Arend — Proof-assistant assisted pedagogy

A graphical proof assistant for undergraduate computer science education

Andrew V. Clifton

April 26, 2015

Department of Computer Science
California State University, Fresno

Arend — Proof-assistant assisted pedagogy

A graphical proof assistant for undergraduate computer science education

Andrew V. Clifton

April 26, 2015

Department of Computer Science
California State University, Fresno
Fresno, CA 93740

Project Advisor
Dr. J. Todd Wilson

Submitted in partial fulfillment of the requirements
for the degree of Master of Science

Abstract

Computer-assisted theorem proving is a valuable tool for the development of formal proofs in computer science and mathematics. Although proof assistants such as Abella [Gacek, 2008], Coq [The Coq development team, 2004], and Twelf [Pfenning and Schürmann, 1999] are widely used in the research and professional domains, their use in computer science education remains relatively unexplored. In particular, *undergraduate* computer science education, where students are first exposed to the concept and elements of a *formal proof*, has largely ignored the possibility of computer-assistance in this area. We present AREND, a proof assistant specifically intended for use as a teaching tool in the undergraduate computer science curriculum.

Arend differs from existing systems in three main areas:

- While traditional proof assistants place equal emphasis on the development of *specifications* and the reasoning about them, Arend places the task of specification-development squarely in the hands of the instructor. The specification language is not intended for student consumption, and its use during the proof process is limited to being the source of the rules and data types being considered.
- While research-level proof assistants often have complex higher-order features, we restrict ourselves to a purely first-order logic. This limits the types of systems which can be specified — although common mathematical and computational structures such as $\mathbb{N}$, sets, lists, trees, and graphs are still readily representable — but also limits the complexity of the proofs that can be required. We feel it is important to avoid requiring proofs that do not obviously correspond to the analogous paper proofs.
- Arend's user interface is highly interactive and non-linear, allowing students to move freely between different cases in a proof, different proofs, new lemmas, etc. The underlying specification is presented visually, as a collection of *inference rules*, while proofs themselves are constructed and presented visually as *derivation trees*, rather than in the traditional written format (e.g., "by induction on ... by inversion..." etc.).

We hope that Arend will offer undergraduate students an early, and not unpleasant, exposure to the important realm of computer-assisted theorem proving, as well as offering instructors a useful new tool in the presentation and use of formal proofs.

A. Gacek. The Abella interactive theorem prover (system description). In A. Armando, P. Baumgartner, and G. Dowek, editors, *Proceedings of IJCAR 2008*, volume 5195 of *Lecture Notes in Artificial Intelligence*, pages 154–161. Springer, August 2008

The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. URL `http://coq.inria.fr`. Version 8.0

F. Pfenning and C. Schürmann. System description: Twelf—a meta-logical framework for deductive systems. In *Automated Deduction—CADE-16*, pages 202–206. Springer, 1999

Contents

Preface

FORMAL PROOFS ARE AN INTEGRAL PART of the undergraduate computer science curriculum. The techniques of proofs are introduced early, often within the first semester, and are used in both theoretic (e.g., language and automata theory) and practical (e.g., analysis of algorithms) coursework. Despite this, the use of computerized *proof assistants* at the undergraduate level is surprisingly rare. We believe that an early introduction to computer-assisted proofs can be a valuable and useful component of the undergraduate curriculum. With appropriate software tooling, the process of developing a formal proof can become an exercise in the same kind of algorithm reasoning used in other forms of programming.[1]

[1] Indeed, in the logic which undergirds our system, proofs *are* a limited kind of program.

AREND is a proof assistant designed specifically for use in the undergraduate sphere. This narrow focus allows it to deviate significantly from the designs of other proof assistants. In particular, while commercial and research proof assistants often aim for more *powerful* logics, so as to facilitate more powerful proof techniques, Arend's logic is intentionally simple, with just enough power to formalize the systems commonly introduced at the undergraduate level (set theory, number theory and induction over $\mathbb{N}$, finite automata, regular expressions, and formal languages). This allows Arend's proofs to proceed in an uncomplicated (though sometimes verbose) fashion.

In a further deviation from existing applications, Arend presents its proofs graphically, rather than textually. Proofs are constructed by interacting with a visual representation of the proof tree. The user selects the element of the proof on which they wish to act and the system presents the results of their action by updating the tree.

Arend's underlying logic is *constructive*, the intuitionistic logic of Brouwer [Brouwer, 1907], Heyting [Heyting, 1966], and Gentzen [Gentzen, 1935]. (Arend is named for Arend Heyting, one of the fathers of intuitionism.) A logic based on intuitionism has several advantages over one based on classical logic:

L. E. J. Brouwer. On the foundations of mathematics. *1975) LEJ Brouwer: Collected Works*, 1:11–101, 1907

A. Heyting. *Intuitionism*, volume 41. Elsevier, 1966

G. Gentzen. Untersuchungen über das logische schließen. i. *Mathematische zeitschrift*, 39(1):176–210, 1935

- Intuitionism emphasizes *evidence*: to prove a proposition P true is to give evidence for its truth. To prove a proposition false is to prove that it implies a contradiction. In particular, proving $\forall x, P(x)$ implies that we have a method, an algorithm, for producing a proof of P, given any x. Similarly, $\exists x, P(x)$ implies that we have a method for finding at least one x such that $P(x)$. This grounding in the concrete, in forcing every proof to answer the question of *how* it is true, we hope will remove some of the abstractness of formal proofs that is often problematic for beginning students.

Many a student has stared at the Law of Noncontradiction, $P \vee \neg P$ and wondered, "Which one is it?" In intuitionistic logic, $P \vee \neg P$ is *not* axiomatic, but rather a statement about the proposition P: the problem P is *solved*, it is either true or false, and *we know which one*.

- An intuitionistic proof is inherently *hierarchical*. A proof is a tree of sub-proofs. This presentation of proofs, as instances of a familiar data structure, is emphasized by Arend's visual presentation of proof-trees, as the traditional paragraph-proof notation tends to obscure the fact that a proof is composed of subproofs.

- Because proofs are hierarchical, their construction is very similar to that of other kinds of programming. A proof is iteratively broken down into smaller parts, lemmas can be used, like functions, to abstract out commonly-used subproofs, or to break down proofs that would otherwise be too large and unwieldy to understand.
- The lack of traditional negation eliminates some of the more-confusing elements of traditional logic programming. In particular, Arend has nothing corresponding to "negation as failure"; it is not possible in Arend to know, in a computational sense, when a proposition P fails. Thus, reasoning in Arend is entirely about things that are known to be true, never about things that are assumed to be false.

Visually, Arend presents proofs as trees of subproofs. A proof in this presentation has the form

$$\frac{P_1 \quad P_2 \quad \dots \quad P_n}{A_1, A_2, \dots, A_m \vdash G}$$

where the P_i are (possibly trivial) subproofs. The user navigates the proof process by *elaboration*. An incomplete proof is presented as

$$\frac{?}{A_1, A_2, \dots, A_m \vdash G}$$

Selecting one of the A_i or the goal G is generally sufficient to identify the action to be performed.[2] The action specifies what subproofs will be replace the ?, and how their A'_i and G' will differ from those of the root of the proof. Thus, each user action "grows" the proof upward and outward, until the axiomatic leaves are reached. Note that it is not possible in Arend to construct an invalid proof; the only actions which can be taken are those which make sense in a given proof state, and the resulting proof state is always consistent.[3]

[2] The exception is when G has the form $G_1 \vee G_2$; in this case, a particular branch of the disjunction must be specified.

[3] It is not hard to imagine an alternate mode of interaction, in which the user constructs the subproofs, rather than having them presented by the system. In such a mode the system would function as a pure proof checker, validating proofs only when marked as complete.

We believe that this emphasis on *concrete* proofs, proofs which present evidence, and on *direct interaction* will bring formal proofs out of the unfortunate corner to which they are often relegated, and enable students to see that formal methods can be both useful and enjoyable.

Background

Related work

AREND IS A *proof assistant*, a software tool to aid in the application of formal methods. We can partition this range of software tools into three not-necessarily disjoint sets:

- *Automated theorem provers* aim to automate as much of the work of proving a given theorem statement as possible. The hope is that the system will, if sufficiently "smart", be able to prove any theorem which a human with near-infinite patience could prove. Proofs produced by automated theorem provers are often quite lengthy, some stretching into the thousands-of-pages. Automated theorem provers must often employ sophisticated heuristics in their proof search algorithms, to avoid the exponential run times that would otherwise be required.

- *Model checkers* are concerned the automatic checking of a given model against a formal specification. For example, given a state machine as a model of a computational system, we may wish to verify that the system never enters some particular set of error states. Since model checking is only tangentially related to our subject, we will not discuss it further.

- *Proof assistants* aim to aid in the development and checking of formal proofs. While some systems only check proofs for completeness, more modern systems will typically also aid the user in the development of a proof. Recent examples include Coq [The Coq development team, 2004], Twelf [Pfenning and Schürmann, 1999], and Abella [Gacek, 2008]. Several element of Arend's design were influenced by Abella, most notably the use of a constructive two-level logic for describing and reasoning about systems. For a full history of the development of computer-assisted proof systems, see Geuvers [2009].

- More recently, several systems have emerged which try to bridge the gap between traditional functional programming and theorem proving. Systems such as Agda [Norell, 2007], Idris [Brady, 2011], and Beluga [Pientka and Dunfield, 2010] are based on the Curry-Howard isomorphism (see below) and thus represent propositions as types, and proofs as functional programs. These systems offer varying degrees of support for "real-world" usage: interfaces to system libraries, optional non-termination (logical soundness traditionally requires that all programs terminate), and friendly syntax. At the same time, their type systems (typically dependently-typed) are sufficiently powerful to represent interesting propositions about the systems being implemented.

The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. URL `http://coq.inria.fr`. Version 8.0

F. Pfenning and C. Schürmann. System description: Twelf—a meta-logical framework for deductive systems. In *Automated Deduction—CADE-16*, pages 202–206. Springer, 1999

A. Gacek. The Abella interactive theorem prover (system description). In A. Armando, P. Baumgartner, and G. Dowek, editors, *Proceedings of IJCAR 2008*, volume 5195 of *Lecture Notes in Artificial Intelligence*, pages 154–161. Springer, August 2008

H. Geuvers. Proof assistants: History, ideas and future. *Sadhana*, 34(1):3–25, 2009

U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007

E. C. Brady. Idris—: systems programming meets full dependent types. In *Proceedings of the 5th ACM workshop on Programming languages meets program verification*, pages 43–54. ACM, 2011

B. Pientka and J. Dunfield. Beluga: A framework for programming and reasoning with deductive systems (system description). In *Automated Reasoning*, pages 15–21. Springer, 2010

Many proof assistants are based on the *Curry-Howard isomorphism* which roughly states that *types* are *propositions* and *programs* are *proofs* of their types. That is, saying that a term t has type P is the same as saying that t is a proof of P. Under this view, proof checking is the same as type checking. Arend is not directly based on the Curry-Howard isomorphism; although its proofs are terms, it does not check them by computing their types, since terms in Arend are untyped. Nonetheless, we believe that it should be possible to derive a consistent type system from Arend's reasoning logic, which would support this interpretation.

Proof assistants that present a graphical interface are rare. While graphical "wrappers" exist for some systems (for example, ProofWeb[4] for Coq), these

[4] `http://prover.cs.ru.nl/login.php`

were not a part of the design of their underlying systems. The only proof assistant of which we are aware which makes graphical interaction as much an integral feature as Arend is HLM [Reichelt, 2010]. Like Arend, HLM encourages direct manipulation of in-progress proofs and is explicitly designed with that hope that it will make building proofs "fun", but unlike Arend, it is based on classical, rather than constructive logic, and has a more mathematical, and less pedagogical, aim.

