GENTZEN

A beginner's proof assistant based on
Higher Order Logic

A Special Project

LIAM O'CONNOR-DAVIS

Supervised By

MANUEL M. T. CHAKRAVARTY

submitted in fulfillment of
the requirements for the honours degree of
Bachelor of Science (Computer Science)

School of Computer Science and Engineering
University of New South Wales

THE UNIVERSITY OF
NEW SOUTH WALES

SYDNEY · AUSTRALIA

February 15th 2013

Abstract

Existing proof assistants such as Isabelle (Nipkow *et al.*, 2002) or Coq (Coq Development Team, 2011) are geared towards large-scale, complicated verification projects, and are not ideal educational tools for topics of a fairly formal nature, such as the theoretical foundations of programming languages. In particular, they opt for a proof language that resembles a programming language, rather than the structure typically seen in pen-and-paper proofs. Gentzen is a simple theorem prover we are developing in Haskell (Marlow Ed, 2010), which will be tightly integrated with its user interface, using a structural editor for graphically presented proofs. This allows proof scripts to resemble pen-and-paper proofs, and assists students to focus on the proof, rather than the proof assistant. This report outlines the current state of Gentzen development, and gives a formal treatment of its proof checker semantics.

Contents

1 Motivation 4

2 Design and Semantics 5

2.1 Term Language 5

2.2 Proof Language 6

2.3 Proof Checking 10

3 Future Work 11

3.1 Pattern Unification 11

3.2 Graphical Editor 13

3.3 Data types and other extensions 13

4 Conclusion and Related Work 13

References 14

"I tried reading Hilbert. Only his papers published in mathematical periodicals were available at the time. Anybody who has tried those knows they are very hard reading. "

— Alonzo Church

1 Motivation

Logic, proofs, and proof techniques form the foundation for a variety of topics in Computer Science. In particular, simple, inductively defined structures and proofs about them are pervasive in fields such as programming language theory and formal methods.

From an instructor's perspective, the assumption that any given student has the necessary mathematical maturity to understand such material has proven to be highly tenuous. In particular, students often lack the ability to read and write proofs.

This leads to three possible course curricula, all problematic:

1. A curriculum that simply assumes the necessary mathematical maturity anyway, and suffers from perpetually low enrollments as a result[1].
2. A curriculum that attempts to "hide" the mathematical underpinnings of the field, instead focusing on more "practical" considerations; an approach which Edsger Dijkstra rightly decried as "[lacking] the courage to teach hard science" and "misguiding students". Moreover, his prediction that "each next stage of infantilization of the curriculum will be hailed as educational progress" was sadly correct (Dijkstra, 1988).
3. A curriculum that is forced to reduce the amount of difficult content so that preliminary mathematical skills can be taught first. Our focus is on this type of course.

When investigating this problem for his own Software Foundations course, Benjamin Pierce advocated the use of a proof assistant, to enable students to work with proofs and check them without requiring intensive training and a fast feedback loop with teachers (Pierce, 2009). This approach yielded promising results, enabling his syllabus to be significantly expanded with no substantial effect on student exam scores.

Pierce chose Coq (Coq Development Team, 2011), a dependently-typed theorem prover based on the calculus of constructions (Coquand, 1986), as the proof assistant for his course. He encountered a few difficulties with this choice:

- Being based on a *constructive* logic formed by an intensional type theory, via the Curry Howard correspondence (Howard, 1980), Coq quickly plunges students into deep theoretical waters if they try to encode a proof by contradiction or extensionality. Explaining the theoretical foundations of programs being proofs requires a background in type theory and logic, which leads to a difficult conundrum when the goal of the course is to provide said background in type theory and logic.
- The user interface of Coq is not well-suited to beginners: the overall proof obligation can sometimes be lost in a sea of confusing variable names and seemingly unrelated subgoals.

Based on this information, we have begun implementation of Gentzen, a theorem prover designed *specifically* for this educational use-case.

The meta-logic is based on Natural Deduction (Gentzen, 1935), with a higher-order term language based on the simply typed lambda calculus (Church, 1940). By avoiding Curry-Howard, we eliminate the circular pedagogy problem that exists with a choice like Coq.

Unlike Coq, Gentzen will be tightly integrated with our own custom *structural editor* that operates on syntax trees rather than raw text. This enables us to represent proofs *graphically* in a way that Coq cannot. Rules are written in vertical form, proofs are written in the "proof tree" style of Gentzen (Gentzen, 1935), and propositions can be displayed with any desired notation without fear of ambiguity, as they are never parsed from that format. In this way, a proof in Gentzen would resemble a proof written by hand on a paper or blackboard, rather than a series of inscrutable tactics and rules. Thus, we hope to avoid distracting students with learning the operation of the proof assistant, and help them instead to focus on learning to prove and reason mathematically.

One of the positive aspects Pierce mentions for choosing Coq is its powerful automation facilities. Gentzen,

[1] I refer to this as the Engelhart Method

Term variables		$\in$	$x, y, z, f, g, h, \cdots$
Base Types		$\in$	$\mathtt{T}, \mathtt{U}, \cdots$
Types	τ	$::=$	$\mathtt{T} \mid \tau_1 \to \tau_2$
Free (schematic) variables		$\in$	A_τ, B_τ, C_τ
Expression heads	h	$::=$	$x \mid A_\tau$
β-normal expressions	e	$::=$	$\lambda(\overline{x : \tau}).\ h\ \overline{e}$
Type environments	Γ	$::=$	$\overline{x : \tau}$

Fig. 1: Syntax of Term Language

by contrast, has little automation — indeed, it has no notion of a tactic script, so automated tactics are difficult to express. As an experiment, we encoded some of our basic course materials in Agda (Norell, 2008), a proof assistant which also lacks automation, and have found that the lack of automation is not a hindrance to encoding the simple definitions and proofs used commonly in our Computer Science courses.

