The Lean Theorem Prover (System Description)

Leonardo de Moura[1](✉), Soonho Kong[2], Jeremy Avigad[2], Floris van Doorn[2], and Jakob von Raumer[2]

[1] Microsoft Research, Redmond, USA
`leonardo@microsoft.com`
[2] Carnegie Mellon University, Pittsburgh, USA
`soonhok@cs.cmu.edu`, `{avigad,fpv,javra}@andrew.cmu.edu`

Abstract. Lean is a new open source theorem prover being developed at Microsoft Research and Carnegie Mellon University, with a small trusted kernel based on dependent type theory. It aims to bridge the gap between interactive and automated theorem proving, by situating automated tools and methods in a framework that supports user interaction and the construction of fully specified axiomatic proofs. Lean is an ongoing and long-term effort, but it already provides many useful components, integrated development environments, and a rich API which can be used to embed it into other systems. It is currently being used to formalize category theory, homotopy type theory, and abstract algebra. We describe the project goals, system architecture, and main features, and we discuss applications and continuing work.

1 Introduction

Formal verification involves the use of logical and computational methods to establish claims that are expressed in precise mathematical terms. These can include ordinary mathematical theorems, as well as claims that pieces of hardware or software, network protocols, and mechanical and hybrid systems meet their specifications. In practice, there is not a sharp distinction between verifying a piece of mathematics and verifying the correctness of a system: formal verification requires describing hardware and software systems in mathematical terms, at which point establishing claims as to their correctness becomes a form of theorem proving. Conversely, the proof of a mathematical theorem may require a lengthy computation, in which case verifying the truth of the theorem requires verifying that the computation does what it is supposed to do.

Automated theorem proving focuses on the "finding" aspect, and strives for power and efficiency, often at the expense of guaranteed soundness. Such systems can have bugs, and typically the correctness relies on extensive testing. In

J. Avigad—Work supported by the AFOSR under MURI Grant Number FA9550-15-1-0053.

J. von Raumer—Visiting student from Karlsruhe Institute of Technology, sponsored by the Baden-Württemberg-Stipendium.

A.P. Felty and A. Middeldorp (Eds.): CADE-25, LNAI 9195, pp. 378–388, 2015.
DOI: 10.1007/978-3-319-21401-6_26

contrast, interactive theorem proving focuses on the *verification* aspect of theorem proving, requiring that every claim is supported by a proof in a suitable axiomatic foundation. This sets a very high standard: every rule of inference and every step of a calculation has to be justified by appealing to prior definitions and theorems, all the way down to basic axioms and rules. In fact, most such systems provide fully elaborated *proof objects* that can be communicated to other systems and checked independently. Constructing such proofs typically requires much more input and interaction from users, but it allows us to obtain deeper and more complex proofs. Finally, we remark that some automated theorem provers do generate proof certificates that can be verified by other systems [8].

The *Lean Theorem Prover*[1] aims to bridge the gap between interactive and automated theorem proving, by situating automated tools and methods in a framework that supports user interaction and the construction of fully specified axiomatic proofs. The goal is to support both mathematical reasoning and reasoning about complex systems, and to verify claims in both domains. Lean is released under the Apache 2.0 license, a permissive open source license that permits others to use and extend the code and mathematical libraries freely. At Carnegie Mellon University, Lean is already being used to formalize category theory, homotopy type theory, and abstract algebra. Lean is an ongoing, long-term effort, and much of the potential for automation will be realized only gradually over time.

Lean's small, trusted kernel is based on dependent type theory, with several configuration options. It can be instantiated with an impredicative sort or propositions, `Prop`, to provide a version of the Calculus of Inductive Constructions (CIC) [5,6]. Moreover, `Prop` can be marked proof-irrelevant if desired. Without an impredicative `Prop`, the kernel implements a version of Martin-Löf type theory [12,23]. In both cases, Lean provides a sequence of non-cumulative type universes, with universe polymorphism.

Lean is meant to be used both as a standalone system and as a software library. SMT solvers can use the Lean API to create proof terms that can be independently checked. The API can be used to export Lean proofs to other systems based on similar foundations (e.g., Coq [3] and Matita [1]). Lean can also be used as an efficient proof checker, and definitions and theorems can be checked in parallel using all available cores on the host machine. When used as a proof assistant, Lean provides a powerful elaborator that can handle higher-order unification, definitional reductions, coercions, overloading, and type classes, in an integrated way. Lean allows users to provide definitions and theorems using a declarative style resembling Mizar [20] and Isabelle/Isar [24]. Lean also provides *tactics* as an alternative (more imperative) approach to constructing (proof) terms as in Coq, HOL-Light [11], Isabelle [17] and PVS [19]. Moreover, the declarative and tactic styles can be freely mixed together.

