# User guide for *Cubint*

Christian Müller

November 9, 2004

**Abstract**

The simply typed $\lambda$-calculus (S), system F, F$\omega$ and the calculus of construction (CC) offer a good framework to understand the underlying theory of programming languages. *Cubint* is an interpreter for these calculi that lets you play around with examples to deepen your understanding. This document should provide all the things that you need to know in order to work with the interpreter.

The first section *crash course* is intended for the audience of the *semantics* lecture at the computer science department of UdS. It offers introductory material that enables students to use the interpreter in the first few weeks of the lecture.

## Contents

**1 Crash Course** **2**
- 1.1 First Steps . . . 2
- 1.2 Untyped $\lambda$-Calculus . . . 2
- 1.3 Simply Typed $\lambda$-Calculus . . . 3

**2 Getting *Cubint*** **4**

**3 *rlwrap* or *emacs*** **5**

**4 Commands** **5**

**5 Expressions** **6**
- 5.1 *S* - Simply Typed $\lambda$-Calculus . . . 6
- 5.2 *F* - System F . . . 7
- 5.3 *Fw* - F$\omega$ . . . 7
- 5.4 *CC* - Calculus of Construction . . . 7
- 5.5 *R* - Type Reconstruction for Prenex Polymorphism . . . 8
- 5.6 *RG* - Type Reconstruction Based on Graphs . . . 8
- 5.7 *U* - Untyped Lambda Cube . . . 8
- 5.8 *UD* - User Defined . . . 8
- 5.9 *SUB* - Subtyping . . . 8

**6 Stepwise Reduction** **8**

**7 Further Readings** **9**

# 1 Crash Course

## 1.1 First Steps

Download the pre-compiled package for Windows or Linux from the web page[1] and extract the package with *tar xfz cubint_linux.tar.gz* on Linux or your favourite tool (e.g. WinZip[2]) on Windows respectively. Start the interpreter with a double click on the executable (or alternatively *./cubint* on Linux). Type `help;<RETURN>` for information about available commands.

## 1.2 Untyped $\lambda$-Calculus

To switch to the *untyped* $\lambda$*-calculus* type `switch tm u;<RETURN>`. The syntax for untyped terms is as follows:

| calculus | interpreter input |
|---|---|
| $x$ | `x` |
| $\lambda\, x.t$ | `\x.t` |
| $t_1\ t_2$ | `t1 t2` |

Let's look at some examples:

- We apply the identity function several times to itself:

  $$(\lambda x.x)(\lambda x.x)(\lambda x.x)$$

  can be fed to the interpreter as

  ```
  (\x.x) (\x.x) (\x.x)
  ```

  We can also bind the identity function to the identifier *id* and apply the identifiers several times. This is shown in figure 1.

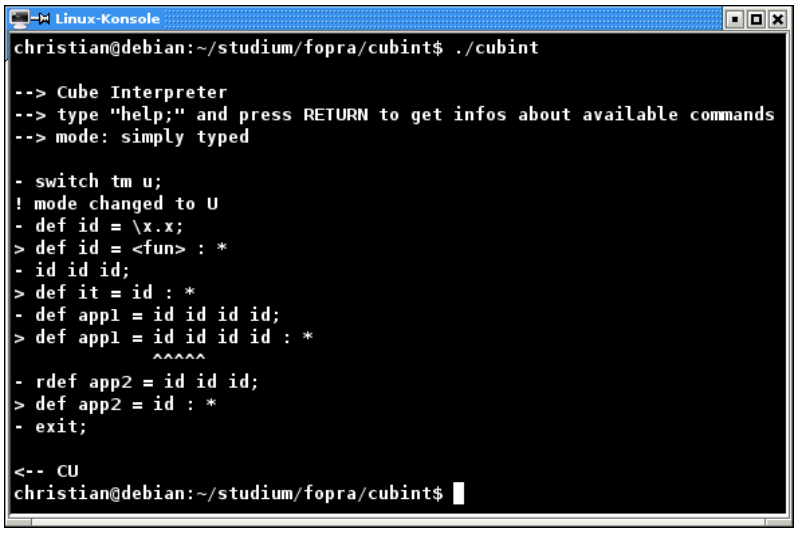


Figure 1: definition and use of *id*

As you probably noticed there are to binding constructs, namely `def` and `rdef`. The first one only binds a term to an identifier whereas the second one performs $\beta$-reduction before binding.

