A Reference Manual for Bedwyr v1.4 *

David Baelde[1], Andrew Gacek[2], Quentin Heath[3],
Dale Miller[3], Gopalan Nadathur[4], Alwen Tiu[5]

[1]LSV, ENS Cachan
[2]Rockwell Collins
[3]INRIA Saclay and LIX/École polytechnique
[4]University of Minnesota
[5]Australian National University

July 4, 2013

Contents

*Support has been obtained for this work from the following sources: from INRIA (first through the "Équipes Associées" Slimmer, and the "ADT" BATT), and from the NSF Grants CCR-0429572 (that also included support for Slimmer) and OISE-0553462.

Foreword

This manual documents the release 1.4 of the Bedwyr prover. It is organized as follows:

- Part I, a user's guide
- Part II, a collection of tutorials and examples
- Part III, a deeper system description

For a faster start, please refer to the Quick-Start Guide to Bedwyr and then read the examples in Part II.

The source code documentation can be found here.

The complete distribution of the system, including documentation, examples, documentation sources and links to related publications can be found on the same website. Development versions and old releases are available.

Listings

List of Figures

Part I
A User's Guide to Bedwyr

1 Overview

Some recent theoretical work in proof search has illustrated that it is possible to combine the following two computational principles into one computational logic:

1. a symmetric treatment of finite success and finite failure – this allows capturing both aspects of *may* and *must* behavior in operational semantics, and mixing model checking and logic programming;
2. direct support for λ-tree syntax, as in λProlog, via term-level λ-binders, higher-order pattern unification, and the ∇ quantifier.

All these features have a clean proof theory. The combination of these features allow, for example, specifying rather declarative approaches to model checking syntactic expressions containing bindings. The Bedwyr system is intended as an implementation of these computational logic principles.

Why the name Bedwyr? In the legend of King Arthur and the round table, several knights shared in the search for the holy grail. The name of one of them, Parsifal, is used for an INRIA team where Bedwyr is currently developed. Bedwyr was another one of those knights. Wikipedia (using the spelling "Bedivere") mentions that Bedwyr appears in *Monty Python and the Holy Grail* where he is "portrayed as a master of the extremely odd logic of ancient times". Bedwyr is a re-implementation and rethinking of an earlier system called Level 0/1 written by Alwen Tiu and described in [TNM05]. It was an initial offering from "Slimmer", a jointly funded effort between INRIA and the University of Minnesota on "Sophisticated logic implementations for modeling and mechanical reasoning" from 2005 to 2010. For more information, see `http://slimmer.gforge.inria.fr/`.

What is the difference between *hoas* and *λ-tree syntax*? The term "higher-order abstract syntax" (hoas) was originally coined by Pfenning and Elliott in [PE88] and names the general practice (that was common then in, say, λProlog [MN87]) of using an abstraction in a programming or specification language to encode binders in an object-language. Since the choice of "meta-language" can vary a great deal, the term "hoas" has come to mean different things to different people. When hoas is used directly within functional programming or constructive type systems, syntax with bindings contains functional objects, which make rich syntactic manipulations difficult. Bedwyr, on the other hand, follows the *λ-tree* approach [Mil00] to hoas: in particular, Bedwyr's use of λ-abstraction is meant to provide an abstract form of syntax in which only the names of bindings are hidden: the rest of the structure of syntactic expressions remains.

Is Bedwyr efficient? Some care has been taken to implement the novel logical principles that appear in Bedwyr. In particular, the system makes extensive use of the implementation of the suspension calculus [Nad99] and other implementation ideas developed within the Teyjus [NM99] implementation of λProlog [NM88]. Aspects of tabled deduction have also been added to the system [RRR+97, Pie05]. We have

found that Bedwyr's performance is good enough to explore a number of interesting examples. It is not likely, however, that the current implementation will support large examples. For example, the system implements the occur-check within logic: this is, of course, necessary for sound deduction but it does slow unification a lot. As a result, the append program is quadratic in the size of its first argument. There are a number of well-known improvements to unification that make it possible to remove many instances of the occur-check (and making append linear). As of this time, such an improvement has not been added to Bedwyr.

An open source effort: Can I help? The Bedwyr system was conceived as a prototype that could help validate certain proof theory and proof search topic. In the end, this prototype has illustrated the main principles that we hoped that it would. It has also pointed out a number of new topics to be explored. If you are interested in contributing examples, features, or performance enhancements, or if you are interested in considering the next generation of a system like this, please let an author of this guide know: we are looking for contributions.

Background assumed To read this guide, we shall assume that the reader is familiar with the implementation of proof search that is found in, say, Prolog, λProlog, or Twelf. While familiarity with various foundations-oriented papers (particularly, [MMP03, MT05, Tiu04, Bae08b, TM10]) is important for understanding fully this system, much can be learned from studying the examples provided in the distribution.

2 Getting Bedwyr

Different means of getting Bedwyr are listed on Slimmer's INRIA Gforge project site: `http://slimmer.gforge.inria.fr/bedwyr/#download`. You can either download tarballs, get any development version using SVN, or use Slimmer's unofficial Apt repository – instructions are provided on the project page. The development of Bedwyr is meant to be an open source project. If you are keen to work on the source code and/or examples, please contact one of the "Project Admins" of the project (as listed at `https://gforge.inria.fr/projects/slimmer/`).

2.1 Distribution layout

The Bedwyr distribution is organized as follows:

`src/`	Source code
`doc/`	Documentation – you're reading it
`contrib/`	Auxiliary files – e.g. Emacs and Vim support
`examples/`	Examples – reading them helps

2.2 Build

Bedwyr's main build dependency is on the OCaml compiler suite. You also need some standard tools you probably already have, especially the GNU build system (a.k.a. the autotools, especially autoconf and GNU make), bash, tar, gzip, bzip2, etc. Most of these are looked for by the configuration step.

Then, the procedure is quite simple:

```
$ make
```

You'll get a link to the Bedwyr executable as `./bedwyr`. You can also use the expanded form to choose your building options:

```
$ autoconf
$ ./configure <configure-options>
$ make <make-targets>
```

The available options and targets are listed with

```
$ ./configure --help
$ make help
```

By default, Bedwyr is built using the bytecode compiler `ocamlc`, since compilation with it is much faster. If you don't want this (no frequent recompilation, no need for debugging), you can generate more efficient native-code instead with `ocamlopt` by passing the option `--enable-nativecode` to `./configure`.

You can also enable the documentation generation by using `--enable-doc` and then `make doc`. This userguide, the quick-start guide the `ocamldoc` documentation will be generated in `doc/`.

Last, `--with-xmlm`/`--without-xmlm` will add/remove a dependency on the `xmlm` OCaml library, needed to produce XML output. By default, the library is not required, and the feature is included if and only if the library is found.

2.3 Test

Individual components can be tested for bugs with individual targets:

```
$ make test_ndcore
$ make test_batyping
$ make test_input
$ make test_prover
```

This longer test runs the complete program on a set of example files:

```
$ make test_bedwyr
```

The `test` target runs all the previous tests, and serves as a correctness and performance test.

3 User interface

When you run Bedwyr, you specify a list of definition files, which contain objects to be declared and defined. You can then use the toplevel to ask queries against those definitions.

There is also a special brand of commands, meta-commands, which can appear anywhere.

As a general rule, any kind of input ends with a full-stop. Commands start with uppercase letters, meta-commands with hashes, and queries are just formulae.

