The Bedwyr system for model checking over syntactic expressions

David Baelde[1], Andrew Gacek[2], Dale Miller[1], Gopalan Nadathur[2], and Alwen Tiu[3]

[1] INRIA & LIX, École Polytechnique
[2] Digital Technology Center and Dept of CS, University of Minnesota
[3] Australian National University and National ICT Australia

1 Overview

Bedwyr is a generalization of logic programming that allows model checking directly on syntactic expressions possibly containing bindings. This system, written in OCaml, is a direct implementation of two recent advances in the theory of proof search. The first is centered on the fact that both finite success and finite failure can be captured in the sequent calculus by incorporating inference rules for *definitions* that allow *fixed points* to be explored. As a result, proof search in such a sequent calculus can capture simple model checking problems as well as may and must behavior in operational semantics. The second is that higher-order abstract syntax is directly supported using term-level λ-binders and the quantifier known as ∇. These features allow reasoning directly on expressions containing bound variables.

2 Foundations

The logical foundation of Bedwyr is the logic called LINC [9], an acronym for "lambda, induction, nabla, and co-induction" that also describes its major components. LINC extends intuitionistic logic in two directions.

Fixed points via definitions. Clauses such as $A \stackrel{\triangle}{=} B$ are used to provide (mutual) recursive definitions of how instances of the atom A can be unfolded to the same instance of the body B. After fixing a set $\mathcal{D}$ of definition clauses, LINC contains the following inference rules for introducing defined atomic formulas. Unfolding atoms on the right of the sequent arrow is specified by the following *definition-right* rule.

$$\frac{\Sigma : \Gamma \vdash B\theta}{\Sigma : \Gamma \vdash A}\text{, provided } A' \stackrel{\triangle}{=} B \in \mathcal{D} \text{ and } A'\theta = A$$

This rule resembles backchaining in more conventional logic programming languages. The *definition-left* rule is a case analysis justified by a closed-world reading of a definition.

$$\frac{\{\Sigma\theta : \Gamma\theta, B\theta \vdash G\theta \mid A' \stackrel{\triangle}{=} B \in \mathcal{D} \text{ and } \theta \in csu(A, A')\}}{\Sigma : \Gamma, A \vdash G}$$

Notice that this rule makes uses unification: the eigenvariables of the sequent (stored in the signature Σ) are instantiated by θ, which is a member of a complete set of unifiers (csu) for atoms A and A'. Bedwyr implements a subset of this rule that is restricted to *higher-order pattern unification* and, hence, to a case where *csu* can be replaced by *mgu*. If an atom on the left fails to unify with the head of any definition, the premise set of this inference rule is empty and, hence, the sequent is proved: thus, a unification *failure* is turned into a proof search *success*.

Notice that this use of definitions as fixed points implies that logic specifications are not treated as part of a *theory* from which conclusions are drawn. Instead, the proof system itself is parametrized by the logic specification. In this way, definitions remain fixed during proof search and the *closed world assumption* can be applied to the logic specification. For earlier references to this approach to fixed points see [2, 8, 3].

Nabla quantification Bedwyr supports the λ-*tree syntax* [4] approach to higher-order abstract syntax [7] by implementing a logic that provides (i) terms that may contain λ-bindings, (ii) variables that can range over such terms, and (iii) equality (and unification) that follows the rules of λ-conversion. Bedwyr shares these attributes with systems such as λProlog: however, the closed-world aspects of LINC could not be fully exploited without the addition of the ∇-quantifier. This quantifier can be read informally as "for any fresh variable", and is accommodated easily within the sequent calculus with the introduction of a new kind of local context scoped over formulas. We refer the reader to [5] for more details. We point out here, however, that ∇ can always be given minimal scope by using the equivalences $\nabla x.(Ax \supset Bx) \equiv (\nabla x.Ax) \supset (\nabla x.Bx)$ (also where $\wedge$ and $\vee$ replace $\supset$) and the fact that ∇ is self-dual: $\nabla x.\neg Bx \equiv \neg\nabla x.Bx$. When ∇ is moved under $\forall$ and $\exists$, it *raises* the type of the quantified variable: in particular, in the equivalences $\nabla x\forall y.Fxy \equiv \forall h\nabla x.Fx(hx)$ and $\nabla x\exists y.Fxy \equiv \exists h\nabla x.Fx(hx)$, the variable y is replaced with a functional variable h. Finally, when ∇ is scoped over equations, the equivalence $\nabla x(Tx = Sx) \equiv (\lambda x.Tx) = (\lambda x.Sx)$ allows it to be completely removed. As a result, no fundamentally new ideas are needed to implement ∇ in a framework where λ-term equality is supported.