As Gentzen is designed to work on small, simple problems that would be commonly encountered in a tutorial or exam, there is no need for a highly efficient implementation. This means that, unlike Isabelle or Coq, we can focus on making the implementation simple and easily understood, and exploit language features to guide us to a correct implementation more straightforwardly. In addition, as our prover is not designed to be trusted for safety-critical systems, we do not need to go to LCF-style pains to ensure soundness (Milner, 1972), unlike Isabelle.

In the world of proof assistant implementation, there are very few tutorials or simple implementations to study. The core theorem prover components of Gentzen are designed to be the *simplest possible* implementation of a proof assistant. This motivates many design decisions, from our use of a simply typed (non-polymorphic) lambda calculus, to conventional higher order unification (see section 2). In this way, we hope to make the first version of Gentzen into a *tutorial implementation* that smooths the road for others that wish to follow us into this sparsely charted territory.

2 Design and Semantics

There are two main languages at work in any HOL-style theorem prover. The language of *terms*, which are usually expressions in some (normalising) lambda calculus, and the language of *proof*, also known as the *meta-logic*, which in our case is based on the style of natural deduction.

2.1 Term Language

The language of terms is defined in Figure 1. For our modest goals, we have chosen to use Church's $\lambda_{\to}$, the simply typed lambda calculus, equipped with atomic base types.

For our term representation, we want structural equality between terms to equate: terms with cosmetic differences in names (α-equivalence), terms in different forms but which normalise to the same result (β-equivalence), and terms with expanded λ-abstractions around variables of function types (η-equivalence). By making this $\alpha\beta\eta$-equivalence into a structural equality we simplify a number of problems when proof-checking, chief among them being the unification of higher-order terms.

Making α-equivalence into structural equality can be solved trivially by using a nameless representation such as de Bruijn indices (de Bruijn, 1972). We present our rules here, however, using a named representation, as de Bruijn terms can be difficult to read. Our choice of name representation is essentially based on de Bruijn indices, however we parameterise the type of terms by the type used for indices, which turns terms into a monad, with substitution as the monadic "bind" function (Bird & Paterson, 1999).

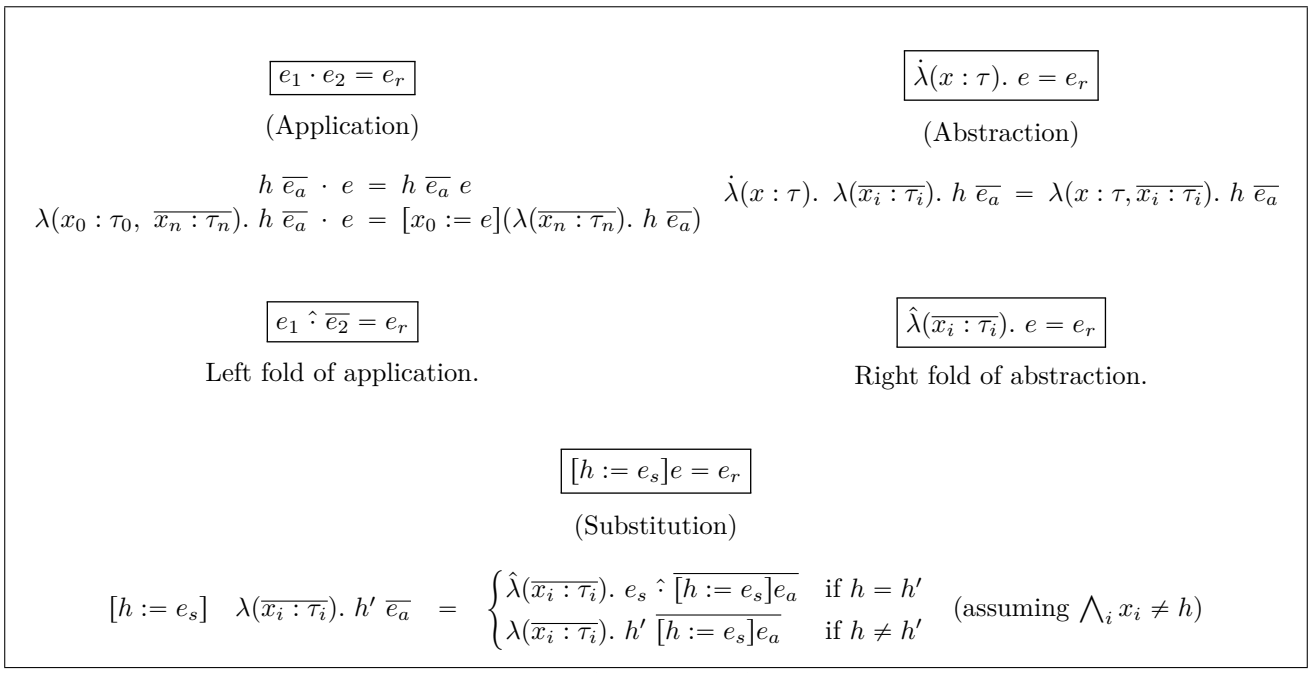

Fig. 2: Recovering traditional Application, Abstraction, and Substitution

To make β-equivalent terms structurally equal, we remove expressions of the form $(\lambda(x : \tau).\ e_r)\ e_a$ from our language. All terms consist of some number of lambda abstractions, a *head* term (which is essentially a variable), and some number of argument terms. Thus, all our terms are automatically in β-normal form. We recover the expressivity of the more traditional, non-normalised representation by encoding it as explicit combinators that result in β-normal terms. Figure 2 defines application, abstraction and substitution this way.

