Bedwyr,
a proof-search approach to model-checking

David Baelde, INRIA / École Polytechnique

Bedwyr was developed with Gacek, Miller, Nadathur & Tiu.
What/who is Bedwyr?

Bedwyr is a proof-search engine featuring:
– finite failure thanks to definitions;
– λ-tree approach to HOAS;
– reasoning about generic variables thanks to ∇;
– and tabling.

It allows to execute and reason about logic specifications of:
– λ-calculus: typing and evaluation;
– π-calculus: transitions, (bi)simulations;
– provability in object logics, e.g. HH;
– model-checking on graphs, winning strategies in games, etc.

(And Bedwyr is a knight of the round table, known as a not-so-sound logician in The Quest for the Holy Grail…)
The logic is parametrized by a set of definitions:

\[ nat \triangleq \lambda x. \; x = 0 \lor \exists y. \; x = s \; y \land \text{nat} \; y \]

Some unusual rules in FOLDN, nothing really new to implement:

\[
\begin{align*}
\Gamma, (\sigma, x) & \vdash Hx \vdash \sigma' \vdash G & \Gamma \vdash (\sigma, x) \vdash Gx \\
\Gamma, \sigma & \vdash \nabla x.Hx \vdash \sigma' \vdash G & \Gamma \vdash \sigma \vdash \nabla x.Gx \\
\Gamma, \sigma & \vdash B \vec{t} \vdash \sigma' \vdash G & \Gamma \vdash \sigma \vdash B \vec{t} \; p \overset{\triangle} = B \\
\Gamma, \sigma & \vdash p \vec{t} \vdash \sigma' \vdash G & \Gamma \vdash \sigma \vdash p \vec{t} \\
\Sigma, h; \Gamma, \sigma & \vdash F(h\sigma) \vdash \sigma' \vdash G & \Sigma; \Gamma \vdash \sigma \vdash G(t\sigma) \\
\Sigma; \Gamma, \sigma & \vdash \exists x.\; Fx \vdash \sigma' \vdash G & \Sigma; \Gamma \vdash \sigma \vdash \exists x.\; Gx \\
\{ (\Gamma \vdash \sigma' \vdash G) \theta : \theta \in \text{csu}(\lambda \sigma \cdot s \overset{=} = \lambda \sigma \cdot t) \} & \Gamma, \sigma \vdash s = t \vdash \sigma' \vdash G & \Gamma \vdash \sigma \vdash t = t
\end{align*}
\]
Reasoning about provability in HH

Let’s define Hereditary Harrop provability in Bedwyr:

\[
\begin{align*}
\text{seq } L \ (\forall B) & \ := \ \text{nabla } x \ \text{seq } L \ (B \ x). \\
\text{seq } L \ (D \rightarrow G) & \ := \ \text{seq } (\text{and } D \ L) \ G. \\
\text{seq } L \ A & \ := \ \text{atom } A, \ \text{bc } L \ L \ A.
\end{align*}
\]

... 

Not much thinking is needed to prove that

\[
\begin{align*}
\pi t \ \pi u \ \pi w \\
\text{seq } tt \ (\forall x \ \forall y \ (p \ x \ t) \rightarrow (p \ y \ u) \rightarrow (p \ x \ w)) & \implies w = t
\end{align*}
\]

Unfold the definition of \text{seq} on the left, two cases remain:

\[
\begin{align*}
x, y \triangleright \text{bc } \Gamma \ pxt \ pxw & \vdash w = t & x, y \triangleright \text{bc } \Gamma \ pyu \ pxw & \vdash w = t \\
x, y \triangleright pxt = pxw & \vdash w = t & x, y \triangleright pyu = pxw & \vdash w = t \\
\lambda x.\lambda y. pxt = \lambda x.\lambda y. pxw & & \lambda x.\lambda y. pyu = \lambda x.\lambda y. pxw
\end{align*}
\]
Proof-search in Level 0/1

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

\[
\mathcal{L}_0 ::= \mathcal{L}_0 \land \mathcal{L}_0 \mid \mathcal{L}_0 \lor \mathcal{L}_0 \mid s = t \mid p \vec{t} \\
\mid \forall x.\mathcal{L}_0x \mid \exists x.\mathcal{L}_0x
\]

\[
\mathcal{L}_1 ::= \mathcal{L}_1 \land \mathcal{L}_1 \mid \mathcal{L}_1 \lor \mathcal{L}_1 \mid s = t \mid p \vec{t} \\
\mid \forall x.\mathcal{L}_1x \mid \exists x.\mathcal{L}_1x
\]

Implicitely : 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.
The treatment of implication

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

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

In particular a finite failure on a level-0 formula \( F \) yields success on \( F \supset \bot \).

