POLYTYPIC PROVING

by

Anne Mulhern

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2010

ACKNOWLEDGMENTS

I wish to thank Charles Fischer and Ben Liblit, sine quibus non.

Others who have contributed significantly to the progress of this thesis are:

- The people who have attended my Coq seminars.
- The members of the CBI team, past and present.
- The members of the administrative staff in the department, past and present.

I would like to thank them as well.

I would like to thank the members of my committee and Robert Wilson.

I would like to thank Jon Giffin for giving me honest, clear, and unaffected answers to puzzling C++ questions, among other things.

I would like to thank Tristan Ravitch for introducing me to presentations with Beamer, among other things.

I would like to thank Will Benton for many things, including providing the LaTeX infrastructure for formatting this document.

I would like to thank Kim Nichols.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

POLYTYPIC PROVING

Anne Mulhern

Under the supervision of
Professor Charles Fischer
and
Assistant Professor Benjamin Liblit
At the University of Wisconsin-Madison

Formal methods are a class of techniques for automatically verifying software correctness. They are a common topic in computer science research. However, they are less well-known in software development and in undergraduate computer science education.

As society's reliance on software increases so does the potential severity of the consequences of software failure. Formal methods are the surest way of guaranteeing software correctness. Moreover, the study of formal methods leads to better informal understanding of software correctness and thus to better software.

Unfortunately, the study and use of formal methods can be difficult and this impedes their adoption in software engineering and the computer science curriculum. Coq is an automated proof-assistant for developing certified programs and proofs. It is powerful and expressive and has been used in a large variety of significant developments.

Experienced developers are accustomed to picking up new languages rapidly. Unfortunately, it is difficult for a new user to learn the use of any proof-assistant with anything like the same rapidity. Furthermore, developers have become accustomed to sophisticated IDEs with tools for refactoring and for auto-generating code. Very little such support is available for proof-assistants. Coq suffers from both these drawbacks.

Our work addresses these problems in several ways. We provide two alternative approaches that facilitate general induction and recursion. We

demonstrate methods for enhancing structural recursion principles to increase their generality and transparency. We demonstrate graphical techniques for improved proof visualization and an impact analysis tool for predicting the consequences of a proof refactoring.

Charles Fischer

ABSTRACT

Formal methods are a class of techniques for automatically verifying software correctness. They are a common topic in computer science research. However, they are less well-known in software development and in undergraduate computer science education.

As society's reliance on software increases so does the potential severity of the consequences of software failure. Formal methods are the surest way of guaranteeing software correctness. Moreover, the study of formal methods leads to better informal understanding of software correctness and thus to better software.

Unfortunately, the study and use of formal methods can be difficult and this impedes their adoption in software engineering and the computer science curriculum. Coq is an automated proof-assistant for developing certified programs and proofs. It is powerful and expressive and has been used in a large variety of significant developments.

Experienced developers are accustomed to picking up new languages rapidly. Unfortunately, it is difficult for a new user to learn the use of any proof-assistant with anything like the same rapidity. Furthermore, developers have become accustomed to sophisticated IDEs with tools for refactoring and for auto-generating code. Very little such support is available for proof-assistants. Coq suffers from both these drawbacks.

Our work addresses these problems in several ways. We provide two alternative approaches that facilitate general induction and recursion. We demonstrate methods for enhancing structural recursion principles to increase their generality and transparency. We demonstrate graphical techniques for improved proof visualization and an impact analysis tool for predicting the consequences of a proof refactoring.

1 INTRODUCTION

1.1 Motivation

Automated theorem proving (Moore, 2004a) is a mature field of study dating back to the 1950s. Over the last few decades numerous automated theorem provers and proof assistants have been developed (Wiedijk, 2006). They are used in industry to prove the correctness of critical software and hardware (Moore, 1999, 2004b); they are used in mathematics to prove significant theorems (Gonthier, 2005); they are used in computer science to prove properties of programming languages and programs (Aydemir et al., 2008; Chlipala, 2006, 2007a,b; Swords and Cook, 2006); they are used in classes to teach logic and computer science (Asperti et al., 2005; Mulhern, 2009; Pierce, 2008). Over the last couple of decades the use of automated theorem provers and proof assistants has increased. More and more, persons with less and less experience in automated theorem proving are expected to make use of theorem provers and proof assistants. The complexity of the projects which experts are addressing has increased. It is desirable to make formal proving more accessible to the novice and more efficient for the expert.