Note that these functions normalise as per the *untyped* lambda calculus — they completely ignore types when normalising. This means that an attempt to construct a non-normalising term will cause these functions to diverge. As our *typed* lambda calculus is strongly normalising, we can guarantee that these combinators will converge when used with well-typed terms. In practice, these combinators are only used in two situations: performing substitutions, and expansion to η-long normal form. In both of these situations, it is easy to show that terms have the correct types[2].

It is not possible, however, to make any two η-equivalent terms structurally equal, as any sound η-normalisation is dependent on type-checking information. This necessitates some initial term representation, which is not necessarily η-normal, on which the type checker operates. The typing rules of our lambda calculus are shown in Figure 3, along with the expansion rules to *η-long normal form*, where all function-typed variables are fully saturated with applications. While our typing rules and η-long normalisation steps are shown here as separate phases, our implementation interleaves type checking and η-long normalisation. In addition to some minor performance benefits, this also guarantees that, for *type checked terms* at least, η-equivalent terms are structurally equal.

In addition to bound variables, a *head* term can also be a *schematic* or unification variable. A schematic variable A_τ can be freely substituted for any term of type τ, and the user or the unification algorithm is free to do so at any time. These variables are effectively global in scope, and therefore no α-equivalence problems arise from using names to represent these variables.

2.2 Proof Language

Our meta-logic must provide for a means of expressing *statements* in our logic, such as assumptions or proof obligations, which we refer to collectively as *rules*, as well as some language for expressing the proofs themselves, which we call *statements*.

[2] Assuming substitution is sound.

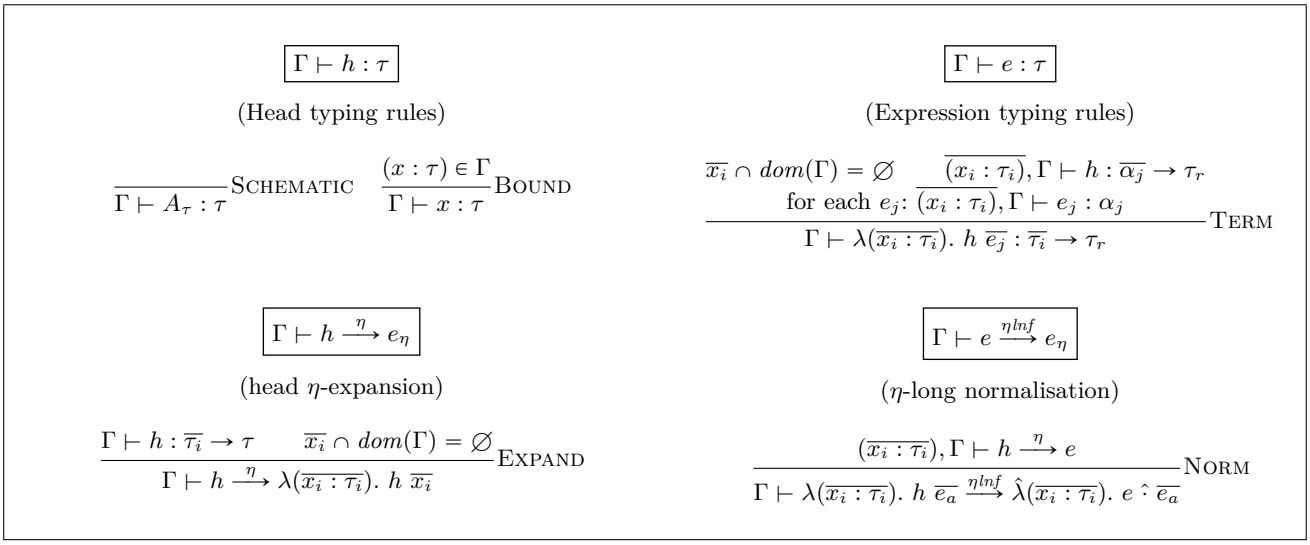

Fig. 3: Typing and η-long normalisation rules

2.2.1 Rules

Our notion of a rule is derived from the *Natural Deduction* of Gentzen (Gentzen, 1935), where rules are expressed as rules of inference of the following form:

$$\dfrac{\begin{array}{cccc}\rho_1 & \rho_2 & \cdots & \rho_n\end{array}}{e}\textsc{Name}$$
abc.

