A Quick-Start Guide to Bedwyr v1.4

Quentin Heath
INRIA Saclay and LIX/École polytechnique

June 11, 2013

1 Overview

Bedwyr is a generalization of logic programming that allows model checking directly on syntactic expression possibly containing bindings. It identifies finite successes and finite failures under the closed world assumption. The logic manipulated, a subset of LINC (for *lambda, induction, nabla, co-induction*), contains object-level λ-abstractions, three quantifiers (including the ∇ quantifier), and inductive and co-inductive definitions. While LINC is a extension of higher-order intuitionistic logic, formulae are restricted to a fragment where connectives on the left of an implication must have invertible rules (i.e. no universal quantifier nor implication – this enables the closed-world assumption), while equalities are restricted to the L_λ fragment (allowing for the use of higher-order pattern unification).

The system, written in OCaml, is part of an open source project. The web page offers multiple ways to get it:

- read-only SVN repository
- sources tarball
- precompiled binaries (win32, GNU/Linux, OCaml bytecode)
- GNU/Linux (Gentoo and Debian) unofficial packages
- Windows installer

The documentation includes this quick-start guide, a reference manual and the source-code documentation.

2 Input format

Bedwyr accepts three kinds of input: commands, queries and meta-commands.

Commands are grouped in definition files (plain text files, usually named `*.def`). They declare new types and constants, declare and define new predicates, and define theorems (note that theorems are not proved, but are used as lemmas).

Queries are entered at the toplevel (either in batch-mode or interactively). They are plain formulae that Bedwyr attempts to solve. If it succeeds in its search, it either displays the list of solutions (substitutions of the free variables) or a negative answer;

Listing 1: `maxa.def`

```
Kind ch type.

Type z ch.
Type s ch -> ch.

% The predicate a holds for 3, 5, and 2.
Define a : ch -> prop by
  a (s (s (s z))) ;
  a (s (s (s (s (s z))))) ;
  a (s (s z)).

% The less-than-or-equal relation
Define inductive leq : ch -> ch -> prop by
  leq z N ;
  leq (s N) (s M) := leq N M.

% Compute the maximum of a
Define maxa : ch -> prop by
  maxa N := a N /\ forall x, a x -> leq x N.
```

otherwise, if the formula is not handled by the prover (non-invertible connective on the left) or by the unifier (not L_λ), it aborts with an appropriate error message.

Meta-commands can be entered either in files or at the toplevel. Most of them aim at improving the user experience by executing strictly non-logical tasks, mainly input (`#include "inc.def".`), output (`#debug on.`, `#typeof X Y :: nil.`) and testing (`#assert true.`, `#assert_not false.`). A few of them change the order of computations, and thus performances, but not provability (`#freezing 4.`, `#saturation 2.`).

3 Sample files

Listing 1 shows a complete sample definition file, with the declarations for a type and two constants, along with a few predicates.

The predicate `a` is a typical example of what must be done to build a Bedwyr example: even with a theoretically infinite search space (here, Church numerals), Bedwyr only does finite reasoning, and hence must be given an explicit description of its finite actual search space.

The use of the `inductive` keyword has two consequences. Firstly, memoization is used on the corresponding predicate; secondly, it has an impact on the way loops in computation are handled. Since the `leq` predicate obviously cannot loop, only the first aspect is used here (meaning we might as well have used the `coinductive` keyword instead).

4 REPL demo

Listing 2 shows an example of use of the interactive toplevel following the invocation of `bedwyr maxa.def`.

Listing 2: Interactive session

```
?= #env.
*** Types ***
   list : * -> *
   ch : *
*** Constants ***
   (::) : A -> (list A) -> list A
   nil : list A
   s : ch -> ch
   z : ch
*** Predicates ***
   a : ch -> prop
 I leq : ch -> ch -> prop
   maxa : ch -> prop
   member : A -> (list A) -> prop
?= leq X (s (s z)).
Solution found:
 X = z
More [y] ? y
Solution found:
 X = s z
More [y] ? y
Solution found:
 X = s (s z)
More [y] ? y
No more solutions.
?= maxa X.
Solution found:
 X = s (s (s (s z)))
More [y] ? y
No more solutions.
?= #exit.
```

The `#env.` meta-command shows all declared objects (including the standard predeclared list-related objects), and displays `leq` as inductive. The call to the query `leq X (s (s z)).` offers to display all solutions, one by one. The call to the query `maxa X.` does the same, but a subsequent call to `#show_table leq.` would then show the table filled with `leq`-headed atoms, either marked as proved or disproved.