This situation mirrors that of programming. Experts now write much larger and more ambitious programs than they were expected to construct just a few decades ago. More and more novices enter introductory programming courses. To assist the novice simpler or more helpful integrated development environments (IDEs) have been developed (Barnes and Kolling, 2006; Cross and Hendrix, 2007; Dann et al., 2008; Hasker, 1995; Resnick et al., 2009). Sophisticated and powerful IDEs and numerous tools for software analysis (Anderson, 2004; Xu et al., 2009) have been developed to assist the expert programmer.

Today there is an increasing need for proof engineering(Pons, 2000a;

Mulhern et al., 2006) and perhaps integrated proof environments (IPEs). A few tools now exist to make proving more accessible to the novice (Bornat, 2005; Dillinger et al., 2007) and to make proving in certain domains easier for the expert (Aydemir et al., 2009; Chlipala, 2010).

There are hundreds of programming languages in existence today; many of these languages differ from each other in fundamental ways. There are about twenty proof assistants and theorem provers in the world today; almost every one is profoundly different from any other (Wiedijk, 2006). Thus, a technique that facilitates development with one proof assistant might be entirely irrelevant to another.

Formal proof tools may be domain specific (Kalman, 2001) or they may be quite general (Nipkow et al., 2002). They may be fully automatic, in which case they are *theorem provers*. Alternatively, they may require interactive assistance, in which case they are *proof-assistants*. Coq (The Coq Development Team, 2008) is a general purpose proof-assistant that makes explicit use of the Curry-Howard isomorphism. This important result identifies proofs with programs (Wadler, 2000). A lemma is a type in the input language of Coq, the Calculus of Co-inductive Constructions(CoC) (Paulin-Mohring, 1993). A satisfactory proof of that lemma is any program in the CoC that inhabits that type. Such programs are generally large, complex, and uninteresting in themselves; the user builds these proof-programs by judicious use of Coq's automation and tactics rather than by writing them out explicitly. A sequence of tactics is known as a *proof script*.

Paradoxically, while these proof-programs are uninteresting in themselves the method of their construction is the subject of considerably research. Different approaches are advanced by opposing camps (Bertot and Casteran, 2004; Chlipala, 2009; Sozeau, 2007b). The development of useful libraries and tactics for facilitating metatheoretic developments has been ongoing for over half a decade (Aydemir et al., 2005) and the efforts

show no signs of abating (Aydemir et al., 2009). In practice, it should be possible for the user to ignore the inner workings of all these techniques just as it should be possible for a program developer to ignore the inner workings of a compiler. Unfortunately, this is not the case. A developer must consider the compiler only occasionally since compilers do, by and large, generate semantically correct code. However, users of any of these proof-assistant techniques must be aware of their principles at every step.

In developing a proof the user is likely to go through two stages. In the first stage, the user does not know how to prove the desired goal. In this stage, the user proceeds haphazardly. If the lemma she must prove is fairly general, she may hypothesize that a proof already exists in Coq's standard libraries. In this case, she will attempt to use the *auto* tactic or some of its more powerful variants which make use of a Prolog-like decision procedure to solve the goal. Failing that, she will attempt to proceed by structural induction on some term in the goal; once presented with the resulting subgoals she will continue in much the same vein. Eventually, the user will reach the second stage. By proceeding through the first stage, she will have discovered the essential nature of the proof, at least for the particular inductive definitions for which she has developed it. At the second stage she will attempt to clean up the proof to make it more robust, clearer, and simpler.

