
The Bedwyr system for model-checking over syntactic expressions

David Baelde¹ Andrew Gacek²
Dale Miller¹ Gopalan Nadathur² Alwen Tiu³

¹ INRIA & LIX, École Polytechnique

² Digital Technology Center and Dept of CS, University of Minnesota

³ Australian National University and NICTA

Bedwyr is a proof-search engine featuring *a symmetric treatment of success and failure* thanks to definitions (a.k.a. fixed points), a "recent" proof theoretic concept reflecting a *closed world assumption*.

It allows *executing and reasoning about* purely logical specifications of :

- various graph properties, *e.g.*, cyclicity, (un)reachability ;
- winning strategies in games ;
- more generally, model checking.

Higher-order abstract syntax is treated and the ∇ quantifier allows reasoning about bindings. Hence, Bedwyr also handles specifications such as :

- π -calculus : transitions, (bi)simulations ;
- λ -calculus : typing and evaluation ;
- provability in object logics.

LINC extends intuitionistic logic with *definitions* [Gir92,SH93,MT03].
The logic is parametrized by a set of definitions :

$$\text{nat } z \stackrel{\Delta}{=} \top \quad \text{nat } (s X) \stackrel{\Delta}{=} \text{nat } X$$

Definitions provide both backchaining

$$\frac{\Sigma; \Gamma \vdash B\theta}{\Sigma; \Gamma \vdash A} \quad A = A'\theta \text{ for some } A' \stackrel{\Delta}{=} B$$

and case-analysis

$$\frac{\{\Sigma\theta; \Gamma\theta, B\theta \vdash G\theta \mid A' \stackrel{\Delta}{=} B \text{ and } \theta \in \text{csu}(A, A')\}}{\Sigma; \Gamma, A \vdash G}$$

Notice that the substitution θ ranges over *eigenvariables*. In Bedwyr, unifications are restricted to a fragment where *csu* becomes *mgu*.

A logical specification of winning at the tic-tac-toe game, that can be used in LINC to compute winning strategies and reason about them :

```
% Assume we have defined the following:  
% (move B P B')    if the board B' is the same as B  
%                  except for one move by player P  
% (winner P B)     if player P has won on B  
% (nowinner B)     if no player has won on B
```

```
xwins B := winner x B.
```

```
xwins B :=
```

```
  pi B' \ move B o B' =>
```

```
    (nowinner B',
```

```
      sigma M \ move B' x B'', xwins B'').
```

Proving `xwins` (*i.e.*, finding a winning strategy) involves enumerating opponent moves (case analysis) and finding corresponding player moves (backchaining).

Bedwyr searches for proofs in a fragment of LINC. Given its power, one may still call it a (pure) logic programming language.

$$\begin{aligned} L0 & ::= L0 \wedge L0 \mid L0 \vee L0 \mid s = t \mid p\vec{t} \\ & \mid \nabla x.L0x \mid \exists x.L0x \\ L1 & ::= L1 \wedge L1 \mid L1 \vee L1 \mid s = t \mid p\vec{t} \\ & \mid \nabla x.L1x \mid \exists x.L1x \\ & \mid \forall x.L1x \mid L0 \supset L1 \end{aligned}$$

Implicitly : syntactic conditions on the bodies of the defined atoms p .

On the left of the implication there are only *invertible* connectives. The strategy is to introduce them eagerly.

How to find θ (ranging over logic variables) such that $\vdash (A \supset B)\theta$?

1. Collect all σ_i (ranging over eigenvariables) such that $\vdash A\sigma_i$.
2. Find θ such that for all i , $\vdash B\sigma_i\theta$.

In particular a finite failure on a level-0 formula A yields success on $A \supset \perp$.

Bedwyr's engine is actually a standard depth-first proof-search procedure, except that :

- it only accepts \forall and \supset on the right ;
- it unifies logic variables on the right, but eigenvariables on the left ;
- logic variables are not allowed on the left.

λ Prolog does not support case-analysis :

```
p (f a) :- true.
```

```
p (f b) :- true.
```

```
?- pi x\ p x => sigma y\ x = f y.
```

► Fails in λ -Prolog, succeeds in Bedwyr.