S. Reichelt. Treating sets as types in a proof assistant for ordinary mathematics. Institute of Informatics, University of Warsaw, Warszawa, Poland, 2010. URL `http://hlm.sourceforge.net/types.pdf`

Use of proof assistants within education has been largely limited to the graduate sphere. Nonetheless, some pertinent work on their use at the undergraduate level does exist:

- Kadoda et al. [1999] present the results of a survey on the desirable user-interface features in an educational proof assistant. One surprising result they describe is a positive correlation between both the interactivity of a system, and its ability to work with explicit proof trees, and the "error-proneness" of a system. This strikes us as surprising, as Arend is both interactive and works with explicit proof trees, and yet has virtually no error-proneness during the proof construction process, as users are limited strictly to those actions which will progress the proof in a valid way. However, several features which Kadoda describes as being "highly" desirable are present in Arend: its support for visualizing proof structure, its consistency, and in the degree of "assistance" it gives to the user.

G. Kadoda, R. Stone, and D. Diaper. Desirable features of educational theorem provers–a cognitive dimensions viewpoint. In *Collected Papers of the 11th Annual Workshop of the Psychology of Programming Interest Group (PPIG-11)*, pages 18–23, 1999

It should be noted this author did not discover Kadoda's work until after Arend's design, with its emphasis on these features, was largely finalized.

- Suppes [1981] offers an interesting perspective, that of an instructor of mathematics, not computer science, at Stanford University, and furthermore, one who had, at the time of this publication, been using interactive theorem proving in undergraduate coursework for almost twenty years. Suppes raises several points of concern which are still relevant, most notably the conflict between providing a system which is "intelligent" and, hence, easy to use, and one which meets the pedagogical goals of teaching *introductory* material (the sort of material which a more intelligent proof assistant would tend to elide). Thus, although Arend is actually capable of automatically proving a reasonable large set of propositions, during proof construction it intentionally refrains from doing so, thus forcing the user to carry out — and, hopefully, to learn — the elementary steps of proof construction.

P. Suppes. Future educational uses of interactive theorem proving. In *University-level Computer-assisted Instruction at Stanford: 1968-1980*, pages 399–430. Stanford University, Instute for Mathematical Studies in the Social Sciences; Stanford, CA, 1981

Inference rules and natural deduction

Arend presents its specifications and proofs as *inference rules* and *derivations*, respectively. Since this presentation is central to the system, we review its basic elements here, by incrementally developing a simple zeroth-order intuitionistic logic with syntax-directed rules.[5]

[5] *Syntax-directed* implies that in each rule, everything above the line is a subterm of something below the line. This implies that new terms are not magically created out of nothing, so that proof search may proceed in a straightforward bottom-up order.

An *inference rule* is written

$$\text{Rule-Name}\,\frac{P_1 \quad P_1 \quad \ldots \quad P_n}{Q}$$

and should be read as "Q is true when P_1, P_2, etc. are." We call Q the *conclusion* of the rule and the P_i its *premises*. For logical conjunction and disjunction, we have the rules

$$\text{True}\,\frac{}{\top}$$

$$\text{And}\,\frac{P_1 \quad P_2}{P_1 \wedge P_2} \qquad \text{Or-1}\,\frac{P_1}{P_1 \vee P_2} \qquad \text{Or-2}\,\frac{P_2}{P_1 \vee P_2}$$

Note: the "elimination rules" for $\wedge$,

$$\frac{P \wedge Q}{P} \qquad \frac{P \wedge Q}{Q}$$

are not syntax directed but appear in an alternate form below.

Already from these we can construct some simple proofs-as-derivations:

$$\text{And}\,\frac{\text{True}\,\frac{}{\top} \qquad \text{Or-R}\,\frac{\text{True}\,\frac{}{\top}}{(\bot \vee \top)}}{\top \wedge (\bot \vee \top)}$$

If we wish to add implication to our system we run into a complication: we want $A \rightarrow B$ to express "B is true *given* A, or *assuming* A." That is, a valid derivation for B is now allowed to terminate, not just with the axiomatic rules of the system, but also with

$$\text{Assume-A}\,\frac{}{A}$$

However, this assumption only applies within the "scope" of the subproof of $A \rightarrow B$.

In order to express this, we introduce the *hypothetical judgments* using the symbol $\vdash$. $A_1, A_2, \ldots A_n \vdash C$ should be read "C is true *assuming* A_1, etc." We call the A_i the *antecedents* of the judgment, and C the *consequent*. (Although not strictly correct, we will sometimes refer to the antecedents and consequent of a *rule*, since a rule can have at most one hypothetical judgment as its conclusion.) [6] We use Γ to signify any collection of P_i. To enable the use of $\vdash$ in our derivations we add the following axiom, known as the *assumption rule*:

[6] Note that $\vdash$ is *not* a logical connective; in particular, it is not valid to say, e.g., $A \wedge (B \vdash C)$.

$$\text{Assume}\,\frac{}{A_1, A_2, \ldots, A_n \vdash A_{i, 1 \le i \le n}}$$

That is, "A_i is true *assuming* A_i".

Using $\vdash$ we can now formulate the rule for $\rightarrow$:

$$\rightarrow\,\frac{\Gamma, P \vdash Q}{\Gamma \vdash P \rightarrow Q}$$

This states that $P \rightarrow Q$ is true if assuming P allows us to prove Q.

We also have to reformulate the existing rules to specify how Γ is carried during a derivation:

$$\text{Assume}\,\frac{}{A_1, A_2, \ldots, A_n \vdash A_{i, 1 \le i \le n}} \qquad \text{True}\,\frac{}{\Gamma \vdash \top}$$

$$\text{And}\,\frac{\Gamma \vdash P_1 \quad \Gamma \vdash P_2}{\Gamma \vdash P_1 \wedge P_2}$$

$$\text{Or-1}\,\frac{\Gamma \vdash P_1}{\Gamma \vdash P_1 \vee P_2} \qquad \text{Or-2}\,\frac{\Gamma \vdash P_1}{\Gamma \vdash P_1 \vee P_2}$$

$$\rightarrow\frac{\Gamma, P \vdash Q}{\Gamma \vdash P \rightarrow Q}$$

However, the introduction of $\vdash$ and Γ, while solving the problem of implication, introduces another difficulty. Consider the proof of $(P \wedge Q) \rightarrow P$, which we would expect to be true.

$$\rightarrow\frac{P \wedge Q \vdash P}{(P \wedge Q) \rightarrow P}$$

We cannot complete the proof, because the assumption rule only allows us to conclude $P \wedge Q$. Despite the fact that we are assuming "*P and Q*" to be true, and, according to the rules of our system, this can only be the case if P is true, and Q is true, independently, we have no way of "extracting" the truth of P from the known truth of the conjunction.

In fact, we now need another set of rules telling us what operations are permitted on logical connectives on the LEFT side of the $\vdash$. All the rules we have presented so far have been RIGHT rules, in that they described the valid operations on the *conclusion* of the $\vdash$. Furthermore, we require that the LEFT and RIGHT rules of our system (indeed, any reasonable system) be consistent; it should always be the case that $P \vdash P$, given P (for any valid proposition P), we can prove exactly P.

It is perfectly acceptable to prove, given P, *less* than P. For example, in our system we have $P \vdash \top$. $\top$ carries less information than any particular P, precisely because it is provable in any context.

There is no True-L rule. $\quad \text{False-L}\frac{}{\Gamma, \bot \vdash C}$ $\quad$ There is no False-R rule.

$$\text{And-L}\frac{\Gamma, P, Q \vdash C}{\Gamma, P \wedge Q \vdash C} \qquad \text{Or-L}\frac{\Gamma, P \vdash C \quad \Gamma, Q \vdash C}{\Gamma, P \vee Q \vdash C}$$

(We omit the $\rightarrow$ Left rule as it is somewhat unintuitive and is not used in our system.) The somewhat-surprising FALSE-L rule corresponds to the logical principle of *ex falso quodlibet*; i.e., if false is assumed to be true, then any proposition can be proved.

In comparison to $\top$ being the minimal element in our system, the element which conveys the *least* information, $\bot$ is the maximal element, because from it we can prove anything. Thus it can be thought of as "containing" all possible proofs, even those that are contradictory!

With both LEFT and RIGHT rules we can now present proofs of simple logical tautologies, beginning with the formerly-unprovable $(P \wedge Q) \rightarrow P$:

$$\rightarrow\text{R}\frac{\wedge\text{-L}\dfrac{\dfrac{}{P, Q \vdash P}}{P \wedge Q \vdash P}}{(P \wedge Q) \rightarrow P}$$

(This exposition of natural deduction is by necessity quite simplified. For an exposition of the modern form of natural deduction, see Gentzen [1964]. For a more rigorous presentation of the logical connectives in natural deduction, see Martin-Löf [1996].)

G. Gentzen. Investigations into logical deduction. *American philosophical quarterly*, pages 288–306, 1964

P. Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. *Nordic journal of philosophical logic*, 1(1):11–60, 1996

Arend System Description

Specifications

Arend presents the rules of the underlying system as a collection of *inference rules*. For example, figure 1 presents the rules defining the set of natural numbers, together with natural number addition and the less-than relation.

Figure 1: Inference rules defining natural numbers, addition, and $<$

$$\text{Nat-}z\,\frac{}{\mathsf{nat}(z)} \qquad \text{Nat-Succ}\,\frac{\mathsf{nat}(X)}{\mathsf{nat}(s(X))}$$

$$\text{Add-}z\,\frac{}{\mathsf{add}(z, X, X)} \qquad \text{Add-Succ}\,\frac{\mathsf{add}(X, Y, Z)}{\mathsf{add}(s(X), Y, s(Z))}$$

$$>\text{-}z\,\frac{}{s(X) > z} \qquad >\text{-Succ}\,\frac{X > Y}{s(X) > s(Y)}$$

(This specification is a fragment of the ℕ specification included with Arend, described in appendix II.)

Observant readers may note that this definition of the set ℕ is missing the traditional third clause required in inductive definitions: the statement that "nothing else" is an element of the set in question. This omission is intentional, because the *closure property* of the rules is implicit in the logic of our system.

In the syntax of the specification language, the above can be written as

```
"Nat-0":    nat(0).
"Nat-Succ": nat(s(X)) :- nat(X).

"Add-0":    add(0,X,X).
"Add-Succ": add(s(X),Y,s(Z)) :- add(X,Y,Z).

"<-0":    0    < s(_).
"<-Succ": s(X) < s(Y) :- X < Y.
```

Under this system, we can ask for a *derivation* (i.e., a proof) that, for example, $\mathsf{add}(s(z), s(z), s(s(z)))$:

$$\text{Add-Succ}\,\dfrac{\text{Add-}z\,\dfrac{}{\mathsf{add}(z, s(z), s(z))}}{\mathsf{add}(s(z), s(z), s(s(z)))}$$

Proofs-as-derivations proceed upward, from the statement to the use of axiomatic rules at the "leaves" of the proof tree.[7]

[7] A rule is an axiom if it has no premises; nothing above the line.