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

- it carries the extra generic context;
- it only accepts \( \forall \) and \( \supset \) in right-mode;
- it unifies logic variables on the right, eigenvariables on the left.
Comparison with $\lambda$Prolog

$\lambda$Prolog does not support case-analysis or negation-as-failure:

\[
p(f\ a).
\]
\[
p(f\ b).
\]
\[
?-\ \text{forall}\ x\ \ p\ x\ \rightarrow\ \text{exists}\ y\ \ x = f\ y.
\]

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

\[
nat\ z.
\]
\[
nat\ (s\ X) := nat\ X.
\]
\[
?=\ \pi\ x\ \ nat\ x\ \rightarrow\ nat\ x.
\]
Details: not so simple...  

Bedwyr suffers from the usual incompletenesses of depth-first engines, but also from more specific problems.

– How to handle logic variables on the left?

\[
X=1 \Rightarrow X=1
\]

We would need to mix disunification and unification, there would easily be an infinity of solutions... so we just give up.

– We restrict ourselves to higher-order patterns, and give up on more complicated problems.

We use Nadathur and Linnell’s implementation, which makes use of a level annotation to represent raising efficiently. We extended it with \( \nabla \) indices, local level annotations and corresponding constraints, which allows to avoid errors on goals like

\[
\text{nabla } y \backslash \sigma a \backslash \pi x \backslash a \ x = a \ x
\]

– Finally, we must check that the instantiations of eigenvariables on the left hand-side do not make right hand-side problems fall outside of higher-order patterns...
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 definition but also the encountered ones for loop detection.

\[
\frac{?}{\vdash d \bar{x}}
\]

\[
\vdash d \bar{x}
\]

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

Tabling is the only use of the (co)induction rules of LINC.
Miller and Tiu’s formalization of open bisimulation for $\pi$-calculus in LINC fits in Level 0/1. The one-step transition specification is within Level 0, and bisimulation roughly goes as follows:

$$\text{coinductive bisim } P \ Q :=$$

$$\quad (\pi A \ \pi P1 \ step \ P \ A \ P1 \Rightarrow$$

$$\quad \quad \sigma Q1 \ step \ Q \ A \ Q1, \ bisim \ P1 \ Q1),$$

$$\quad (\pi A \ \pi Q1 \ step \ Q \ A \ Q1 \Rightarrow$$

$$\quad \quad \sigma P1 \ step \ P \ A \ P1, \ bisim \ P1 \ Q1).$$

It means that writing it down in Bedwyr will give an *executable specification* of it, that is a bisimulation checker. All that without knowing any implementation detail, the ability to modify the spec easily, etc.
\[ \pi \text{-calculus} \]

% bound input
onep (in X M) (dn X) M.
% free output
one (out X Y P) (up X Y) P.

% comm
one (par P Q) tau (par (M Y) T) :=
    onep P (dn X) M & onep Q (up X Y) T.

% open
onep (nu x\ M x) (up X) N :=
    nabla y\ one (M y) (up X y) (N y).
% close
one (par P Q) tau (nu y\ par (M y) (N y)) :=
    sigma X\ onep P (dn X) M & onep Q (up X) N.
The real specification of simulation is as follows:

\[
\text{coinductive sim } P \sim Q := \text{pi } A \setminus \text{pi } P_1 \setminus \text{pi } M \setminus
\]
\[
\text{(one } P A P_1 \Rightarrow \text{one } Q A Q_1 \& \text{sim } P_1 Q_1),
\]
\[
\text{(onep } P \text{ (dn } X) M \Rightarrow \text{onep } Q \text{ (dn } X) N,
\]
\[
\text{pi } w \setminus \text{sim } (M w) (N w),
\]
\[
\text{(onep } P \text{ (up } X) M \Rightarrow \text{onep } Q \text{ (up } X) N,
\]
\[
\text{nabla } w \setminus \text{sim } (M w) (N w)).
\]

Bisimulation is twice as large but similar.

Again, binding and freshness issues are completely expressed by the three binders of LINC, as shown in simples examples:

\[
a(x).a(y).0 \sim a(x).\nu z.a(y).0
\]
\[
a(x).\nu y.[x = y].P \sim a(x).0
\]
\[
\nu x.\bar{a}(x).c(y).[x = y].P \sim \nu x.\bar{a}(x).c(y).0
\]
More complex examples involving weak bisimulation and encodings of natural numbers benefit a lot from tabling... but we still can’t compete with dedicated tools such as MWB.

\[ 5 + 5 = 10 \]

```lisp
#assert
(weak_bisim
  (church s z
    (ss (ss (ss (ss (ss (ss (ss (ss zz))))))))))))
  (nu s1\ nu z1\ nu s2\ nu z2\n    (par (church s1 z1 (ss (ss (ss (ss zz))))))
    (par (church s2 z2 (ss (ss (ss (ss zz))))))
    (add s1 z1 s2 z2 s z))).
```
Conclusion

Regarding Bedwyr:
- ongoing work on tabling: make it sound, extend it;
- suspend non-Lλ unifications;
- try to generalize and re-use the term unification library;

Beyond Bedwyr:
- work on the restrictions of LINC’s (co)induction, or move to LG;
- design tools with real support for (co)induction, but still using focused proof-search disciplines.

Thank you!