The tactics that Coq provides are eminently suitable for the first stage of development. They are like probes; the user applies a tactic to a subgoal and gets a response; a new subgoal that gives further information about the nature of the proof. The tactics are less suitable for the second stage; they are restricted in scope and the choice of the correct tactic is often dependent on the intuition of the user. Although there are conflicting approaches to proof development in Coq they are similar in that they rely on sophisticated super-tactics. These tactics are meant to be deployed after an initial induction step to completely solve all remaining goals. The

super-tactics are constructed by gluing together tactics using Coq's *Ltac* language (Delahaye, 2000). The proponents of different schools of proof development all have different super-tactics. These are a wonderful time-saver; unless they fail. When they do fail the user is likely to have no intuition whatsoever as to the cause of the failure.

Much ingenuity has been spent in developing super-tactics and domain-specific libraries for Coq. We address several obstacles to proof development in Coq which have received little attention.

Our solutions are *polytypic*; that is they are generally applicable to inductive definitions with any structure (Gibbons, 2007). Polytypic solutions are valuable because they are generally applicable and because they are robust to changes in the structure of the inductively defined types of the CoC.

1.2 Generalizing General Induction

The CoC is a small language. Syntactically it is extraordinarily simple. On the other hand its type-system is sophisticated enough to express quite interesting properties. Due to its extraordinary simplicity the CoC itself only includes one logical connective, implication. The connectives *and* and *or*, for example, are defined in Coq's standard libraries using the CoC's inductive datatypes which are analogous to the algebraic datatypes of ML or Haskell. Because almost everything is inductively defined induction and recursion are ubiquitous.

Coq provides an *induction* tactic. To use this tactic the user must prepare the goal so that induction will yield the proper induction hypotheses. The user then invokes the *induction* tactic passing the appropriate induction variable as an argument. The *induction* tactic then applies the default induction principle to the goal and the user is presented with subgoals, as many as there are constructors for the corresponding inductive definition.

The default induction principle, which is generated automatically for every inductive definition is a structural induction principle. Unfortunately, structural induction is sometimes insufficient to prove a goal.

The quicksort algorithm (Hoare, 1962) is a familiar example of an algorithm which cannot proceed by structural recursion. A quicksort implementation may be made extremely terse and may be written very rapidly by an experienced ML programmer. However, in Coq, it is necessary to prove that all functions terminate since a single non-terminating function yields an inconsistent logic. A structural recursion principle is always terminating but the quicksort algorithm cannot make use of structural recursion over the list or array which contains the elements to be sorted. Hence, it is necessary to make use of some general recursive approach.

The need for general recursion has been widely recognized and considerable effort and ingenuity has been devoted to the development of general induction and recursion techniques and supporting libraries (Balaa and Bertot, 2000; Bove and Capretta, 2005; Charguéraud, 2009, 2010; Danielsson and Altenkirch, 2009; Megacz, 2007; Sozeau, 2007b).

However, very little study has been devoted to making these techniques programmatically accessible to a user or to generalizing these techniques to proofs where the type of the term on which general induction is to be performed is composed of several mutually inductive types. In Part I we discuss some techniques to address these problems.

We provide a new tactic, *principle*, which populates the proof-context with a requested general recursion principle. This tactic is suitable for approaches to general recursion which makes use of recursion combinators (Balaa and Bertot, 2000; Charguéraud, 2009).

We address the problem of mutually inductive types with two alternative approaches. The first, a dependent type transformation, is described in Chapter 2. In this chapter, we demonstrate that a mutually inductive datatype can be transformed to an equivalent simple inductive datatype

by a syntactic transformation obviating the need for a general recursion principle for mutually inductive types. The second approach allows a developer to specify the structure of a general recursion principle using a Parameterized Context-free Grammar (PCFG). This technique is described in Chapter 5. General recursion and induction techniques often rely on *measures* which are a special case of *uniform* functions which we discuss in Chapter 3 and Chapter 4.

1.3 Extending Structural Induction and Recursion

Even where available Coq's automatically generated structural induction principles may be inadequate. In some cases there exist special vernacular commands which can be used to automatically construct better induction principles but in many cases there are none. Chlipala (Chlipala, 2009) frequently eschews the use of the *induction* tactic in favor of using the *refine* tactic to explicitly write out fixed point functions leaving holes for proof terms that must be later filled in by automation.

