Proof Weaving

Anne Mulhern [1]

Computer Sciences Department
University of Wisconsin-Madison
Madison, WI USA

[1] Email: `mulhern@cs.wisc.edu`

Automated proof assistants provide few facilities for incremental development. Generally, if the underlying structures on which a proof is based are modified, the developer must redo much of the proof. Yet incremental development is really the most natural approach for proofs of programming language properties [5, 12]. We propose "proof weaving", a technique that allows a proof developer to combine small proofs into larger ones by merging *proof objects*. We automate much of the merging process and thus ease incremental proof development for programming language properties.

To make the discussion concrete we take as an example the problem of proving type-soundness by proving progress and preservation [17] in Coq [3, 7]. However we believe that the methods can be generalized to other proof assistants which generate proof objects, and most directly to those proof assistants which exploit the Curry-Howard isomorphism in representing proof terms as λ-terms [16], e.g. Isabelle and Minlog.

We rely on the proof developer to initially prove type-soundness for "tiny" languages. Each of these languages encapsulates a single well-defined programming feature. For example, a tiny language of booleans can be restricted to the terms *True*, *False*, and *If* and their accompanying typing and evaluation rules. Tiny languages have relatively small and easily developed proofs of type-soundness.

Having developed a repository of small languages along with their associated soundness proofs, the developer builds larger languages by combining small languages in a pairwise progression. Proofs are woven together using a multiphase approach. In the first phase, we rely explicitly on the fact that the proofs are developed using induction on the structure of the terms of the language[2]. The two proofs being merged thus have a similar structure; each proof is the application of the induction principle for the terms of the language to subproofs corresponding to each term. We extract these subproofs from the already existing proofs for each tiny language and apply the induction principle for the combined language to each extracted proof.

Generally there will not be a completed proof at the end of the first phase. Each subproof may contain transparent dependencies[3] on parts of the language in the same way that the whole proof contains a transparent dependency on the structure of terms. In the second phase, we rewrite each term that contains a transparent dependency introducing place holders, i.e., "holes", for each constructor not in the original term. We extract a proof template from existing proof terms that share the same context as the desired term and use the template to generate a likely subproof for the "hole". If the system fails to find a valid subproof, we leave the hole "as is". If there are any unproven subgoals remaining after the second phase we present them to the developer to prove using the tactics of the automated proof assistant.

[2] Informally the proofs are by induction on the structure of typing derivations [12], but it is necessary to perform induction on the structure of terms in order to determine which typing rules are applicable.

[3] A transparent dependency [4] between a proof term T and an inductive type I occurs when T has a dependency with an induction principle of I or if a case analysis on type I is performed in T.

Discussion

Motivation

Programming language properties are difficult to prove formally. Anybody who has used an automated proof assistant is familiar with the difficulty of expressing a theorem and selecting the appropriate tactics to prove the theorem. In proving properties of programming languages the proof developer may have to treat separately numerous distinct cases. Moreover, there is the likelihood that the language may be extended in which case the original proof will no longer be correct.

Automated proof assistants provide few facilities for extending and modifying an existing proof. Generally, the proof developer must rely on experience and intuition to decide where to introduce new tactics or modify existing ones so that the proof can be completed. A single extension may require that the proof be changed in many places; the proof itself may increase in size by an order of magnitude.

Programming language properties are generally proved by induction, whether it be on the terms, the types, or the typing rules of the language. Additional terms, types, or typing rules generally leave the overall structure of a proof unchanged. Yet the addition of certain features may require the entire proof to be fundamentally modified. For example, if a functional language is augmented with references, it becomes necessary to represent the store throughout a proof. Properties which present no difficulty in an informal proof may be quite difficult to formalize in a tractable way. For example, a number of solutions for representing binders exist [1] but each one has some drawback which makes its use difficult in a formal proof.

Languages are assembled according to certain organizing principles. TinkerType [10] exploits these relationships to automatically generate interpreters for various λ-calculi. In the TinkerType system, typing and evaluation rules are clauses, these clauses are associated with features and the dependencies among the features are encoded as a graph. The user specifies a set of features; TinkerType calculates any necessary dependent features, instantiates the typing and evaluation rules associated with all the features, and synthesizes an interpreter for the language.