3.1 Definition files

Definition files are usually named with a `.def` extension. You can find several of them in the `examples/` directory of the Bedwyr distribution. They contain declarations for types (`Kind <id> type.`), declarations for constants (`Type <id> type.`), declarations and definitions for predicates (`Define <id> : <type> by <definitions>.`), and theorems (`Theorem <id> : <formula>.`).

The only meta-command that is really intended for definition files is the include command: `#include "another/file.def"`. This can really be seen as the plain inclusion of another file, as Bedwyr doesn't have any namespace or module system. If the path is not absolute, it is relative to the path of the current file, or the execution path for the toplevel.

3.1.1 Emacs mode

Assuming Bedwyr is installed in standard Linux system folders, you can use the Emacs mode for Bedwyr by adding these two lines to your `~/.emacs` file:

```
(load "/usr/share/bedwyr/contrib/emacs/bedwyr.el")
(setq bedwyr-program "/usr/bin/bedwyr")
;; Of course you can change both locations to wherever you want.
```

Then you should be able to load any `.def` file and have syntax highlighting and some rough auto-indenting. Also if you do `C-c C-c` it will start Bedwyr and load the current file you are working on.

3.1.2 Vim syntax highlighting

There is also a basic syntax highlighting file for Vim. With a standard system installation, the files `/usr/share/vim/vimfiles/[ftdetect|syntax]/bedwyr.vim` should suffice; otherwise do the following:

- copy `contrib/vim/syntax/bedwyr.vim` to your `~/.vim/syntax/` to make highlighting available
- copy `contrib/vim/ftdetect/bedwyr.vim` to your `~/.vim/ftdetect/` to have it used automatically for all `.def` files

3.2 Toplevel

The interactive toplevel is automatically launched once the files have been parsed, unless the flag `-I` is passed to Bedwyr. In it, you can either query a formula, or run a meta-command. In queries, free and bound variables are the only objects that can be used without prior declaration, and the solutions are displayed as instantiations of free variables.

Queries can also be given in batch mode, to a non-interactive toplevel, via the command-line option `-e` (e.g. `bedwyr -e 'X = 0.'`). In this case, they are processed after the files and before the interactive toplevel.

In Listing 1 we load a set of definitions and prove (twice) that the untyped λ-term $\lambda x.x\ x$ has no simple type.

Notice that we had to use the term `(abs x\ app x x)` instead of `(x1\ x1 x1)`: the former encodes the untyped λ-term $\lambda x.x\ x$ by mapping object-level abstraction to `abs`

Listing 1: Run on `examples/lambda.def`.

```
?= (exists T, wt void (abs x\ app x x) T).
No.
?= (exists T, wt void (abs x\ app x x) T) -> false.
Yes.
More [y] ?
No more solutions.
?=
```

and object-level application to `app`, while the latter would map them directly to logic-level abstraction and application, and therefore is not a legal term in Bedwyr. (Prior to version 1.3, this was allowed as Bedwyr did not use simple typing on its own terms.)

Most of the errors that can stop the reading of a file (parsing or typing error, undeclared object, etc.) are correctly caught by the toplevel, though the line number reported for the error is often not useful.

3.2.1 Line editing

Bedwyr has no line editing facilities at all. Thus, we recommend using `ledit` or `rlwrap`, which provides such features. Get one of them from your usual package manager or at `http://pauillac.inria.fr/~ddr/ledit/` or `http://utopia.knoware.nl/~hlub/rlwrap/#rlwrap`.

Then you can simply run `ledit bedwyr`. One can also define an alias in his `~/.bashrc`, such as the following which also makes use of `~/.bedwyr_history` to remember history from one session to another:

```
alias bedwyr="ledit -h ~/.bedwyr_history -x /path/to/bedwyr"
```

3.3 Meta-commands

Meta-commands are used to change the state or the program, or for non-logical tasks. They can be used any time a command or query is allowed.

3.3.1 Session management

Those commands alter the set of definitions the current session of Bedwyr holds. An empty session actually means that only pervasive types, constants and predicates are known. The session's initial state is the list of files given on the command-line, and it can grow anytime with the use of `#include`. It should be noted that, although Bedwyr has no solid notion of what a module is, it tries to do the smart thing by ignoring the request to include a file if it appears to be already loaded in the current session, as failure to do this would result in fatal multiple declarations. This only works if the same path is used for each inclusion; for instance, `./file.def` and `file.def` will be seen as different files.

- `#include` adds a `.def` file to the current session. It is designed to be used in definition files.

- `#session` is an advanced `#include` meant for query mode. It accepts any number of filenames as parameters, and this set of files will be remembered as the new session. When you pass filenames on Bedwyr's command line, it is equivalent to a call to `#session` with these definition files.
- `#reload` clears all the definitions, and then reloads the session's initial state, i.e. the definition files given on the command-line. It is useful if they have been changed.
- `#reset` clears all the definitions and empties the session. It is synonymous to `#session` with no arguments.

3.3.2 Assertions

Three kinds of assertions can be used in definition files. These tests are not executed unless the `-t` flag has been passed on Bedwyr's command-line, in which case any assertion failure is fatal.

- `#assert` checks that a formula has at least one solution.
- `#assert_not` checks that a formula has no solution.
- `#assert_raise` checks that the proof-search for a formula triggers a runtime error.

Our examples include a lot of assertions, to make sure that definitions have (and keep) the intended meaning. These assertions are also the basis of Bedwyr's correctness and performance tests ran using `make test`.

3.3.3 Other commands

- Output
  - `#env` displays the current session (types, constants and predicates).
  - `#typeof` displays the (monomorphic) type of a formula and of its free variables.
  - `#show_def` displays the definition of a predicate.
  - `#show_table` displays the content of the table of an inductive of co-inductive predicate.
  - `#save_table` writes the table of an inductive of co-inductive predicate in a fresh definition file.
  - `#export` exports a structured aggregate of all tables in a file. This functionality has to be enabled at compile-time.
- Tabling (i.e. memoization or caching)
  - `#equivariant` enables an alternative tabling mode.
  - `#clear_table` clears the results cached for a predicate.
  - `#clear_tables` clears all cached results.
  - `#freezing` sets the depth of backward-chaining during proof-search.
  - `#saturation` sets the depth of forward-chaining during proof-search.

- General purpose
  - `#debug` adds a lot of output during the proof search.
  - `#time` displays computation times between two results.
  - `#help` is what you should type first.
  - `#exit` is what you should type last.

Part II
Tutorials

4 Released examples

> Few things are harder to put up with
> than the annoyance of a good example.
> – Mark Twain

The distribution of Bedwyr comes with several examples of its use. These examples can be classified roughly as follows.

Basic examples These examples are small and illustrate some simple aspects of the system.

Model checking Some simple model-checking-style examples are provided.

Games Bedwyr allows for a simple approach to explore for winning strategies in some simple games, such as tic-tac-toe.

λ-calculus Various relations and properties of the λ-calculus are developed in some definition files.

Simulation and bisimulation These relationships between processes where an important class of examples for which the theory behind Bedwyr was targeted. Examples of checking simulation is done for abstract transition systems, value-passing CCS, and the π-calculus. The π-calculus examples are of particular note: all side-conditions for defining the operational semantics and bisimulation are handled directly and declaratively by the logic underlying Bedwyr. See section 6 below for some more details about the π-calculus in Bedwyr.

