCamlType`.

Kind checking is performed by the type checker and ill-kinded types are reported as type errors. Consider the following ill-formed type annotation, where the variable `x` is given the wrong type ' (OCaml_lst):

```
let x : ' (OCaml_lst) = [];;
```

The type checker detects that the index type ' (OCaml_lst) is of the wrong kind to classify values, and reports the error as follows:

```
Characters 6-24:
  let x : ' (OCaml_lst) = 5;;
      ^^^^^^^^^^^^^^^^^^^
Kind error: Type ' (OCaml_lst) is expected to be of kind
' (OCamlType) but got ' (OCamlType -> OCamlType)
#
```

3.3 First-class Polymorphism

Concoction extends the OCaml type system with a System F-style universal quantifier over types. Universally quantified types are introduced using the keyword `forall` with the following syntax:

$$\text{type} ::= \text{forall } a[:\text{kind}].\text{type} \mid \dots$$

The kind annotation may be omitted, in which case it is assumed to be ' (OCamlType). The type variable `a` is bound by the quantifier and occurs *inside* index type expressions in the type quantifier. For example:

```
type identityType = forall a.' (a) -> ' (a)
let f (x:identityType) = x
```

3.3.1 Type Abstraction

Expressions of `forall` type are constructed using type abstraction syntax.

$$\text{expr} ::= /\!a:\text{kind}.\text{expr}|\dots$$

Again, if the kind annotation is omitted it is assumed to be `OCamlType`:

```
let id = /\!a. fun (x:' (a)) -> x;;
val id : (forall a . ' (a) -> ' (a)) = <forall>

# let id' = f id;;
val id' : identityType = <forall>
#
```

3.3.2 Type Application

Universally quantified type can be explicitly instantiated using the *type application* syntax:

$$\text{expr} ::= \text{expr} . | \text{type} | | \dots$$

Note the asymmetric braces `. | ... |` are used to enclose any OCaml type that the expression `expr` is instantiated to. For example:

```
# let id_int = id .|int|;;
val i : int -> int = <fun>
# let id_int2 = id .|'(OCaml_int)|;;
val id_int2 : int -> int = <fun>
#
```

3.3.3 Syntactic Sugar

Just as OCaml provides syntactic sugar for `fun` abstraction, where

$$\text{let } f \ x = e$$

is shorthand for

$$\text{let } f = \text{fun } x \rightarrow e,$$

we provide similar syntactic sugar for type abstraction as well. The following two `let` definitions are equivalent:

$$\text{let } f \ .|a| \ (x : ' (a)) = x$$
$$\text{let } f' = /\a. \text{fun } (x : ' (a)) \rightarrow x$$

Type abstraction and functional abstraction can be mixed in `let` definitions as long as the type abstracted parameters are enclosed in the type braces `. | ... |`.

3.4 Prooflets

Concoction extends OCaml declarations with the notion of *prooflets*. A prooflet is a Coq Vernacular script (the same language used to interact with the theorem prover) delimited by the keywords `coq` and `end`. Any declarations, definitions or Coq proofs written in the prooflet are added to the Coq environment and visible in the following index type expressions. The most common use of sections is to add definitions of new index types. By issuing commands to Coq in the prooflet, the programmer can import any standard or separately compiled Coq library.

Prooflets also allow the programmer to state properties of indices as Coq theorems and then prove them. For example, one might wish to prove that

for any type constructor f over natural numbers $'(f (m+n))$ is equal to $'(f (n+m))$.

The proofs of this and similar properties can be constructed using tactics:

```
coq
Require Arith.
Lemma comm_eq :
  forall m (n:nat) (f: nat->OCamlType),
    (f(m+n))=(f(n+m)).
  intros; eauto with arith. Qed.
end
```

After stating the lemma `comm_eq` in prooflets, Coq goes into proof mode. Issuing the tactic `intros; eauto with arith` proves the lemma. At the Vernacular command `Qed`, Coq checks and accepts the theorem, which is then available in the rest of the program as an index-type function named

```
comm_eq : forall m n:nat, (f :nat->OCamlType),
  (f(m+n)) = (f(n+m))
```

3.5 Datatypes

Concoction extends the OCaml type declarations in two ways. First, parameters of type constructors can range over any specified kind. The type constructor `container` represents collections of integers, and is declared to range over container types of kind `OCamlType->OCamlType`:

```
type 'c:'(OCamlType -> OCamlType) container =
  | C of let 'c:'(OCamlType -> OCamlType) in
        '(c OCaml_int) : '(c) container
  | Empty
```

Note that the following declaration does not kind check:

```
let x : int container = Empty
```

However, the following two expressions are correct:

```
let x : '(OCaml_list) foo = C .|'(OCaml_list)| [1;2;3];;
let y : '(OCaml_array) foo = Empty
```

Second, in algebraic data-type declarations, the OCaml restriction that the result type of each data-constructor must be polymorphic in the type's parameters is relaxed. For example, the OCaml type `'a list` tells us nothing about the structure the list.

