Arend — Proof Assistant Assisted Pegagogy

A graphical proof assistant for undergraduate computer science education

Andrew V. Clifton

Department of Computer Science
California State University, Fresno

May 2015

Formal proofs

Formal proofs — an important component of computer science education.

Prove

- $\forall x, y \in \mathbb{N} : x + y = y + x$.
- If T is a complete binary tree with $n = |T|$ nodes, then the height of any node is at most $\lfloor \log_2 n \rfloor$.
- The reverse of a regular language L^R is itself regular.

Paper proofs

Paper proofs are common, but problematic for education:

- Too flexible; allow a wide variety of “almost correct” answers.

Paper proofs

Paper proofs are common, but problematic for education:

- Too flexible; allow a wide variety of “almost correct” answers.
- Delayed results; turn in a proof assignment, get results back a week later.

Paper proofs

Paper proofs are common, but problematic for education:

- Too flexible; allow a wide variety of “almost correct” answers.
- Delayed results; turn in a proof assignment, get results back a week later. Batch processing for proofs.

Paper proofs

Paper proofs are common, but problematic for education:

- Too flexible; allow a wide variety of “almost correct” answers.
- Delayed results; turn in a proof assignment, get results back a week later. Batch processing for proofs.
- Non-interactive.

Computer-assisted logic

Using computers to do logic is not a new idea:

Computer-assisted logic

Using computers to do logic is not a new idea:

- Automated theorem provers (e.g., AUTOMATH)

Computer-assisted logic

Using computers to do logic is not a new idea:

- Automated theorem provers (e.g., AUTOMATH)
- Model checkers

Computer-assisted logic

Using computers to do logic is not a new idea:

- Automated theorem provers (e.g., AUTOMATH)
- Model checkers
- Proof assistants (Abella, Coq, Arend, etc.)

Proof assistants

A proof assistant

Proof assistants

A proof assistant

- Assists the user in constructing a valid proof.

Proof assistants

A proof assistant

- Assists the user in constructing a valid proof.
- Forbids the construction of invalid proofs.

Proof assistants

A proof assistant

- Assists the user in constructing a valid proof.
- Forbids the construction of invalid proofs.
- Presents proofs, complete or not, to the user in a comprehensible format.

Proof assistants, cont.

Some well-known proof assistants:

- Twelf (previously used in CSCI 217)
- Coq
- Abella (currently used in CSCI 217)
- Agda

Aside: the Curry-Howard Isomorphism

An aside:
Some proof assistants bridge the gap between functional programming and proofs, thanks to the Curry-Howard isomorphism.

Definition

The Curry-Howard isomorphism states that *proofs* are to *propositions* as *programs* are to *types*.

$a : A$ can mean "a is a program with type A", or "a is a proof of the proposition A".

Curry-Howard isomorphism, cont.

Some examples:

- If $p : P$ and $q : Q$ then the pair $(p, q) : P \wedge Q$.
- If $p : P$ and $q : Q$ then either

$$\text{left}(p) : P \vee Q$$

  or

$$\text{right}(q) : P \vee Q$$

- More interesting: $P \rightarrow Q$ means "P implies Q".

Curry-Howard isomorphism, cont.

Some examples:

- If $p : P$ and $q : Q$ then the pair $(p, q) : P \wedge Q$.
- If $p : P$ and $q : Q$ then either

  $$\text{left}(p) : P \vee Q$$

  or

  $$\text{right}(q) : P \vee Q$$

- More interesting: $P \to Q$ means "P implies Q".
  But it is also the type of functions from P to Q. A proof of $P \to Q$ is a *program* that converts a proof (value) of P into a proof (value) of Q!

(End of aside.)

Proof assistants in education

We are interested in the application of proof assistants to CSCI education.

Why?

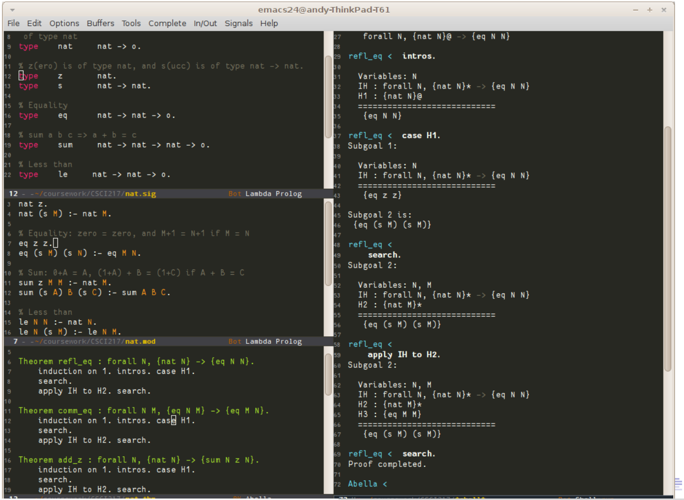
- Fixed notion of what a valid proof is (and isn't).
- Instant results: yes, this proof is correct; no, it isn't.
- Interactive.