5 Hypothetical reasoning

For those familiar with λProlog, a key difference between Bedwyr and λProlog is that the latter allows for "hypothetical" reasoning and such reasoning is central to the way that λProlog treats bindings in syntax. Bedwyr treats implication and universals in goal formulas in a completely different way: via the closed world assumption.

Sometimes, when dealing with λ-tree syntax in Bedwyr, one wishes to program operations as one might do in λProlog. This is possible in the sense that one can write in Bedwyr an interpreter for suitable fragments of λProlog. This is done, for example, in the `seq.def` definition file. There is a goal-directed proof search procedure for a small part of hereditary Harrop formulas (in particular, the minimal theory of the fragment based on $\top$, $\wedge$, $\supset$, and $\forall$). This interpreter is prepared to use a logic program that is stored as a binary predicate. For example, in λProlog, one would write type checking for simple types over the untyped λ-calculus encoded using `app` and `abs` as in Listing 2. The hypothetical reasoning that is involved in typing the object-level λ-binder in the second clause above is not available directly in Bedwyr. One can, however, rewrite these clauses as simply "facts" in Bedwyr (Listing 3).

The first definition describes a logic program called `simple` that directly encodes the above λProlog program; the second definition tells the interpreter

Listing 2: Simple typing in Teyjus.

```
typeof (app M N) B :- typeof M (arrow A B), typeof N A.
typeof (abs R) (arrow A B) :- pi x\ typeof x A => typeof (R x) B.
```

Listing 3: Simple typing in Bedwyr (from `examples/progs_small.def`).

```
Define simple : form -> form -> prop by
  simple (type_of (app M N) Tb)
    (type_of M (Ta ~> Tb) && type_of N Ta);
  simple (type_of (abs R) (Ta ~> Tb))
    (for_all x\ type_of x Ta --> type_of (R x) Tb).

Define atom : form -> prop by
  atom (type_of X T).
```

in `seq.def` how to recognize an object-level atomic formula. A call to `seq atom simple tt (type_of Term Ty)` will now attempt to perform simple type checking on `Term`. Specifically, it is possible to prove in Bedwyr the goal

```
(exists Ty, seq atom simple tt (type_of (abs x\ app x x) Ty))
  -> false.
```

or, in other words, that the self-application $\lambda x(xx)$ does not have a simple type.

This "two-level approach" of specification uses Bedwyr as a meta-language in which a simple intuitionistic logic is encoded as an object logic: computations can then be specified in the object-logic in the usual way and then Bedwyr can be used to reason about that specification. This general approach has been described in more detail in [Mil06, GMN10].

6 The π-calculus example in more detail

To illustrate another example and how it can be used, consider the implementation of the π-calculus that is contained in the example file `pi/pi.def`. Of the several things defined in that file, the operational semantics for the π-calculus is given using one-step transitions: for a specific example, see Listing 4.

Beyond the syntactic differences, the operational semantics of λProlog and Bedwyr differ significantly. If a specification is simply a Horn clause program, the two systems coincide. They differ in the operational interpretation of implication: in Bedwyr, to prove $A \supset B$, all possible ways to prove A are explored and for each answer substitution θ that is found, the goal $B\theta$ is attempted (see subsection 7.5). Bedwyr also contains the ∇-quantifier [MT05].

Returning to the example in Listing 4, notice that two predicates are defined: `one` and `onep`. The first one relates a process, an action, and a process. The second one relates a process, an abstraction of an action, and an abstraction of a process. The `one` predicate is used to capture "free transitions" and the "τ-transition" while the second

Listing 4: Some one-step transitions (from `examples/pi/pi.def`).

```
Define
  one  :  p ->          a  ->          p  -> prop,
  onep :  p -> (name -> a) -> (name -> p) -> prop
by
  onep (in X M) (dn X) M;
  one  (out X Y P) (up X Y) P;
  one  (taup P) tau P;
  one  (match X X P) A Q := one P A Q;
  onep (match X X P) A M := onep P A M;
```

Listing 5: (Open) bisimulation (from `examples/pi/pi.def`).

```
Define coinductive bisim : p -> p -> prop by
  bisim P Q :=
    (forall A P1, one P A P1 ->
      exists Q1, one Q A Q1 /\ bisim P1 Q1) /\
    (forall X M, onep P (dn X) M ->
      exists N, onep Q (dn X) N /\
      forall w, bisim (M w) (N w)) /\
    (forall X M, onep P (up X) M ->
      exists N, onep Q (up X) N /\
      nabla w, bisim (M w) (N w)) /\
    (forall A Q1, one Q A Q1 ->
      exists P1, one P A P1 /\ bisim P1 Q1) /\
    (forall X N, onep Q (dn X) N ->
      exists M, onep P (dn X) M /\
      forall w, bisim (M w) (N w)) /\
    (forall X N, onep Q (up X) N ->
      exists M, onep P (up X) M /\
      nabla w, bisim (M w) (N w)).
```

Listing 6: Run on `examples/pi/pi.def`

```
?= bisim (in a x\ in a y\ z)
     (in a x\ nu w\ in a y\ out w w z).
Yes.
More [y] ? y
No more solutions.
?= bisim (in a x\ nu y\ match x y (out c c z))
     (in a x\ z).
Yes.
More [y] ? y
No more solutions.
?= bisim (nu x\ out a x (in c y\ match x y (out c c z)))
     (nu x\ out a x (in c y\ z)).
No.
?=
```

is used to capture bounded transitions. See [TM04, Tiu05] for more details on this encoding strategy for the π-calculus.

Listing 5 provides all that is necessary to specify (open) bisimulation for (finite) π-calculus. The keyword `coinductive` tells the system that it will be attempting to explore a greatest fixed point. That keyword also enables tabling, which avoids redundant computations and accept loops as successes (see section 12). The other cases should look natural, at least once one understands the λ-tree approach to representing syntax and the use of the ∇-quantifier. The main thing to point out here is that in the specification, no special side conditions need to be added to the system: all the familiar side conditions from the usual papers on the π-calculus are treated by the implementation of the Bedwyr logic: the user of the system no longer needs to deal with them explicitly but implicitly and declaratively (via quantifier scope, $\alpha\beta\eta$-conversion, etc.).

It is now possible to test some simple examples in the system, for example Listing 6. These query prove that $a(x).a(y).0$ and $a(x).(\nu w).a(y).w!w.0$ are bisimilar, that $a(x).(\nu y).[x = y].c!c.0$ and $a(x).0$ are bisimilar, and that $(\nu x).a!x.c(y).[x = y].c!c.0$ and $(\nu x).a!x.c(y).0$ are not bisimilar.

Several other aspects of the π-calculus are explored in the examples files of the distribution. For example, the file `pi/pi_modal.def` contains a specification of the modal logics for mobility described in [MPW93], the file `pi/corr-assert.def` specifies the checking of "correspondence assertions" for the π-calculus as described in [GJ03], and the file `pi/pi_abscon.def` specifies the polyadic π-calculus following [Mil99].