Unfortunately, only an expert Coq developer can correctly predict where the holes should be placed within the partial proof. A single misplaced hole will generally result in a bewildering type-error. Moreover, this approach is extremely brittle. Should the inductive definition on which the proof is based be changed every proof constructed in this way must be edited. Software engineering principles suggest that rather than distributing these changes throughout the code it is better to localize them in a single place.

In Part II, we set out to address some of the defects in the autogenerated structural induction and recursion principles. All structural induction and recursion principles are homogeneous, i.e., the induction or recursion principle applies only to a single type. In Chapter 6, we discuss how a

principle may be extended to include a parameter type. The *induction* tactic replaces a single goal with multiple subgoals, one for each principal premise of the principle. The subgoals themselves contain no indication of the principal premise to which they correspond. In Chapter 7 we discuss a simple extension to the automatically generated induction and recursion principles so that the corresponding principal premise becomes part of the proof-context. Variadic types are types with constructors that may contain any number of arguments. Such types are necessary to express certain concepts but are not available in the CoC. In Chapter 8 we discuss a proof developer's idiom for expressing variadic types and making use of them in real developments.

1.4 Lemma Extraction and Proof Analysis

Lemma Extraction

Procedure extraction (Komondoor and Horwitz, 2000, 2003) is a source code transformation which discovers selected statements within a procedure and extracts these statements into a separate procedure. The motivation for this transformation is code reuse and simplification of existing code.

In Coq, the motivation for certain kinds of lemma extraction (Pons, 2000a) is much stronger. Certain expressions, if inlined within a larger proof, will not be accepted by the type-checker. The very same expression, encapsulated within a lemma, and used in the same context, will be accepted by the type-checker.

This odd situation arises due to the distinction between *Prop* and the other sorts, i.e., *Set* and *Type*. To facilitate certified programming, Coq has a mechanism which extracts a formal proof to source code in a target language (Letouzey, 2003, 2004). This extraction mechanism elides expressions with types in *Prop* but preserves expressions in *Set* and *Type*. The

user must put proofs of properties in *Prop* but specifications in *Set* or *Type*. It is forbidden to deconstruct an expression with a type in *Prop* in order to build an expression with a type in *Set* or *Type*.

For this reason, some *match* expressions in the CoC are forbidden because the expression being matched has a type in *Prop*, but the type of the *match* expression itself is not in *Prop*. Very commonly, however, the expression being matched has the property that only one of the cases in the *match* expression fails to yield a contradiction. In this case, it is possible to extract the *match* expression into a function. Because the *match* expression is concealed within the function Coq's type-checker can be satisfied. The result of the extracted function is exactly the additional hypotheses that would be made available in the one case in the *match* expression that does not lead to a contradiction.

In Chapter 9 we describe a tactic, *canonical_inversion*, that we have developed which automatically extracts a suitable lemma whenever possible and applies it in the appropriate context. This tactic is strictly more powerful than the Coq *inversion* tactic that it supersedes. Our methods are novel; we believe that they may be broadly applicable.

Proof Analysis

Coq's support for tasks that are likely to be necessary during the haphazard initial stage of development is exceptional. The user can query loaded libraries using sophisticated pattern matching to discover lemmas that appear relevant. On the other hand, support for examining proof developments in the large or for anticipating the impact of a refactoring step is non-existent.

For example, a lemma is transparently dependent (Bertot et al., 2000; Pons et al., 1998) on an inductive definition if it contains a *match* expression which matches some term in the type defined by that inductive definition. The *match* expression itself has a type, which may also be defined using

an inductive definition. If the type of the *match* expression is in *Prop* then the type of the expression matched must also be in *Prop*. If the inductive definition that defines the type of the *match* expression is translated to *Set* or *Type* the proof will be invalidated since expressions in *Prop* can not be deconstructed to build expressions in *Set* or *Type*. Probably the inductive definition that defines the type of the matched expression will also be translated to *Set* to repair the proof. Of course, this can result in most or all inductive definitions in a proof development being migrated from *Prop* due to the transitive effects of the initial change.