Lean includes two libraries of formally verified mathematics and basic datastructures. The standard library uses a kernel instantiated with an impredicative and proof-irrelevant `Prop`. This library supports constructive and classical users,

[1] http://leanprover.github.io.

and the following axioms can be optionally used: propositional completeness, function extensionality, and strong indefinite description. Lean also contains a library tailored for Homotopy Type Theory (HoTT) [23], using a predicative and proof relevant instantiation of the kernel and higher inductive types (HITs). Future plans to support HoTT include sorts for fibrant type universes.

2 The Kernel

Lean's trusted kernel is implemented in two layers. The first layer contains the type checker and APIs for creating and manipulating terms, declarations, and the environment. This layer consists of 6k lines of C++ code. The second layer provides additional components such as inductive families and quotient types. The inductive family component consists of 700 lines of code. When the kernel is instantiated, one selects which of these components should be used. We have tried to maintain the number of objects manipulated by the kernel to a minimum: the list consists of terms, universe terms, declarations, and environments. Identifiers are encoded as *hierarchical names* [14], i.e. lists of strings/numbers, such as $x.y.1$.

Terms. The term language is a dependent λ-calculus. A term can be a free variable (also called a local constant), a bound variable, a constant (parameterized by universe terms), a function application $f\ t$, a lambda abstraction $\lambda x : A, t$, a function space $\Pi x : A, B$, a sort `Type` u (where u is a universe term), a metavariable, or a macro $m[t_1, \ldots, t_n]$.

Sorts. The sorts `Type` u are used to encode the infinite sequence of universes $\texttt{Type}_0, \texttt{Type}_1, \texttt{Type}_2, \ldots$ An *explicit* universe term is of the form $\texttt{s}^k\ \texttt{z}$ (for $k \geq 0$), where `z` denotes the base universe zero, and `s` denotes the *successor* universe operator. We use `Type z` to represent `Prop` in kernel instantiations that support it. To support universe polymorphism, we also have universe parameters (an identifier), and the operators `max` $u_1\ u_2$ and `imax` $u_1\ u_2$. The universe term `max` $u_1\ u_2$ denotes the universe that is greater than or equal to u_1 and u_2, and is equal to one of them. The universe term `imax` $u_1\ u_2$ denotes the universe zero if u_2 denotes zero, and `max` $u_1\ u_2$ otherwise. The operator `imax` is only needed for kernel instantiations that have an impredicative `Prop`. In these kernels, given $A :$ `Type` u_1 and $B :$ `Type` u_2, the type of $\Pi x : A, B$ is `Type` (`imax` $u_1\ u_2$). The `imax` operator makes sure that $\Pi x : A, B$ is a proposition when B is a proposition.

Free and bound variables. Free variables have a unique identifier and a type, and bound variables are just a number (a de Bruijn index). By storing the type with each free variable, we do not need to carry around contexts in the type checker and normalizer. As described in [14], this representation simplifies the implementation considerably, and it also minimizes the number of places where calculations with de Bruijn indices must be performed.

Metavariables. In Lean, users may provide *partial constructions*, i.e., constructions containing "holes" that must be filled by the system. These holes (also known as placeholders) are internally represented as metavariables that must be replaced by closed terms that are synthesized by the system. Since only closed terms can be assigned to metavariables, a metavariable that occurs in a context records the parameters it depends on. For example, we encode a hole in the context $(x : nat)$ $(y : bool)$ as $?m\ x\ y$, where $?m$ is a fresh metavariable. As with free variables, every metavariable has a type. We also have universe metavariables to represent "holes" in universe terms.

Macros. Macros, which can be viewed as *procedural attachments*, provide more efficient ways of storing and working with terms. Each macro must provide two procedures, namely, type inference and macro expansion. The type inference procedure `minfer` is responsible for computing the type of a macro application $m[t_1, \ldots, t_n]$, and the macro expansion procedure `mexpand` must expand/eliminate the macro application. The point is that, given a term t of the form $m[t_1, \ldots, t_n]$, `minfer`(t) may be able to infer the type of `mexpand`(t) more efficiently than the kernel type checker, and t may be more compact than `mexpand`(t).