Problems with existing systems

But when it comes to undergrad education, there are some problems with existing systems:

- Complexity: powerful logics create complexity in even simple proofs.
- Not user-friendly: Emacs + ProofGeneral are hardly intuitive.
- Unfamiliar: Syntax often is often wildly different from any kind of paper proof

What we don't want



What we do want

Prove $\forall X\colon \mathsf{nat}(X) \rightarrow \mathsf{add}(X,z,X)$

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

Demo

A quick demo of a proof in Arend

What is Arend?

Arend is a web-based proof assistant designed for use in undergraduate CSci education.

- Based on a simple, familiar first order logic ($\forall$, $\exists$, $\wedge$, $\vee$, and $\rightarrow$).
- Specifications (systems to be reasoned about) are constructed by instructors, as are proof statements ($\forall X : \exists Y : \ldots$)
- Students construct proofs by direct interaction: "point-and-click".
- Invalid proofs cannot be constructed, and incomplete proofs are marked as such

Specification logic

Arend's specification logic is used to describe the systems to be reasoned about. E.g., a specification for $\mathbb{N}, +$:

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

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

Specification logic, cont.

- A specification consists of a series of definitions.
- A definition consists of one or more clauses.
- Each clause has a name, a head, and an (optional) body.
- The body of each clause must be a pure conjuction of atomic goals (calls to definitions)

Almost Prolog...

It looks like Prolog, but not quite:

- No disjunction, except that implicit in multiple clauses.
- No negation ("as failure", or otherwise).
- No proof search control structures: `!`, `->`, etc.

Proof search (by resolution) is largely the same. (I.e., ordering of clauses is significant for execution, but *not* for proofs.)

Specifications as rules

Clauses in the specification logic correspond almost exactly to inference rules:

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

becomes

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

Reasoning logic

Proofs are about things in the specification logic, but proofs themselves are in the reasoning logic.

The reasoning logic has everything the specification logic has, plus

- Implication: $P \to Q$. (Note that P cannot contain further implications!)
- Explicit quantification: $\forall X : \ldots$ and $\exists Y : \ldots$
- Free use of $\wedge$ and $\vee$

Embedding

Thus, the specification logic can be embedded in the reasoning logic:

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

becomes

$$\forall X, Y, Z : \mathsf{add}(X, Y, Z) \to \mathsf{add}(succ(X), Y, succ(Z))$$

Reasoning about specifications

This allows us to use the specification logic to reason about specifications. E.g.

Prove:

$$\forall X, Y : \mathsf{nat}(X) \land \mathsf{nat}(Y) \to \exists Z : \mathsf{add}(X, Y, Z)$$

This proof will be about nat and add.

Implementation statistics

Arend's implementation consists of:

- 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

Development details

Arend's development:

- Tracked using the Fossil version control system (http://fossil-scm.org)
- 294 commits
- Spans eight months of development

Development tools

Some libraries and tools used:

- Node.JS – Offline Javascript runtime
- SWI-Prolog – Prolog environment
- Lodash – Javascript utility library
- jQuery – Javascript+HTML utility library
- qUnit – Javascript test framework
- Pengines – Prolog HTTP server framework

Web client overview

Arend's user interface is a fairly straightforward web client, with a few twists:

- Full `Term` datatype (incl. atoms, logic variables, and compounds). This allows terms to be communicated to/from the backend without any special-purpose translation.
- Unification of terms is also present in the client codebase, currently unused. Eventually will form part of a term pattern-matching library.
- Pengines allows (nearly) transparent JS/Prolog interop., almost as if Prolog was running in the browser.

