



The Imandra Automated Reasoning System (System Description)


Grant Passmore^(✉), Simon Cruanes, Denis Ignatovich, Dave Aitken,
Matt Bray, Elijah Kagan, Kostya Kanishev, Ewen Maclean,
and Nicola Mometto

Imandra Inc., Austin, USA
grant@imandra.ai

Abstract. We describe Imandra, a modern computational logic theorem prover designed to bridge the gap between decision procedures such as SMT, semi-automatic inductive provers of the Boyer-Moore family like ACL2, and interactive proof assistants for typed higher-order logics. Imandra’s logic is computational, based on a pure subset of OCaml in which all functions are terminating, with restrictions on types and higher-order functions that allow conjectures to be translated into multi-sorted first-order logic with theories, including arithmetic and datatypes. Imandra has novel features supporting large-scale industrial applications, including a seamless integration of bounded and unbounded verification, first-class computable counterexamples, efficiently executable models and a cloud-native architecture supporting live multiuser collaboration. The core reasoning mechanisms of Imandra are (i) a semi-complete procedure for finding models of formulas in the logic mentioned above, centered around the lazy expansion of recursive functions, (ii) an inductive waterfall and simplifier which “lifts” many Boyer-Moore ideas to our typed higher-order setting. These mechanisms are tightly integrated and subject to many forms of user control.

1 Introduction

Imandra is a modern computational logic theorem prover built around a pure, higher-order subset of OCaml. Mathematical models and conjectures are written as executable OCaml programs, and Imandra may be used to reason about them, combining models, proofs and counterexamples in a unified computational environment. Imandra is designed to bridge the gap between decision procedures such as SMT [2], semi-automatic inductive provers of the Boyer-Moore family like ACL2 [1, 6], and interactive proof assistants for typed higher-order logics [4, 5, 7, 8]. Our goal is to build a friendly, easy to use system by leveraging strong automation in proof search that can also robustly provide counterexamples for false conjectures. Imandra has novel features supporting large-scale industrial applications, including a seamless integration of bounded and unbounded verification, first-class computable counterexamples, efficiently executable models and a cloud-native architecture supporting live multiuser



REASONING AS A SERVICE™

View all notebooks
Imandra terminal

In [4]:

```
let rec rev x =
  match x with
  | [] -> []
  | x::xs -> rev xs @ [x]
```

Out[4]: val rev : 'a list -> 'a list = <fun>

> termination proof

In [6]:

```
verify (fun x -> rev x <> [5;6;7])
```

Out[6]: - : Z.t list -> bool = <fun>
 module CX : sig val x : int list end
 Counterexample (after 10 steps, 0.026s):
 let (x : int list) = [7; 6; 5]

⊗ Refuted

call graph
proof

Load graph

In [8]:

```
rev CX.x
```

Out[8]: - : int list = [5; 6; 7]

In [18]:

```
verify -upto:25 (fun x -> rev (rev x) = x)
```

Out[18]: - : 'a list -> bool = <fun>

✔ Proved up to 25 steps

Load graph

In [7]:

```
verify (fun x -> rev (rev x) = x) [@@auto]
```

Subgoal 1.1'''.1':

```
H0. gen_1 <> []
H1. rev (List.append (List.tl gen_1) [x1]) = x1 :: (rev (List.tl gen_1))
-----
List.append (rev (List.append (List.tl gen_1) [x1])) [List.hd gen_1] =
x1 :: (List.append (rev (List.tl gen_1)) [List.hd gen_1])
```

But we verify Subgoal 1.1'''.1' by recursive unrolling.

① Rules:

```
(:def List.append)
(:def rev)
(:induct List.append)
(:induct rev)
```

✔ Proved

In [15]:

```
instance (fun f x -> List.length x = 5 && List.fold_left (fun x y -> f x + y) 0 x = 100)
```

Out[15]: - : (Z.t -> Z.t) -> Z.t list -> bool = <fun>
 module CX : sig val f : int -> Z.t val x : int list end

Instance (after 12 steps, 0.025s):

```
let f = function
| 609 -> 2437
| 11292 -> 1796
| 8945 -> 5853
| 13951 -> 6283
| _ -> 38
let (x : int list) = [571; 8855; 7149; 8098; (-6183)]
```

✔ Instance

call graph
proof

Load graph

In [17]:

```
List.fold_left (fun x y -> CX.f x + y) 0 CX.x
```

Out[17]: - : int = 100

Fig. 1. An example Imandra session illustrating recursive definitions, computable counterexamples (CX), bounded verification (**verify upto**), unbounded verification with automated induction (**@@auto**), and higher-order instance synthesis.

collaboration. Imandra is already in use by major companies in the financial sector, including Goldman Sachs, Itiviti and OneChronos [9].