Listing 7: Computing the maximum of a defined predicate (λProlog).

```
% The predicate a holds for 3, 5, and 2.
a (s (s (s z))).
a (s (s (s (s (s z))))).
a (s (s z)).

% The less-than-or-equal relation
leq z N.
leq (s N) (s M) :- leq N M.

% Compute the maximum of a
maxa N :- a N, pi x\ a x => leq x N.
```

Part III
System Description

7 The logic LINC

The logic behind Bedwyr, named LINC, is an extension to a higher-order version of intuitionistic logic that has been developed over the past few years. The acronym LINC, which stands for "lambda, induction, nabla, and co-induction", lists the main novel components of the logic. In particular, λ-terms are supported directly (and, hence, the λ-tree syntax approach to higher-order abstract syntax is supported [Mil00]). Induction and co-induction are also available. The nabla (∇) quantifier has been added to this logic in order to increase the expressiveness of programs using λ-tree syntax in negated situations. The proof theory of LINC is given in [MT05, Tiu04]. Since this earlier work on LINC, more recent work on the logic $\mathcal{G}$ [GMN10, GMN11] and with fixed points in linear logic [Bae08a, Bae12] has further improved our understanding of using fixed points, induction, co-induction, and ∇-quantification.

Below we provide a high-level overview of the logical aspects of Bedwyr. More explicit information on this system can be found in [TNM05]. (N.b. the name "Level 0/1" in that paper has now been replaced by Bedwyr) Next, we describe the two orthogonal extensions to higher-order intuitionistic logic that have been incorporated into Bedwyr.

7.1 Built-in treatment of bindings

Bedwyr treats λ-abstractions within terms as primitives as well as allowing for variables of function type and quantifiers within formulas ($\forall$, $\exists$, ∇). The system implements "higher-order pattern unification" (see section 8). This kind of unification appears to be the weakest extension to first-order unification that treats bindings as a primitive. A number of automated deduction systems implement this kind of unification (e.g. Twelf, Teyjus, Coq, and Minlog). Full β-conversion is implemented by Bedwyr as well.

7.2 Syntax and semantics of definitions

Some systems implementing aspects of higher-order logic programming, such as λProlog, accept the "open-world assumption": any conclusion drawn in their logic

Listing 8: Computing the maximum of a defined predicate (Bedwyr).

```
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 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.
```

will hold in any extension of the underlying logic programming language. For example, consider the λProlog program in Listing 7 (the signature has been left out), where the last clause has an implication `=>` in the goal. During proof search, this implication causes λProlog to add a new eigenvariable, say `c`, to the runtime signature and to extend the current program with an atomic fact about it: `(a c)`. In such a new world, however, the `leq` relation does not have any information about this "non-standard" number `c`.

Bedwyr on the contrary accepts the "closed-world assumption": the notion of programs is replaced by *definitions* that capture the "if-and-only-if" closure of logic programs (Listing 8). One of the syntactic difference between the syntax of clauses and that used in λProlog is that the head and body of clauses are separated from each other using the `:=` instead of the `:-` (turnstile). The former symbol is used to remind the Bedwyr user of that "if and only if" completion of specifications.

Bedwyr takes the assumption `(a c)` and asks "Given the assumption that `(a c)` is true, how could have it been proved?" The natural answer to this is that that assumption could have been proved if `c` was either 3 or 5 or 2. Thus, this will cause a case analysis: in particular, the query `(maxa N)` will cause the following goals to be considered:

```
(a N)    (leq 3 N)    (leq 5 N)    (leq 2 N)
```

Here we use the numeric symbols '2', '3', etc., as abbreviations of the corresponding terms formed using `z` and `s`. The usual approach to unification and depth-first proof search will now produce the proper maximum value. This change allows Bedwyr to give a computational interpretation to finite failure and to do deduction that encodes model checking.

7.3 Symmetry of finite success and finite failure

The underlying logic of *fixed points* (also known as *definitions*) [Gir92, SH93, MMP03, MT03] contains an inference rule that allows for failure in unification (and, hence, in

simple proof search) to be turned into a success. Thus, simple forms of "negation-as-failure" can be naturally captured in Bedwyr and the underlying logic. It is also possible to describe both *may* and *must* behaviors in process calculi. For example, not only can one code reachability in process calculus but bisimulation is also possible. One way to view this enhancement to proof search is the following: Let A and B be two atomic formulas. Then, finite success is captured by proving the sequent $\longrightarrow A$, finite failure is captured by proving the sequent $A \longrightarrow$, and simulation is captured by proving the sequent $A \longrightarrow B$.

7.4 The ∇ quantifier

In order to treat specifications using λ-tree syntax properly, it appears that a new quantifier, called ∇, is necessary. If finite success is all that is needed, the ∇ can be replaced with the universal quantifier. When finite failure is involved, however, the ∇ quantifier plays an independent role. See [MT05, Tiu04, Tiu05] for more on this quantifier. It is worth pointing out that we know of no examples involving ∇ that do not also involve λ-tree syntax.

7.5 Proof search within LINC

Bedwyr is a proof search engine for a small fragment of the LINC logic. In principle, Bedwyr uses two provers. *Prover-1* is similar to the depth-first interpreter used in λProlog. The main difference is in the proof of an implication. To prove an implication $A \Rightarrow B$, prover-1 calls *prover-0* to enumerate all possible solutions $\{\theta_i \mid i = 1, \ldots, n\}$ for A, and then prover-1 tries to prove $B\theta_1 \wedge \cdots \wedge B\theta_n$. If A has no solution (that is, if $n = 0$), the implication is true. The substitutions generated by prover-1 are for existential[1] variables, as usual in logic programming. On the other hand, the substitutions generated by prover-0 are for universal variables.

To illustrate this, consider the following goal:

$$\forall x \,.\, (\exists y \,.\, x = s\ y) \Rightarrow x = 0 \Rightarrow false$$

(This formula formalizes the fact that if x is the successor of some number then x is not zero.) Bedwyr will call prover-1 on it. The prover introduces a universal variable and reaches the first implication. It then calls prover-0 on $(\exists y \,.\, x = s\ y)$. Prover-0 introduces an existential variable y, and the unification instantiates x to get the only solution. Back to prover-1, we have to prove $(s\ y = 0 \Rightarrow false)$ where y is still an existential variable. Prover-0 is given $s\ y = 0$ to prove and fails to do so: that failure is a success for prover-1.

We've seen in section 7 with the `maxa` example (Listing 8) how this treatment of the implication allows Bedwyr to check formulas which are not provable in traditional (pure) logic programming languages such as λProlog. As often, this novelty has a price. The systematic enumeration leads to infinite search for simple formulas like $(A \Rightarrow A)$ as soon as A does not have a finite number of solutions. Further development of Bedwyr may provide real support for induction and co-induction.

Prover-0 is similar to prover-1. The first difference is this dual treatment of variables; soundness requires another one. Because it needs to completely destruct for-

[1] We avoid the usual names (logic variables for existential variables and *eigenvariables* for universal variables) in order to clearly separate the high-level description given here from the implementation, which is not detailed here, but in which the class of a variable isn't static.

mulas in order to enumerate solutions, prover-0 requires its connectives to be *asynchronous* on the left: they can be immediately destructed (introduced, in sequent calculus terminology) without restricting provability. This means that implication and universal quantification are forbidden on the left of implications.

Prover-1 instantiates existential variables, and considers universal variables as (scoped) constants. Prover-0 produces substitutions for universal variables, considers existential variables introduced in prover-0 as constants, but we have no satisfactory answer for existential variables introduced in prover-1. As a consequence, in prover-0, unification raises an run-time error when the instantiation of an existential variable is needed. More details about that can be found in Section 13.2.