[1] http://www.ps.uni-sb.de/ cmueller/cubint.html
[2] http://www.winzip.de

- Now we define *church booleans*.

| | |
|---|---|
| $tru = \lambda t.\lambda f.t$ | `def tru  = \t.\f.t` |
| $fls = \lambda t.\lambda f.f$ | `def fls  = \t.\f.f` |
| $test = \lambda l.\lambda m.\lambda n.l\ m\ n$ | `def test = \l.\m.\n. l m n` |

  On the left hand side you can see the conventional definition in the untyped $\lambda$-calculus and on the right hand side there is the corresponding interpreter input. We use these definitions in the following examples:

```
- def v = \x.x;
> def v = <fun> : *
- def w = \x.\y.x;
> def w = <fun> : *
- test tru v w;
> def it = v : *
- rdef result = test fls v w;
> def result = w : *
```

- As a last example we give a definition of a fixpoint combinator:

$$\lambda f.(\lambda x.\lambda y.f(xx)y)(\lambda x.\lambda y.f(xx)y)$$

```
def Fix = \f.(\x.\y.f (x x) y) (\x.\y.f (x x) y)
```

  If you are not convinced that it works, conduct the following test:

```
def gauss_sum = Fix (\f.\x.if x<=0 then 0 else x+f(x-1));<RETURN>
gauss_sum 6;<RETURN>
```

  You should get `def it = 26`.

## 1.3 Simply Typed $\lambda$-Calculus

At the end of the last section we saw that *Cubint* supports built-in integers. As you know from the lecture this extension is very error-prone in the untyped calculus, i.e. it is very simple to input terms whose evaluation get stuck (e.g. `3+(\x.x)`). Therefore types are introduced. The syntax looks as follows:

| **$\lambda$calculus** | **interpreter input** |
|---|---|
| $x$ | `x` |
| $\lambda x : \mathbf{T}.t$ | `\x:T.t` |
| $t_1\ t_2$ | `t1 t2` |
| $T \to T$ | `T->T` |
| $Int$ | `Int` |

The $\lambda$-abstraction needs a type annotation. Types can be built with `->`. `Int` is the base type for integers. For a full overview about the syntax look at 5.1.

The interpreter is started in simply typed mode at the beginning. You can also change the mode *on-the-fly* by typing `switch tm s;<RETURN>`.

Let's look at some basic examples:

- Term definitions work as before:

$$\lambda f : Int \to Int.\lambda x : Int.f\ x$$

```
\f:Int->Int.\x:Int.f x
```

- You also have the possibility to bind types to an identifier and use it in a term:

```
def T = Int -> Int;
def f  = \g:T.\x:Int.g x
```

This concludes the short overview about the base calculi. The following sections describe the usage of *Cubint* in more detail. All available interpreter commands and syntactic entities are listed. If you need more examples, use the *tests library*.

# 2 Getting *Cubint*

There are to separate packages for Windows and Linux available on the web page[3]. To start the interpreter on your Linux machine you have to install *mosml*[4]. To recompile the system you need *make*[5]. For Windows users I suggest *cygwin*[6] in this case.

To start the interpreter, use:

```
./cubint [-lw n] [-s|-f|-fw|-cc|-r|-rg|-u|-ud] [files]
```

For windows users:

```
cubint.exe [-lw n] [-s|-f|-fw|-cc|-r|-rg|-u|-ud] [files]
```

| **Parameter** | **Meaning** |
|---|---|
| `-s` | simply typed $\lambda$-calculus with iso-recursive types, integers, boolean, records, variants |
| `-f` | System F, i.e. terms depending on types, with the same extensions as S |
| `-fw` | $F_\omega$, i.e. in addition to F this mode provides type operators |
| `-cc` | Calculus of Construction, i.e. types may depend on terms |
| `-r` | ML type reconstruction, i.e. you can leave out type annotations in $\lambda$-abstractions |
| `-rg` | same as R, but realized with cyclic graphs, so this mode provides equi-recursive type |
| `-u` | untyped $\lambda$-calculus; expressions are not type checked |
| `-ud` | user defined mode |
| `-lw n` | set line width to n characters, default is 80 |
| `files` | arbitrary number of files |

All arguments are optional, but the order is significant. If you do not choose a typing mode, then `s` will be chosen by default.