An online version may be found at <https://try.imandra.ai> (Fig. 1).

2 Logic

Imandra’s logic is built on a mechanized formal semantics for a pure, higher-order subset of OCaml. Foundationally, the subset of OCaml Imandra supports (called the ‘Imandra Modelling Language’) corresponds to a (specializable) computational fragment of HOL equivalent to multi-sorted first-order logic with induction up to ϵ_0 extended with theories of datatypes, integer and real arithmetic. Theorems are implicitly universally quantified and expressed as Boolean-valued functions. Proving a theorem establishes that the corresponding function always evaluates to **true**. As in PRA (Primitive Recursive Arithmetic) and Boyer-Moore logics, existential goals are expressed with explicit computable Skolem functions [1, 3, 11].

2.1 Definitional Principle

Users work with Imandra by incrementally extending its logical world through definitions of types, functions, modules and theorems. Each extension is governed by a definitional principle designed to maintain the consistency of Imandra’s current logical theory through a discipline of *conservative extensions*. Types must be proved well-founded. Functions must be proved terminating. These termination proofs play a dual role: Their structure is mined in order to instruct Imandra how to construct induction principles tailored to the recursive function being admitted when it later appears in conjectures.

Imandra’s definitional principle is built upon the ordinals up to ϵ_0 . Ordinals are encoded as a datatype (`Ordinal.t`) in Imandra using a variant of Cantor normal form, and the well-foundedness of `Ordinal.<<`—the strict less-than relation on `Ordinal.t` values—is an axiom of Imandra’s logic.

To prove a function f terminating, an ordinal-valued *measure* is required. Measures can often be inferred (e.g., for structural recursions) and may be specified by the user. To establish termination, all recursive calls of f are collected together with their guards, and their arguments must be proved to conditionally map to strictly smaller ordinals via the measure. Imandra provides a shorthand annotation for specifying lexicographic orders (`@@adm`), and explicit measure functions may be given using the `@@measure` annotation.

Example 1 (Ackermann). We can define the Ackermann function and prove it terminating with the attribute `[@@adm m,n]` which maps `ack m n` to the ordinal $m \cdot \omega + n$. Alternatively, we could use `[@@measure Ordinal.(pair (of_int m) (of_int n))]` to give an explicit measure via helper functions in Imandra’s `Ordinal` module.

```
let rec ack m n =
  if m <= 0 then n + 1 else if n <= 0 then ack (m-1) 1 else ack (m-1) (ack m (n-1))
[@@adm m,n]
```

Example 2. Here we have a naive version of the classic *left-pad* function [13], where termination depends on both arguments in a non-lexicographic manner:

```
let rec left_pad c n xs =
  if List.length xs >= n then xs else left_pad c n (c :: xs)
[@@measure Ordinal.of_int (n - List.length xs)]
```

2.2 Lifting, Specialization and Monomorphization

Imandra definitions may be polymorphic and higher-order. However, once Imandra is tasked with determining the truth value of a conjecture, the goal and its transitive dependencies are transformed into a family of ground, monomorphic first-order (recursive) definitions. These transformations include lambda lifting, specialization and monomorphization. Imandra's supported fragment of OCaml is designed so that all admitted definitions may be transformed in this way.