We also use macros to store annotations and hints used by automation such as rewriters and decision procedures. Each macro has a *trust level* represented by a natural number. When the Lean kernel is initialized, the user must provide a trust level ℓ, and the kernel then refuses any term that contains a macro with trust level greater than or equal to ℓ. A kernel initialized with trust level zero does not accept any macro, forcing any macro occurring in declarations to be expanded. The idea is that macros are not part of the trusted code base, but users may choose to trust them "most of the time" when formalizing a system and/or theorem. Note that an independent type checker for Lean does not need to implement support for metavariables or macros.

Environments. An environment stores a sequence of declarations. The kernel currently supports three different kinds of declarations: axioms, definitions and inductive families. Each has a unique identifier, and can be parameterized by a sequence of universe parameters. Every axiom has a type, and every definition has a type and a value. A constant in Lean is just a reference to a declaration. The main task of the kernel is to type check these declarations and refuse type incorrect ones. The kernel does not allow declarations containing metavariables and/or free variables to be added to an environment. Environments are never destructively updated, and are implemented using pure red-black trees, where the keys are hierarchical names.

Inductive families. Inductive families [9] are a form of simultaneously defined collection of algebraic data-structures which can be parameterized over values as well as types. Each inductive family definitions produces introduction rules, elimination rules, and computational rules as described in [9]. As in the CIC, the instances of an inductive family can be in `Prop`, and special rules are used to make sure the eliminator is compatible with proof irrelevance. Finally, when proof irrelevance is enabled in the kernel, axiom K [22] "computes" in Lean (a similar feature is available in Agda [18]). In contrast to Coq, Lean does not have fix-point expressions, `match` expressions, or a termination checker in the kernel. Instead, recursive definitions and pattern matching are compiled into eliminators outside of the kernel.

The type checker. To minimize the amount of code duplication, the type checker plays two roles. First, it is used to validate any declaration sent to the kernel before adding it to an environment. Second, it is used by elaboration procedures that try to synthesize holes in terms provided by the user. Consequently, the type checker is capable of processing terms containing metavariables. When a term contains metavariables, the type checker may produce unification constraints, in which case the resultant type is correct only if the unification constraints can be resolved.

3 Elaboration

The task of the elaborator is to convert a partially specified expression into a fully specified, type-correct term. When typing in a term, users can leave arguments implicit by entering them with an underscore (i.e., a "hole"), leaving it to the elaborator to infer a suitable value. One can also mark arguments implicit by putting them in curly brackets when defining a function, to indicate that they should generally be inferred rather than entered explicitly. For example, the standard library defines the identity function as:

```
definition id {A : Type} (a : A) : A := a
```

As a result, the user can write `id a` rather than `id A a`. It is fairly routine to infer the type `A` given `a : A`. Often the elaborator needs to infer an element of a Π-type, which constitutes a *higher-order* problem. For example, if `e : a = b` is a proof of the equality of two terms of some type `A`, and `H : P` is a proof of some expression involving `a`, the term `subst e H` denotes a proof of the result of replacing some or all the occurrences of `a` in `P` with `b`. Here not just the type `A` is inferred, but also an expression `C : A → Prop` denoting the context for the substitution, that is, the expression with the property that `C a` "reduces" to `P`. Such expressions can be ambiguous. For example, if `H` has type `R (f a a) a`, then with `subst e H` the user may have in mind `R (f b b) b` or `R (f a b) a` among other interpretations, and the elaborator has to rely on context and a backtracking search to find an interpretation that fits. Similar issues arise with proofs by induction, which require the system to infer an induction predicate.

The elaborator should also respect the computational interpretation of terms. It should recognize the equivalence of terms `(λx, t)s` and `t[s/x]` under beta reduction, as well as `(s, t).1` and `s` under the reduction rule for pairs. (Terms that are equivalent modulo such reductions are said to be *definitionally equal.*) Unfolding definitions and reducing projections is especially crucial when working with algebraic structures, where many basic expressions cannot even be seen to be type correct without carrying out such reductions.

Lean's elaborator also supports ad-hoc overloading; for example, we can use notation `a + b` for addition on the natural numbers, integers, and additive groups simultaneously. Each possible interpretation becomes a choice-point in the elaboration process. The elaborator can also detect the need to insert a coercion, say, from `nat` to `int`, or from the class of rings to the class of additive groups.