It is difficult and time-consuming for the user to discover whether and how other inductive definitions may be affected through these dependencies. Most proofs are many orders of magnitude larger than the proof scripts that generate them. The user may use Coq's *Print* command to print the body of each of her proofs. However, the bodies are very long and she has no programmatic way to scour the text of the proofs to locate relevant *match* expressions. Coq has a facility for generating an XML representation of its compiled proof files (Asperti et al., 2004, 2000b,a, 2001; Sacerdoti Coen, 2003). Unfortunately, the user experiences as much difficulty scouring the XML files as she does scouring Coq's text output. Under these circumstances, iteratively changing the sorts of inductive definitions and repeatedly re-running Coq to discover broken proof scripts may be the most efficient way to discover these dependencies.

The information necessary to predict the impact of a translation from *Prop* to another sort exists, however it is unavailable to the user. In Chapter 10 we demonstrate a facility that makes this information readily available.

Our facility makes use of the XML representation of Coq developments. This XML representation is easily explored via XPATH queries. We have shown that another use that can be made of this representation is to examine the dependencies among individual lemmas in a given devel-

opment. We present a graphical representation of these dependencies and show how further post-processing of the graphical structure may assist the developer in partitioning the proof and the user in comprehending the proof.

We have demonstrated that the XML representation of proofs accommodates a variety of analyses. In no case does implementing these analyses require an expert understanding of the Coq system. We believe that further exploration of this area may expose the opportunity for other useful analyses and lead to further innovation.

Part I

Generalizing General Induction

Induction is a fundamental proof technique; recursion a fundamental programming technique. Because Coq makes explicit use of the Curry-Howard isomorphism induction and recursion are the same. Conventionally, induction is used to describe the occurrence of a recursive function which inhabits the sort *Prop* while recursion is used to describe the occurrence of a recursive function which inhabits the sort *Set*. In the first case recursive functions are used to describe the construction of proofs; in the second case recursive functions are used to describe the construction of values. Otherwise, Coq makes no distinction among recursive functions.

In Coq, all functions that cannot be shown to terminate under some evaluation order are forbidden. If this were not the case, it would be possible to derive a contradiction. Therefore, Coq must forbid terms that do not have a normal form.

For every inductive definition Coq automatically generates structural induction and recursion principles. Coq enforces the property that either the inductive definition is uninhabitable or the principles are terminating. Consequently, these principles can never result in a contradiction.

However, structural induction and recursion are often inadequate. Consequently, many solutions to the problem of general recursion have been put forward (Balaa and Bertot, 2000; Bove and Capretta, 2005; Charguéraud, 2009; Danielsson and Altenkirch, 2009; Megacz, 2007).

Coq provides tactical support for structural induction. The *induction* tactic selects the appropriate automatically generated structural induction or recursion principle and automatically applies it to the goal. No general induction or recursion principle is made programmatically available to the user in the same way as the structural induction and recursion principles. Without a strong knowledge of the current literature on the subject the user who must proceed by general induction is stymied. Moreover, general induction solutions do not generally address the practical problem of applying the solutions where the type of the induction term is mutually

inductive.

To address this problem we have implemented a tactic, *principle*, described in Chapter 5, which autogenerates a recursion principle based on arguments supplied by the user and inserts that recursion principle in the proof-context. In this way, we give the user programmatic access to a variety of the general recursion principles described in the literature.

We have remarked that the libraries and tactical support for many general induction solutions do not offer any support for mutually inductive datatypes. We provide two alternative solutions to this problem.

In Chapter 2 we describe a syntactic transformation from a mutually inductive type to a simple inductive type that makes use of dependent types. This transformation obviates the need for a general recursion principle over mutually inductive types.

Many general recursion principles make use of a *measure*. Usually, the measure is a special instance of a *uniform* function. Uniform functions are functions which are generic in the sense that every constructor of an inductive type is treated the same. A common example of a uniform measure is a size function which counts the number of constructors in a term. Uniform functions are common in many proof developments. In Chapter 3 we describe an approach for autogenerating uniform functions which is robust in the presence of changes to the definition of the inductive type on which it is defined. In Chapter 4 we discuss measure functions particularly.

An alternative approach to the problem of general recursion on mutually inductive types is described in Chapter 5. We define a specification language for describing the overall form of a general recursion principle and an algorithm for expanding a specification into a general recursion principle for an inductive type with any number of mutually inductive definitions.