Major backend modules

Arend's backend (exposed via HTTP) consists of three main modules:

- `subst` – Unification and substitution
- `program` – Goal expansion and execution for specifications
- `checker` – Elaboration and checking of proofs (reasoning logic)

Substitution and unification

Because proofs may have different substitutions in different parts of the tree, we cannot use Prolog's (global) unification and substitution. We reimplement logic variables, unification, and substitution.

$$
\dfrac{X \mapsto z \quad \dfrac{\vdots}{\vdash \mathsf{add}(z, z, z)} \qquad X \mapsto s(N) \quad \dfrac{\vdots}{\mathsf{nat}(N) \vdash \mathsf{add}(s(N), z, s(N))}}{\mathsf{nat}(X) \vdash \mathsf{add}(X, z, X)}\ \text{Case on nat}
$$

`subst` module

The `subst` module implements:

- Custom variable type (encoded as special atoms)
- Robinson unification algorithm over terms containing these variables
- Application of substitutions to terms

`program` module

Module `program` is responsible for handling specifications:

- Expanding calls to atomic goals (e.g., `add(z,s(z),X)`) requires renaming variables in the body, so they don't conflict with variables in scope.
- Execution of specification queries follows the resolution proof search procedure. Note that Arend lacks "negation as failure".
- Execution produces proof objects compatible with those used by the full proof checker.
- Execution of queries is exposed via the `repl` Pengine application.

`checker` module

The most complex module in the backend, `checker` handles elaboration and checking of proofs in the full reasoning logic.

- Proof completeness – Does a proof contain any holes? (Simple recursive predicate)
- Proof elaboration – Expanding a hole into a 1-level subproof
- Proof checking – Is a proof correct, according to a specification and the rules of the reasoning logic?

Proof elaboration

Proof elaboration, in tandem with proof checking, is at the heart of incremental proof construction. Consider the proof state:

$$\frac{?}{\vdash P \wedge Q}$$

If we elaborate $P \wedge Q$, what should replace ?.

Proof elaboration, cont.

$$\dfrac{\dfrac{?}{\vdash P} \qquad \dfrac{?}{\vdash Q}}{\vdash P \wedge Q}$$

Elaboration expands a ?, in combination with either the consequent or an antecedant, so that the result is a valid proof tree, one level deeper.

Proof checking

Checking a proof object proceeds by checking it against the rules of the specification logic.

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

(E.g.: Rules for $\wedge$)

Proof checking, cont.

Each node of the proof tree includes:

- Node type (e.g., `product`, `induction`, etc.)
- Subproof(s) (child nodes)
- Consequent (proposition to the right of $\vdash$)
- Antecedents (propositions to the left of $\vdash$)
- Current substitution
- Variables in scope

Proof checking, cont.

Substitutions and variable bindings flow through the tree nontrivially:

- Substitutions flow from leaves to root, but also left-to-right in conjunctions.
- Variable scopings flow from root to leaves, but also left-to-right in conjunctions.

Formalization of the complete proof checking procedure, including substitutions and variable scopings, is ongoing.

Proof construction procedure

1. User selects an element (antecedent or consequent) in the current proof state.
2. Path to the element along with the proof tree is passed to the server.
3. Server calls `checker:elaborate` to elaborate the desired element.
4. Elaborated proof is returned to client.
5. New proof is checked for completeness. Complete? then Stop, else GoTo 1.

The future of Arend

Arend is far from complete; enhancements can be divided into three areas:

- Necessary features
- Enhancements
- Formalization

Necessary features

Arend is missing many features that would be necessary in a large-scale deployment:

- Centralized storage of specifications, assignments
- Interop with grading backend, for storage of (in)complete assignments
- Richer user interface: lemma construction, instantiation of $\exists$ variables, etc. are all unspecified
- Easy-to-deploy packaging of the entire system

Enhancements

Although not strictly necessary, there are still many enhancements that would make Arend a better system, either more powerful, easier to use, or both.

- Enhanced proofs: tactics, instructor-controlled proof automation.
- Support for student-authored specifications
- Alternate proof interfaces: traditional paragraph, mixed, etc.
- Functional language for reasoning about programs, equational reasoning

Formalization

Although we believe Arend's systems to be fully adequate, being based on existing well-studied systems, a full formalization of our systems and their integration would be a useful addition.

- Full operational semantics of the specification logic
- Proof of soundness and non-deterministic completeness of the specification logic (all things proven are true, and nothing false can be proven)
- Full semantics for reasoning logic, incl. substitutions and bindings
- Proof of adequecy of the reasoning logic with regard to the specification logic.

Conclusions

We believe that Arend's design will make it a valuable addition to the undergraduate computer science curriculum. We are currently working to get Arend into a suitable state for use in our own courses, and hope to have feedback from real student usage in the future.

For Further Reading

- A. V. Clifton
  Arend — Proof-assistant Assisted Pedagogy
  CSU Fresno, 2015.
- H. Geuvers
  Proof assistants: History, ideas and future
  Sadhana, 31(1):3–25, Springer, 2009.