This rule says that, for all values of the metavariables a, b and c, the conclusion (which is some concrete proposition e, an expression of type `Prop`) can be derived if all of the premises ρ_1, ρ_2, $\cdots$, ρ_n can be derived. Note that these premises can themselves be rules. For example, here is an induction principle for Peano style natural numbers, where $0 : \mathbb{N}$ and $\mathbf{succ} : \mathbb{N} \to \mathbb{N}$:

$$\dfrac{\begin{array}{cc}P(0) & \begin{array}{c}P(x)\\ \vdots \\ P(\mathbf{succ}\ x)\end{array}\end{array}}{P(n)}\textsc{Induct}$$
P n. *x.*

This rule says that for any P, presumably of type $\mathbb{N} \to \texttt{Prop}$, and any $n : \mathbb{N}$, if $P(0)$ is derivable and, for any x, we can derive $P(\mathbf{succ}\ x)$ from $P(x)$, then $P(n)$ is derivable.

Note that the vertical dots ($\vdots$) are used for the subrule rather than the horizontal vinculum — this notational difference distinguishes hypothetical derivations like those used above from the stacked rule applications used in the "proof tree" notation of natural deduction proofs, which will be discussed in Section 2.2.2.

While the graphical interface of GENTZEN would display this rule much like it is presented here, we will use the more compact syntax described in Figure 4 to simplify presentation. In this syntax, the above rule would be represented by:

$$\Lambda(P : \mathbb{N} \to \texttt{Prop}, n : \mathbb{N}).\ [\![P(0);\ \Lambda(x : \mathbb{N}).\ [\![P(x)]\!] \Rightarrow P(\mathbf{succ}\ x)]\!] \Rightarrow P(n)$$

Figure 5 describes a well-formedness relation on rules, which simply ensures that all terms inside the rule are of the correct type — the special propositional type `Prop`. It also defines substitution on rules, and instantiation of rules with some metavariable assignment.

Rule names		$\in$	$\dot{q} \mid \dot{r} \mid \dot{s} \mid \cdots$
Rules	ρ	$::=$	$\Lambda(\overline{x : \tau}).\ [\![\overline{\rho_p}]\!] \Rightarrow e$
Statements	π	$::=$	$\texttt{for all}\ x : \tau.\ \pi$
		$\mid$	$\texttt{assuming}\ \dot{r} = \rho,\ \pi$
		$\mid$	$\texttt{lemma}\ \dot{r} = \pi_1,\ \pi_2$
		$\mid$	ψ
Proof Trees	ψ	$::=$	$\texttt{sorry}\ \rho$
		$\mid$	$\texttt{show}\ \rho\ \texttt{by}\ \dot{r}\ \texttt{with}\ \sigma$
		$\mid$	$\texttt{step}\ e\ \texttt{by}\ \dot{r}\ \texttt{with}\ \sigma\ \texttt{giving}\ \overline{\psi}$
Substitutions	σ, θ	$::=$	$[\overline{h := e}]$
Rule environments	Δ	$::=$	$\overline{\dot{r} = \rho}$
Context stack	Φ	$::=$	$\Phi \blacktriangleleft \phi \mid \epsilon$
Context frame	ϕ	$::=$	$\texttt{for all}\ x : \tau.\ \square$
		$\mid$	$\texttt{assuming}\ \dot{r} = \rho,\ \square$
		$\mid$	$\texttt{lemma}\ \dot{r} = \square,\ \pi$
		$\mid$	$\texttt{lemma}\ \dot{r} = \pi,\ \square\ (\rho)$
		$\mid$	$\texttt{step}\ e\ \texttt{by}\ \dot{r}\ \texttt{with}\ \sigma\ \texttt{giving}\ \overline{\psi}\ \square\ \overline{\psi}$
		$\mid$	$\texttt{flex-flex}\ e_1 \sim e_2,\ \square$
Equations	ξ	$::=$	$\overline{e_1 \sim e_2}$
Proof states	Q	$::=$	$\Gamma;\ \Delta;\ \xi;\ \Phi \rhd \pi$
		$\mid$	$\Gamma;\ \Delta;\ \xi;\ \Phi \lhd \pi; \rho$

Substitution Image: $\sigma'' \overline{h} = \{x := e \mid x \in \overline{h},\ (x := e) \in \sigma\}$

Fig. 4: Syntax for Rules and Statements

2.2.2 Statements

Suppose we encode in natural deduction rules for implication and conjunction:

$$\frac{\begin{array}{c}A\ B.\\ \\ \overline{A}\ u\\ \vdots\\ B\end{array}}{A \supset B}\ \supset_I^u \qquad\qquad \frac{A\ B.}{\dfrac{A \quad B}{A \wedge B}\ \wedge_I}$$

In natural deduction, a proof of a statement is given by a tree-like application of rules, where each sub-goal is produced by an application of some instantiation of a rule to the desired conclusion. Here is a proof that, assuming P (A_1), Q (A_2), and R (A_3), then $P \wedge (Q \wedge R)$[3]:

$$\dfrac{\overline{P}\ A_1 \quad \dfrac{\overline{Q}\ A_2 \quad \overline{R}\ A_3}{Q \wedge R}\ \wedge_I\ [\![A := Q,\ B := R]\!]}{P \wedge (Q \wedge R)}\ \wedge_I\ [\![A := P,\ B := Q \wedge R]\!]$$

This proof can be read either in a *forward* direction, establishing the truth of the overall statement from the assumptions of P, Q and R, or a *backward* direction, decomposing each proof obligation into smaller obligations via rule application.