2.1 Introduction

Languages with algebraic types generally allow mutually inductive type definitions. Figure 2.1(b) shows an O'Caml mutually inductive definition for a type of trees and forests where forests are composed of trees and trees of forests. This definition is extracted from a corresponding definition in the CoC; Figure 2.1(a) shows the equivalent inductive definition. It is possible to transform such a type into an equivalent singly inductive type by making use of dependent types. Figure 2.1(c) shows the equivalent singly inductive type in the CoC. It is dependent on the simple type `tf_enum`, also shown. It is easy to demonstrate, through an informal argument, that such a transformation yields an equivalent type. In any particular case, it is possible to formally demonstrate a bijection between the sets defined by the equivalent types.

Figure 2.2 shows the relationships between the type `ntree` in Coq, the type `tf` defined by the dependent type transformation and the extracted O'Caml datatypes corresponding to both `ntree` and `tf`. Since the types are defined in different languages it is impossible to formally define a bijection. However, it is still possible to reason informally about the set of objects which the different definitions define. Because dependent types are elided by the extraction mechanism the extracted datatypes are in general less precise than their corresponding CoC types, i.e., they define supersets of the sets defined by the CoC types. Observe that the only bijection in the figure is that between the two CoC datatypes. Since the CoC definition of `ntree` made no use of dependent types there exists an informal bijection between it and its corresponding extracted type. However, in the general case the set it defines must be a subset of the set defined by its extracted type. Since the CoC definition of `tf` is the result of

```
Inductive ntree (A : Set) : Set :=
    node : A -> nforest A -> ntree A
  with nforest (A : Set) : Set :=
    nil : nforest A
  | cons : ntree A -> nforest A -> nforest A
```

(a) A mutually inductive definition of trees and forests in the CoC.

```
type 'a ntree =
  | Node of 'a * 'a nforest
and 'a nforest =
  | Nil
  | Cons of 'a ntree * 'a nforest
```

(b) The definition in Figure 2.1(a) extracted to O'Caml.

```
Inductive tf_enum : Set :=
    tf_tree : tf_enum
  | tf_forest : tf_enum

Inductive tf (A : Set) : tf_enum -> Set :=
    tf_node : A -> tf A tf_forest -> tf A tf_tree
  | tf_nil : tf A tf_forest
  | tf_cons :
      tf A tf_tree ->
      tf A tf_forest ->
      tf A tf_forest
```

(c) The definition in Figure 2.1(a) transformed into an equivalent singly inductive dependent type.

```
type tf_enum =
  | Tf_tree
  | Tf_forest

type 'a tf =
  | Tf_node of 'a * 'a tf
  | Tf_nil
  | Tf_cons of 'a tf * 'a tf
```

(d) The definition in Figure 2.1(c) extracted to O'Caml.

Figure 2.1: Several definitions of trees and forests.