Lean also supports the use of Haskell-style *type classes*. For example, we can define a class `has_mul A` of types `A` with an associated multiplication operator, and a class `semigroup A` of types `A` with semigroup structure, as follows:

```
structure has_mul [class] (A : Type) :=
(mul : A → A → A)
structure semigroup [class] (A : Type) extends has_mul A :=
(mul_assoc : ∀a b c, mul (mul a b) c = mul a (mul b c))
```

We can then declare appropriate instances of these classes, and instruct the elaborator to synthesize such instances when processing the notation `a * b` or the generic theorem `mul.assoc`.

Finally, definitions and proofs can invoke *tactics*, that is, user-defined or built-in procedures that construct various subterms. The elaborator needs to call these procedures at appropriate times during the elaboration process to fill in the corresponding components of a term.

The interactions between these components are subtle, and the main difficulty is that the elaborator has to deal with them all at the same time. A definition or proof may give rise to thousands of constraints requiring a mixture of higher-order unification, disambiguation of overloaded symbols, insertion of coercions, type class inference, and computational reduction. To solve these, the elaborator uses nonchronological backtracking and a carefully tuned algorithm [7].

Recursive equations. Lean provides natural ways of defining recursive functions, performing pattern matching, and writing inductive proofs. Behind the scenes, these are "compiled" down into eliminators and auxiliary definitions automatically generated by Lean whenever we declare an inductive family. This compiler is based on ideas from [4,10,13,21]. The default compilation method supports structural recursion, i.e. recursive applications where one of the arguments is a subterm of the corresponding term on the left-hand-side. Lean can also compile recursive equations using well-founded recursion. The main advantage of the default compilation method is that the recursive equations hold definitionally.

The compiler also supports dependent pattern matching for indexed inductive families. For example, we can define the type `vector A n` of vectors of type `A` and length `n` as follows:

```
inductive vector (A : Type) : nat → Type :=
| nil {} : vector A zero
| cons   : Π  {n : nat}, A → vector A n → vector A (succ n)
```

We can then define a function `map` that applies a binary function `f` to elements of vectors of type `A` and `B`, to produce a vector of elements of type `C`:

```
definition map {A B C : Type} (f : A → B → C) :
      Π {n : nat}, vector A n → vector B n → vector C n
| map nil     nil    := nil
| map (a: :va) (b: :vb) := f a b : : map va vb
```

Note that we can omit "unreachable" cases such as `map nil (a :: va)` because the input vectors have the same length. Behind the scenes, a lot of boilerplate code is needed to reduce these definitions to eliminators for the inductive family.

Type classes. Any family of inductive types can be marked as a *type class*. Then we can declare particular elements of a type class to be *instances*. These provide hints to the elaborator: any time the elaborator is looking for an element of a type class, it can consult a table of declared instances to find a suitable element. What makes type class inference powerful is that one can *chain* instances, that is, an instance declaration can in turn depend on other instances. This causes class inference to chain through instances recursively, backtracking when necessary. The Lean type class resolution procedure can be viewed as a simple λ-Prolog interpreter [15], where the Horn clauses are the user declared instances.

For example, the standard library defines a type class `inhabited` to enable type class inference to infer a "default" or "arbitrary" element of types that contain at least one element.

```
inductive inhabited [class] (A : Type) : Type :=
mk : A → inhabited A
```

Elements of the class `inhabited A` are of the form `inhabited.mk a`, for some element `a : A`. The following function extracts the corresponding element:

```
definition default (A : Type) [H : inhabited A] : A :=
inhabited.rec (λa, a) H
```

The annotation `[H : inhabited A]` indicates that `H` should be synthesized from instance declarations using type class resolution. We can then declare suitable instances for types like `nat` and `Prop`. The following declaration shows that if two types `A` and `B` are inhabited, then so is their product:

```
definition prod.is_inhabited [instance] {A B : Type}
  (H1 : inhabited A) (H2 : inhabited B) : inhabited (A × B) :=
inhabited.mk (default A, default B)
```