8 Unification

A subset of λProlog, called L_λ, was presented in [Mil91] where it was shown that an implementation of proof search could be written in which only a small subset of higher-order unification was required. Furthermore, that subset was decidable, unary, and did not need typing information. This subset of unification was called L_λ-unification in [Mil91] but is now more commonly referred to as *higher-order pattern unification* [Nip93, NL05]. In that subset, variables in functional position are applied to distinct variables which must be bound in the scope of the binding of the functional variable.

For instance,

```
exists X, forall y z, X y z = y
```

can be solved, but

```
exists X, forall y, X y y = y
```

can't, as it violates the first constraint (so that `X` can take at least two values, `x1\ x2\ x1` and `x1\ x2\ x2`), and

```
forall y, exists X, forall z, X y z = y
```

can't either, as it violates the second constraint and therefore could be rewritten

```
exists X', forall y z, (X' y) y z = y
```

Bedwyr uses an extension of this higher-order pattern unification which handles ∇.

9 Typing

Terms in Bedwyr form a strongly typed language with polymorphism and type constructors. This language is statically type-checked; once definition files are loaded and queries are read, types are discarded and the prover handles only untyped terms. Therefore, to ensure that "well-typed formulae don't go wrong", a form of the Hindley-Milner type system is used instead of the full System F_ω. The polymorphism has therefore those properties:

parametric type parameters are given as uppercase letters in constant or predicate declarations

Listing 9: Polymorphism in Bedwyr.

```
Kind option   type -> type.
Type none     option A.
Type some     A -> option A.

Define print? : option A -> prop by
  print? none ;
% print? (some 42) ;
  print? (some X) := println X.
```

predicative terms are always monomorphic (apart from definitions), so the type parameters of a polymorphic object have to be instantiated with monotypes whenever it is used in a term

prenex type quantifiers can only occur at the outermost level of a type, and therefore can be omitted

As the language is fairly specific, what we have is not *let-polymorphism* but "*define-polymorphism*": while it is not possible to give a polytype to a bound variable (whether it is bound by an abstraction or a quantifier), a definition can be polymorphic, and must be if the predicate was declared so. With the syntax Bedwyr uses for clauses, this means that the type of the occurence of a predicate at the head of the application that is itself the head of a clause is not instantiated. In Listing 9, the commented out clause wouldn't type-check as it assumes `print?` has type `option int` instead of `option A`, and the last clause type-checks as `println` is itself polymorphic and adds no constraints on the type of `X`.

The only two constraints on type parameters are that they must be of kind `type`, and therefore cannot appear at the head of an type application, and that types must be definite, i.e. a type parameter that appears in a type must appear in the goal of that type, so as to forbid heterogeneous wrappers like `Type c A -> t`.

Recursive algebraic types are de facto available via constant declarations, e.g. the predeclared type constructor `list` is morally defined as

```
list A = nil | (::) of (A * list A)
```

in pseudo-OCaml notation. It is even possible to emulate type deconstruction (matching) by using clause heads that cannot unify simultaneously with a ground term, as is done in Listing 9 with the constants `none` and `some`.

10 Definition files

Type declarations only use `type` and `->`, and as type constructors can only be applied on proper types, `type` never appears under more than one `->` (Figure 1).

Constant declarations have the structure described in Figure 2, with additional constraints on type variables as described in section 9.

Definitions are given as blocks with a header containing predicate declarations, and an optional body containing a set of clauses, in which uppercase variables are implicitly universally quantified (Figure 3). A predicate with no definition clauses is always false; the head of a bodiless clause is always true. A predicate can only depend

```
type_decl   ::=  Kind <id> <kind>.
kind        ::=  type -> <kind>
             |   type
```

Figure 1: Grammar of type declarations.

```
constant_decl   ::=  Type <id> <type>.
type            ::=  <type> -> <type>
                 |   <type_atom>
type_atom       ::=  <type_atom> (<type_atom>)
                 |   <type_atom> <type_variable>
                 |   <id>
```

Figure 2: Grammar of constant declarations.

on predicates defined up to its definition block, so multiple predicates in one block is the only way to achieve mutual recursion. One can see definition blocks as groups of predicates belonging to the same stratum; stratification forbids predicates from the same block to depend negatively one on the other, as usual, but here it also forbids the use of `inductive` and `coinductive` in the same block for similar reasons.

```
definition_block   ::=  Define <declarations>.
                    |   Define <declarations> by <clauses>.
declarations       ::=  <declaration> , <declarations>
                    |   <declaration>
declaration        ::=  <flavour> <id> : <type>
flavour            ::=  inductive
                    |   coinductive
                    |
clauses            ::=  <clause> ; <clauses>
                    |   <clause>
clause             ::=  <head> := <body>
                    |   <head>
head               ::=  <id> <atom>*
body               ::=  <formula>
```

Figure 3: Grammar of predicates declarations and definitions.

Theorems are horn-like formulae which head must be an atom obtained by the application of an already defined predicate (Figure 4). For a non-recursive predicate, they can be seen as additional clauses, admissible by the definition with respect to the logic, if not to Bedwyr. Their names holds no semantic value.

11 Concrete syntax

Although the concrete syntax of Bedwyr was originally derived from that of λProlog in the Teyjus implementation[NM99], it has now evolved in such a way that, by design, it resembles that of the Abella proof assistant [Gac10]. This explains the lack of syntactic compatibility with versions earlier than 1.3.

theorem ::= `Theorem` <id> `:` <body> `->` <head>`.` <proof> `Qed.`

Figure 4: Grammar of theorems specifications.

formula	::=	<term> `=` <term>	equality
	|	<formula> `/\` <formula>	conjunction
	|	<formula> `\/` <formula>	disjunction
	|	<formula> `->` <formula>	implication
	|	<quantifier> <bound_variable>`+,` <formula>	
	|	<term>	
quantifier	::=	**`forall`**	universal
	|	**`exists`**	existential
	|	**`nabla`**	generic

Figure 5: Grammar of formulae.

11.1 Formulae

Formulae are described in Figure 5, separately from terms, as it is customary to have the former contain the later and not the other way around. However, one can actually write `((x\ (p x /\ q x)) c)` instead of `(p c /\ q c)`, and the parser takes it into account by allowing a formula to be interpreted as a term.

Quantifiers are n-ary, and their scope extends to the right as far as possible:

`(forall x y, f y x = g x y)` ≡ `(forall x y, (f y x = g x y))`

11.2 Terms

Within terms, the highest priority goes to the regular application (Figure 6). In particular, it has precedence over the application of an infix constant:

`(w x ** y z)` ≡ `(w x) ** (y z)`

All infix constants have the same priority, and are right-associative, to mimic the behavior of `->`.

An infix constant usually has to be at least of arity 2 to be read: `(x **)` raises a parsing error, while `(x ** y)` can be applied to another term if `**` is of arity 3 or more. It is also possible to use the prefix version of an infix constant by surrounding it with parentheses, in which case any arity is permitted: `((**) x)` and `((**) x y z)` are both syntactically legal.