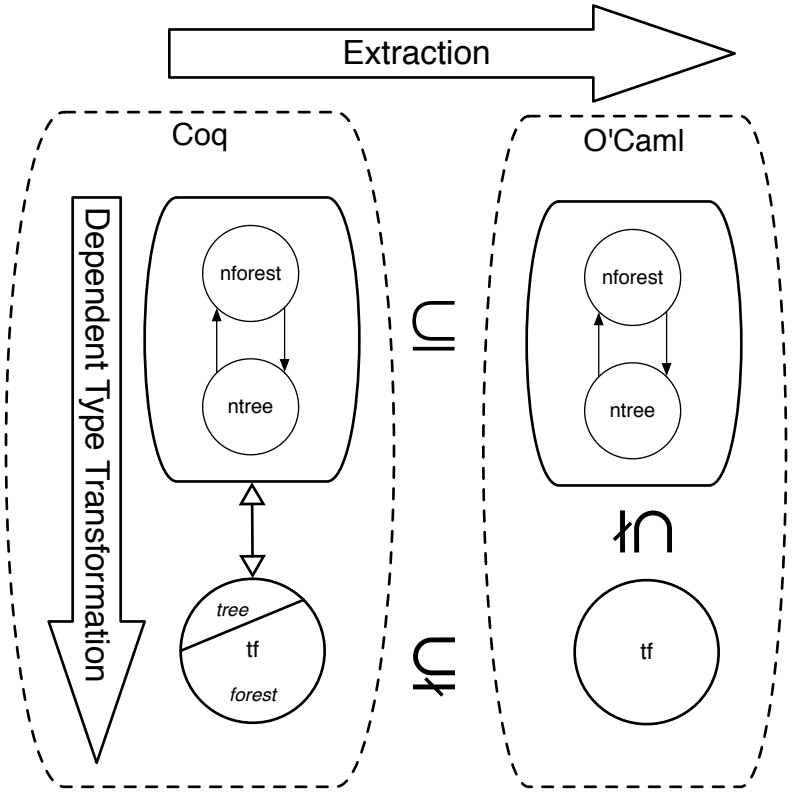


Figure 2.2: Subset relationships of transformed and extracted types for trees and forests.

the dependent type transformation it must always make use of dependent types; its corresponding O'Caml type will always represent a superset of the set it represents.

Because O'Caml does not allow dependent types the extracted type corresponding to `tf_enum` is elided in the extracted version of `tf`. Figure 2.1(d) shows the result of extracting the dependently typed datatype. Note that `tf` has no dependence on `tf_enum`. It is easy to see that the there is no bijection between the two extracted datatypes; the set defined by the extracted version of `tf` is strictly larger than the set defined by the extracted version of `ntree`.

The contributions of this chapter are twofold. First, we demonstrate that while expressing a mutually recursive type as a dependent type may at first seem a retrograde step, the dependently typed representation has advantages in the context of proof development. Second, we demonstrate that the transformation can be done using a relatively simple, syntax-based approach.

2.2 Theory of the Transformation

Demonstrating a Bijection

This transformation is only possible where the mutually recursive type is *homogeneous*, i.e., where each component type has the same type. In the example, `ntree` and `nforest` both had type `Set` $\rightarrow$ `Set` and so the whole type is homogeneous. Had their types been different the single dependent type would have been required to also have two different types, which is impossible.

We give an informal argument to show that the result of the dependent type transformation must always represent an equivalent inductive datatype. We also describe how this can be proved formally within Coq by defining inverse mappings.

Informal Presentation

We present an informal argument that the transformation yields a type which represents an equivalent set. We define the depth of a term in the usual way. We call the type on which the dependent type depends the *enum* type.

The transformation yields several related bijections between the structure of the two representations.

- A bijection between constructors. In the ongoing example the bijection is {`node` $\leftrightarrow$ `tf_node`, `nil` $\leftrightarrow$ `tf_nil`, `cons` $\leftrightarrow$ `tf_cons`}.
- A bijection between types and enum constructors. In the ongoing example the bijection is {`ntree` $\leftrightarrow$ `tf_tree`, `nforest` $\leftrightarrow$ `tf_forest`}.
- A bijection between sets of constructors. Sets of constructors are grouped according to the bijection between types and enum types. In the ongoing example the bijection is {`node` $\leftrightarrow$ `tf_node`, {`nil`, `cons`} $\leftrightarrow$ {`tf_nil`, `tf_cons`}}.

Our argument is by induction over sets of terms of depth at most n. The base case is the set of terms of depth 1 in both representations. Terms of depth 1 can be constructed only using nullary constructors; the bijection between constructors implies a bijection between terms built of nullary constructors. The induction step is from sets of terms of depth at most n to sets of terms of depth at most $n + 1$. Terms of depth $n + 1$ can only be constructed by application of a constructor to at least one term of depth n. We assume that there exists a bijection between the set of all terms of depth `n` in the mutually inductive representation and in the dependently typed representation. Moreover, the bijection only maps terms with corresponding constructors. The arguments for constructors in both representations are restricted to those in corresponding sets. Under these assumptions the induction step is obvious if laborious.