Example 3. To prove the following higher-order theorem

```
theorem same_len l =
  List.length (List.map (fun x -> x+1) l) = List.length l
```

we obtain a set of lower level definitions, where the anonymous function was lifted, the type list was monomorphised, and map and length were specialised:

```
type int_list = Nil_int | Cons_int of int * int_list
let rec length_int = function
  | Nil_int -> 0
  | Cons_int (_, tl) -> 1 + length_int tl
let map_lambda0 x = x+1
let rec map1 = function
  | Nil_int -> Nil_int
  | Cons_int (x, tl) -> Cons_int (map_lambda0 x, map1 tl)
theorem same_len (l:int_list) : bool =
  length_int (map1 l) = length_int l
```

3 Unrolling of Recursive Functions

A major feature of Imandra is its ability to automatically search for proofs and counterexamples in a logic with recursive functions. When a counterexample is found, it is reflected as a first-class value in Imandra's runtime and can be directly computed with and run through the model being analysed. In fact, the statement **verify** (**fun** $x \rightarrow \dots$) does not try any inductive proving unless requested; the default strategy is recursive function *unrolling* for a fixed number of steps, a form of bounded symbolic model-checking.

Our core unrolling algorithm is similar in spirit to the work of Suter et al. [12] but with crucial strategic differences. In essence, Imandra uses the *assumption* mechanism of SMT to block all Boolean assignments that involve the evaluation

of a (currently) uninterpreted ground instance of a recursive function. A refinement loop, based on extraction of unsat-cores from this set of assumptions, then expands (interprets) the function calls one by one until a model is found, an empty unsat-core is obtained, or a maximal number of steps is reached.

Definition 1 (Function template). *A function template for f is a set of tuples $(g, \mathbf{t}, \mathbf{p})$ such that the body of f contains a call to $g(\mathbf{t})$ under the path \mathbf{p} .*

Example 4

- $\text{fact}(x) = \text{if } x > 1 \ (x * \text{fact}(x - 1)) \ 1$ has as template $\{(\text{fact}, (\mathbf{x} - \mathbf{1}), (x > 1))\}$
- $f(x) = 1 + \text{if } g(0) \ h(g(x)) \ h(42)$
has as template
 $\{(g, (0), \top), (h, (g(\mathbf{x})), (g(0) = \top)), (g, (\mathbf{x}), g(0) = \top), (h, (42), (g(0) = \perp))\}$

We use what we call *reachability literals* to prevent the SMT solver from picking assignments that use function calls that are not expanded yet. A reachability literal is a Boolean atom that doesn't appear in the original problem, and that we associate to a given function call $f(\mathbf{t})$ regardless of where it occurs. This is to be contrasted with Suter et al.'s notion of *control literals* associated with individual occurrences of function calls within the expanded body of another function call. We denote by $\mathbf{b}[f(\mathbf{t})]$ the unique reachability literal for $f(\mathbf{t})$.

```

1  def calls_of_term(t: Term):
2    return {b[f(u)] | f(u) < t}
3
4  def subcalls_of_call(f(t): Term, expanded: Set[Term]):
5    return {(b[g(u)], p) | (g, u, ^ p) ∈ template(f)[t/x] ∧ g(u) ∉ expanded}
6
7  def unroll(goal: Formula) -> SAT|UNSAT:
8    q = calls_of_term(goal), expanded = ∅
9    F = goal ∧ ∧_{a ∈ q} a
10   while True:
11     is_sat, unsat_core = check_sat(F, assume={¬a | a ∈ q})
12     if is_sat == SAT: return SAT
13     else if is_sat == UNSAT:
14       if unsat_core == ∅: return UNSAT
15       b[f(t)] = pick_from(unsat_core) # next call to expand
16       expanded = {f(t)} ∪ expanded
17       {(a_i, p_i)}_i = subcalls_of_call(f(t), expanded)
18       q = q ∪ {a_i}_i \ b[f(t)]
19       F = F ∧ b[f(t)] ∧ f(t) = body_f[t/x] ∧ ∧_i (b[f(t)] ∧ p_i ⇒ a_i)

```

Fig. 2. Unrolling algorithm

The main search loop is presented in Fig. 2, where body_f is the body of f (i.e. $f(\mathbf{x}) \stackrel{\text{def}}{=} \text{body}_f$) and $t < u$ means t is a proper subterm of u . We start with F

initialized to the original goal, and the queue q containing function calls in the goal (computed by `calls_of_term`). Each iteration of the loop starts by checking validity under the assumption that all reachability literals in q are false (line 11). If no model is found, we pick an unexpanded function call $f(\mathbf{t})$ from the unsat core (line 15). Selection must be *fair*: all function calls must eventually be picked.