The specification logic of Arend is based on a limited subset of an intuitionistic first-order logic. The full syntax of specifications is given in Appendix I. Note that while the syntax is superficially that of traditional Prolog, several extensions and restrictions are applied:

- The body of any clause must consist of a conjunction of atomic goals. Nested disjunctions, the "cut" operator, etc. are all forbidden.
- Rules (definitions) may have *names*, which are displayed in the user interface.
- *Render declarations* describe how to convert goals into HTML for rendering in the user interface. This is used to translate the internal representation

of goals as terms into a more mathematically-familiar form. (Eventually, one of the LaTeX-to-HTML translators may be used to allow more succinct mathematical renderings.)

- New user-defined infix operators may be created, simply by using a token whose text consists entirely of symbolic characters. Infix operators that would otherwise be parsed as ordinary atoms may be declared infix, via an `infix` declaration.

The restriction that all clauses of a rule consist solely of a conjunction is present for two reasons: one, it allows the specification logic to have very simple execution semantics, and two, it allows the specification logic to be easily embedded in the reasoning logic.

Specification execution

It is important that Arend specification language be not just logically consistent, but also *executable*; users can enter queries against a specification and receive answers, in the form of a unifier (as in traditional Prolog) coupled with a *derivation*. The derivation illustrates the tree of rule-applications that must be followed in order to produce the associated unifier. Note that derivations produced by queries are a restricted form of the derivations constructed as proofs. Thus, queries against the specification can serve as an introduction to the creation of simple proofs.

Since Arend's specification logic is a restricted form of the Horn-clause classical logic used in traditional Prolog, a resolution-style proof search procedure is sufficient to execute queries against a specification. Note that in keeping with its intuitionistic foundation, Arend has no negation operator. (Intuitionistically, $\neg P$ can be defined as $P \to \bot$, but Arend's specification logic lacks the rightward $\to$ implication operator. This operator *is* present in the reasoning logic, which thus requires a more sophisticated handling.)

Reasoning logic

AREND'S REASONING LOGIC is closer to a full expression of first-order intuitionistic logic, with extensions to support proofs by single-induction. The inference rules defining the reasoning logic are presented in Figure 2. It supports rightward implication ($\to$), with the restriction that the left-hand side cannot contain nested implications, only conjunctions, disjunctions, and atomic goals.[8] This implies that antecedents cannot contain implications, since they cannot be added by $\to_R$, and are not allowed in definitions in the specification logic.

[8] This restriction subsumes the *stratification* restriction in Abella; stratification in some form is required to ensure monotonicity of the logic.

Figure 2: Semantics of the reasoning logic

$$\text{Assume}\,\frac{}{P_1,\ldots,P_n \vdash P_i}$$

$$\top_R\,\frac{}{\Gamma \vdash \top} \qquad \bot_L\,\frac{}{\Gamma, \bot \vdash P}$$

$$\wedge_R\,\frac{\Gamma \vdash P \qquad \Gamma \vdash Q}{\Gamma \vdash P \wedge Q} \qquad \wedge_L\,\frac{\Gamma, P, Q \vdash G}{\Gamma, P \wedge Q \vdash G}$$

$$\vee_{R_1}\,\frac{\Gamma \vdash P}{\Gamma \vdash P \vee Q} \qquad \vee_{R_2}\,\frac{\Gamma \vdash Q}{\Gamma \vdash P \vee Q}$$

$$\vee_L\,\frac{\Gamma, P \vdash G \qquad \Gamma, Q \vdash G}{\Gamma, P \vee Q \vdash G}$$

$$\to_R\,\frac{\Gamma, P \vdash Q}{\Gamma \vdash P \to Q}$$

(There is no $\to_L$ rule, because implications are not allowed in antecedents.)

$$\forall_R\,\frac{\Gamma \vdash P(y)}{\Gamma \vdash \forall x : P(x)} \qquad \forall_L\,\frac{\Gamma, \forall x : P(x), P(t) \vdash G}{\Gamma, \forall x : P(x) \vdash G}$$

$$\exists_R\,\frac{\Gamma \vdash P(t)}{\Gamma \vdash \exists x : P(x)} \qquad \exists_L\,\frac{\Gamma, P(y) \vdash G}{\Gamma, \exists x : P(x) \vdash G}$$

$$=_R\,\frac{}{\Gamma \vdash t = t} \qquad =_L\,\frac{\Gamma_{[t_1=t_2]} \vdash G_{[t_1=t_2]}}{\Gamma, t_1 = t_2 \vdash G}$$

$$\text{Atomic-Right}\,\frac{\begin{array}{c} G \equiv C_1 \vee C_2 \vee \ldots \vee C_n \\ \Gamma \vdash C_1 \vee C_2 \vee \ldots \vee C_n \end{array}}{\Gamma \vdash G}$$

$$\text{Atomic-Left}\,\frac{\begin{array}{c} A \equiv C_1 \vee C_2 \vee \ldots \vee C_n \\ \Gamma, C_1 \vee C_2 \vee \ldots \vee C_n \vdash G \end{array}}{\Gamma, A \vdash G}$$

Notes:

- In $\forall_R$ and $\exists_L$, the variable y must be "fresh" in Γ.
- In $\forall_L$, $=_R$, and $\exists_R$, t is a term, which may use variables from Γ.
- In $=_L$, the notation $P_{[x=y]}$ indicates that the unification $x = y$ should be performed and the resulting substitution applied to P. Similarly, $\Gamma_{[x=y]}$ indicates that the substitution is to be applied to every goal in Γ.
- In Atomic-Left and -Right, each clause will contain the unification(s) necessary to unify its actual head with the atomic goal.

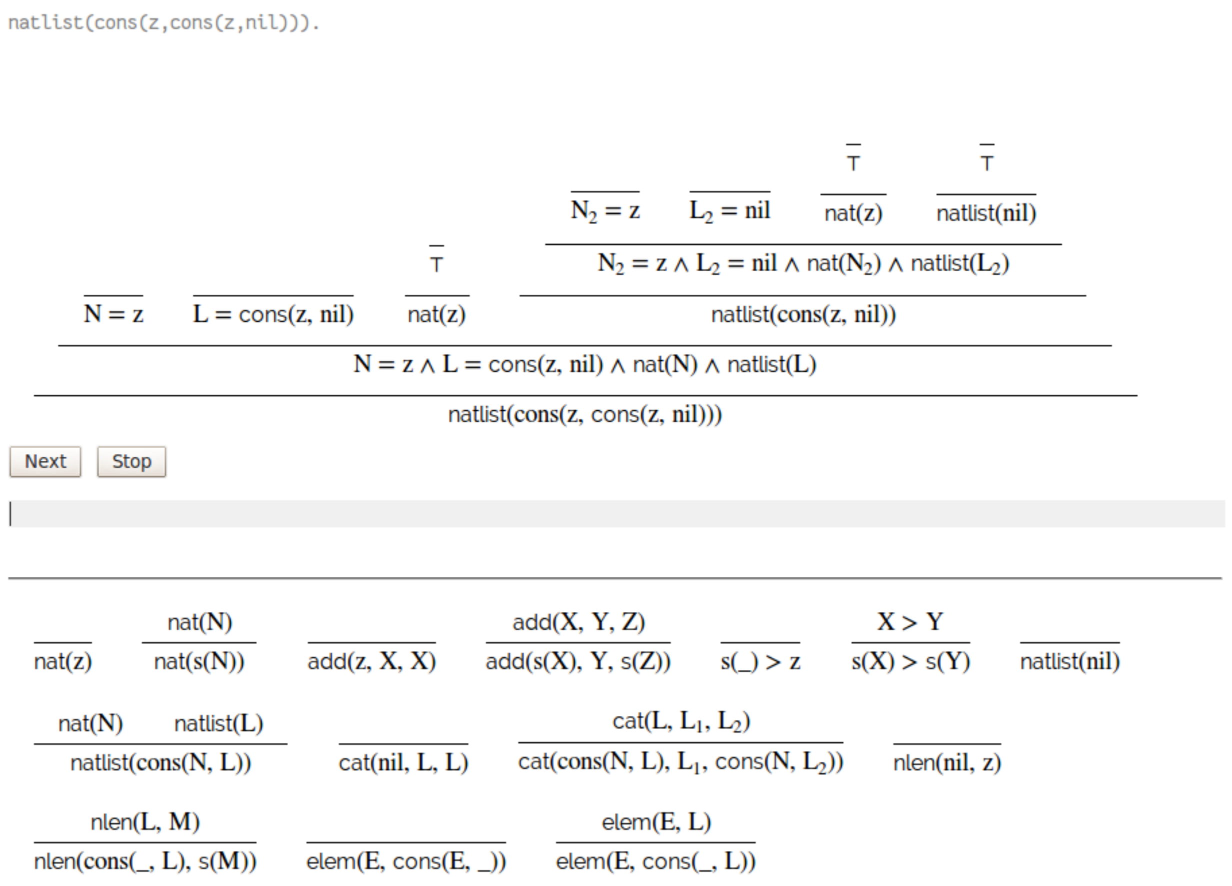
User Interface

Arend has two web-based user interfaces that support different kinds of interaction with the system:

- The *run-eval-print loop* (figure 3) allows the user to run queries against a specification, view the results (substitution and derivation), and view the complete set of rules from the current specification. In figure 3 the user has issued the query `natlist(cons(z,cons(z,nil))).` and is viewing the resulting derivation (because this query has no free variables, no substitution is displayed). The rules of the specification (the $\mathbb{N}$ specification, given in full in appendix II) are displayed below the input line.

- The interactive *proof assistant* allows the user to construct proofs for given propositions, in the context of a particular specification. Figure 4 illustrates the proof assistant interface: the rules of the current specification are displayed in the pane on the left; since the current proof is inductive, the inductive hypothesis is included. (Lemmas, once proved, are also displayed as rules.)

  The right pane displays the current proof statement, and the proof, here, in progress. The user can double-click on any antecedent, or any goal, to perform the default elaboration for that element. For example, double-clicking an antecedent $P \wedge Q$ would apply the $\wedge$-L rule, producing a subproof with antecedents P, Q. For situations in which there is more than one possible action (e.g., the $\vee$-R rules, or when backchaining against the IH), keyboard interaction is used to select the appropriate action.



```
natlist(cons(z,cons(z,nil))).
```