What we propose is essentially a TinkerType for developing formal proofs. *Proof weaving* is the underlying technique that allows us to automatically synthesize subproofs by taking advantage of the regular structure of programming language proofs.

Technical Discussion

The developer must specify each tiny language in a uniform way so that the languages can be combined easily. In our work, we express everything, including syntax of terms, evaluation rules, and typing rules using Coq's Inductive types. The union of two languages is the union

```
Inductive ty : Set :=
 | TyBool : ty
 | TyUnit : ty
.

Inductive term : Set :=
 | TmTrue : term
 | TmFalse : term
 | TmIf : term -> term -> term -> term
 | TmUnit : term
.

Inductive isval : term -> Prop :=
 | VTrue : isval TmTrue
 | VFalse : isval TmFalse
 | VUnit : isval TmUnit
.
```

```
Inductive eval : term -> term -> Prop :=
 | EIfTrue : forall (t2 t3 : term), eval (TmIf TmTrue t2 t3) t2
 | EIfFalse : forall (t2 t3 : term), eval (TmIf TmFalse t2 t3) t3
 | EIf : forall (t1 t1' t2 t3 : term),
    eval t1 t1' -> eval (TmIf t1 t2 t3) (TmIf t1' t2 t3)
.

Inductive typeof : term -> ty -> Prop :=
 | TTrue : typeof TmTrue TyBool
 | TFalse : typeof TmFalse TyBool
 | TIf : forall (t1 t2 t3 : term) (T : ty),
    typeof t1 TyBool ->
      typeof t2 T ->
        typeof t3 T ->
          typeof (TmIf t1 t2 t3) T
 | TUnit : typeof TmUnit TyUnit
.
```

Fig. 1. The language of booleans defined in Coq. The language is being combined with the language of *Unit*. Constructors taken from the language of *Unit* are in bold type.

of the constructors for each Inductive type.

A tiny language need not even be useful. The language consisting of the single term *isZero*, a function that takes a natural number and returns true if the natural number is zero but otherwise false is degenerate since no terms can be built; however a proof of type-soundness is easily developed. This proof is easily combined with a proof of type-soundness for the language of booleans and the language of natural numbers.

Example

We define the very simple language of booleans combined with *Unit* in Figure 1[4]. We use Coq's Inductive types to define the typing and evaluation rules as well as the types and terms of the language.

Recall that to prove progress we must prove that if a term is typable in the language then either the term is a value or a step can be taken in the evaluation. The subproof corresponding to *TmFalse* has the form

```
(fun _ : exists x : ty, typeof TmFalse x =>
  left (exists x : term, eval TmFalse x) VFalse)
```

Since *TmFalse* is a value it is only necessary to use the proof that it is a value, *VFalse*. The '_' indicates that the hypothesis that *TmFalse* has a type is not employed in the proof. The subproofs for *TmTrue* and *TmUnit* follow the same pattern, since each is a value. On the other hand, a well-typed term constructed using *TmIf* can always take a further step of evaluation. The subproof for *TmIf* uses the inductive hypothesis as well as the hypothesis that the term is well-typed and is considerably more involved. The fragment shown in Figure 2 is a small part of the subproof for *TmIf*. We assume we have already a proof of progress for the language of booleans alone. The proof has been modified so that