3 Architecture

Bedwyr implements a fragment of LINC which still allows interesting applications of fixed points and ∇. In this fragment, *all* the left rules are invertible. Consequently, we can use a simple proof strategy that alternates between left and right-rules, with the left-rules taking precedence over the right rules.

Two provers. The fragment of LINC implemented in Bedwyr is given by the following grammar:

$$L0 ::= \top | A | L0 \wedge L0 | L0 \vee L0 | \nabla x.\ L0 | \exists x.\ L0$$
$$L1 ::= \top | A | L1 \wedge L1 | L1 \vee L1 | \nabla x.\ L1 | \exists x.\ L1 | \forall x.\ L1 | L0 \supset L1$$

The formulas in this fragment are divided into *level-0* formulas, given by $L0$ above, and *level-1* formulas, given by $L1$. Implicit in the above grammar is the partition of atoms into level-0 atoms and level-1 atoms. Restrictions apply to goal formulas and definitions: goal formulas can be level-0 or level-1 formulas, and in a definition $A \stackrel{\triangle}{=} B$, A and B can be level-0 or level-1 formulas, provided that the level of A is higher or equal to the level of B.

Level-0 formulas are essentially a subset of goal formulas in λProlog (with ∇ replacing $\forall$). Proof search for a defined atom of level-0 is thus the same as in λProlog. We can think of a level-0 definition, say, $p\,x \stackrel{\triangle}{=} B\,x$, as defining a set of elements x satisfying $B\,x$. A successful proof search for $p\,t$ means that t is in the set characterized by B. A level-1 statement like $\forall x.p\,x \supset R\,x$ would then mean that R holds for all elements of the set characterized by p. That is, this statement captures the enumeration of a *model* of p and its verification can be seen as a form of model checking. To reflect this operational reading of level-1 implications, the proof search engine of Bedwyr uses two subprovers: the Level-0 prover (a simplified λProlog engine), and the Level-1 prover. The latter is a usual depth-first goal-directed prover but with a novel treatment of implication. When the Level-1 prover reaches the implication $A \supset B$, it calls the Level-0 prover on A and gets in return a stream of answer substitutions: the Level-1 prover then checks that, for every substitution θ in that stream, $B\theta$ holds. In particular if Level-0 finitely fails with A, the implication is proved.

As with most depth-first implementations of proof search, Bedwyr suffers from some aspects of incompleteness: for example, the prover can easily loop during a search although different choices of goal or clause ordering can lead to a proof, and certain kinds of unification problems should be delayed instead of attempted eagerly. For a more detailed account on the incompleteness issues, we refer the reader to [11]. Bedwyr does not currently implement static checking of types and the stratification of definitions (which is required in the cut-elimination proof for LINC). This allows us to experiment with a wider range of examples than those allowed by LINC.

Higher-order pattern unification. We adapt the treatment of higher-order pattern unification due to Nadathur and Linnell [6]. This implementation uses the *suspension calculus* representation of λ-terms. We avoid explicit raising, which is expensive, by representing ∇-bound variables by indices and associating a *global* and a *local* level annotations with other quantified variables. The global level ignores the ∇-quantifiers and counts alternations in the universal and existential quantifier prefix. The local level counts the number of ∇ quantifiers under which the variable occurs. Using this annotation scheme, the scoping aspects of ∇ quantifiers are reflected into new conditions on local levels but the overall structure of the higher-order pattern unification problem and its mgu properties are preserved.

Tabling. We introduced tabling in Bedwyr to cut-down exponential blowups caused by redundant computations and to detect loops during proof-search. The first optimization is critical for applications such as weak bisimulation checking. The second one proves useful when exploring reachability in a cyclic graph.

Tabling is currently used in Bedwyr to experiment with proof search for inductive and co-inductive predicates. A loop over an inductive predicate causes a divergence that can be recognized instead as failure. Conversely, in the co-inductive case, loops yield success. This interpretation of loops as failure or success is not part of the meta-theory of LINC. Its soundness is currently conjectured, although we do not see any inconsistency of this interpretation on the numerous examples that we tried.