[3] http://www.ps.uni-sb.de/ cmueller/cubint.html
[4] http://www.dina.dk/ sestoft/mosml.html
[5] http://www.gnu.org/software/make/
[6] http://www.cygwin.com

# 3 *rlwrap* or *emacs*

Of course you want to enjoy features like command line history, parenthesis matching and navigating through an expression with the arrow keys. Suppose you work on a Linux machine, you have two possibilities to get all these things. First you can install *rlwrap*[7]. If you then invoke the interpreter with `rlwrap ./cubint ...`, the tool will catch all your input and you can replay it with the arrow keys. The second possibility is to use the editor *emacs*[8] with *sml-mode*[9]. If you set up emacs for SML, follow these steps:

- Load a file and switch to the SML mode (`M-x sml-mode`).
- Press CTRL-C + CTRL-B. Now you (hopefully) see an input prompt like "ML command: " in the mini-buffer.
- Input here the location of the compiled interpreter and press ENTER.
- Answer "Any Args: " with any of the arguments listed in section 2 and press ENTER.
- Now the content of the buffer is passed to the interpreter and you can go on working with it in the usual sml-mode fashion.

This approach also work with Emacs on Windows. Alternatively you can use the Windows XP shell that offers a command line history which also works in the interpreter. To input more complicated expressions use an arbitrary text editor and load the file with `use "file"` (cf. section 4).

# 4 Commands

The following commands are available in the toplevel environment:

| **Command** | **Description** |
|---|---|
| `def id = expression` | binds the *expression* to *id*. The expression is type checked but not evaluated. Therefore this command is good to define abbreviations for types. If a $\beta$ redex is found, it is marked. |
| `rdef id = expression` | binds the type checked and reduced *expression* to *id*. So this is the usual way to bind terms. |
| `rstep expression` | performs one evaluation step on *expression* and binds it to the default identifier *it*. |
| `protocol e` | outputs an execution protocol of the expression *e*. |
| `equiv E1, E2` | checks whether *E1* and *E2* are equal modulo full (!!!) $\beta$-reduction. |
| `use "filename"` | loads a file into the interpreter |
| `exit` | exits the interpreter (CTRL+D on Linux has the same effect) |
| `help` | shows some information about the commands |

To pass these commands to the interpreter, type "`<command>; <RETURN>`".

Furthermore there is the possibility to enter an expression at the toplevel. This is an abbreviation for "`rdef it = exp`". If you only type "`rstep`", then the interpreter tries to perform an evaluation step on the expression bound to *it*. Further abbreviations are "`def id = rstep`" and "`def id = rstep expression`". You can find a short example in section 6.

To change the behaviour of the interpreter use the `switch` command. The following alternatives are available:

[7] http://freshmeat.net/projects/rlwrap/
[8] http://www.gnu.org/software/emacs/emacs.html
[9] http://www.smlnj.org/doc/Emacs/sml-mode.html

| Command | Description |
|---|---|
| `switch pm o` | changes the pretty printer to opaque output, i.e. hide definitions |
| `switch pm t` | switches the pretty printer to transparent output to expose definitions. |
| `switch pm ows` | changes the pretty printer to opaque output and tries in addition to synchronise the expressions with the environment. This check requires full $\beta$-reduction and might therefore diverge in some cases. |
| `switch pm terms` | outputs only terms but no types. |
| `switch pm types` | outputs only types but no terms. |
| `switch pm all` | switches back to default to see both terms and terms. |
| `switch tm s` | switches the typing mode to `s`. Replace `s` by `f, fw, cc, un, r, rg` to switch to another mode. `switch [no]sub` enables or disables subtyping |

*Note:* On switching the environment, it is always modified so that it only contains definitions which are well typed in the new mode.