[4] The definitions of the terms, types, typing rules and evaluation rules for the language of booleans are taken directly Figure 3-1, Figure 8-2 and Figure 11-2 of "Types and Programming Languages" [12]

```
right (isval (TmIf tm1 tm2 tm3))
 match H with
 | ex_intro x H0 =>
   match
     H0 in (typeof t t0)
     return
       (t = TmIf tm1 tm2 tm3 ->
        t0 = x -> exists x0 : term, eval (TmIf tm1 tm2 tm3) x0)
   with
   | TUnit => _ : (TmUnit = TmIf tm1 tm2 tm3 -> TyUnit = x ->
        exists x0 : term, eval (TmIf tm1 tm2 tm3) x0)
   | TTrue =>
       fun (H1 : TmTrue = TmIf tm1 tm2 tm3) (_ : TyBool = x) =>
       False_ind (exists x0 : term, eval (TmIf tm1 tm2 tm3) x0)
         (eq_ind TmTrue
            (fun e : term =>
             match e with
             | TmTrue => True
             | TmFalse => False
             | TmIf _ _ _ => False
             | TmUnit => False
             end) I (TmIf tm1 tm2 tm3) H1)
   | TFalse =>
       fun (H1 : TmFalse = TmIf tm1 tm2 tm3) (_ : TyBool = x) =>
       False_ind (exists x0 : term, eval (TmIf tm1 tm2 tm3) x0)
         (eq_ind TmFalse
            (fun e : term =>
             match e with
             | TmTrue => False
             | TmFalse => True
             | TmIf _ _ _ => False
             | TmUnit => False
             end) I (TmIf tm1 tm2 tm3) H1)
```

The type of each matching subproof. t and t0 are instantiated with appropriate values for each subproof.

Proof that the typing rule TTrue cannot be applied

Proof that the typing rule TFalse cannot be applied

Fig. 2. A fragment of a proof of progress for the language of booleans and *Unit*. The hole corresponding to *TUnit* is annotated with its type which can be inferred from the `return` value of the match clause.

the added *TmUnit* constructor appears in the *match* statement. The abstract syntax trees for the subproof corresponding to *TmTrue* and the subproof corresponding to *TmFalse* are identical. The constants *False_ind*, *eq_ind*, *I* and *TmIf* also occur at corresponding locations in both subproofs. The variables *H1*, *tm1*, *tm2*, and *tm3* correspond exactly.

The reason for this strong similarity is that the subproof corresponding to *TTrue* and the subproof corresponding to *TFalse* are proofs that the typing rules *TTrue* and *TFalse* can not be applied to a term constructed using *TmIf*. In an informal proof a statement such as "*TIf* is the only typing rule that applies,..." would be all that was required, however, in a formal proof we must prove for each rule separately that it does not apply. Of course, the typing rule *TUnit* does not apply either, so we would expect its subproof to be similar to the subproofs for *TTrue* and *TFalse*.

The alert reader will have noticed that we have already filled in some holes; the anonymous function in each subproof returns `False` in the *TmUnit* case. We chose `False` arbitrarily; any proposition at all will satisfy the typechecker.

Figure 3 shows the template constructed from the subproofs corresponding to $TTrue$ and $Tfalse$. The holes in the template are marked with '?'s. We leave holes wherever there is

```
fun (H1 : ?0 : term = TmIf tm1 tm2 tm3) (_ : ?1 : ty = x) =>
False_ind (exists x0 : term, eval (TmIf tm1 tm2 tm3) x0)
  (eq_ind ?2 : term
    (fun e : term =>
     match e with
     | TmTrue => ?3 : Prop
     | TmFalse => ?4 : Prop
     | TmIf _ _ _ => False
     | TmUnit => ?5 : Prop
     end) I (TmIf tm1 tm2 tm3) H1)
```

Fig. 3. A proof template. The holes are annotated with their types.

a constructor from our language definition, the two subproofs disagree, or we have chosen a value arbitrarily as discussed above. Using the type of the subproof for $TUnit$ we can immediately fill in ?0 with $TmUnit$ and ?1 with $TyUnit$.

At this point we must find an assignment for the other holes. We use a type reconstruction algorithm to infer the values for ?2 and ?5. In this case we find that ?2 $=$ $TmUnit$ and ?5 $=$ *True*. Again, any proposition will do for ?3 and ?4 so we choose them arbitrarily. The term typechecks, so we are done. Figure 4 shows the proof fragment from Figure 2 including the synthesized term for $TUnit$. If $TmUnit$ were a constructor with arguments or if $TUnit$ were a typing rule with hypotheses the correct subproof would still be found by our algorithm.