Declarative Proofs. Lean provides a rich notation declaration system [2], and it is used to support human readable proofs similar to the ones found in Mizar and Isabelle/Isar. For example, the `have` construct introduces an auxiliary subgoal in a longer proof. Internally, the notation `have H : p, from s, t` produces the term `(λ(H : p), t) s`. Similarly, `show p, from t` does nothing more than annotate `t` with its expected type `p`. Lean also provides alternative Mizar/Isar-inspired syntax for lambda abstractions: `assume H : p, t` and `take x : A, t`. Calculational proofs, which begin with the keyword `calc`, are a convenient notation for chaining intermediate results that are meant to be composed by basic principles such as the transitivity of equality. The set of binary relation predicates supported in calculational proofs can be freely extended by users. In the following example, we demonstrate some of these features:

```
theorem le.antisymm : ∀ {a b : ℤ}, a ≤ b → b ≤ a → a = b :=
take a b : ℤ, assume (H₁ : a ≤ b) (H₂ : b ≤ a),
obtain (n : ℕ) (Hn : a + n = b), from le.elim H₁,
obtain (m : ℕ) (Hm : b + m = a), from le.elim H₂,
have H₃ : a + of_nat (n + m) = a + 0, from
... -- suppressed rest of the proof due to space limitations
have H₆ : n = 0, from nat.eq_zero_of_add_eq_zero_right H₅,
show a = b, from
  calc
    a = a + 0      : add_zero
      ... = a + n  : H₆
      ... = b      : Hn
```

Namespaces. Lean provides the ability to group definitions, as well as meta-objects such as notation declarations, coercions, rewrite rules and type classes, into nested, hierarchical *namespaces*. The `open` command brings the shorter names and all meta-objects into the current context.

The tactic framework. Tactics provide an alternative approach to constructing terms. We can view a term as a representation of a construction or mathematical proof; tactics are commands, or instructions, that describe how to build such a term. Most automation available in Lean is integrated into the system as tactics. For example, Lean contains a `rewrite` tactic that provides a basic mechanism for performing rewriting. The tactic framework provides a general mechanism for synthesizing metavariables. In this framework, we say a metavariable is a *goal*. A *proof state* contains a sequence of goals; postponed unification constraints; and a substitution which stores already assigned metavariables. A *tactic* is a function that maps a proof state into a stream of proof states, implemented as a lazy list [16]. This is important because some tactics may produce an unbounded stream of proof states. Lean provides all usual combinators (also known as *tacticals*) available in other interactive theorem provers, such as `andthen`, `orelse`, and `try`. Lean also provides the tacticals `par` (for executing tactics concurrently in multiple cores), and `tryfor T n` that fails if tactic `T` does not terminate in `n` milliseconds. Lean also comes equipped with basic tactics such as `apply`, `intro`, `generalize`, `rewrite`, etc. The complete list of tactics is described in [2]. Wherever a term is expected, Lean allows us to insert instead a `begin` ... `end` block, composed of a sequence of tactics separated by commas. Here is a small example using tactics:

```
theorem test (p q : Prop) : p → q → p ∧ q ∧ p :=
begin
  intro Hp, intro Hq,
  apply and.intro, exact Hp, apply and.intro,
  exact Hq, exact Hp
end
```

4 The User Interface

Lean's standard integrated development environment (IDE) is based on the Emacs editor, and provides continuous elaboration and checking. In the background, the source text is continuously analyzed and annotated with semantic information as it is being edited by the user. The interaction between editor and prover is performed by an asynchronous protocol which exploits parallelism, multi-core hardware, and incremental compilation. The native interface provides all standard features found in advanced IDEs, such as hyperlinks, auto-completion, syntax highlighting, error highlighting, etc. Users can view automatically synthesized terms, implicit coercions, and overloading resolution. If a user makes changes to a file higher in the dependency chain, everything is recompiled in the background, and with caching the changes are propagated almost immediately.

The Javascript bindings for Lean do not contain any native code, and can be used in any modern web browser. They are intended for web applications such as web IDEs[2], "live" tutorial/documentation[3] and online exercises. We have used this infrastructure to develop course material for an interactive theorem proving course[4] being offered in the spring of 2015 at CMU.

5 Conclusion

Lean has been designed with the goal of obtaining a theorem proving system which has all of the following features: an expressive logical foundation for writing mathematical specifications and proofs; an interactive and supportive user interface and environment; a flexible framework for supporting automation; and a rich API that can be used to embed this functionality into other systems. Lean already provides a novel elaboration procedure that can handle higher-order unification, definitional reductions, coercions, overloading, and type classes, in an integrated way. It has a relatively small trusted kernel, making the task of implementing a reference/independent type checker for Lean much simpler. It is also quite fast, with support for multi-core machines and coarse and fine grain parallelism. Lean is an ongoing and long-term effort, and future plans include extensive search procedures, decision procedures, better support for homotopy type theory, and an independent type checker.

