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;
– $\lambda$-tree approach to HOAS;
– reasoning about generic variables thanks to $\nabla$;
– and tabling.

It allows to execute and reason about logic specifications of:
– $\lambda$-calculus: typing and evaluation;
– $\pi$-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:

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

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

\[
\begin{align*}
\Gamma, (\sigma, x) & \triangleright Hx \triangleright \sigma' \triangleright G \\
\Gamma, \sigma & \triangleright \nabla x. Hx \triangleright \sigma' \triangleright G \\
\Gamma, \sigma & \triangleright Bt \triangleright \sigma' \triangleright G \\
\Gamma & \triangleright \sigma \triangleright pt \triangleright p \triangledown = B \\
\Sigma, h; \Gamma, \sigma & \triangleright F(h\sigma) \triangleright \sigma' \triangleright G \\
\Sigma; \Gamma, \sigma & \triangleright \exists x. Fx \triangleright \sigma' \triangleright G \\
\Sigma; \Gamma & \triangleright \sigma \triangleright G(t\sigma) \\
\Sigma; \Gamma & \triangleright \sigma \triangleright \exists x. Gx \\
\end{align*}
\]

\[\{(\Gamma \triangleright \sigma' \triangleright G)\theta : \theta \in csu(\lambda \sigma. s \equiv \lambda \sigma. t)\}\]

\[
\begin{align*}
\Gamma, \sigma & \triangleright s = t \triangleright \sigma' \triangleright G \\
\Gamma & \triangleright \sigma \triangleright t = t \\
\end{align*}
\]
Reasoning about provability in HH

Let’s define Hereditary Harrop provability in Bedwyr:

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

Not much thinking is needed to prove that

\[
\pi \ t \ \pi \ u \ \pi \ w \ \\
\text{seq } tt \ (\text{forall } x \ \text{forall } y \ (p \ x \ t) \rightarrow (p \ y \ u) \rightarrow (p \ x \ w)) \Rightarrow w = t
\]

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

\[
x, y \triangleright bc \ \Gamma \ pxt \ pxw \vdash w = t \quad \quad x, y \triangleright bc \ \Gamma \ puu \ pxw \vdash w = t
\]
\[
x, y \triangleright pxt = pxw \vdash w = t \quad \quad x, y \triangleright puu = pxw \vdash w = t
\]
\[
\lambda x.\lambda y.\ pxt = \lambda x.\lambda y.\ pxw \quad \quad \lambda x.\lambda y.\ puu = \lambda x.\lambda y.\ pxw
\]
Bedwyr searches for proofs in a fragment of FOLDN. Given its power, one may still call it a (pure) logic programming language.

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

\[
\mathcal{L}_1 \ ::= \ \mathcal{L}_1 \land \mathcal{L}_1 \mid \mathcal{L}_1 \lor \mathcal{L}_1 \mid s = t \mid p \vec{t} \\
| \ \nabla x. \mathcal{L}_1 x \mid \exists x. \mathcal{L}_1 x \\
| \ \forall x. \mathcal{L}_1 x \mid \mathcal{L}_0 \supset \mathcal{L}_1
\]

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.
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.
\textbf{Comparison with $\lambda$Prolog}

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

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

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

\[
\text{nat } z.
nat (s\ X) := \text{nat } X.
?= \pi x\ \text{nat } x \Rightarrow \text{nat } x.
\]
Bedwyr suffers from the usual incompletenesses of depth-first engines, but also from more specific problems.

– How to handle logic variables on the left?

\[ X = 1 \implies X = 1 \]

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

\[ \nabla y \sigma a \pi x \ a \ x = a \ x \]

– 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.

\[
\begin{array}{c}
? \\
\vdash d \bar{x} \\
\vdash d \bar{x} \\
\vdash d \bar{x}
\end{array}
\]

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:

\[
\text{coinductive bisim } P \ Q := \\
\quad (\pi \ A \ \pi \ P1 \ \text{step } P \ A \ P1 \Rightarrow \\
\qquad \sigma \ Q1 \ \text{step } Q \ A \ Q1, \ \text{bisim } P1 \ Q1), \\
\quad (\pi \ A \ \pi \ Q1 \ \text{step } Q \ A \ Q1 \Rightarrow \\
\qquad \sigma \ P1 \ \text{step } P \ A \ P1, \ \text{bisim } P1 \ Q1).
\]

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.
\[ \begin{align*} 
\% \text{ bound input} 
on & \text{onep (in } X \ M) \ (dn \ X) \ M. 
\% \text{ free output} 
& \text{one (out } X \ Y \ P) \ (up \ X \ Y) \ P. 
\% \text{ comm} 
on & \text{one (par } P \ Q) \ \tau \ (par \ (M \ Y) \ T) := 
& \text{onep } P \ (dn \ X) \ M \ & \text{onep } Q \ (up \ X \ Y) \ T. 
\% \text{ open} 
on & \text{onep (nu } x\backslash M \ x) \ (up \ X) \ N := 
& \text{nabla } y\backslash \text{one (M } y) \ (up \ X \ y) \ (N \ y). 
\% \text{ close} 
on & \text{one (par } P \ Q) \ \tau \ (nu \ y\backslash \text{par (M } y) \ (N \ y)) := 
& \text{sigma } X\backslash \text{onep } P \ (dn \ X) \ M \ & \text{onep } Q \ (up \ X) \ N. 
\end{align*} \]
The real specification of simulation is as follows:

\[
\text{coinductive } \text{sim } P \text{ Q} := \pi A \ \pi P1 \ \pi M \\[
\text{(one } P \text{ A } P1 \Rightarrow \text{one } Q \text{ A } Q1 \ & \ \text{sim } P1 \ Q1),
\text{(onep } P \text{ (dn } X) \ M \Rightarrow \text{onep } Q \text{ (dn } X) \ N,
\pi w \ \text{sim } (M \ w) \ (N \ w)),
\text{(onep } P \text{ (up } X) \ M \Rightarrow \text{onep } Q \text{ (up } X) \ N,
nabla w \ \text{sim } (M \ w) \ (N \ w)).
\]
\]

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:

\[
a(x).a(y).0 \sim a(x).\nu z.a(y).0
\]
\[
a(x).\nu y.[x = y].P \sim a(x).0
\]
\[
\nu x.\overline{a}(x).c(y).[x = y].P \sim \nu x.\overline{a}(x).c(y).0
\]
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

#assert

(weak_bisim
  (church s z
   (ss (ss (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-L\(\lambda\) 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!