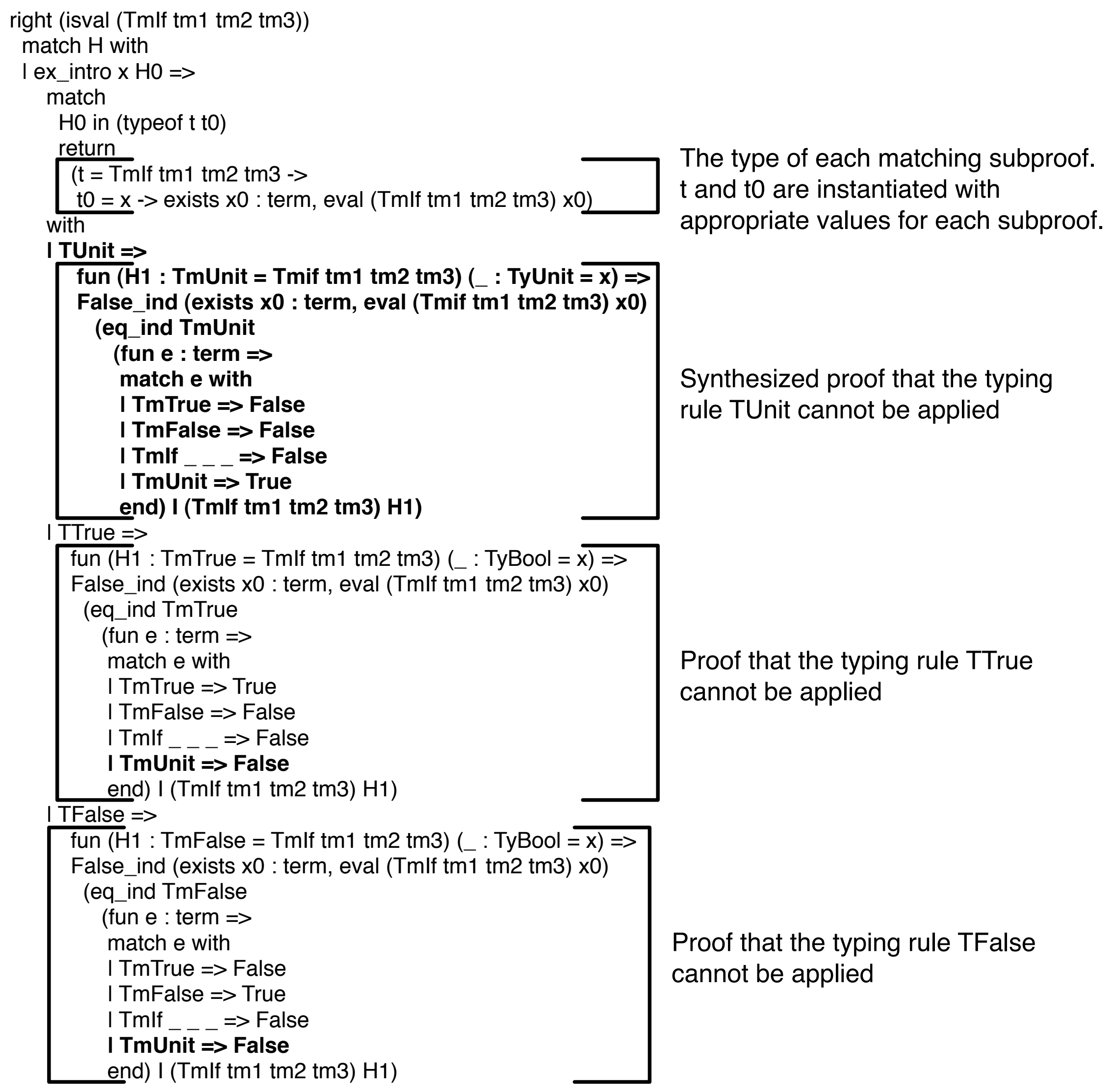
Comparison with the use of tactics

Tactics, although an excellent tool for interactive use, are not a good choice as the basic building blocks for automation. In the above example, a skillful user might observe that two of the subgoals can be solved by the sequence of tactics `intros; simplify_eq in H.` and conclude that most of the subgoals could be solved in the same way. In anticipation of additions to the language the user would construct the sequence of tactics thus:

```
...; try (intros; simplify_eq in H).
(* additional tactics for the case where the typechecking rule does apply *)
```

But this approach is extremely brittle. First, the hypothesis `H` must be named in the `simplify_eq` tactic. Should the name of the hypothesis change the tactic will fail. A more sophisticated tactic, which searches the available hypotheses for one of the correct form and applies the `simplify_eq` tactic to the chosen hypothesis, could be built from existing tactic primitives. But this requires much additional effort on the part of the proof developer. Second, an additional typing rule that does apply to $TmIf$ could be added to the language. In this case, subsequent tactics may be ordered wrongly and the proof is almost certain to fail.