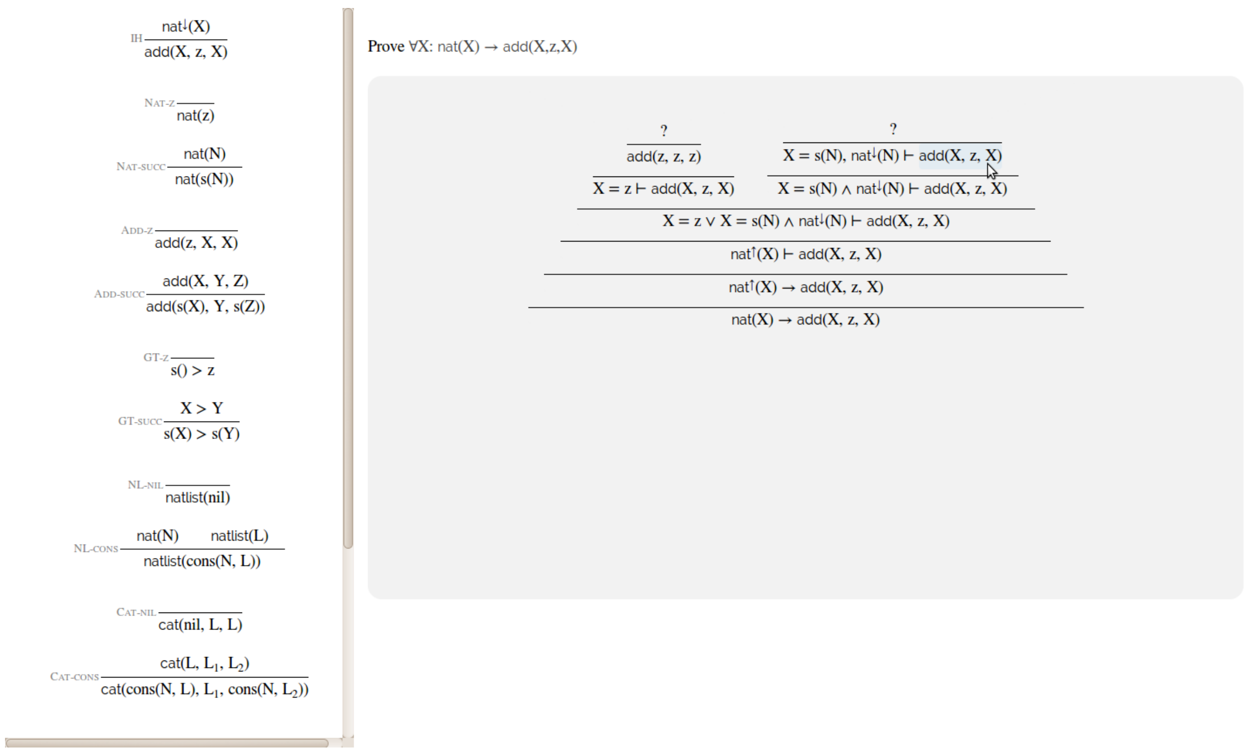
$$\frac{\dfrac{\overline{N = z} \quad \overline{L = \mathrm{cons}(z, nil)} \quad \dfrac{\overline{\top}}{\mathrm{nat}(z)} \quad \dfrac{\dfrac{\overline{N_2 = z} \quad \overline{L_2 = nil} \quad \dfrac{\overline{\top}}{\mathrm{nat}(z)} \quad \dfrac{\overline{\top}}{\mathrm{natlist}(nil)}}{N_2 = z \wedge L_2 = nil \wedge \mathrm{nat}(N_2) \wedge \mathrm{natlist}(L_2)}}{\mathrm{natlist}(\mathrm{cons}(z, nil))}}{N = z \wedge L = \mathrm{cons}(z, nil) \wedge \mathrm{nat}(N) \wedge \mathrm{natlist}(L)}}{\mathrm{natlist}(\mathrm{cons}(z, \mathrm{cons}(z, nil)))}$$

$$\frac{}{\mathrm{nat}(z)} \qquad \frac{\mathrm{nat}(N)}{\mathrm{nat}(s(N))} \qquad \frac{}{\mathrm{add}(z, X, X)} \qquad \frac{\mathrm{add}(X, Y, Z)}{\mathrm{add}(s(X), Y, s(Z))} \qquad \frac{}{s(\_) > z} \qquad \frac{X > Y}{s(X) > s(Y)} \qquad \frac{}{\mathrm{natlist}(nil)}$$

$$\frac{\mathrm{nat}(N) \quad \mathrm{natlist}(L)}{\mathrm{natlist}(\mathrm{cons}(N, L))} \qquad \frac{}{\mathrm{cat}(nil, L, L)} \qquad \frac{\mathrm{cat}(L, L_1, L_2)}{\mathrm{cat}(\mathrm{cons}(N, L), L_1, \mathrm{cons}(N, L_2))} \qquad \frac{}{\mathrm{nlen}(nil, z)}$$

$$\frac{\mathrm{nlen}(L, M)}{\mathrm{nlen}(\mathrm{cons}(\_, L), s(M))} \qquad \frac{}{\mathrm{elem}(E, \mathrm{cons}(E, \_))} \qquad \frac{\mathrm{elem}(E, L)}{\mathrm{elem}(E, \mathrm{cons}(\_, L))}$$

Figure 3: The browser-based run-eval-print-loop interface



$$\textsc{IH}\ \frac{\mathsf{nat}^{\downarrow}(X)}{\mathsf{add}(X, z, X)}$$

$$\textsc{Nat-z}\ \frac{}{\mathsf{nat}(z)}$$

$$\textsc{Nat-succ}\ \frac{\mathsf{nat}(N)}{\mathsf{nat}(s(N))}$$

$$\textsc{Add-z}\ \frac{}{\mathsf{add}(z, X, X)}$$

$$\textsc{Add-succ}\ \frac{\mathsf{add}(X, Y, Z)}{\mathsf{add}(s(X), Y, s(Z))}$$

$$\textsc{GT-z}\ \frac{}{s() > z}$$

$$\textsc{GT-succ}\ \frac{X > Y}{s(X) > s(Y)}$$

$$\textsc{NL-nil}\ \frac{}{\mathsf{natlist}(nil)}$$

$$\textsc{NL-cons}\ \frac{\mathsf{nat}(N) \quad \mathsf{natlist}(L)}{\mathsf{natlist}(cons(N, L))}$$

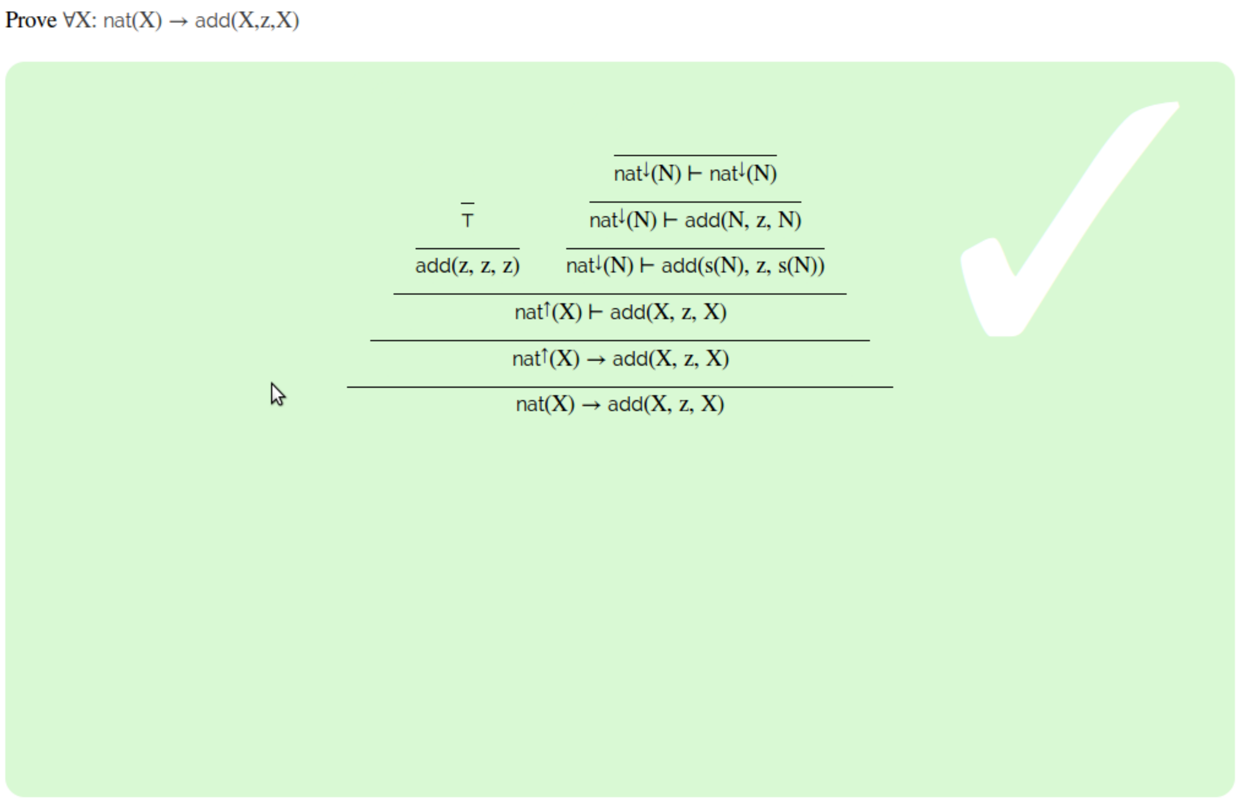
$$\textsc{Cat-nil}\ \frac{}{\mathsf{cat}(nil, L, L)}$$

$$\textsc{Cat-cons}\ \frac{\mathsf{cat}(L, L_1, L_2)}{\mathsf{cat}(cons(N, L), L_1, cons(N, L_2))}$$

Prove ∀X: nat(X) → add(X,z,X)

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{?}{\mathsf{add}(z, z, z)}}{X = z \vdash \mathsf{add}(X, z, X)} \qquad \dfrac{\dfrac{?}{X = s(N), \mathsf{nat}^{\downarrow}(N) \vdash \mathsf{add}(X, z, X)}}{X = s(N) \land \mathsf{nat}^{\downarrow}(N) \vdash \mathsf{add}(X, z, X)}}{X = z \lor X = s(N) \land \mathsf{nat}^{\downarrow}(N) \vdash \mathsf{add}(X, z, X)}}{\mathsf{nat}^{\uparrow}(X) \vdash \mathsf{add}(X, z, X)}}{\mathsf{nat}^{\uparrow}(X) \to \mathsf{add}(X, z, X)}}{\mathsf{nat}(X) \to \mathsf{add}(X, z, X)}$$

Figure 4: The proof assistant interface, with



Prove ∀X: nat(X) → add(X,z,X)

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\top}}{\mathsf{add}(z, z, z)} \qquad \dfrac{\dfrac{\dfrac{}{\mathsf{nat}^{\downarrow}(N) \vdash \mathsf{nat}^{\downarrow}(N)}}{\mathsf{nat}^{\downarrow}(N) \vdash \mathsf{add}(N, z, N)}}{\mathsf{nat}^{\downarrow}(N) \vdash \mathsf{add}(s(N), z, s(N))}}{\mathsf{nat}^{\uparrow}(X) \vdash \mathsf{add}(X, z, X)}}{\mathsf{nat}^{\uparrow}(X) \to \mathsf{add}(X, z, X)}}{\mathsf{nat}(X) \to \mathsf{add}(X, z, X)}$$

Figure 5: A completed proof

Implementation

Arend is implemented as a web-based system, with a server component, written in Prolog and running in the SWI-Prolog[9] and a browser-based frontend. Currently, the implementation of Arend consists of

[9] `http://swi-prolog.org`

- 1,401 lines of Prolog
- 6,198 lines of Javascript (of which 442 lines are test code)
- 493 lines of PEG grammar specification
- 501 lines of HTML
- 129 lines of CSS
- 41 source code files in total

The following libraries and applications are used in the development of Arend; these will be described in detail in the following section.

- Node.js[10]
- SWI-Prolog[11]
- Pengines[12]
- jQuery[13] — General utilities for Javascript in a browser environment
- Lodash[14] — Collection utilities for Javascript
- PEG.js[15] — Parsing Expression Grammar parser generator for Javascript.
- QUnit[16] — Javascript test framework
- JSCheck[17] — Randomized testing engine for Javascript.

[10] `https://nodejs.org/`

[11] `http://www.swi-prolog.org/`

[12] `http://pengines.swi-prolog.org/docs/index.html`

[13] `https://jquery.com/`

[14] `https://lodash.com/`

[15] `http://pegjs.org/`

[16] `https://qunitjs.com/`

[17] `http://www.jscheck.org/`

Arend's development is tracked using the Fossil[18] source control management system. As of April 18, 2015, the project history consisted of 294 commits spanning eight months of development. Arend, its source code and project history, can be found on the web at `http://fossil.twicetwo.com/arend.pl`.