The abstraction over variable `x` in `term` is denoted by `x\ term` – which is read as $\lambda x.term$. The scope of the infix λ-abstraction extends to the right as far as possible inside of a term, but not across formula operators:

`(x\ y\ f y x = g x y)` ≡ `((x\ (y\ (f y x))) = (g x y))`

11.3 Tokens

Names for objects such as types, predicates, constants and variables are character strings built with letters, digits and the special characters {`-^<>=~+*&:|`}, {`'$?`} and

term	::=	<atom>+	application
	|	<atom>* <abstraction>	application on an abstraction
	|	<term> <infix_id> <term>	(partial) infix application
abstraction	::=	<bound_variable>\ <term>	
atom	::=	**true**	**prop**
	|	**false**	**prop**
	|	"<string>"	**string**
	|	[0-9]+	**nat**
	|	(<formula>)	
	|	(<infix_id>)	prefix form of an infix constant
	|	<id>	declared object
	|	<bound_variable>	bound (or free) variable

Figure 6: Grammar of terms.

{_/@#!} (Figure 7). Names must be separated by space characters (SPACE, TAB, CR, LF), parentheses or C-style inline nested comments (/* */). As a general rule, the first character of a name determines the kind of name it is, and cannot be a digit.

More precisely, we divide the special characters into three categories:

- infix characters: -^<>=~+*&:|
- prefix characters: ‘’$?
- tail characters: _/@#!

which gives us three token categories:

upper names start with A-Z; contain any letter, digit, prefix character or tail character (i.e. anything but an infix): Foo?0, B@r, My_Var'

prefix names start with a-z or a prefix character; contain any letter, digit, or prefix or tail character (i.e., anything but an infix): l33t, h#sh, ?Your_Var

infix names contain only infix characters: -->, |=, ^^

Keywords are implicitly excluded from those definitions.

Types and predicates must have prefix names, constants can have either prefix or infix names, and bound variables (from quantifiers of λ-abstractions) can have either upper or prefix names, though it is customary to use an upper name for an existentially quantified variable and a prefix name for the others.

In a term or a type, all unbound infix or prefix names must be declared, and unbound upper names (which cannot be declared objects) are free variables. Those are implicitly universally quantified in files (i.e. in types and clauses), and existentially quantified in queries. Though no name can begin with the special character _[2], it can serve as a placeholder: it is a fresh one-time free variable, except when used instead of a variable name in a binding, where it denotes a vacuous abstraction.

One more constraint restrict the range of names. As already said, names must be separated by space characters or comments. This is true even if they are not names of

[2] Actually, such names exist and are accepted by the parser, but are rejected in type, constant and predicate declarations, as they are read-only names, only used for undocumented, internally defined predicates (usually experimental, non-logical, and with side-effects).

id	::=	<prefix_name>
	|	<infix_name>
bound_variable	::=	<upper_name>
	|	<prefix_name>
type_variable	::=	<upper_name>
infix_id	::=	<infix_name>
upper_name	::=	`[A-Z][a-zA-Z0-9‘’$?_/@#!]*`
prefix_name	::=	`[a-z‘’$?][a-zA-Z0-9‘’$?_/@#]*`
infix_name	::=	`[-^<>=~+*&:|]+`

Figure 7: Grammar of type declarations.

the same kind, e.g. infix characters and other characters cannot be contiguous. This explains why the spaces are mandatory in `(X -> Y)` or even `(X = Y)`. The only allowed exceptions are the special sequences `/*` and `*/`, which can appear right after (resp. before) a prefix name, and which always start (resp. end) a level of inline comment[3], except when in a quoted string or a single-line comment.

12 Tabling

Proof search for a defined atom is done by unfolding the definition for the atom, i.e. by replacing it with the body of the definition. Since (mutually) recursive definitions are allowed, it is possible that loops occur in the proof search. The same goals can also arise several times in different searches. By default, Bedwyr doesn't detect any of these issues, which makes the proof search much longer than needed, or even infinite. To address this, Bedwyr can use tabling to keep records of certain proved, disproved or ongoing formulae, hence avoiding redundant search.

Tabling is used in both prover-0 and prover-1 (see Section 7.5). The current implementation restricts tabling to atomic goals with no occurrence of existential variables in prover-1. In prover-0, only ground atomic goals (no occurrence of existential or universal variables) are tabled.

The use of tabling as a memoization mechanism is straightforward: once an atom is *proved* or *disproved*, it is marked as such in the table, and this result can be reused in any later computation. On the other hand, while the proof search for an atom is still ongoing, the atom is marked as *working*, and any new occurrence of it will mean that a loop in the search has been found. This can have several interpretations, depending on whether we consider the predicate as *inductive* or *co-inductive*. In the former case, that means that the atom is not provable, since otherwise it would contradict the well-foundedness of inductive definitions. In the latter case, we would have a proof of the atom. This simple loop checking makes it possible to do proof search for some non-terminating definitions.

Tabling is by default not enabled in Bedwyr. To enable it, two keywords are provided: `inductive` and `coinductive`. To use tabling on a predicate `p`, one of them has to be added in the declaration of `p`, in the header of the definition block. Note that, while one can mix tabled and non-tabled predicates in the same definition block by only applying such a keyword to some of the predicates, the scope of the inductive or co-inductive trait is the whole block of mutually recursive definitions. This means

[3]Contrary to Teyjus, Bedwyr doesn't see two variables in the expression `X/* Y*/`.

that a definition block cannot contain both inductive and co-inductive predicates at the same time, as it might lead to contradictions – see [MT03] for more details.

12.1 Table output

The command `#show_table pred.` allows one to inspect the contents of `pred`'s table outside of any computation, when it only contains proved and disproved atoms. The output displays one formula per line, with the prefix `[P]` for proved and `[D]` for disproved. The formulas are abstracted over by their generic and universal variables. The relative scopings of generic and universal variables is not displayed although that information is present internally: such information is needed, for example, to avoid that a proof of $(\forall x \nabla y \,.\, p\ x\ y)$ is used as a proof for $(\nabla y \forall x \,.\, p\ x\ y)$. The displaying of this information will be fixed with planned extensions of the tabling mechanisms that will implicitly allow extra axioms on ∇ (see [Tiu06]) in order to be able to inspect in a meaningful way one predicate's table from another logic program.

For example, if we define

```
Define inductive neq : nat -> nat -> prop by
  neq X Y := X = Y -> false.

Define query1 : prop, query2 : prop by
  query1 := forall x, nabla y, neq x y ; % true
  query2 := nabla y, forall x, neq x y.  % false
```

and ask the queries `query1.` and `query2.` , we end up with the following puzzling table:

```
?= #show_table neq.
Table for neq contains (P=Proved, D=Disproved):
 [P] nabla x1, x2\ neq x2 x1
 [D] nabla x1, x2\ neq x2 x1
?=
```

The two entries are indistinguishable by the user, but internally some extra information does separate them.

Tables can be reset with the commands `#clear_table` and `#clear_tables`.

12.2 Table extraction

Two means of extracting tabled information exist in Bedwyr. The first is the `#save_table` command, which is similar to `#show_table` but outputs the table in a definition file as a pair of predicates (Listing 10), `proved` and `disproved`. This way, it is possible for Bedwyr to reason about its own tables.

The other method is the `#export` command. It outputs the whole set of tabled atoms of the current session in a structured way (Listing 11), not unlike the *trees of multicut derivations* described in [Nig08]. Note that this tree can contain atoms from multiple predicates, and therefore cannot be built if some tables were selectively removed by `#clear_table`.