[3] Typically instantiating assignments are not given in these proofs, but we include them for clarity

$$\boxed{\Gamma \vdash \rho \textbf{ wellformed}}$$

$$\frac{(\overline{x_i : \tau_i}), \Gamma \vdash e : \texttt{Prop} \qquad \text{for each } \rho_p\text{: } (\overline{x_i : \tau_i}), \Gamma \vdash \rho_p \textbf{ wellformed} \qquad \overline{x_i} \cap dom(\Gamma) = \varnothing}{\Gamma \vdash \Lambda(\overline{x_i : \tau_i}).\ [\![\overline{\rho_p}]\!] \Rightarrow e \textbf{ wellformed}}\textsc{Well-formed}$$

$$\boxed{[x := e_s]\rho = \rho_r}$$

(Rule Substitution)

$$[x := e_s] \quad \Lambda(\overline{x_i : \tau_i}).\ [\![\overline{\rho_r}]\!] \Rightarrow e \quad = \quad \Lambda(\overline{x_i : \tau_i}).\ [\![\overline{[x := e_s]\rho_r}]\!] \Rightarrow [x := e_s]e \quad \text{(Assuming } x \notin \overline{x_i})$$

$$\boxed{\textbf{instantiate}(\rho, \sigma) = \rho_r}$$

(Rule instantiation)

$$\textbf{instantiate}(\Lambda(\overline{x_i : \tau_i}).\ [\![\overline{\rho_r}]\!] \Rightarrow e, \sigma) \quad = \quad \Lambda(\overline{x_i : \tau_i}).\ [\![\overline{\sigma\rho_r}]\!] \Rightarrow \sigma e \quad \text{where } dom(\sigma) = \overline{x_i}$$

Fig. 5: Rule Well-formedness relation and utility functions

Our proof language, fully outlined in Figure 4, mirrors the structure of this proof quite closely[4]:

$$\begin{array}{l}\texttt{for all } P : \texttt{Prop}.\ \ \texttt{for all } Q : \texttt{Prop}.\ \ \texttt{for all } R : \texttt{Prop}.\\ \quad \texttt{assuming } A_1 = P,\ \ \texttt{assuming } A_2 = Q,\ \ \texttt{assuming } A_3 = R,\\ \quad\quad \texttt{step } (P \wedge (Q \wedge R)) \texttt{ by } \wedge_I \texttt{ with } [A := P, B = Q \wedge R]\\ \quad\quad\quad \texttt{giving show } P \texttt{ by } A_1 \texttt{ with } \epsilon;\\ \quad\quad\quad\quad \texttt{step } (Q \wedge R) \texttt{ by } \wedge_I \texttt{ with } [A := Q, B := R]\\ \quad\quad\quad\quad\quad \texttt{giving } (\texttt{show } Q \texttt{ by } A_2 \texttt{ with } \epsilon, \texttt{show } R \texttt{ by } A_3 \texttt{ with } \epsilon)\end{array}$$

Here is a proof of $P \supset (Q \supset P \wedge Q)$, for all P and Q:

$$\dfrac{\dfrac{\dfrac{\overline{P}^{\,u} \quad \overline{Q}^{\,v}}{P \wedge Q} \wedge_I [\![A := P,\ B := Q]\!]}{Q \supset P \wedge Q} \supset_I^v [\![A := Q,\ B := P \wedge Q]\!]}{P \supset (Q \supset P \wedge Q)} \supset_I^u [\![A := P,\ B := Q \supset P \wedge Q]\!]$$

This proof demonstrates a number of characteristics of natural deduction proofs. Most notably, rules with *hypothetical derivations* in their premises (like $\supset_I$) produce *local assumptions* in the proof tree, which are only available to the subgoals produced by that rule application.

These local assumptions prove to be a considerable complication when implementing proof checking, and they can be quite confusing to struggling students. We solve this problem by restricting proof trees to include *no* hypothetical assumptions. Instead, goals which include hypothetical derivations must be solved by directly instantiating a separate lemma. The above proof would therefore be written in our proof language as:

$$\begin{array}{l}\texttt{for all } P : \texttt{Prop}.\ \ \texttt{for all } Q : \texttt{Prop}.\\ \quad \texttt{lemma } \ell = \texttt{assuming } \alpha_1 = P,\\ \quad\quad\quad \texttt{lemma } \ell_1 = \texttt{assuming } \alpha_2 = Q,\\ \quad\quad\quad\quad\quad\quad \texttt{step } (P \wedge Q) \texttt{ by } \wedge_I \texttt{ with } [A := P, B := Q]\\ \quad\quad\quad\quad\quad\quad\quad\quad \texttt{giving } (\texttt{shows } P \texttt{ by } \alpha_1 \texttt{ with } \epsilon, \texttt{shows } Q \texttt{ by } \alpha_2 \texttt{ with } \epsilon),\\ \quad\quad\quad \texttt{step } (Q \supset P \wedge Q) \texttt{ by } \supset_I \texttt{ with } [A := Q, B := P \wedge Q]\\ \quad\quad\quad\quad\quad \texttt{giving } (\texttt{shows } [\![Q]\!] \Rightarrow P \wedge Q \texttt{ by } \ell_1 \texttt{ with } \epsilon),\\ \quad \texttt{step } (P \supset (Q \supset P \wedge Q)) \texttt{ by } \supset_I \texttt{ with } [A := P, B := Q \supset P \wedge Q]\\ \quad\quad\quad \texttt{giving } (\texttt{shows } ([\![P]\!] \Rightarrow (Q \supset P \wedge Q)) \texttt{ by } \ell \texttt{ with } \epsilon)\end{array}$$

[4] Assuming $\supset_I$ and $\wedge_I$ are already in scope

Note: The graphical editor for GENTZEN will display these proof objects in a much more visually appealing way, using vertical rules of inference and proof trees where possible.