[18] `http://fossil-scm.org`

Automated testing

Arend's Javascript code is run through a test suite consisting of 133 automated tests, however, of these, 12 central tests are randomized, running, by default, 20 randomly generated tests each. Thus, a total of 361 individual tests are run. The test suite can be run in the browser, or offline, via Node.js. Testing is handled via the QUnit[19] test framework; a small compatibility layer was written to allow QUnit tests to run offline. Randomized testing is provided by JSCheck[20] a "port" of the Haskell QuickCheck framework to Javascript. We have extended JSCheck with support for randomized generation of Arend data types: atoms, variables, and ground and non-ground terms up to a limited depth.

[19] `https://qunitjs.com/`

[20] `http://www.jscheck.org/`

QUnit is designed for automated testing in a browser environment. Arend's development, as far as is possible, tries to target both browser and offline environments; this is true even for Javascript code. Thus, we wrote "QNodeit", a small compatibility layer that allows the complete suite of QUnit tests to run offline, via Node.js. All non-user-interface tests run successfully in both the browser and offline environments. QNodeit also integrates JSCheck into QUnit, allowing JSCheck's randomized tests to be used naturally within QUnit.

Client-side implementation

Although Arend's client-side code only serves to provide a user-interface to the backend's proof-checking and manipulation engine, it contains a significant amount of logic itself. Early in Arend's development we felt that the best course would be to implement the entire proof checking engine in Javascript, thus allowing proof-handling with no server at all. Although this did not prove feasible, a significant amount of logic-handling code in Javascript was written, and, so far from being a redundancy, this has proved to be an asset. Arend's client is not "dumb", but in fact understands a great deal of the structure of the proofs it is presenting. This allows for richer user-interface possibilities, and more flexible coordination between front- and back-end.

The Lodash[21] Javascript library provides a set of general utilities, mostly aimed at manipulation of collections (arrays and objects) and enabling a functional style of programming; Lodash is used extensively throughout Arend. Lodash makes its facilities available as methods of a global _ object. Thus, filtering out the odd elements of an array in "stock" Lodash would take the form

[21] `https://lodash.com/`

```
_.filter([1,2,3,4], function(e) { return e % 2 == 0; });
```

We have created a wrapper library around Lodash called "Nodash" which integrates Lodash's utility methods into the global prototypes of the datatypes which they operate on. Thus, for example, the `filter` method, which can be applied to any "collection" – array, object, or string — would be installed on the global `Array.prototype`, `Object.prototype`, and `String.prototype` objects, so that it is accessible simply as

Historically, "monkey-patching" the global prototypes was considered highly unsafe, as new properties would be "enumerable" and would thus become visible in, for example, `for-in` loops over the properties of *any* object. However, all modern Javascript implementations support the `defineProperty` method, which allows the creation of *non-enumerate* properties on objects which do not have this problem. Nodash uses `defineProperty` to safely add Lodash's methods to the global prototypes.

```
[1,2,3,4].filter(function(e) { return e % 2 == 0; });
```

Note that in Nodash the global _ object is still available, for those methods which do not fit with any of the global prototypes.

The client-side implementation includes a complete parser for the specification language. This allows user input to be fully syntax-checked before being sent to the server, and allows for faster feedback to the user when syntax errors occur. The parser is written using PEG.js[22], a parsing expression grammar (PEG) [Ford, 2004] parser-generator. The specification grammar in PEG form consists of 28 non-terminals and 53 terminal tokens. Of note is the fact that the grammar uses the standard Unicode character classes in its definition

[22] `http://pegjs.org/`

B. Ford. Parsing expression grammars: A recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 111–122, New York, NY, USA, 2004. ACM. ISBN 1-58113-729-X. DOI: 10.1145/964001.964011. URL `http://doi.acm.org/10.1145/964001.964011` of "identifier" and "operator"; an operator, in particular, is defined as any sequence of characters from the symbolic or punctuation classes[23]. This allows for traditional mathematical operators such as $\in$ or $\leq$ to be used directly in specifications.

[23] Classes Sc, Sm, Sk, So, Pc, Pd, and Po.

Other modules of note in the client-side implementation include:

- `core` — contains a small amount of compatibility code that allows the other non-user-interface modules to operate transparently in either the browser or Node. In particular, it provides a browser implementation of the `require()` function in Node, used to load modules. In the browser, these modules must have already been loaded via standard `<script>` tags (i.e., dynamic loading is not provided), but, once loaded, they are installed into a global module repository; `require` then simply returns a reference to the appropriate module. This allows Javascript code to transparently access other modules, without knowing whether they are being dynamically loaded within Node, or have already been loaded in the browser.

- `terms` — contains a complete representation of *terms*, the fundamental datatype of logic programming. The `terms` module supports walking the structure of terms, converting Prolog-style term-lists to Javascript Arrays and vice versa, rendering terms to either specification or Prolog-compatible strings, and enumerating the variables in a term.

- `unify` — contains a fully-functional implementation of the Robinson unification algorithm [Robinson, 1965]. Although this module is exhaustively tested, it is currently minimally used. In the future, we hope to build a pattern-matching utility library on it, to allow for more natural examination and manipulation of term-structures on the client-side.

J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM (JACM)*, 12(1):23–41, 1965

- `term_render` — contains the rendering engine for converting terms, rules, and derivations to HTML structures. Note that derivation rendering is extensible via a number of "hook" functions, which allow client code to extend or replace the rendering of the various proof components: antecedents, consequents, disjunctive consequents, and consequents which could be the target of the inductive hypothesis.

Server-side implementation

The core of Arend's proof-checking engine is written in Prolog, and is made available to the client via a HTTP interface. SWI-Prolog provides both a standard HTTP server module, as well as a specialized "app engine" module called PENGINES, both of which are used in Arend. HTML, CSS, and Javascript files for the client-side interface are served via the standard HTTP server, while two PEngine "applications", `repl` and `passist` provide the interface to the proof-checker itself.

PEngines provides a transparent interface between Javascript code running in the browser, and Prolog code running on the server. With it, our client-side code can directly execute queries against the exported predicates of the two aforementioned applications. The results (success, failure, and substitution) of those queries can then be enumerated. Terms are encoded as JSON objects (our `terms` module can decode JSON terms into its own types).

It should be noted that the Prolog implementation of Arend is mostly portable, relying only on ISO-Prolog predicates, commonly-available libraries, and a few SWI-Prolog-specific extensions and modules (most notably the HTTP server module). It would not be difficult to port the proof-checker itself to another Prolog system.

The Prolog core proof-checker implementation consists of the following modules, the functionality of which will be described in detail in the following sections:

- `subst` — provides support for working with *explicit substitutions*. As described below, Arend cannot use Prolog's own substitution, as the substitutions that are applied to a proof object may differ in subtrees; Prolog applies a substitution globally. Thus, we re-implement both variables and substitutions for our own use.

- `program` — provides support for expanding atomic goals in the specification language, and for executing goals in the specification. The `run/3` predicate produces proof objects compatible with those produced by the full proof checker, but is an independent implementation.

- `checker` — the heart of the proof checking system, provides routines for dealing with proof objects, elaborating incomplete proofs, and checking and generating proof objects corresponding to particular statements.

Proof representation

Informally, the content of a derivation is relatively simple: a tree of -LEFT and/or -RIGHT rule applications, drawn from the rules of the reasoning logic in figure 2. In practice, a more detailed representation is required, one which, in particular, necessitates the use of *manual substitution*, rather than the implicit substitution that would result from naive use of (for example) the system unification in Prolog.

For derivations purely derived from the specification language, manual substitution is not required. This is because for any valid derivation of a proposition expressed in the specification language, the substitution is *consistent* throughout the derivation tree; it is not possible for different "branches" to have different substitutions. (Disjunction may, of course, result in the generation of multiple distinct derivations, each with its own unifying substitution, but within any single derivation there is always a single consistent substitution.)

However, because the reasoning language possesses rightward implication, a new element is introduced to derivations: the list of *antecedents*. Case analysis on a disjunction antecedent is superficially similar applying the $\wedge$-R rule, with an important distinction: the branches produced each have *independent* bindings. Consider, for example, case analysis on $\mathsf{nat}(X)$ in the course of a proof:

$$\frac{X = z, \mathsf{nat}(z) \vdash \ldots \qquad X = s(X'), \mathsf{nat}(X') \vdash \ldots}{\mathsf{nat}(X) \vdash \ldots}$$

In the left branch, the substitution is $X = z$; in the right, it is $X = s(x')$.

In order for an analogous situation to arise in the specification language, we would need to have a conclusion $X = z \wedge X = s(x')$, which is already inconsistent, and actually impossible to express, because Horn clauses forbid such conjunctions as conclusions. Conversely, given the conclusion $X = z \vee X = s(x')$ we have the choice of which unification to use, and thus there is no inconsistency.

Because of this, each branch of a derivation must maintain its own substitution, applied to its goals as they are expanded. Thus, Arend contains facilities for implementing its own notion of logical variables, unifying terms containing these variables (producing a substitution object), and applying substitution objects to arbitrary terms.

Proofs are represented as a term of the form

```
proof(Goal,Ctx,Subst,Proof)
```

where `Goal` is the goal to be proved, `Ctx` is the list of antecedents (initially empty for most goals), `Subst` is the substitution that results if the goal is proved, and `Proof` is a proof term for this particular goal.

The proof terms vary, depending on the structure of the goal and the contents of the context. Non-axiomatic proof terms include one or more subproofs. The possible proof terms are

- `induction(N,Proof)` — proves a $\forall$ inductively, on the N-th element of the context.
- `ih(Proof)` — Proves an atomic goal by backchaining it against the inductive hypothesis.
- `generic(Proof)` — Proves a $\forall$ generically (i.e., by substituting a unique constant for the quantified variable).
- `instan(V,Proof)` — Proves a $\exists$ by giving a value for the quantified variable.
- `product([Proofs...])` — Proves a conjunction by providing a subproof for each conjunct.
- `choice(N,Proof)` — Proves the N-th branch of a disjunction.
- `implies(Proof)` — Proves an implication, by adding the left-hand-side to the context as an assumption and then proving the right-hand-side.
- `ctxmem(N)` — Implements the "assumption rule"; i.e., proves an atomic goal by unifying it with the N-th element of the context.
- `backchain(Proof)` — Proves an atomic goal by backchaining it; i.e., by expanding it into its definition, the disjunction of its clauses.
- `unify(A,B)` — Proves a goal of the form `A = B` (this has the side-effect of adding the substitution produced by $A = B$ to the `Subst` for this proof subtree).

- `case(T,N,Keep,[Proofs...])` — Performs case-analysis on an element of the context. The type `T` corresponds to the various -Left rules of the reasoning logic.
- `trivial` — Proves $\top$.
- `hole` — Proves any goal, but represents an incomplete proof.

Capture avoidance

The use of manual substitution means that the proof-checker must also contend with another troublesome problem which Prolog conceals: avoiding *accidental capture* when expanding a call to a goal. Consider the goal $\mathsf{nat}(X)$. We would expect this goal to succeed twice, first with the solution $X = z$. However, if we naively unify this goal with the head of the Nat-Succ rule we get the solution $X = s(X)$ which fails the occurs check. The problem is that X as present in our goal is not the same variable as X in the Nat-Succ rule. Variables in rule definitions are *schematic*, their names should be irrelevant.