Listing 10: `#save_table path "path-table.def".`

```
% Table for path contains :
Define proved : (s -> s -> prop) -> s -> s -> prop,
  disproved : (s -> s -> prop) -> s -> s -> prop by
  disproved path (state 1 1 0) (state 5 5 1) ;
  disproved path (state 1 1 0) (state 5 5 1) ;
[...]
  disproved path (state 1 1 0) (state 5 5 0) ;
  disproved path (state 2 1 1) (state 5 5 0).
```

Listing 11: `#export "tables.xml".`

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE skeleton SYSTEM "bedwyr.dtd">
<?xml-stylesheet type="text/xsl" href="bedwyr-skeleton.xsl"?>
<skeleton timestamp="1365771164">
  <son value="disproved" id="57">
    <atom>path (state 3 1 1) (state 5 5 1)</atom>
    <son value="disproved" id="113">
      <atom>path (state 1 1 0) (state 5 5 1)</atom>
      <son value="disproved" id="33">
        <atom>path (state 2 1 1) (state 5 5 1)</atom>
        <loop value="disproved" ref="57">
          path (state 3 1 1) (state 5 5 1)
        </loop>
[...]
```

12.3 Tabling modulo

Version 1.3 introduced tabling modulo theorems, where simples lemmas can be used to improve the efficiency of tabling in two ways:

backward-chaining uses a lemma as an additional definition clause, and unifies its head with a queried atom to expand the range of the search in the table. For instance, if the lemma $A \wedge B \Rightarrow C$ is known and the table contains $A\theta$ and $B\theta$, then the query $C\theta$ can be solved without unfolding a (possibly complicated) definition.

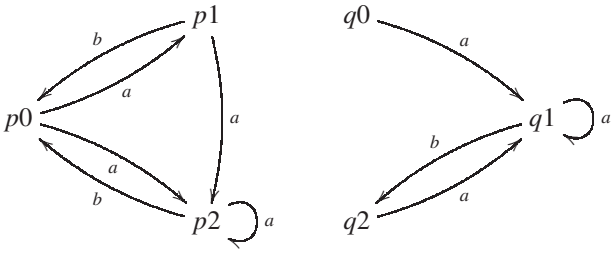
forward-chaining uses a lemma to fill the table faster: with the same lemma, if the table contains $A\theta$, then upon solving and tabling $B\theta$, $C\theta$ is de facto solved and can be tabled without even having been queried.

Lemmas obviously have to be admissible by the definitions; they are merely shortcuts that ease the access to results too complex for Bedwyr to compute quickly or at all. The first examples that come to mind are symmetry or transitivity lemmas. They can be added to files as theorems, using the Abella syntax, and the subsequent text is ignored until the command `Qed` is met. That way, parsing and checking the proof is left to Abella.

12.4 A bisimulation example

In some cases the table contents has important uses: for example, once the co-inductive predicate `bisim` (for bisimulation in some of the example files) has been checked, the table for the predicate `bisim` describes a bisimulation. We give here a simple example of checking bisimulation of finite state automata. The example is distributed with Bedwyr as `bisim.def`. For more sophisticated examples involving the π-calculus, we refer the reader to section 6.

Consider the following transition system (taken from [Mil99], page 19):



The state `p0` and `q0` are bisimilar (see [Mil99] for a proof). This transition system and the bisimulation relation are encoded in Bedwyr as shown in Listing 12. Using this definition of bisimulation, Bedwyr is able to prove that `p0` and `q0` are indeed bisimilar (Listing 13).

Listing 12: Excerpt from `examples/bisim.def`.

```
Define next : state -> trans -> state -> prop by
  next p0 a p1;
  next p0 a p2;
  next p1 b p0;
  next p1 a p2;
  next p2 a p2;
  next p2 b p0;
  next q0 a q1;
  next q1 a q1;
  next q1 b q2;
  next q2 a q1.

Define coinductive bisim : state -> state -> prop by
  bisim P Q :=
    (forall P1 A, next P A P1 ->
                  exists Q1, next Q A Q1 /\ bisim P1 Q1) /\
    (forall Q1 A, next Q A Q1 ->
                  exists P1, next P A P1 /\ bisim P1 Q1).
```

The table produced gives exactly the bisimulation set needed to prove the bisimilarity of `p0` and `q0`, i.e. the set $\{(p0, q0), (p0, q2), (p1, q1), (p2, q1)\}$.

13 Limitations of the interpreter

The strategy used by Bedwyr for attempting proofs is not complete. That strategy involves using two provers (prover-0 and prover-1), tabling, and depth-first search. Many

Listing 13: Run on `examples/bisim.def`.

```
?= bisim p0 q0.
Yes.
More [y] ? y
No more solutions.
?= #show_table bisim.
Table for bisim contains (P=Proved, D=Disproved):
 [P] bisim p0 q0
 [P] bisim p0 q2
 [P] bisim p1 q1
 [P] bisim p2 q1
?=
```

of the incompleteness that one encounters in traditional logic programming languages, such as Prolog and λProlog, resulting from depth-first search certainly reappear in Bedwyr. We mention two additional sources of incompleteness in the proof search engine of Bedwyr.

13.1 L_λ and non-L_λ unification problems

Bedwyr allows for unrestricted applications of variables to argument but it is only willing to solve L_λ-unification problems. As a result, Bedwyr will occasionally complain that it needs to solve a "not LLambda unification problem" and stop searching for a proof.

To illustrate this aspect of Bedwyr's incompleteness, consider the problem of specifying the instantiation of a first-order quantifier. In particular, consider the specification

```
Kind tm, fm   type.
Type all   (tm -> fm) -> fm.
Type p          tm -> fm.
Type a          tm.
Define instan : fm -> tm -> fm -> prop by
   instan X T (B T) := X = (all B).
```

Thus, `instan` relates a universally quantified formula and a term to the result of instantiating that quantifier with that term. It is the case, however, that a unification problem containing `(B T)` does not belong to the L_λ subset. As a result, the following query results in a runtime error.

```
?= instan (all x\ p x) a (p X).
At line 1, byte 28: Not LLambda unification encountered: a.
?=
```

In some situations, a specification can be written so that the problematic unification is delayed to a point where the unification problem is within the L_λ restriction. In this particular case, if the definition of `instan` is rewritten with the logically equivalent clause

```
Define instan : fm -> tm -> fm -> prop by
   instan X T Y := X = (all B) /\ Y = (B T).
```

this same query now returns an appropriate solution.

```
?= instan (all x\ p x) a (p X).
Solution found:
 X = a
More [y] ?
No more solutions.
?=
```

An improvement to Bedwyr would be for it to automatically delay unification problems that are outside the L_λ-subset: delaying "difficult" unification problems in the hope that future instantiations and β-reduction will make them "simple" is employed in such systems as Twelf and the second version of the Teyjus implementation of λProlog [GHN+08].

13.2 Restriction on the occurrences of logic variables

As we have already noted, in the current implementation of Bedwyr there are restrictions on negative occurrences of logic variables – i.e. to the left of an implication. This restriction arises from the fact that we do not have a satisfactory and comprehensive understanding of unification in the prover-1 that incorporates such variables. As a result, Bedwyr is incomplete since it generates a run-time error in these cases. Consider the following two queries.

```
?= nabla f, exists X, X = 42 -> false.
At line 1, byte 35: Logic variable encountered on the left: H.
?= nabla f, exists X, f X = 42 -> false.
Yes.
More [y] ?
No more solutions.
?=
```

The first query is certainly meaningful and is provable if there is a term different from 42 (say, 43): in Bedwyr, this query generates a run-time error since it requires dealing with a prover-1 existential variable within prover-0 unification. The second query illustrates that some instances of prover-0 unification can tolerate the occurrences of prover-1 existential variables.

Sometimes, one can change a specification to avoid this runtime error. A simple example is provided by the following two queries.