To expand $f(\mathbf{t})$, the corresponding reachability literal becomes true, we instantiate the body of f on \mathbf{t} , and use `subcalls_of_call` to compute the set of subcalls along with their control path within $f(\mathbf{t})$ (using f 's template). For each $b[g(\mathbf{u})]$ occurring under path p inside $\text{body}_f(\mathbf{t})$, we need to block models that would make p valid until $g(\mathbf{u})$ gets expanded. The assertions $\bigwedge_i (b[f(\mathbf{t})] \wedge p_i \Rightarrow a_i)$ delegate to SMT the work of tracking which paths are forbidden. This way, expanding one function call might lead to many paths becoming “unlocked” at once.

4 Induction

Imandra has extensive support for automated induction built principally around Imandra’s *inductive waterfall*¹. This combines techniques such as symbolic execution, lemma-based conditional rewriting, forward-chaining, generalization and the automatic synthesis of goal-specific induction principles. Induction principle synthesis depends upon data computed about a function’s termination obtained when it was admitted via our definitional principle. Imandra’s waterfall is deeply

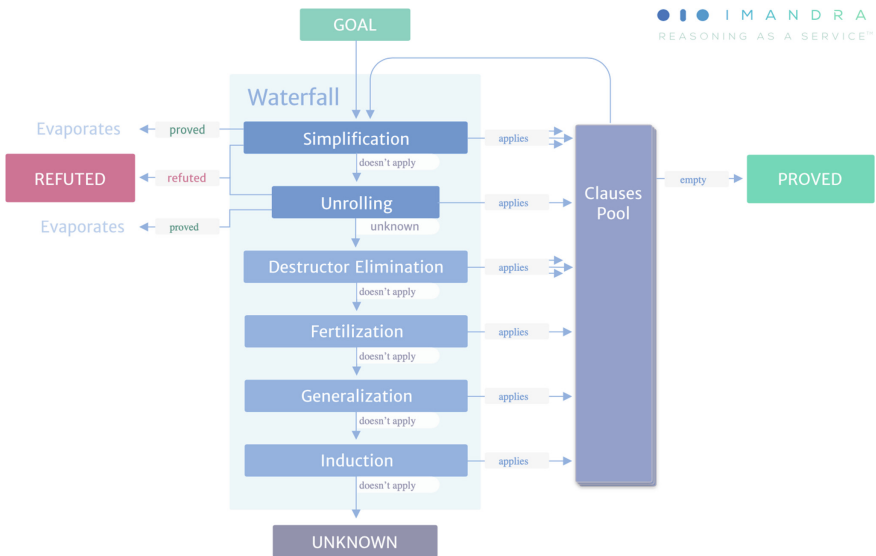


Fig. 3. Imandra’s inductive waterfall

¹ More details about Imandra’s waterfall and rule classes may be found in our online documentation at <https://docs.imandra.ai>.

inspired by the pioneering work of Boyer-Moore [1, 6], and is in many ways a “lifting” of the Boyer-Moore waterfall to our typed, higher-order setting (Fig. 3).

Imandra’s waterfall contains a simplifier which automatically makes use of previously proved lemmas. Once proved, lemmas may be installed as rewrite, forward-chaining, elimination or generalization rules. Imandra gives users feedback in order to help them design efficient collections of rules. With a good collection of rules (especially rewrite rules), it is hoped that “most” useful theorems over a given domain will be provable by simplification alone, and induction will only be applied as a last resort. In these cases, the subsequent waterfall moves are designed to prepare the simplified conjecture for induction (via, e.g., generalization) before goal-specific induction principles are synthesized.

Imandra’s inductive waterfall plays an important role in what we believe to be a robust verification strategy for applying Imandra to real-world systems. Recall that all Imandra goals may be subjected to bounded verification via unrolling (cf. Sect. 3). In practice, we almost always attack a goal by unrolling first, attempting to verify it up to a bound before we consider trying to prove it by induction. Typically, for real-world systems, models and conjectures will have flaws, and unrolling will uncover many counterexamples, confusions and mistakes. As all models are executable and all counterexamples are reflected in Imandra’s runtime, they can be directly run through models facilitating rapid investigation. It is typically only after iterating on models and conjectures until all (bounded) counterexamples have been eliminated that we consider trying to prove them by induction. Imandra’s support for counterexamples also plays another important role: as a filter on heuristic waterfall steps such as generalization.