In order to enforce this, we rename variables in a clause so as to not conflict with any variables already in scope. (Note that the expansion of a clause may thus introduce new variables into the scope.) At the same time, we want to avoid obscurely-generated names such as `_G356` (a variable as renamed by SWI-Prolog), in order to avoid confusing the user. Thus, we adopt a numerical scheme for variable renaming: if a variable name X is already in scope, then we try the names $X_2, X_3, \ldots$. Under this scheme, the Nat-Succ rule succeeds with $X = s(X_2)$

Of course, X_2 is constrained to be $\mathsf{nat}(X_2)$, so the second real solution will be $X_2 = z, X = s(z)$, followed by an infinite sequence of successes, for all $X \in \mathbb{N}$.

Capture-avoiding substitution becomes even more complex in the reasoning language, where the quantifiers $\forall S$ and $\exists T$ act as *binders*, capturing the names S and T, respectively, within their bodies. Thus, to apply a substitution $X \mapsto 1$ to $\forall X, P(X)$ it is *not* correct to give $\forall 1, P(1)$. In addition, consider the problem of applying the substitution $X \mapsto Y$ to $\forall Y, P(X, Y)$; clearly, one of the Y's will need to be renamed. Since the Y in the substitution is most likely bound somewhere else in the derivation, we choose to rename the Y bound within the $\forall$.

Unification

The Robinson unification algorithm is presented, in natural deduction style, in figure 6. Note that the same algorithm is used, with minor modification, in both the Javascript and Prolog implementations (for a direct comparison of these two implementations, see our comments below).

(This presentation is based in part on that in Pfenning [2006].)

F. Pfenning. Logic programming. *Course notes*, 2006. URL `http://www.cs.cmu.edu/~fp/courses/lp/lectures/lp-all.pdf`

It is interesting to directly compare the two implementations of unification (one of the few modules which is semantically similar in both implementations). Both systems implement the same algorithm. The Javascript implementation consists of 136 lines of non-comment code and is quite difficult to

Figure 6: Rules for the Robinson unification algorithm

$$\text{Wildcard}\ \frac{}{\_ \equiv t \mid \varepsilon}$$

$$\text{Var-Refl}\ \frac{}{X \equiv X \mid \varepsilon} \qquad \text{Var-Atom}\ \frac{}{X \equiv a \mid \{x = a\}}$$

$$\text{Var-Term}\ \frac{X \notin \hat{t}}{X \equiv f(\hat{t}) \mid \{X = f(\hat{t})\}} \qquad \text{Var-Swap}\ \frac{X \equiv t \mid \theta}{t \equiv X \mid \theta}$$

$$\text{Term-Term}\ \frac{\hat{t} \equiv \hat{s} \mid \theta}{f(\hat{t}) \equiv f(\hat{s}) \mid \theta}$$

$$\text{Nil}\ \frac{}{[] \equiv [] \mid \varepsilon} \qquad \text{Cons}\ \frac{t \equiv s \mid \theta_1 \qquad \hat{t}\theta_1 \equiv \hat{s}\theta_1 \mid \theta_2}{(t : \hat{t}) \equiv (s : \hat{s}) \mid \theta_2}$$

follow; the Prolog implementation requires only 69 lines, an almost 50% reduction! This despite the fact that the Prolog implementation does *not* use the built-in unification to simplify the algorithm at all; it would be largely the same if only traditional assignment were used.

Proof Checking

The proof checker is implemented in approximately 450 lines of Prolog code. It essentially implements the rules of the reasoning logic presented in figure 2, by recursively checking the proof tree. That is, it first checks the root node to ensure that the proof object is consistent with the conclusion (antecedents and consequent) by ensuring that it has the correct number and type of subproofs. The subproofs are then recursively checked, working through the tree until the axiomatic rules are reached (or, in the case of an incomplete proof, a `hole` is found, indicating an unproved subtree).

There are two complications to a straightforward top-down recursive implementation:

- When expanding an atomic goal (i.e., a call to a predicate in the specification) the expansion must be done in such a way that variables in the body of the expansion do not conflict with variables already in scope. In order to ensure this, we track the set of variables that are in scope as we proceed through the tree, and use it to rename variables in expansions before they are placed in the tree.

- Similarly, checking a unification requires applying the unification in a non-trivial way. For example, in $t_1 = t_2 \wedge P$ the substitution resulting from unifying t_1 and t_2 will be applied to P. This is consistent with Arend's (and, indeed, Prolog's) left-to-right evaluation order for conjunctions, but does mean that substitutions must be "accumulated" bottom-up while traversing the tree, and then applied when appropriate.

`check` is written in such a way that it can be called with the proof tree argument bound or unbound. In the former mode, it functions as a pure proof checker. If the proof is unbound, however, it functions as an automated proof assistant, attempting to construct a proof object for the given goal. In this mode it can prove any proposition in the pure specification logic, and even a few that lie outside it; these extensions include:

- Simple implications: for example, $P \to P$ can be proved.
- Proofs involving $\bot$-Left, for example $\bot \to P$.
- Proofs involving limited use of other -Left rules. For example, $P \wedge Q \to P$ can be proved.[24]

[24] $P \wedge Q \to P$ produces an infinite sequence of proofs, because the list of antecedents, while properly speaking a set, is implemented as a list without duplicate removal, for efficiency reasons. Thus it is always possible to case on $P \wedge Q$, keep the original antecedent, and thus generate an infinite collection of P's, each of which gives, theoretically, a distinct proof.

More interestingly, `check` can be called with the *goal* unbound and the proof bound, albeit for a very limited set of proofs. In this mode it will construct a proposition which is proved by the given proof. This behavior is not unexpected; by the Curry-Howard isomorphism proofs are to propositions as values are to types. Without a "type" given `check` functions as a limited *type inference* engine, finding a type (proposition) for a given value (proof).

Proof Elaboration

The heart of the incremental proof checking algorithm is the predicate `elaborate` (defined in file `checker.pl`). `elaborate` works in conjunction with `check`, the proof checker itself. The purpose of elaborate is to take an incomplete proof, together with a reference to one of its leaves (i.e., to a ?), and to "elaborate" it with respect to either its consequent, or one of its antecedents. As described above, the form of the selected element dictates the forms of the subproofs. `elaborate` "fills in" these subproofs, to a single level only (i.e., all the subproofs it constructs are themselves ?).

In a logic which did not include unification or the ability to call defined predicates (atomic goals), we could elaborate any node of the tree, independent of the rest of the tree. Unification and backchaining both introduce complications:

- Unification changes the substitution, which "flows" through the proof tree in a non-trivial way. Extending substitution may have far-reaching effects on variables in other parts of the tree.
- Backchaining a goal can only be done with the knowledge of what variables are already in scope, because it is necessary to rename the variables within the definition before expanding it, in order to avoid accidental capture. As with unification, the set of variables in scope is not trivial. Although Arend is already overly conservative in its determination of what variables are in scope — preferring to harmlessly rename variables rather than risking a naming collision — computing this set still requires more than traversing the path from the root to a leaf.

In particular, every conjunct in a conjunction has in scope all the variables in scope in every *other* conjunct as well as its own.

Fortunately, there is a simple solution to both these difficulties: the proof checker `check`:

- In the case of a unification, which has no subproofs, `elaborate` simply instantiates the proof to `unify(_,_)` and then runs `check` on the resulting tree. `check` will instantiate the arguments of the `unify` term with references to the actual variables, while at the same time propagating the new substitution to the rest of the tree.

- For an atomic goal expansion, `elaborate` will instantiate the proof to `backchain(proof(_,_,_,hole))` and then run `check`. The presence of the `backchain` tactic will force the proof checker to apply backchaining at this point in the proof. It will then instantiate the first three arguments to the subproof with the (correctly renamed) expansion of the goal, the correct context, and the correct substitution. It will likewise propagate the new set of in-scope variables to the rest of the tree. Finally, the presence of `hole` (i.e., ?) in place of the subproof will terminate the proof checker's recursion along this branch. (If we had left the fourth argument uninstantiated, the proof checker would continue to search for a complete subproof for this branch of the tree.)

The right and left rules for elaboration are given in figures 7 and 8. Proof state terms are written with the notation

$$\text{Premises} \vdash \text{Goal} \rightarrow \text{Proof}$$

This should be read as "Premises $\vdash$ Goal is proved by Proof." We use the notation $\Gamma \vdash G \rightarrow \ ? \implies$ Proof to signify that the the $?$ should be filled by Proof.

During interactive proof construction, elaboration is used to construct the subproofs after the user has selected the element of the current state on which they wish to act. For example, if the user selects the consequent, and it has the form $P \wedge Q$, then the output proof will be elaborated to

$$\Gamma \vdash P \wedge Q \rightarrow \text{product}([\Gamma \vdash P \rightarrow \text{hole}, \Gamma \vdash Q \rightarrow \text{hole}])$$

The subproofs of the product will have the correct consequents and antecedents, but their own proofs will be empty, ready for further elaboration.

Inductive reasoning

Arend's reasoning logic supports proofs of a universal quantification both generically and by *induction*. Arend's induction is technically on the height of derivations, although from the perspective of the user it supports full structural induction on terms; this subsumes the usual natural number induction.

Arend supports only single-induction proofs (i.e., induction on multiple antecedents is not allowed) and supports only induction global to a proof (i.e.,

Figure 7: Right rules for elaboration of proof states

$$\text{Trivial}\;\frac{}{\Gamma \vdash \top \to ? \qquad \Longrightarrow \qquad \text{trivial}}$$

$$\text{Assumption}\;\frac{\Gamma = \ldots, P_i, \ldots \qquad P_i = G}{\Gamma \vdash G \to ? \qquad \Longrightarrow \qquad \text{ctxmem}(i)}$$

$$\text{Backchain}\;\frac{G \text{ is atomic }, G \equiv C_1 \vee C_2 \vee \ldots}{\Gamma \vdash G \to \sigma\ ? \qquad \Longrightarrow \qquad \text{backchain}(\Gamma \vdash C_1 \vee C_2 \vee \ldots \to \sigma\ ?\ )}$$

$$\text{Product}\;\frac{}{\Gamma \vdash G_1 \wedge G_2 \wedge \ldots G_n \to ? \qquad \Longrightarrow \qquad \text{product}([\text{Proof}_1, \text{Proof}_2, \ldots, \text{Proof}_n])}$$

(where $\text{Proof}_i = \Gamma \vdash G_i \to ?$)

$$\text{Choice}\;\frac{}{\Gamma \vdash G_1 \vee G_2 \vee \ldots G_n \to ? \qquad \Longrightarrow \qquad \text{choice}(i, \Gamma \vdash G_i \to ?\ )}$$

$$\text{Implies}\;\frac{}{\Gamma \vdash P \to Q \to ? \qquad \Longrightarrow \qquad \text{implies}(\Gamma, P \vdash Q \to \sigma\ ?\ )}$$

$$\text{Unifies}\;\frac{}{\Gamma \vdash t_1 = t_2 \to ? \qquad \Longrightarrow \qquad \text{unifies}(t_1, t_2)}$$

(t_1 is unified with t_2 and the result is merged with the current substitution)

$$\text{Induction}\;\frac{\begin{array}{c}\text{IH} = (P_1 \wedge P_2 \wedge \ldots P_i^{\downarrow} \ldots \to G) \\ \text{IH is added to the specification}\end{array}}{\Gamma \vdash (P_1 \wedge P_2 \wedge \ldots \to G) \to ? \qquad \Longrightarrow \qquad \text{induction}(i, \text{IH}, \Gamma \vdash (P_1 \wedge P_2 \wedge \ldots P_i^{\uparrow} \to G) \to ?\ )}$$

(Induction will never be elaborated into an unbound proof; it can only be checked against an existing proof)

$$\text{IH}\;\frac{\text{IH} = (P_1 \wedge P_2 \wedge \ldots P_i^{\downarrow} \ldots \to G)}{\Gamma \vdash G \to ? \qquad \Longrightarrow \qquad \text{ih}(\Gamma \vdash (P_1 \wedge P_2 \wedge \ldots P_i^{\downarrow} \ldots \to G) \to ?\ )}$$

$$\text{Generic}\;\frac{c \text{ is a fresh constant}}{\Gamma \vdash \forall x : G \to ? \qquad \Longrightarrow \qquad \text{generic}(\Gamma \vdash G_{[x=c]} \to ?\ )}$$

$$\text{Instan}\;\frac{}{\Gamma \vdash \exists x : G \to ? \qquad \Longrightarrow \qquad \text{instan}(t, \Gamma \vdash G_{[x=t]} \to ?\ )}$$

Notes:

- For simplicity, The Atomic rule elides the details of unifying the goal G with the heads of the definitions and applying the resulting substitutions to the clauses C_i.