```
?= exists X, (X = 42 -> false) /\ X = 17.
At line 1, byte 38: Logic variable encountered on the left: H.
?= exists X, X = 17 /\ (X = 42 -> false).
Yes.
More [y] ?
No more solutions.
?=
```

Such reordering of goals is something a future version of Bedwyr might attempt to do automatically.

References

[Bae08a] David Baelde. *A linear approach to the proof-theory of least and greatest fixed points*. PhD thesis, Ecole Polytechnique, December 2008.

[Bae08b] David Baelde. On the expressivity of minimal generic quantification. In A. Abel and C. Urban, editors, *International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP 2008)*, number 228, pages 3–19, 2008.

[Bae12] David Baelde. Least and greatest fixed points in linear logic. 13(1), January 2012. ACM Transactions on Computational Logic.

[Gac10] Andrew Gacek. The Abella interactive theorem prover – version 1.3.5, September 2010. Available from `http://abella-prover.org/`.

[GHN+08] Andrew Gacek, Steven Holte, Gopalan Nadathur, Xiaochu Qi, and Zach Snow. The Teyjus system – version 2, March 2008. Available from `http://teyjus.cs.umn.edu/`.

[Gir92] Jean-Yves Girard. A fixpoint theorem in linear logic. An email posting to the mailing list linear@cs.stanford.edu, February 1992.

[GJ03] Andrew D. Gordon and Alan Jeffrey. Typing correspondence assertions for communication protocols. *Theoretical Computer Science*, 300(1-3):379–409, 2003.

[GMN10] Andrew Gacek, Dale Miller, and Gopalan Nadathur. A two-level logic approach to reasoning about computations. Accepted to the J. of Automated Reasoning, August 2010.

[GMN11] Andrew Gacek, Dale Miller, and Gopalan Nadathur. Nominal abstraction. *Information and Computation*, 209(1):48–73, 2011.

[Mil91] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *J. of Logic and Computation*, 1(4):497–536, 1991.

[Mil99] Robin Milner. *Communicating and Mobile Systems : The π-calculus*. Cambridge University Press, New York, NY, USA, 1999.

[Mil00] Dale Miller. Abstract syntax for variable binders: An overview. In John Lloyd and et. al., editors, *Computational Logic - CL 2000*, number 1861 in LNAI, pages 239–253. Springer, 2000.

[Mil06] Dale Miller. Representing and reasoning with operational semantics. In U. Furbach and N. Shankar, editors, *Proceedings of IJCAR: International Joint Conference on Automated Reasoning*, volume 4130 of *LNAI*, pages 4–20, August 2006.

[MMP03] Raymond McDowell, Dale Miller, and Catuscia Palamidessi. Encoding transition systems in sequent calculus. *Theoretical Computer Science*, 294(3):411–437, 2003.

[MN87] Dale Miller and Gopalan Nadathur. A logic programming approach to manipulating formulas and programs. In Seif Haridi, editor, *IEEE Symposium on Logic Programming*, pages 379–388, San Francisco, September 1987.

[MPW93] Robin Milner, Joachim Parrow, and David Walker. Modal logics for mobile processes. *Theoretical Computer Science*, 114(1):149–171, 1993.

[MT03] Alberto Momigliano and Alwen Tiu. Induction and co-induction in sequent calculus. In Mario Coppo Stefano Berardi and Ferruccio Damiani, editors, *Post-proceedings of TYPES 2003*, number 3085 in LNCS, pages 293 – 308, January 2003.

[MT05] Dale Miller and Alwen Tiu. A proof theory for generic judgments. *ACM Trans. on Computational Logic*, 6(4):749–783, October 2005.

[Nad99] Gopalan Nadathur. A fine-grained notation for lambda terms and its use in intensional operations. *Journal of Functional and Logic Programming*, 1999(2), March 1999.

[Nig08] Vivek Nigam. Using tables to construct non-redundant proofs. In C. Dimitracopoulos In A. Beckmann and B. Loewe, editors, *CiE 2008: Abstracts and extended abstracts of unpublished papers*, pages 354–363, 2008.

[Nip93] Tobias Nipkow. Functional unification of higher-order patterns. In M. Vardi, editor, *Proc. 8th IEEE Symposium on Logic in Computer Science (LICS 1993)*, pages 64–74. IEEE, June 1993.

[NL05] Gopalan Nadathur and Natalie Linnell. Practical higher-order pattern unification with on-the-fly raising. In *ICLP 2005: 21st International Logic Programming Conference*, volume 3668 of *LNCS*, pages 371–386, Sitges, Spain, October 2005. Springer.

[NM88] Gopalan Nadathur and Dale Miller. An Overview of λProlog. In *Fifth International Logic Programming Conference*, pages 810–827, Seattle, August 1988. MIT Press.

[NM99] Gopalan Nadathur and Dustin J. Mitchell. System description: Teyjus—a compiler and abstract machine based implementation of Lambda Prolog. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction*, pages 287–291, Trento, Italy, July 1999. Springer-Verlag LNCS.

[PE88] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*, pages 199–208. ACM Press, June 1988.

[Pie05] Brigitte Pientka. Tabling for higher-order logic programming. In *20th International Conference on Automated Deduction, Talinn, Estonia*, pages 54 – 69. Springer-Verlag, 2005.

[RRR+97] Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, Scott A. Smolka, Terrance Swift, and David Scott Warren. Efficient model checking using tabled resolution. In *Proceedings of the 9th International Conference on Computer Aided Verification (CAV97)*, number 1254 in LNCS, pages 143–154, 1997.

[SH93] Peter Schroeder-Heister. Rules of definitional reflection. In M. Vardi, editor, *Eighth Annual Symposium on Logic in Computer Science*, pages 222–232. IEEE Computer Society Press, IEEE, June 1993.

[Tiu04] Alwen Tiu. *A Logical Framework for Reasoning about Logical Specifications*. PhD thesis, Pennsylvania State University, May 2004.

[Tiu05] Alwen Tiu. Model checking for π-calculus using proof search. In Martín Abadi and Luca de Alfaro, editors, *CONCUR*, volume 3653 of *LNCS*, pages 36–50. Springer, 2005.

[Tiu06] Alwen Tiu. A logic for reasoning about generic judgments. In A. Momigliano and B. Pientka, editors, *International Workshop on Logical Frameworks and Meta-Languages:Theory and Practice (LFMTP'06)*, 2006.

[TM04] Alwen Tiu and Dale Miller. A proof search specification of the π-calculus. In *3rd Workshop on the Foundations of Global Ubiquitous Computing*, volume 138 of *ENTCS*, pages 79–101, September 2004.

[TM10] Alwen Tiu and Dale Miller. Proof search specifications of bisimulation and modal logics for the π-calculus. *ACM Trans. on Computational Logic*, 11(2), 2010.

[TNM05] Alwen Tiu, Gopalan Nadathur, and Dale Miller. Mixing finite success and finite failure in an automated prover. In *Proceedings of ESHOL'05: Empirically Successful Automated Reasoning in Higher-Order Logics*, pages 79 – 98, December 2005.

Any paper listed above that was written by one of the authors of this guide can be found on their respective home pages.