2.3 Proof Checking

Most interactive proof assistants, such as ISABELLE or COQ, structure proofs as a linear series of tactic applications to an invisible, temporal *proof state*. The "proof" that is written is not actually a proof *object*, but a rough series of instructions for *creating* one. Many finer details, such as what instantiating assignments to use when applying rules, are inferred automatically by the prover as it processes the proof script, usually using unification or similar techniques.

This design decision makes sense, considering the use cases for which ISABELLE and COQ are designed: In many large-scale verification efforts, the proof objects themselves become intractably large. For our use-cases though, the added clarity of an explicit proof object is of great benefit, and scalability is not quite so important.

Manually writing out a proof of this nature, including all variable assignments, is an incredibly laborious task. To ease this burden, we add a unification engine to infer assignments, but expose it to the user as an editor feature, rather than as a proof tactic.

2.3.1 Unification

Unification is the problem of taking a set of equality constraints of the form $e_1 \sim e_2$, and devising a substitution σ, which replaces only free schematic unification variables, such that $\sigma e_1 = \sigma e_2$ for each of the constraints in the set.

Now, because our term language is a lambda calculus, the simple, decidable first-order unification algorithm of Robinson (Robinson, 1965) is not sufficient. Instead, we use the semi-decidable, *pre-unification* algorithm of Gérard Huet (Huet, 1975).

For a full introduction to higher order unification, de Moura et al. relate Huet's method to a method based on explicit substitutions, and provide an approachable introduction to the topic (de Moura *et al.*, 2008).

As our terms are automatically in $\beta\eta$ normal form, we can syntactically group equations into three broad categories:

- *Rigid-Rigid* - Where the head of both sides of the equation are a bound variable.
- *Flex-Rigid* - Where the head of one side of the equation is a unification variable.
- *Flex-Flex* - Where the heads of both sides of the equation are unification variables.

Huet's algorithm is based on the interleaving of two procedures:

1. **simpl**, which breaks down *rigid-rigid* equations into *flex-rigid* or *flex-flex* equations.
2. **match**, which produces a number of possible substitutions for *flex-rigid* equations, based on two rules: *imitate*, which attempts to substitute the unification variable for a term involving the head of the rigid term, and *project*, which attempts to reorder the parameters passed into the head to make the flexible term more closely resemble the rigid term.

Imagine a nondeterministic computation tree which starts with the original constraint set, runs **simpl**, and then nondeterministically branches on each substitution returned from **match**, applying the substitution and repeating the process, failing if an unsatisfiable constraint is produced, and succeeding if all equations are *flex-flex*. This tree is known as a *unification tree*. The desired substitution σ is simply the composition of each substitution applied along a successful branch.

As this unification is only semi-decidable, there will be at least one finite path to a *success node* if the constraints are satisfiable, but the tree may have branches that are infinitely long, and never reach either success or failure.

As we have implemented this algorithm in Haskell, a non-strict language, we can conveniently represent this unification tree simply as a tree, which we traverse in breadth-first order up to a depth limit, looking for success nodes. The depth limit can be customised by the user.

Note that Huet's algorithm is merely an method for *pre*-unification, and so *flex-flex* equations are *not* resolved by this algorithm. While any *flex-flex* equation is always unifiable, searching for a solution to multiple simultaneous *flex-flex* equations is difficult indeed. Therefore, our approach, much like that of Isabelle and similar, is to simply *delay* solving nontrivial *flex-flex* constraints, and leave them as unresolved in the proof state. These constraints are then added in to later unification problems, which may result in a substitution that changes the constraint into *flex-rigid* or *rigid-rigid* form, and therefore makes it soluble by the **simpl** and **match** procedures documented above.

2.3.2 Semantics

Figure 6 outlines the proof checker in the style of an operational semantics, however each transition is a *user action*, either to move forward, backward, or to apply a rule with unification. For this reason, this is an *interactive* semantics that describes the changes to the proof state based on the user's behaviour.

States are either of the form $\Gamma;\ \Delta;\ \xi;\ \Phi \rhd \pi$ where Γ is a type environment for all variable introductions in the context stack Φ, Δ is a similar *rule* environment for all known facts (assumptions or lemmas), ξ is a set of unresolved *flex-flex* equations, and π is some proof statement that is currently being written or edited; or of the form $\Gamma;\ \Delta;\ \xi;\ \Phi \lhd \pi, \rho$ where the proof statement π has been checked as a valid proof of the rule ρ, returning into the context waiting in Φ.

This style of presentation, based on Huet's Zipper (Huet, 1997), easily gives rise to a design for a *structural editor*, which edits abstract syntax rather than text. In Isabelle or Coq, the user is equipped with a *cursor* into the proof script, *after* which the code is editable, but before which the code is read-only, as modifying this "earlier" code without re-checking could make the proof state incoherent with respect to the proof script. Similarly, our proof editor can consider any statement syntax that is part of the *context stack* to be inviolate and read-only, and any proof statements to the right of the $\rhd$ to be freely editable.