Figure 8: Left rules for elaboration of proof states

All left-rules instantiate the Proof to a term of the form

$$\text{case}(\text{Type}, N, \text{Keep}, [\text{Proofs}\ldots])$$

In the rules below we leave N (the index in the list of antecedents of the judgment to be cased upon) implicit and omit Keep (a Boolean flag indicating whether the targeted antecedent should remain in the list, or be removed), and write them as

$$\text{Type}(\text{Proofs}\ldots)$$

$$\text{False-Left}\ \frac{}{\Gamma, \bot \vdash G \to ? \quad \Longrightarrow \quad \text{false\_left}()}$$

$$\text{And-Left}\ \frac{}{\Gamma, P_1 \wedge P_2 \wedge \ldots \wedge P_n \vdash G \to ? \quad \Longrightarrow \quad \text{and\_left}(\Gamma, P_1, P_2, \ldots, P_n \vdash G \to ?\ )}$$

$$\text{Or-Left}\ \frac{}{\Gamma, P_1 \vee P_2 \vee \ldots \vee P_n \vdash G \to ? \quad \Longrightarrow \quad \text{or\_left}(\text{Proof}_1, \text{Proof}_2, \ldots, \text{Proof}_n)}$$

(where $\text{Proof}_i = \Gamma, P_i \vdash G \to ?\ )$

$$\forall\text{-Left}\ \frac{}{\Gamma, \forall x : P \vdash G \to ? \quad \Longrightarrow \quad \text{forall\_left}[t](\Gamma, P_{[x=t]} \vdash G \to ?\ )}$$

$$\exists\text{-Left}\ \frac{c \text{ is a fresh constant}}{\Gamma, \exists x : P \vdash G \to ? \quad \Longrightarrow \quad \text{exists\_left}(\Gamma, P_{[x=c]} \vdash G \to ?\ )}$$

$$\text{Unify-Left}\ \frac{}{\Gamma, t_1 = t_2 \vdash G \to ? \quad \Longrightarrow \quad \text{unify\_left}(\Gamma_{[t_1=t_2]} \vdash G_{[t_1=t_2]} \to ?\ )}$$

$$\text{Backchain-Left}\ \frac{P \text{ is atomic} \qquad P \equiv C_1 \vee C_2 \vee \ldots \vee C_n}{\Gamma, P \vdash G \to ? \quad \Longrightarrow \quad \text{backchain\_left}(\Gamma, C_1 \vee C_2 \vee \ldots \vee C_n \vdash G \to ?\ )}$$

nested inductions are not allowed, although they can be "faked" by using lemmas). These restrictions imply that the induction hypothesis can be regarded as being global to a proof, thus eliminating the need to restrict the scope of difference induction hypotheses to different branches of the proof tree.

Internally, induction is implemented by *goal annotation* [Gacek, 2009, sec. 5.2]. When an inductive proof is declared, a particular goal in the antecedents is selected, by the user, to be the target of the induction. This goal must be a user goal; it cannot be a built-in operator such as conjunction, disjunction, or unification. The functor of the goal is internally flagged as being "big" (indi-

A. Gacek. *A Framework for Specifying, Prototyping, and Reasoning about Computational Systems.* PhD thesis, University of Minnesota, September 2009

For example, in a proof of $\mathsf{nat}(X) \vdash \mathsf{add}(X, 0, X)$ we would induct on $\mathsf{nat}(X)$.

cated as ↑) and the induction hypothesis is defined in terms of the same goal, but flagged as "small" (↓). For example, to prove that $\mathsf{nat}(X) \to \mathsf{add}(X, 0, X)$ our proof would proceed by induction on $\mathsf{nat}(X)$, and thus we would have the inductive hypothesis $\mathsf{nat}^{\downarrow}(X) \to \mathsf{add}(X, 0, X)$.

When a ↑ goal is expanded by case analysis or backchaining, any sub-goals in its expansion are flagged as ↓, indicating that they are "smaller" than the original goal. The induction hypothesis can only be applied to goals which are "small", thus enforcing the inductive restriction.

Under this scheme, the induction hypothesis can be viewed as simply another clause in the program. Thus, the IH presented above is added as

```
"IH": add(X,0,X) :- nat↓(X).
```

with the (internal) restriction that the IH will never be used without explicit action on the part of the user. Lemmas, once proved, are represented similarly, as new program clauses, but requiring explicit use.

A somewhat unfamiliar aspect of Arend's use of induction is how the inductive hypothesis is applied: ordinarily we would specify those antecedents of the current proof state which matched the corresponding antecedents of the IH (i.e., "apply IH to $nat^{\downarrow}(X')$"); this would have the effect of introducing the IH, with the appropriate substitutions, as a new antecedent, where the assumption rule could then use it to prove the conclusion. Purely for user-interface reasons, we apply the IH by backchaining it against the current *conclusion*, replacing the goal with the conjunction of the antecedent of the IH (again, with the appropriate substitutions). The user can then use the ∧-R rule to split the proof into subproofs with atomic goals, each of which can be proved by the assumption rule.

The inductive hypothesis is used within a proof by backchaining on the current goal, which must unify with the head of the IH. Recall that backchaining replaces the goal with its definition(s). Ordinarily the disjunction of *all* definitions is used, but when the IH is specified, only the IH will be supplied; this prevents the user from having to "throw away" all the normal disjuncts which are unimportant in an inductive proof. As an example of an inductive proof, we present the aforementioned proof of $\mathsf{nat}(x) \to \mathsf{add}(x, 0, x)$:

$$\text{IH}\,\dfrac{\mathsf{nat}^{\downarrow}(x)}{\mathsf{add}(x, 0, x)}$$

$$\text{By induction}\,\dfrac{\to\text{R}\,\dfrac{\text{Case}\,\dfrac{\text{Add-0}\,\dfrac{}{\mathsf{add}(0,0,0)} \qquad \text{Backchain}\,\dfrac{\text{IH}\,\dfrac{\dfrac{}{\mathsf{nat}^{\downarrow}(x') \vdash \mathsf{nat}^{\downarrow}(x')}}{\mathsf{nat}^{\downarrow}(x') \vdash \mathsf{add}(x', 0, x')}}{\mathsf{nat}^{\downarrow}(x') \vdash \mathsf{add}(s(x'), 0, s(x'))}}{\mathsf{nat}^{\uparrow}(x) \vdash \mathsf{add}(x, 0, x)}}{\mathsf{nat}^{\uparrow}(x) \to \mathsf{add}(x, 0, x)}}{\mathsf{nat}(x) \to \mathsf{add}(x, 0, x)}$$

(This example also demonstrates one of the flaws of the derivation proof format, its insatiable appetite for horizontal space.)

It is interesting to note what happens when induction is applied to a non-terminating definition such as

$$\infty\,\frac{\mathsf{repeat}(x)}{\mathsf{repeat}(x)}$$

If we wish to prove, inductively, that $\mathsf{repeat}(x) \to P$ for any P, we have the inductive hypothesis $\mathsf{repeat}^{\downarrow}(x) \to P$ and the following derivation:

$$\infty\,\dfrac{\text{IH}\;\dfrac{\dfrac{}{\mathsf{repeat}^{\downarrow}(x) \vdash \mathsf{repeat}^{\downarrow}(x)}}{\mathsf{repeat}^{\downarrow}(x) \vdash P}}{\mathsf{repeat}^{\uparrow}(x) \vdash P}$$

From a non-terminating definition we can inductively prove anything! So far from being a flaw in our system, this surprising result reflects the fact that one computational interpretation of $\bot$ is that of a non-terminating computation. In fact, this result is simply the computational analogue of the well-known *ex falso quodlibet* principle of classical logic.

Future work

There are many areas in which Arend could be enhanced and extended. We examine ten possibilities here, including one, equational reasoning, in some depth.

- Arend's specification logic is simple enough that formalization of its properties should not be difficult, nonetheless, for formal correctness and consistency work needs to be done to show that it is both *sound* (everything it proves true is true) and *non-deterministically complete* (when it cannot find a proof none exists). Note that, for a Turing-complete specification language like ours, it is not possible to prove that the language is *complete*; i.e., that everything true is provable. Some logically-valid proofs will fail to terminate.

  More challenging is the task of proving that the reasoning language is *adequate* for proving propositions about the reasoning language. This implies proving that the specification language and its semantics can be embedded in the reasoning language, and that the various meta-level operations correctly preserve these semantics.

- Although we originally envisioned the specification logic as solely the domain of the instructor, there is no reason why, with some work, it could not be exposed to students as well, for the construction of their own specifications. This would lead Arend beyond its original goal of creating an environment for the student construction of proofs, into that of a proper proof assistant, with all levels of the system usable by all its users. In our experience, students' first exposure to declarative programming is often quite challenging; although Arend's specification logic is significantly simpler than Prolog, it remains to be seen whether programming-in-logic can be effectively and usefully presented to students at this stage.

- Currently Arend allows specification authors minimal influence on the *user interface*; their power is limited to declaratively describing how certain predicates should be rendered as HTML. For more complex specifications, this is insufficient. As an example, classical logic can be specified in Arend with relative ease, via a predicate `solve`: `solve(C,G,T)` succeeds if the proposition G evaluates to the truth value T in context C. However, none of the familiar operations for manipulating the consequent and antecedents can be used with this specification, as they are "hidden" inside the arguments to a predicate. A system allowing specification authors to highlight terms in various contexts as targets for user-manipulation would greatly extend the ease-of-use of the system when dealing with non-trivial specifications.

- Many proof assistants allow the user to create *tacticals*, "shortcuts" through the proof process or alternate reasoning strategies. A cursory glance at the *check/3* predicate at the heart of Arend's proof-checking algorithm will reveal a significant amount of similarity between the various cases: to apply a -Left or -Right rule, we apply some kind of transformation(s) to either one of the antecedents or the consequent, and then recursively check zero or more of these transformed subproofs. Tacticals allow the user to specify these transformations themselves, in a way that enforces their consistency with the underlying logic. For some specifications, tacticals could significantly simplify the development of otherwise-tedious proofs, by concealing the irrelevant details.

- In a similar vein, Arend currently requires the user to proceed through *all* steps of a proof. However, often this is not necessary; as can be seen from the specification logic, goals which lie entirely in the specification logic can be *automatically* proved in their entirety! There are several other classes of automation that could also be applied. However, the desire for simplicity and avoidance of tediously-obvious proofs must be balanced against the explicability of the system; some proof assistants are notorious for "magically" completing proofs while giving the user no indication as to *how* the final steps of the proof proceed. In addition, as Arend targets education, not all instances of automation are suitable for all levels of student usage. Some kinds of automation may be suitable for beginning students (e.g., those that conceal the more complex elements of the system), while other kinds may be more suitable for advanced students (e.g., those that do away with now-obvious steps in the proof).