Inductive proof-search with tabling is implemented effectively in provers like XSB using, for example, suspensions. The implementation of tables in Bedwyr fits simply in the initial design of the prover but is much weaker. We only table a goal in Level-1 when it does not have any logic variables and in Level-0 when it does not have any free variables. Nevertheless, this implementation of tabling has proved useful in several cases, ranging from graph examples to bisimulation.

4 Examples

We give here a brief description of the range of applications of Bedwyr. We refer the reader to `http://slimmer.gforge.inria.fr/bedwyr` and Bedwyr's user manual [1] for more details about a range of examples.

Model-checking. If the two predicates P and Q are defined using Horn clauses, then the Level-1 prover is capable of attempting a proof of $\forall x.\ P\ x \supset Q\ x$. This covers most (un)reachability checks common in model-checking. Related examples in the Bedwyr distribution include the verification of a 3 bits addition circuit and graph cyclicity checks.

Games, strategies, and bisimulations. Checking bisimulations and finding winning strategies for games is also achieved through specifications of the form:

$$win\ B \stackrel{\triangle}{=} \forall B'.\ step\ B\ B' \supset \exists B''.\ step\ B'\ B'' \wedge win\ B''$$

If this property is tabled during proof search, the resulting table provides meaningful information, such as the winning strategy or the bisimulation relation.

Meta-level reasoning. Because Bedwyr incorporates λ-tree syntax approach and the ∇ quantifier, it's possible to specify provability in an object logic and to reason to some extent about what is and is not provable. The example `object.def` (see the distribution) defines the predicate pv for object-level provability from a Horn clause program. If we assume that this object-level program states that the ternary predicate q holds whenever two of its parameters are equal, then Bedwyr can prove easily the following meta-theorem, where $\overline{\forall}$ denotes the object-level universal quantification:

$$\forall x \forall y \forall z.\ pv\ (\overline{\forall} u \overline{\forall} v.\ q\ \langle u, x\rangle\ \langle v, y\rangle\ \langle v, z\rangle) \supset y = z$$

Reasoning over syntactic expressions with bindings Along the same lines, specifying and reasoning over one-step transitions for the π-calculus [10] or simple typability for the λ-calculus is supported in a fully declarative way.

5 Future Work

We are working on several improvements to Bedwyr, among others, a more sophisticated tabling, e.g., by allowing suspended goals as in XSB, and allowing non-higher-order-pattern goals, by suspending them until they are instantiated to higher-order pattern goals. We will also explore the use of tabling to export fixed points since these can often act as certificates: for example, in the process of determining if two processes are bisimilar, the table stores an actual bisimulation. Bedwyr is an open source project and we welcome contributions to the project. More details about Bedwyr can be found at `http://slimmer.gforge.inria.fr/bedwyr/`.

Acknowledgments. Support has been obtained for this work from the following sources: from INRIA through the "Equipes Associées" Slimmer, from the ACI grant GEOCAL and from the NSF Grants OISE-0553462 (IRES-REUSSI) and CCR-0429572 that also includes support for Slimmer.

References

1. David Baelde, Andrew Gacek, Dale Miller, Gopalan Nadathur, and Alwen Tiu. *A User Guide to Bedwyr*, November 2006.
2. Jean-Yves Girard. A fixpoint theorem in linear logic. An email posting to the mailing list linear@cs.stanford.edu, February 1992.
3. Raymond McDowell and Dale Miller. A logic for reasoning with higher-order abstract syntax. In Glynn Winskel, editor, *12th Symp. on Logic in Computer Science*, pages 434–445, Warsaw, Poland, July 1997. IEEE Computer Society Press.
4. 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.
5. Dale Miller and Alwen Tiu. A proof theory for generic judgments. *ACM Trans. on Computational Logic*, 6(4):749–783, October 2005.
6. 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, October 2005. Springer.
7. 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.
8. Peter Schroeder-Heister. Definitional reflection and the completion. In R. Dyckhoff, editor, *Proceedings of the 4th International Workshop on Extensions of Logic Programming*, pages 333–347. Springer-Verlag LNAI 798, 1993.
9. Alwen Tiu. *A Logical Framework for Reasoning about Logical Specifications*. PhD thesis, Pennsylvania State University, May 2004.
10. 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.
11. 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.