In this way, applying a rule to an unsolved goal becomes an *editing* action. A known rule is selected by the user, which is unified with the current goal, and the instantiation assignment (a subset of the unifier) is saved into the proof code. This means that, when undertaking normal proof checking, we merely need to check that the only constraints that remain after the instantiating assignment is applied are part of the unsolved constraint set ξ, which is achieved via **simpl**. Thus, unification is not part of the *proof checking* process, merely part of the *proof writing* process.

3 Future Work

3.1 Pattern Unification

Miller's pattern unification (Miller, 1990) is a subproblem of higher order unification where all terms are of a specific *pattern* form, where the position in which schematic unification variables can occur is restricted, eliminating the need for a nondeterministic search, and providing most general unifiers. This does not cover all possible unification cases, however it offers a promising means to solve a broad swathe of these outstanding *flex-flex* constraints, as well as find solutions to other unification problems more simply. Recent versions of Isabelle use pattern unification exclusively, and delay any non-pattern terms. Right now, pattern unification is not implemented in Gentzen, but it our intention to use Miller's method to improve the unification engine in the near future.

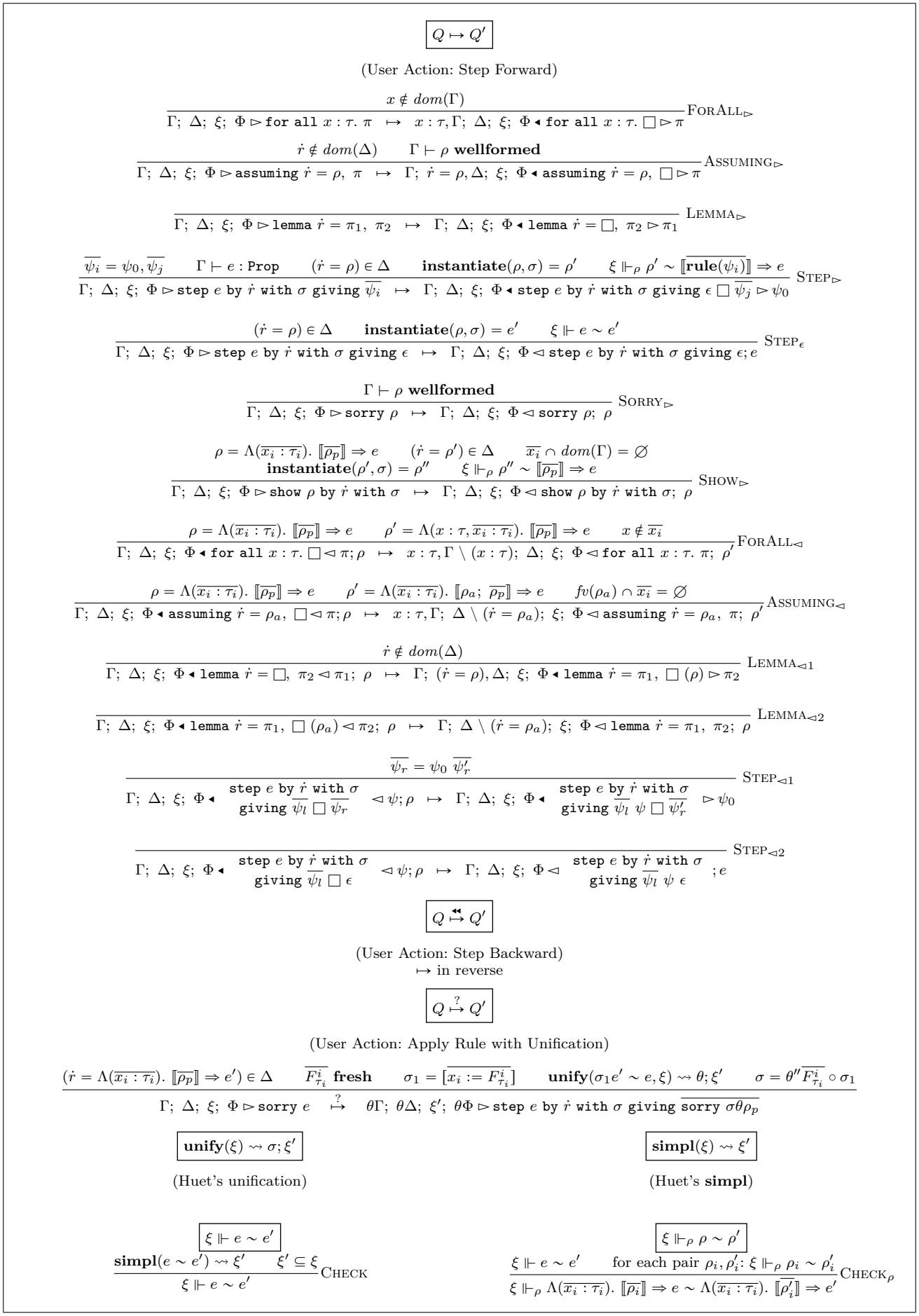

Fig. 6: Interactive step semantics for proof checking

3.2 Graphical Editor

The first release of Gentzen will only feature a command-line driven structural editor, with limited graphical or visual capability. Using the WebKit as a GUI toolkit, we are currently working on a full-featured graphical editor that will make it possible to use Gentzen in an educational context.

3.3 Data types and other extensions

Gentzen currently has no support for algebraic data types. Basic support could be added, and restricted recursion schemes and structural induction could be allowed, based on their Church encoding or catamorphism. Most, if not all of what is necessary to express data types and their associated lemmas is already present in our meta-logic. The meta-logic of Isabelle is very similar, and highly minimal, with a variety of extensions are built upon its foundation, including a very flexible data types extension that merely requires proof of monotonicity (Paulson, 2000). It is likely worth investigating how to make Gentzen's proof language similarly extensible, to enable modular additions to the proof language to express things like data types and inductive predicates.