- The creation of lemmas, and their use in proofs, is currently minimally specified. Although the exact semantics are dictated by the underlying logic, there are a number of questions about presentation and interaction that are currently unresolved. We envision lemmas being applied in a manner similar to the inductive hypothesis, by backchaining on the conclusion. Since one of the difficulties of the derivation-tree proof format is its often-extravagant use of horizontal space, it should be possible to automatically *extract* a sub-proof as a lemma. This could be done even when the lemma is not generally useful, but simply as a way of "naming" a portion of the proof or commenting on its purpose.

- The implementation of Arend is primitively minimal, suitable only for small-scale usage. Real-world usage would require integration with existing grading systems and automatic checks of student-submitted proofs. On a higher level we could easily imagine a set of features aimed at creating an ecosystem around student-created specifications, proofs, and lemmas, perhaps with collaboration features for multi-student use.

- Although we believe Arend's derivation-tree presentation of proofs has distinct advantages over the traditional paragraph format, there is no reason why an alternate user interface could not present proofs in that form, allowing users who feel more comfortable with a textual presentation to navigate their proofs in that way. Users could easily switch between the two presentations, and any other future presentations that might be added, even in the middle of a proof. Indeed, given the hierarchical structure of proofs, it is entirely possible that the two notations could be mixed within a single proof. The outer elements of a proof might be in paragraph form, while the individual cases were presented as derivations, or vice versa.

- The absense of any form of negation in our specification logic is sometimes a source of annoyance for authors, in particular, when the negation of a predicate has an obvious definition (for example, $<$ negated is $\geq$). The two common "solutions", either implementing the predicate and its negation separately, or adding an extra argument which emits a `true/false` value, are less than satisfactory.[25] A more interesting solution would be to add an explicit negation operator to the specification and reasoning languages which was limited to application to predicates which had previously been declared *negatable*, with an implementation of the negated predicate given explicitly. But it seems likely that some amount of inconvenience is unavoidable, simply because negation in a constructive logic functions differently from classical negation.

[25] In particular, the latter typically requires implementing the rest of classical Boolean logic, in order to be able to work with these explicit Boolean values.

Equational reasoning

One very large extension to the logic which we would like to investigate would be the incorporation of *equational reasoning*. Arend's current concept of equality is based entirely on unification: two terms are equal if they can be made equal by some substitution. Equational reasoning allows equality to be defined via rules, for example

For computational purposes equational rules are often regarded as unidirectional *rewrite rules*: $a = b$ becomes $a \mapsto b$.

$$\text{=-Refl}\,\frac{}{x = x} \qquad \text{=-Symm}\,\frac{x = y}{y = x} \qquad \text{=-Trans}\,\frac{x = y \wedge y = z}{x = z}$$

It should be emphasized that in equational reasoning these rules, and others, need not be "built-in" to the system but can be part of any specification that requires them. For example, for reasoning over addition and multiplication we might have the rules

$$\times\text{-dist-+}\,\frac{}{a \times (b + c) = a \times b + a \times c}$$

Equational reasoning is a particularly powerful mechanism for reasoning about *functional programs* and programming languages; we can state equivalences between expressions and then prove that two programs, or two classes of programs, are equivalent under those rules. Even more powerfully, if we wish to prove some property of an entire language, it is sufficient to show that the equivalences preserve that property.

Although the addition of equational reasoning to Arend would give the system a marked increase in logical power, it would also correspondingly increase the complexity of its implementation and presentation. Equivalence rules require a different treatment of terms and variables from unification, and it is not entirely clear how the two will interact, logically. Equivalences over programs require careful handling of *binders*, language constructs that introduce new (program) variables into scope. These variables are distinct from the logic variables, but can both contain, and be contained by, them. Although logics exist which provide support for reasoning about binding structures [Miller and Tiu, 2005] their usage in proofs involves additional complexity that may be at odds with our goal of use in *introductory* curriculum.

D. Miller and A. Tiu. A proof theory for generic judgments. *ACM Transactions on Computational Logic*, 6(4):749–783, October 2005

Appendix I: Syntax of the Specification Logic

The following is a simplified presentation of the grammar of the specification language. In particular, precedence rules for infix operators have been omitted, but are consistent with normal usage. Note that while the grammar includes rules defining infix arithmetic and comparison operators, these operators have no *semantic* significance within the specification logic. They are present under the assumption that specifications may want to use them, and will expect them to have their normal precedence ordering.

```
         ⟨start⟩ ← ⟨definitions⟩?

   ⟨definitions⟩ ← ⟨definition⟩ ⟨definitions⟩ ∗
    ⟨definition⟩ ← Rulename? ⟨predicate⟩
                 | ⟨infix-decl⟩

    ⟨infix-decl⟩ ← "infix" (Atom | Symbol)

     ⟨predicate⟩ ← ⟨term⟩ "."

          ⟨term⟩ ← ⟨infix-term⟩
    ⟨infix-term⟩ ← ⟨term⟩ ":-" ⟨term⟩
                 | ⟨term⟩ (";" | "|") ⟨term⟩
                 | ⟨term⟩ ("," | "&") ⟨term⟩
                 | ⟨term⟩ "=" ⟨term⟩
                 | ⟨term⟩ ⟨user-op⟩ ⟨term⟩
                 | ⟨term⟩ ⟨comparison-op⟩ ⟨term⟩
                 | ⟨term⟩ ⟨arith-op⟩ ⟨term⟩
                 | ⟨prefix-term⟩
 ⟨comparison-op⟩ ← "<=" | "<" | ">=" | ">" | "==" | "!="
      ⟨arith-op⟩ ← "+" | "-" | "*" | "/" | "%" | "^"

   ⟨prefix-term⟩ ← ("-" | "!")
                 | ⟨base-term⟩

     ⟨base-term⟩ ← ⟨compound⟩
                 | "[" ⟨termlist⟩? "]"
                 | "(" ⟨term⟩ ")"
                 | Atom
                 | Variable

      ⟨compound⟩ ← Atom "(" ⟨termlist⟩ ")"
```

Appendix II: Specification for $\mathbb{N}$, lists

The following is an example specification included with Arend, one which desribes natural numbers with addition and the $>$ relation, and lists of natural numbers, with the operations of list concatenation, length, and element testing.

```
"Nat-z":    nat(z).
"Nat-succ": nat(s(N)) :- nat(N).

"Add-z":    add(z,X,X).
"Add-succ": add(s(X),Y,s(Z)) :- add(X,Y,Z).

"GT-z":     s(_) > z.
"GT-succ":  s(X) > s(Y) :- X > Y.

"NL-nil":   natlist(nil).
"NL-cons":  natlist(cons(N,L)) :- nat(N), natlist(L).

"Cat-nil":  cat(nil,L,L).
"Cat-cons": cat(cons(N,L), L1, cons(N,L2)) :- cat(L,L1,L2).

"Len-nil":  nlen(nil,z).
"Len-succ": nlen(cons(_,L),s(M)) :- nlen(L,M).

"Elem-cons": elem(E,cons(E,_)).
"Elem-tail": elem(E,cons(_,L)) :- elem(E,L).
```

Although simplistic, this specification is already suitable for use in proving several interesting properties, for example:

- $\mathsf{nat}(X) \to \mathsf{add}(X, z, X)$. This will require induction on $\mathsf{nat}(X)$.
- Commutativity of add: $\mathsf{nat}(X) \wedge \mathsf{nat}(Y) \to \mathsf{add}(X, Y, Z) \wedge \mathsf{add}(Y, X, Z') \wedge Z = Z'$.
- Totality of add: $\mathsf{nat}(X) \wedge \mathsf{nat}(Y) \to \mathsf{add}(X, Y, Z) \wedge \mathsf{nat}(Z)$.
- Functionality of add: $\mathsf{nat}(X) \wedge \mathsf{nat}(Y) \wedge \mathsf{add}(X, Y, Z_1) \wedge \mathsf{add}(X, Y, Z_2) \to Z_1 = Z_2$.
- Transitivity of $>$: $\mathsf{nat}(X) \wedge \mathsf{nat}(Y) \wedge \mathsf{nat}(Z) \wedge X < Y \wedge Y < Z \to X < Z$.
- Cons is idempotent:
  $\mathsf{natlist}(L) \wedge \mathsf{elem}(N, L) \wedge \mathsf{nat}(N') \wedge L' = cons(N', L) \to \mathsf{elem}(N, L')$

References

E. C. Brady. Idris—: systems programming meets full dependent types. In *Proceedings of the 5th ACM workshop on Programming languages meets program verification*, pages 43–54. ACM, 2011.

L. E. J. Brouwer. On the foundations of mathematics. *1975) LEJ Brouwer: Collected Works*, 1:11–101, 1907.

B. Ford. Parsing expression grammars: A recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 111–122, New York, NY, USA, 2004. ACM. ISBN 1-58113-729-X. DOI: 10.1145/964001.964011. URL `http://doi.acm.org/10.1145/964001.964011`.

A. Gacek. The Abella interactive theorem prover (system description). In A. Armando, P. Baumgartner, and G. Dowek, editors, *Proceedings of IJCAR 2008*, volume 5195 of *Lecture Notes in Artificial Intelligence*, pages 154–161. Springer, August 2008.

A. Gacek. *A Framework for Specifying, Prototyping, and Reasoning about Computational Systems*. PhD thesis, University of Minnesota, September 2009.

G. Gentzen. Untersuchungen über das logische schließen. i. *Mathematische zeitschrift*, 39(1):176–210, 1935.

G. Gentzen. Investigations into logical deduction. *American philosophical quarterly*, pages 288–306, 1964.

H. Geuvers. Proof assistants: History, ideas and future. *Sadhana*, 34(1):3–25, 2009.

A. Heyting. *Intuitionism*, volume 41. Elsevier, 1966.

G. Kadoda, R. Stone, and D. Diaper. Desirable features of educational theorem provers–a cognitive dimensions viewpoint. In *Collected Papers of the 11th Annual Workshop of the Psychology of Programming Interest Group (PPIG-11)*, pages 18–23, 1999.

P. Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. *Nordic journal of philosophical logic*, 1(1):11–60, 1996.

The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. URL `http://coq.inria.fr`. Version 8.0.

D. Miller and A. Tiu. A proof theory for generic judgments. *ACM Transactions on Computational Logic*, 6(4):749–783, October 2005.

U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

F. Pfenning. Logic programming. *Course notes*, 2006. URL `http://www.cs.cmu.edu/~fp/courses/lp/lectures/lp-all.pdf`.

F. Pfenning and C. Schürmann. System description: Twelf—a meta-logical framework for deductive systems. In *Automated Deduction—CADE-16*, pages 202–206. Springer, 1999.

B. Pientka and J. Dunfield. Beluga: A framework for programming and reasoning with deductive systems (system description). In *Automated Reasoning*, pages 15–21. Springer, 2010.

S. Reichelt. Treating sets as types in a proof assistant for ordinary mathematics. Institute of Informatics, University of Warsaw, Warszawa, Poland, 2010. URL `http://hlm.sourceforge.net/types.pdf`.

J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM (JACM)*, 12(1):23–41, 1965.

P. Suppes. Future educational uses of interactive theorem proving. In *University-level Computer-assisted Instruction at Stanford: 1968-1980*, pages 399–430. Stanford University, Instute for Mathematical Studies in the Social Sciences; Stanford, CA, 1981.