# 5 Expressions

In the directory *tests* you can find some files that provide example declarations for the different modes. They might offer a quick overview about the syntax of `s, f, fw, cc, r, rg, un`. The following will give you an overview of the syntax starting with S. For all the other calculi only the extensions are explained.

## 5.1 *S* - Simply Typed $\lambda$-Calculus

The following constructs are available in the *simply typed* mode:

| Expression | Syntax | Note |
|---|---|---|
| $\lambda x : E_1.E_2$ | `\x :E1.E2` | abstraction |
| $\Pi X : E_1.E_2$ | `pi X:E1.E2` | quantification |
| $E_1 \to E_2$ | `E1->E2` | type, abbreviation for $\Pi\_ : E_1.E_2$ |
| $\mu X.E$ | `mu X.E` | built-in iso-recursive types |
| | `E1 E2` | application |
| | `fold E1 E2` | folds the iso-recursive type of E2 once according to E1 |
| | `unfold E1 E2` | unfolds the iso-recursive type E1 of E2 once |
| | `x` | identifier $(a \dots z \vert A \dots Z \vert')(\_ \vert a \dots z \vert A \dots Z \vert 0 \dots 9 \vert')^*$ |
| | `~12+3**(5-4)` | built-in integers and all available arithmetic operations on them |
| | `3=3` | comparison on integers, further operators are `<<,<=, >>, >=, <>` (unequal) |
| | `true` and `false` | boolean values |
| | `not E` | negation of boolean values |

| Expression | Syntax | Note |
|---|---|---|
| | `unit` | unit value |
| | `Int, Bool, Unit` | built-in base types |
| | `if E1 then E2 else E3` | conditional |
| | `E1 as E2` | ascription; both expressions must be atomic |
| | `{x=E1,y=E2}` | records |
| | `record@field` | record projection |
| | `{x:E1,y:E2}` | record types |
| | `<x=E1> as E2` | variant |
| | `<x:E1,y:e2>` | variant types |
| | `case E of`<br>`<l1=x1> => E1`<br>`\| ... \| <ln=xn> => En` | case construct to access a value inside a variant; due to the uniform syntax you have to put parentheses around $E_1$,...,$E_n$ except for atomic and arithmetic expressions. |
| | `ref E` | creates a reference for E |
| | `!E` | dereferences E |
| | `E1 := E2` | assignment |
| | `let x=E1 in E2` | let expression, but only terms can be bound |

Be aware of the operator for multiplication `**`, the smaller test `<<` and the greater test `>>` that are somewhat unconventional. In S,F and Fw records and variants can only contain terms. Moreover a label cannot be used more than once in the same expression.
An expression sequence `(E1;...;En)` is available as syntactic sugar. The sequence `(E1;E2)` is transformed into an application `(\x:Unit.E2) E1`. Therefore type errors in a sequence might be a bit inconvenient. A expression sequence is useful when you are working with references.

## 5.2 *F* - System F

In addition to $S$ terms may depend on types and you have universally quantified types. This is expressed in the following way:

| Expression | Syntax | Note |
|---|---|---|
| $\lambda X.E$ | `\X .E` | type abstraction |
| $\forall X.E$ | `\/X.E` | the type of a type abstraction<br>(abbreviation n for $\Pi X : E_1.E_2$) |

## 5.3 *Fw* - F$\omega$

Type operators are available, i.e. types may be abstracted out of types. Additionally the notion of kinds is introduced. Kinds are the types of types.

| Expression | Syntax | Note |
|---|---|---|
| $\star$ | `*` | base kind $\star$ |
| $\lambda x : E_1.E_2$ | `\x :E1.E2` | $E_1$ might be a kind like $\star \rightarrow \star$ |
| $\forall X : E_1.E_2$ | `\/X:E1.E2` | $E_1$ might be a kind like $\star \rightarrow \star$ |

## 5.4 *CC* - Calculus of Construction

This is the richest available calculus. Now you can abstract terms out of types. The syntax does not change. `if` and `case` can be used on type level, i.e. you can write things like

`if E then Int else Bool`. Records and variants can contain both terms and types at the same time. There is a syntactic sugared version of $\Pi x : Int.T$, namely $(x :: Int) \rightarrow T$.