On the other hand, Bedwyr *always* does a full case-analysis :

```
nat z      := true.
```

```
nat (s X) := nat X.
```

```
?= pi x\ nat x => nat x.
```

► Succeeds in λ -Prolog but loops in Bedwyr.

We are currently experimenting with tabling, in order to avoid redundant and cyclic computations.

When you explicitly declare a definition to be `inductive` or `coinductive`, Bedwyr will remember the proved/disproved instances of the defined atom, as well as those that proof-search encountered (for loop detection).

$$\frac{\frac{?}{\vdash d \bar{x}}}{\vdash d \bar{x}}$$

Loops on inductive definitions are failures, but they yield success for coinductive ones.

Tabling can be justified by the (co)induction rules of LINC.

The proof-theory of definitions extends naturally to higher-order terms ; in the implementation, first-order unification becomes higher-order pattern unification.

One can represent object-level binders by formula-level binders :

$$??? \text{ seq } G \text{ (forall } x \setminus P \ x) := \text{ pi } x \setminus \text{ seq } G \text{ (P } x) ???$$

What can we say about the terms t , u and w if the following is provable :

$$x, y; p \ x \ t, p \ y \ u \vdash p \ x \ w$$

Since x and y are distinct eigenvariables, t and w must be equal.

This cannot be modeled by the universal quantification. Enters ∇ , quantifying over *generic variables* [MT05]. A few theorems about it :

$$\nabla x.(Bx \wedge Cx) \equiv \nabla x.Bx \wedge \nabla x.Cx$$

$$\nabla x_\alpha \forall y_\beta. Bxy \equiv \forall h_{\alpha \rightarrow \beta} \nabla x. Bx(hx)$$

$$\nabla x.(t = s) \equiv (\lambda x.t) = (\lambda x.s)$$

We can now faithfully define provability for a fragment of LJ in Bedwyr :

$\text{seq } L \text{ (forall } x \setminus B \ x) := \text{nabla } x \setminus \text{seq } L \text{ (} B \ x \text{).}$

$\text{seq } L \text{ (} D \text{ } \rightarrow \text{ } G \text{)} := \text{seq (and } D \ L \text{) } G \text{.}$

$\text{seq } L \ A := \text{atom } A, \text{ bc } L \ L \ A \text{.}$

...

Our proof-search strategy is enough to prove that :

$\text{pi } t \setminus \text{pi } u \setminus \text{pi } w \setminus$

$\text{seq } tt \text{ (forall } x \setminus \text{forall } y \setminus$

$\text{(} p \ x \ t \text{) } \rightarrow \text{(} p \ y \ u \text{) } \rightarrow \text{(} p \ x \ w \text{))}$

$\Rightarrow w = t$

Unfold the definition of `seq` on the left, two cases eventually remain :

$$\begin{array}{l|l} \nabla x \nabla y. \text{bc } \Gamma \ pxt \ pxw \vdash w = t & \nabla x \nabla y. \text{bc } \Gamma \ pyu \ pxw \vdash w = t \\ \nabla x \nabla y. (pxt = pxw) \vdash w = t & \nabla x \nabla y. (pyu = pxw) \vdash w = t \\ \lambda x. \lambda y. pxt \doteq \lambda x. \lambda y. pxw & \lambda x. \lambda y. pyu \dot{\neq} \lambda x. \lambda y. pxw \end{array}$$

Conclusion :

- smooth integration of recent proof-theory developments ;
- finite failure in a purely logical way ;
- not as efficient as standard model checkers (*e.g.*, XSB),
but purely declarative and reasonably fast thanks to tabling ;
- reasoning about specifications involving variable bindings.

We are currently investigating further the proof-theory of least and greatest fixed points. In a submitted paper by Miller and Baelde, the proof-search strategy used in Bedwyr appears as a particular case of more general and elegant observations.

Find more examples in the paper, and in the distribution :

<http://slimmer.gforge.inria.fr/bedwyr>

We are open to contributions and have OCaml code to share.