3.5.1 Example: Singleton Booleans

By varying the index parameters in the data-constructor's result type, we can say more about the structure of a value from its type. In the extreme case, this extension allows us to express *singleton* types whose runtime values are fully determined by the type of their indices.

```
type 'b:' (bool) sbool = T : '(true) sbool
                       | F : '(false) sbool
```

The result type of a data constructor is added at the end of its declaration following the colon (:) symbol. The result type must be an instance of the type constructor being defined:

```
# type 'b:' (bool) sbool = T : '(true) sbool | F : bool;;
```

```
Characters 4-56:
.... 'b:' (bool) sbool2 =
| T : '(true) sbool | F : bool ...
Concoction: Return type must be a sbool
but was declared as bool.
```

```
#
```

An expression of type '(true) sbool is statically known to be equal to T.

3.5.2 Example: A Function on Singleton Booleans

Now we can write a type which guarantees that a function `snot: forall b:' (bool). '(b) sbool -> '(negb b) sbool` implements negation as specified by the Coq function `negb`:

```
coq
  Require Export Coq.Bool.Bool.
end

let snot .|b:' (bool)| (x:' (b) sbool) : '(negb b) sbool =
match x in '(negb b) sbool with
| T -> F
| F -> T
```

3.5.3 Example: Sized Lists

A final extension to data-constructor declarations allows the programmer to declare *locally quantified* type variables. For example, consider the type `list1` of lists whose first parameter is a natural number index indicating its length:

```

type ('n:'(nat), 'a) list1 =
| Nil : ('(0), 'a) list1
| Cons of let 'm:'(nat) in 'a * ('(m), 'a) list1
        : ('(m+1), 'a) list1

```

The declaration of the data-constructor `Cons` uses a locally quantified variable `m` of kind `nat` and states that given some natural number `m`, an element of type `a`, and a list of length `m`, `Cons` produces a list of length `m+1`.

3.5.4 Data Constructors

To create an value of an indexed datatype, apply the constructor name to the index arguments and the value arguments. For example, the following function takes an integer and adds it twice to a `list1` increasing its length by two:

```

let add_twice .|m:'(nat)| x xs =
  Cons .|'(1+m)| (x, Cons .|'(m)| (x, xs))

```

Data-constructors that have locally quantified type variables must be fully type-applied to all their type arguments, then applied to any expression arguments they may require.

3.5.5 Pattern Matching

Concoction has an extended form of `match` expressions data-types whose indices may vary for each constructor.

The extended match expression allows two additions. The first is a *type pattern*, introduced by the keyword `as`.

```

let rec app .|m:'(nat), n:'(nat)|
  (l1 : ('(m), _) list1) (l2 : ('(n), _) list1)
  : ('(m+n), _) list1 =
  match l1 as ('i:'(nat), 'a:'(OCamlType)) list1
            in ('(i+n), '(a)) list1 with
| Nil -> l2
| Cons .| m2:'(nat)| (x, xs) ->
  Cons .| '(m2+n) | (x, app .|'(m2), '(n)| xs l2)

```

The type pattern `('i:'(nat), 'a:'(OCamlType)) list1` binds the type variables `i` and `a` in the scope of the rest of the match. The second is the type `('(i+n), '(a)) list1`, following the keyword `in`: it is called a *result type annotation*, and may contain free type variables bound by the type pattern. When type-checking the match expression, the type of the discriminated expression `l1` is matched against the type pattern, obtaining a substitution for the type variables. Applying this substitution to the result type annotation gives the result of the whole `match` expression. In each branch of the case, the type pattern is first matched against the type computed for

the constructor pattern, obtaining a type substitution for that branch. The type of the body of the branch then must be precisely the result type annotation to which this substitution is applied. This allows each branch to have a different type depending on the types of the indices of the constructor in the branch.

For example, in the `Nil` case, `i` is replaced by `' (0)`, allowing the branch expression to be a list of type `' (0+n), ' (a) list1`. In the `Cons` case, `i` is replaced by `' (1+m2)`. This means that the type of the branch expression must be `' ((1+m2)+n), ' (a) list1`. The type computed for the branch expression is `' (1+(m2+n)), ' (a) list1`. By expanding the Coq definitions of `+` the type checker determines that the two types are equal, and accepts the match.