### 5.5 *R* - Type Reconstruction for Prenex Polymorphism

| Expression | Syntax | Note |
|---|---|---|
| $\lambda x.E$ | `\x .E2` | blank lambda abstraction |
| $\lambda f : \alpha \rightarrow \beta$.E | `\f :'a->'b.E` | use of free type variables |

Do not confuse the blank lambda abstraction with the abbreviation in *F, Fw* and *CC* or the abstraction in untyped mode. There is the possibility to annotate a term with free type variables and all types from *S* mode. Therefore the expression `\x.E` can be seen as an abbreviation for `\x:'a.E` where `'a` is not used in the enclosing context.
This mode does not offer the full functionality of the previous ones. It consists of the basic elements which are lambda abstraction, application and variables. Additionally all basic types, terms and operations are integrated, i.e. numbers, boolean, unit, +,-, if-expression, ... . Records are also available. However be aware that these are different from the other constructs in an important way: the constraint typing algorithm may fail if there is a wrong projection.
As a challenge you can try to implement further constructs. Variants are for example very similar to records.

### 5.6 *RG* - Type Reconstruction Based on Graphs

In *RG* there are equi-recursive types. So you do not have to annotate terms with self application with a *fold* or *unfold* expression, i.e. the expression $\lambda x.x\ x$ is type correct and gets the type:

```
- \x. x x;
> def it = <fun> : (mu A.A->'a)->'a
```

In the other respects the mode is just the same as *R* (but records are not available).
Iso-recursive types, let expressions, records, variants and references are not integrated into this mode.

### 5.7 *U* - Untyped Lambda Cube

In this mode the type checker is switched off. Type annotations are ignored. So it is permitted to input senseless expressions without any kind of error message. But of course this will often lead to *stuck terms* which cannot evaluated further.

### 5.8 *UD* - User Defined

In this mode the type checker and evaluator are set to the user's implementation. The user may implement the functions in UserCheckEval.sml. You can find explanation and an example implementation in this file in the directory *src* of the *Cubint* package.

### 5.9 *SUB* - Subtyping

This mode provides subtyping for variants and records and can be enabled in the typing modes s,f,fw and cc. There is no Top type integrated in the calculus. Therefore the interpreter does not accept input like `if true then 3 else true` as a value of type Top is not very valuable. In subtyping mode you can leave out the variant ascription.

## 6 Stepwise Reduction

In figure 2 is a short example for stepwise evaluation. As you can see, the redex is always underlined and if you want to evaluate the identifier to a value, simply pass the variable to the interpreter. Be aware that in transparent mode the output for complicated terms, e.g. a fixpoint computation, is in general very complex.

```
- def inc = \x:Int.x+1;
> def inc = <fun> : Int->Int
- def s = inc (inc (if 3+1<=4 then 2 else ~1));
> def s = inc (inc (if 3+1<=4 then 2 else ~1)) : Int
                        ^^^
- rstep s;
> def it = inc (inc (if 4<=4 then 2 else ~1)) : Int
                         ^^^^
- def s' = rstep;
> def s' = inc (inc (if true then 2 else ~1)) : Int
                     ^^^^^^^^^^^^^^^^^^^^^^
- rstep s';
> def it = inc (inc 2) : Int
                 ^^^^^
- rstep;
> def it = inc (2+1) : Int
                 ^^^
- it;
> def it = 4 : Int
-
```

Figure 2: illustration of the stepping mode

# 7 Further Readings

The examples in directory *tests* might offer you further insight to the usage of *Cubint*. More information can be found in *final_report.pdf*. This document explains the design decisions for *Cubint*, which helps to understand possibly unexpected behaviour of the interpreter.