4 Conclusion and Related Work

Besides other prominent theorem provers like Coq (Coq Development Team, 2011), and more importantly Isabelle (Nipkow *et al.*, 2002), LCF (Milner, 1972) and HOL, which have very similar internal meta logics to Gentzen, there are several other projects that focus more on pedagogical concerns than the concerns of large scale verification projects:

- Hilbert is a prototype interface for a simple, string-matching based propositional theorem prover. We wrote Hilbert at the beginning of this project to investigate the feasibility of Gentzen-tree style graphical interfaces for theorem proving.
- Logitext is a web-based interactive theorem prover for logical statements using the sequent calculus, developed by Edward Yang at MIT. It also visualises proofs using Gentzen trees, however it is restricted to first-order logic, and does not support defining custom rules or connectives.
- JAPE is a Java-based theorem prover that supports Fitch-style proof diagrams, and first-order logic in natural deduction or sequent calculus.
- Pandora is another Java-based natural-deduction theorem prover which uses its own diagram layout, using nested boxes, and first order logic.

To our knowledge, Gentzen will be the first proof assistant based on a *higher order logic* designed for educational purposes. The presence of a full blown higher order logic enables proof work done in Gentzen to be of a much higher degree of sophistication than comparable first-order tools, which are not sufficient for the kinds of use-cases outlined by Benjamin Pierce in (Pierce, 2009). For example, expressing an induction principle or a generalised elimination rule requires variables to stand for propositions, not just objects, which makes higher-order logic a necessity.

By developing Gentzen, we hope to make it possible for computer science students to study rigorous reasoning and develop their reasoning skills. This will also help to inject mathematical logic and formal reasoning back into Computer Science classrooms, where such concepts have been slowly disappearing, without placing an onerous burden upon teaching staff.

Acknowledgements. I would like to thank Andrea Vezzosi, Arseniy Alekseyev and the UNSW PLS group for their assistance in teaching me higher order unification. I would also like to thank my supervisors and assessors, who have been very gracious to agree to my change of project at the last minute.

References

1 Richard S Bird, and Ross Paterson. (1999). de Bruijn Notation as a Nested Datatype. *Journal of functional programming*, **9**(1), 77–91.

2 Alonzo Church. (1940). A Formulation of the Simple Theory of Types. *The journal of symbolic logic*, **5**(2), 56–68.

3 The Coq Development Team. (2011). *The Coq Proof Assistant.* Reference Manual. INRIA: INRIA.

4 Thierry Coquand. (1986). An Analysis of Girard's Paradox. *Pages 227–236 of: In Symposium on Logic in Computer Science.* IEEE Computer Society Press.

5 Nicolaas Govert de Bruijn. (1972). Lambda Calculus Notation with Nameless Dummies: a Tool for Automatic Formula Manipulation with Application to the Church-Rosser Theorem. *Indagationes mathematicae (elsevier)*, **34**, 381–392.

6 Flávio L C de Moura, Mauricio Ayala-Rincón, and Fairouz Kamareddine. (2008). Higher-Order Unification: A structural relation between Huet's method and the one based on explicit substitutions. *Journal of applied logic*, **6**(1), 72–108.

7 Esdger Dijkstra. (1988). *On the Cruelty of Really Teaching Computer Science.*

8 Gerhard Gentzen. (1935). Untersuchungen über das logische Schließen. *Mathematische zeitschrift*, **39**.

9 W A Howard. (1980). *The Formulae-as-Types Notion of Construction.* Letters To H.B. Curry: Essays on Combinatory Logic.

10 Gérard P Huet. (1975). A Unification Algorithm for Typed lambda-Calculus. *Theory of computer science*, **1**(1), 27–57.

11 Gérard P Huet. (1997). Functional Pearl: The Zipper. *Journal of functional programming*, **7**(5), 549–554.

12 Simon Marlow Ed. (2010). *Haskell 2010 Language Report.* Tech. rept.

13 Dale Miller. (1990). A logic programming language with lambda-abstraction, function variables, and simple unification. *Extensions of Logic Programming. Springer Lecture Notes in Artificial Intelligence.*

14 Robin Milner. (1972). *Logic for Computable Functions: description of a machine implementation.* Tech. rept. Stanford, CA, USA.

15 Tobias Nipkow, Lawrence C Paulson, and Markarius Wenzel. (2002). *Isabelle/HOL: A Proof Assistant for Higher-Order Logic.* LNCS, vol. 2283. Springer.

16 Ulf Norell. (2008). Dependently Typed Programming In Agda. *AFP 08: Sixth International Summer School on Advanced Functional Programming.*

17 Lawrence C Paulson. (2000). A Fixedpoint Approach to (Co)Inductive and (Co)Datatype Definitions. *Proof, language, and interaction:essays in honour of robin milner*, 187–211.

18 Benjamin Pierce. (2009). Lambda, the ultimate TA: using a proof assistant to teach programming language foundations. *Pages 121–122 of: Proceedings of the 14th ACM SIGPLAN international conference on Functional programming.* New York, NY, USA: ACM.

19 John Alan Robinson. (1965). A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the acm*, **12**(1), 23–41.