[2] http://leanprover.github.io/live.

[3] http://leanprover.github.io/tutorial.

[4] http://www.cs.cmu.edu/~emc/15815-s15.

References

1. Asperti, A., Ricciotti, W., Sacerdoti Coen, C., Tassi, E.: The Matita interactive theorem prover. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS, vol. 6803, pp. 64–69. Springer, Heidelberg (2011)
2. Avigad, J., de Moura, L., Kong,S.: Theorem Proving in Lean (2015). http://leanprover.github.io/tutorial/tutorial.pdf
3. Barras, B., Boutin, S., Cornes, C., Courant, J., Filliatre, J.-C., Gimenez, E., Herbelin, H., Huet, G., Munoz, C., Murthy, C. et al.: The Coq proof assistant reference manual: Version 6.1 (1997)
4. Cockx, J., Devriese, D., Piessens, F.: Pattern matching without K. In: Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming, pp. 257–268. ACM (2014)
5. Coquand, T., Huet, G.: The calculus of constructions. Inf. Comput. **76**(2–3), 95–120 (1988)
6. Coquand, T., Paulin, C.: Inductively defined types. In: COLOG-88 (Tallinn, 1988), pp. 50–66. Springer, Berlin (1990)
7. de Moura, L., Avigad, J., Kong, S., Roux, C.: Elaboration in dependent type theory. Preprint (arXiv)
8. Delahaye, D., Woltzenlogel Paleo, B. (eds.): All about proofs, proofs for all. Mathematical Logic and Foundations, vol. 55 (2015)
9. Dybjer, P.: Inductive families. Formal Aspects Comput. **6**(4), 440–465 (1994)
10. Goguen, H.H., McBride, C., McKinna, J.: Eliminating dependent pattern matching. In: Futatsugi, K., Jouannaud, J.-P., Meseguer, J. (eds.) Algebra, Meaning, and Computation. LNCS, vol. 4060, pp. 521–540. Springer, Heidelberg (2006)
11. 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)
12. Martin-Löf, P.: Intuitionistic type theory. Bibliopolis (1984)
13. McBride, C., Goguen, H.H., McKinna, J.: A few constructions on constructors. In: Filliâtre, J.-C., Paulin-Mohring, C., Werner, B. (eds.) TYPES 2004. LNCS, vol. 3839, pp. 186–200. Springer, Heidelberg (2006)
14. McBride, C., McKinna, J.: Functional pearl: I am not a number-I am a free variable. In: Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell, Haskell 2004, pp. 1–9. ACM, New York (2004)
15. Miller, D., Nadathur, G.: Programming with Higher-Order Logic. Cambridge University Press, Cambridge (2012)
16. Nipkow, T., Paulson, L.C.: Isabelle-91. In: Kapur, Deepak (ed.) CADE 1992. LNCS, vol. 607. Springer, Heidelberg (1992)
17. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: a proof assistant for higher-order logic, vol. 2283. Springer Science and Business Media (2002)
18. Norell, U.: Dependently typed programming in Agda. In: Koopman, P., Plasmeijer, R., Swierstra, D. (eds.) AFP 2008. LNCS, vol. 5832, pp. 230–266. Springer, Heidelberg (2009)
19. Owre, S., Rushby, J., Shankar, N.: PVS: a prototype verification system. In: Kapur, Deepak (ed.) CADE 1992. LNCS, vol. 607. Springer, Heidelberg (1992)
20. Rudnicki, P.: An overview of the Mizar project. In: Proceedings of the 1992 Workshop on Types for Proofs and Programs, pp. 311–330 (1992)
21. Slind, K.: Function definition in higher-order logic. In: von Wright, Joakim, Harrison, J., Grundy, John (eds.) TPHOLs 1996. LNCS, vol. 1125. Springer, Heidelberg (1996)

22. Streicher, T.: Investigations into intensional type theory. Ph.D. thesis, LMU (1993)
23. The Univalent Foundations Program. Homotopy Type Theory: Univalent Foundations of Mathematics. Institute for Advanced Study (2013)
24. Wenzel, M.M.: Isabelle/Isar - a versatile environment for human-readable formal proof documents. Technical report (2002)