The developer may choose instead to write a problem specific tactic, `eliminate_typing_rules`, and incorporate it into the library of tactics available to the user. But this approach is also quite brittle. It is difficult for the developer to anticipate underlying changes in the structure of the language. In the example above, typing rules are quite straightforward, depending only on the structure of the terms. It is unreasonable to assume that the developer will be



```
right (isval (TmIf tm1 tm2 tm3))
  match H with
  | ex_intro x H0 =>
     match
       H0 in (typeof t t0)
       return
         (t = TmIf tm1 tm2 tm3 ->
          t0 = x -> exists x0 : term, eval (TmIf tm1 tm2 tm3) x0)
     with
     | TUnit =>
          fun (H1 : TmUnit = Tmif tm1 tm2 tm3) (_ : TyUnit = x) =>
          False_ind (exists x0 : term, eval (Tmif tm1 tm2 tm3) x0)
            (eq_ind TmUnit
               (fun e : term =>
                match e with
                | TmTrue => False
                | TmFalse => False
                | TmIf _ _ _ => False
                | TmUnit => True
                end) I (TmIf tm1 tm2 tm3) H1)
     | TTrue =>
         fun (H1 : TmTrue = TmIf tm1 tm2 tm3) (_ : TyBool = x) =>
         False_ind (exists x0 : term, eval (TmIf tm1 tm2 tm3) x0)
           (eq_ind TmTrue
              (fun e : term =>
               match e with
               | TmTrue => True
               | TmFalse => False
               | TmIf _ _ _ => False
               | TmUnit => False
               end) I (TmIf tm1 tm2 tm3) H1)
     | TFalse =>
         fun (H1 : TmFalse = TmIf tm1 tm2 tm3) (_ : TyBool = x) =>
         False_ind (exists x0 : term, eval (TmIf tm1 tm2 tm3) x0)
           (eq_ind TmFalse
              (fun e : term =>
               match e with
               | TmTrue => False
               | TmFalse => True
               | TmIf _ _ _ => False
               | TmUnit => False
               end) I (TmIf tm1 tm2 tm3) H1)
```

The type of each matching subproof. t and t0 are instantiated with appropriate values for each subproof.

Synthesized proof that the typing rule TUnit cannot be applied

Proof that the typing rule TTrue cannot be applied

Proof that the typing rule TFalse cannot be applied

Fig. 4. A fragment of a proof of progress for the language of booleans and *Unit*. A subproof has been correctly synthesized for the hole corresponding to *TUnit*.

able to write a tactic that handles typing environments correctly when they are not in the language for which the tactic was originally developed. Furthermore, developing a tactic and incorporating it into the automated proof assistant's environment is a considerable effort. And this solution does not address the problem of additional typing rules that match *TmIf* any better than the previous solution.

Designing tactic scripts in anticipation of future additions requires foresight. The developer must anticipate what the future additions might be and where they might occur. Proof weaving, on the other hand, is a mechanism that responds to additions, whatever they may turn out to be. Proof weaving is far more general than any problem specific tactic. The developer need only provide examples which are sufficiently similar to the subgoals that need to be proved and the proof weaver will supply the necessary subproof.

Related Work

Reuse of proofs and theorems has been an active area for many years [2, 4, 6, 8, 9, 11, 13, 14]. *Proof abstraction* [8, 9, 13] is concerned with generalizing a proof that works in a specific case to one that works in a more general case. For example, a proof that multiplication on the natural numbers is commutative can be abstracted to a proof that any binary operation that is symmetric and left-associative must be commutative [13]. This sort of abstraction is ubiquitous in mathematics as well as in computer science; a familiar example is the data structure that is parameterized on the type of its elements. However, this approach is not a good fit for proofs of programming language properties which, as noted above, generally proceed by induction.