If the type of the discriminated expression is simple enough, the type pattern may be omitted. In particular, this is the case when the parameters of the type are comprised entirely of variables (`' (i)`) and constant index type expressions (`| ' (0) |`). In this situation, the type checker can infer the particular substitution binding the type variables to more specific types in each branch. The restriction on the discriminated expression's type is necessary to make computing this substitution decidable – in all other cases the programmer must use the more general type-pattern notation. In practice we find that many functions can be written using this simpler syntax. Let consider a simple example of omitted type patterns by writing a `zip` function on lists with length.

```
let rec zip .|n:' (nat)|
  (l1:' (n), 'a) list1) (l2:' (n), 'b) list1)
  : ((' (n), ('a * 'b)) list1) =
  match (l1,l2) in (' (n), 'a*'b) list1 with
  | Nil,Nil -> Nil
  | Cons .|i:' (nat)| (x,xs), Cons .|j:' (nat)| (y,ys) ->
    Cons .|' (j)| ((x,y), zip .|' (i)| xs ys)
```

The type of the expression `(l1, l2)` is a pair of lists of length `n`. In the first branch, Concoqton infers that `n` must be equal to zero, substituting `0` for `n` in the result type annotation when checking the right-hand side.

In the next case, the pattern has the type `' (S i), 'a) list1 * (' (S j), 'b) list1` where the sub-lists `xs` and `ys` are have lengths `' (i)` and `' (j)` respectively. The type checker, concludes that since both `' (S i)` and `' (S j)` must be equal to `n`, `i` and `j` must be equal. This allows us to apply `zip` to `xs` and `ys` although the variables representing their length are different.

A OCAMLTYPE CONSTANTS

We list some of the more interesting built-in OCaml_ constants.

Constant	Kind
OCaml_Int	OCamlType
OCaml_string	OCamlType
OCaml_Arr	OCamlType -> OCamlType -> OCamlType
OCaml_Tuple	ot_list -> OCamlType
OT_List_nil	ot_list
OT_List_cons	OCamlType -> ot_list -> ot_list
OCaml_list	OCamlType -> OCamlType
OCaml_code	OCamlType -> OCamlType -> OCamlType
OCaml_All	forall A : Set, (A -> OCamlType) -> OCamlType

B SOFTWARE LICENSE

Copyright 2002-2006
Walid Taha's research group at Rice University and collaborators
All Rights Reserved

MetaOCaml and MetaOCaml Concoqtion licensed as a patch to the respective distribution of OCaml and Coq upon which they based. OCaml and Coq are developed and copyright by INRIA. Please see OCaml and Coq licences their respective licences, and for status of a patch.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal with the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimers.

- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimers in the documentation and/or other materials provided with the distribution.
- Neither the names of MetaOCaml, Concoqtion, Rice University, nor the names of its contributors may be used to endorse or promote products derived from this Software without specific prior written permission.
- Products derived from this software may not be called "MetaOCaml", "MetaOCaml Concoqtion", "Concoqtion", or an extension of these names without prior written permission from the RAP group.

Commercial use is prohibited without prior written permission.

Permissions must be granted by

Walid Taha (taha@rice.edu)
Department of Computer Science
Rice University, Houston, TX 77025.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE CONTRIBUTORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS WITH THE SOFTWARE.

C CREDITS

MetaOCaml and
MetaOCaml Concoqtion (Concoqtion for short)

Developers

Walid Taha, Rice University
Cristiano Calcagno, Imperial College
Xavier Leroy, INRIA
Oleg Kiselyov, FMNOC
Emir Pasalic, Rice University
Jeremy Siek, Rice University
Edward Pizzi, Rice University
Roumen Kaiabachev, Rice University
Jason Lee Eckhardt, Rice University
Kedar Swadi, Rice University

Conception and Design
Staging constructs
Compiler specifics
Native code compiler, cylce timer
Concoqtion, tag elimination
Concoqtion
Pretty printer
Offshoring
Offshoring
Offshoring

D SOFTWARE README FILE

DOCUMENTATION

A brief user guide can be found at `doc/users_guide.pdf`

QUICK INSTALLATION INSTRUCTIONS

To build concoqtion:

1. `cd ometaconc`
2. `./configure -prefix `pwd``
3. `make concoqtion`

KNOWN PROBLEMS

We are aware of the following liminations:

1. Nested patterns. Currently, the type-checker does not properly handle nested patterns for data-constructors that bind local type variables. However, such patterns can always be replaced by using nested match statements which are handled correctly now.
2. The current release cannot be compiled under Cygwin
3. The current release does not support the native code compiler.
4. The current release does not support `camp4`