5 Architecture and User Interfaces

Imandra is developed in OCaml and integrates with its compiler libraries. Arbitrary OCaml code may interact with Imandra models and counterexamples through the use of Imandra’s `program mode` and reflection machinery. Imandra integrates with Z3 [2] for checking satisfiability of various classes of ground formulas. Imandra has a client-server architecture: (i) the client parses and executes models with an integrated toplevel; (ii) the server, typically in the cloud, performs all reasoning. Imandra’s user interfaces include:

Command line for power users, with tab-completion, hints, and colorful messages. This interface is similar in some ways to OCaml’s `utop`.

Jupyter notebooks hosted online or via local installation through Docker [10].

This presents Imandra through interactive notebooks in the browser.

VSCoDe plugin where documents are checked on the fly and errors are underlined in the spirit of Isabelle’s Prover IDE [14].

6 Conclusion

Imandra is an industrial-strength reasoning system combining ideas from SMT, Boyer-Moore inductive provers, and ITPs for typed higher-order logics. Imandra delivers an extremely high degree of automation and has novel techniques

such as reflected computable counterexamples that we now believe are indispensable for the effective industrial application of automated reasoning. We are encouraged by Imandra’s success in mainstream finance [9], and share a deep conviction that further advances in automation and UI—driven in large part by meeting the demands of industrial users—will lead to a (near-term) future in which automated reasoning is a widely adopted foundational technology.

References

1. Boyer, R.S., Moore, J.S.: *A Computational Logic*. Academic Press Professional Inc, Cambridge (1979)
2. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) *TACAS 2008*. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
3. Goodstein, R.L.: *Recursive Number Theory: A Development of Recursive Arithmetic in a Logic-Free Equation Calculus*. North-Holland Pub. Co., Amsterdam (1957)
4. Gordon, M.J.C., Melham, T.F.: *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, Cambridge (1993)
5. Harrison, J.: HOL light: an overview. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) *TPHOLs 2009*. LNCS, vol. 5674, pp. 60–66. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03359-9_4
6. Kaufmann, M., Moore, J.S.: ACL2: an industrial strength version of Nqthm. In: *Computer Assurance (COMPASS 1996)*, pp. 23–34. IEEE (1996)
7. Nipkow, T., Wenzel, M., Paulson, L.C. (eds.): *Isabelle/HOL*. LNCS, vol. 2283. Springer, Heidelberg (2002). <https://doi.org/10.1007/3-540-45949-9>
8. Owre, S., Rushby, J.M., Shankar, N.: PVS: a prototype verification system. In: Kapur, D. (ed.) *CADE 1992*. LNCS, vol. 607, pp. 748–752. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-55602-8_217
9. Passmore, G.O., Ignatovich, D.: Formal verification of financial algorithms. In: de Moura, L. (ed.) *CADE 2017*. LNCS (LNAI), vol. 10395, pp. 26–41. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63046-5_3
10. Pérez, F., Granger, B.E.: IPython: a system for interactive scientific computing. *Comput. Sci. Eng.* **9**(3), 21–29 (2007)
11. Skolem, T.: The foundations of elementary arithmetic established by means of the recursive mode of thought, without the use of apparent variables ranging over infinite domains. In: van Heijenoort, J. (ed.) *From Frege to Gödel*. Harvard University Press, Cambridge (1967)
12. Suter, P., Köksal, A.S., Kuncak, V.: Satisfiability modulo recursive programs. In: Yahav, E. (ed.) *SAS 2011*. LNCS, vol. 6887, pp. 298–315. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23702-7_23
13. Hillel, W.: *The Great Theorem Prover Showdown* (2018). <https://www.hillelwayne.com/post/theorem-prover-showdown/>
14. Wenzel, M.: Isabelle/jEdit – a prover IDE within the PIDE framework. In: Jeur-ing, J., et al. (eds.) *CICM 2012*. LNCS (LNAI), vol. 7362, pp. 468–471. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31374-5_38