Boite's work on *proof extension* [4] is concerned with reusing proofs when additional constructors are added to inductive types and is much closer in spirit to our work. There are two significant ways in which our work differs. In the first place, we combine two languages by taking the union of the constructors of their inductive types while Boite adds constructors to an existing inductive type, thereby forming an extended type. Boite's approach enforces an extension relationship much like the inheritance relationship of object oriented languages. Using this approach the user must choose whether the language of natural numbers and booleans is an extension of the language of natural numbers or an extension of the language of booleans though neither relationship is superior. In our approach the constructors from each language are equal citizens in the combined language. Furthermore, where Boite leaves "holes" corresponding to every new constructor, an approach that can result in a vestigial proof when constructors have been added to many types, we synthesize proofs for as many of the subgoals as possible, ideally leaving only the interesting work to the developer.

Conclusion

We introduce a new technique for proof reuse, *proof weaving*. This technique is quite general but it is particularly useful when the underlying definitions on which a proof is built may be extended and when the structure of a proof is quite repetitive. Proofs of programming language properties are likely to have both these characteristics; hence this technique is particularly suitable.

References

[1] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The POPLmark Challenge. In *Proceedings of TPHOLs 2005: the 18th International Conference on Theorem Proving in Higher Order Logics (Oxford), LNCS 3603*, August 2005.

[2] Gilles Barthe and Olivier Pons. Type isomorphisms and proof reuse in dependent type theory.

In *FoSSaCS '01: Proceedings of the 4th International Conference on Foundations of Software Science and Computation Structures*, pages 57–71, London, UK, 2001. Springer-Verlag.

[3] Yves Bertot and Pierre Casteran. *Interactive Theorem Proving and Program Development : Coq'Art: The Calculus of Inductive Constructions*, volume XXV of *Texts in Theoretical Computer Science*. Springer, 2004.

[4] Olivier Boite. Proof reuse with extended inductive types. In Slind et al. [15], pages 50–65.

[5] Kim B. Bruce. *Foundations of Object-oriented Languages: Types and Semantics*. MIT Press, 2002.

[6] Joshua E. Caplan and Mehdi T. Harandi. A logical framework for software proof reuse. In *SSR '95: Proceedings of the 1995 Symposium on Software reusability*, pages 106–113, New York, NY, USA, 1995. ACM Press.

[7] The Coq Proof Assistant. http://coq.inria.fr/.

[8] Amy Felty and Douglas J. Howe. Generalization and reuse of tactic proofs. In *LPAR '94: Proceedings of the 5th International Conference on Logic Programming and Automated Reasoning*, pages 1–15, London, UK, 1994. Springer-Verlag.

[9] Einar Broch Johnsen and Christoph Lüth. Theorem reuse by proof term transformation. In Slind et al. [15], pages 152–167.

[10] Michael Y. Levin and Benjamin C. Pierce. Tinkertype: A language for playing with formal systems. *Journal of Functional Programming*, 13(2), 2003. A preliminary version appeared as an invited paper at the *Logical Frameworks and Metalanguages Workshop (LFM)*, June 2000.

[11] Z. Luo. Developing reuse technology in proof engineering. In *Proceedings of AISB95, Workshop on Automated Reasoning: bridging the gap between theory and practice,*, Sheffield, U.K., April 1995.

[12] Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.

[13] Olivier Pons. Proof generalization and proof reuse.

[14] Axel Schairer, Serge Autexier, and Dieter Hutter. A pragmatic approach to reuse in tactical theorem proving. *Electr. Notes Theor. Comput. Sci.*, 58(2), 2001.

[15] Konrad Slind, Annette Bunker, and Ganesh Gopalakrishnan, editors. *Theorem Proving in Higher Order Logics, 17th International Conference, TPHOLs 2004, Park City, Utah, USA, September 14-17, 2004, Proceedings*, volume 3223 of *Lecture Notes in Computer Science*. Springer, 2004.

[16] F. Wiedijk. The seventeen provers of the world. 2005.

[17] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, 1994.