Formal Presentation

To formally demonstrate a bijection in any particular case it is necessary to define two total functions which can be proven to be inverses. We show by an example on the equivalent representations of trees and forests how this can be done. Figure 2.3 shows the type of the mapping from the dependently typed representation to the mutually recursive representation and of its inverse and the statement of the bijection lemma. The mapping

```
Q =
fun (A : Set) (x : tf_enum) =>
  match x with
  | tf_tree => ntree A
  | tf_forest => nforest A
  end
     : Set -> tf_enum -> Set
```

(a) A mapping from enum constructors to types.

```
P =
fun (A : Set) (x : tf_enum) (_ : tf A x) => Q A x
     : forall (A : Set) (x : tf_enum), tf A x -> Set
```

(b) A mapping from elements in the dependent type to their corresponding mutually inductive type.

```
ntree_or_nforest_of_tf
     : forall (A : Set) (x : tf_enum) (t : tf A x),
         P A x t
```

(c) A mapping from `ntree` to `tf`.

```
tf_of_ntree_or_nforest
     : forall (A : Set) (x : tf_enum), Q A x -> tf A x
```

(d) A mapping from `tf` to `ntree`.

```
bijection
     : forall (A : Set) (x : tf_enum) (t : tf A x),
       t = tf_of_ntree_or_nforest A x
         (ntree_or_nforest_of_tf A x t)
```

(e) The type of a proof that `ntree_or_nforest_of_tf` and `tf_of_ntree_or_nforest` form a bijection.

Figure 2.3: Types of mapping between equivalent representations and statement of the bijection lemma.

functions show an interesting use of dependent types. The return type of `ntree_or_nforest_of_tf` is dependent on the value of the `x` argument; while the type of an argument of `tf_of_ntree_or_nforest` is dependent on the value of `x`. It is possible to define equivalent mappings in several alternate ways; these definitions facilitate the proof of the bijection lemma by forcing Coq's type-checker to reduce the *match* expression.

Such a complicated type cannot be extracted to O'Caml. Figure 2.4 shows the extracted versions of the inverses. Note that both functions depend on the type `'a q` which has the value `__`. This value represents a type that cannot be translated into O'Caml's non-dependent type system. However, all uses of arguments of this type are proven safe by the Coq system.

A Dependent Type Transformation as a View

A simpler dependent type transformation yields a *view* of the mutually recursive type. Figure 2.5 shows the view type. Views were introduced in Wadler (1987) as a way of allowing induction on an abstract type or of allowing a different form of induction on a non-abstract type.

A view must be accompanied by an *in* function which constructs a view from the underlying type and an *out* function which performs the inverse operation. These functions are not recursive.

As a computation progresses, the mutually recursive datatype is transferred by stages to the view type using the *in* function. When a recursive call is necessary the term in the view type is translated to the non-view type using the *out* function.

The necessary calls to the *in* and *out* functions make structural recursion impossible because it is not possible to demonstrate termination. In Chapter 5 more general approaches to induction are discussed.

```
type 'a q = __
```

(a) A mapping from enum constructors to types.

```
type 'a p = 'a q
```

(b) A mapping from elements in the dependent type to their corresponding mutually inductive type.

```
(** val ntree_or_nforest_of_tf :
    tf_enum -> 'a1 tf -> 'a1 p **)

let rec ntree_or_nforest_of_tf t = function
  | Tf_node (a, t1) ->
      Obj.magic (Node (a,
        (Obj.magic
          (ntree_or_nforest_of_tf Tf_forest t1))))
  | Tf_nil -> Obj.magic Nil
  | Tf_cons (t1, t2) ->
      Obj.magic (Cons
        ((Obj.magic
           (ntree_or_nforest_of_tf Tf_tree t1)),
         (Obj.magic
           (ntree_or_nforest_of_tf Tf_forest t2))))
```

(c) A mapping from `ntree` to `tf`.

```
(** val tf_of_ntree_or_nforest :
    tf_enum -> 'a1 q -> 'a1 tf **)

let tf_of_ntree_or_nforest x x0 =
  match x with
    | Tf_tree ->
        Obj.magic (fun x1 -> tf_of_ntree x1) x0
    | Tf_forest ->
        Obj.magic (fun x1 -> tf_of_nforest x1) x0
```

(d) A mapping from `tf` to `ntree`.

Figure 2.4: Extracted versions of mappings between equivalent representations.

```
Inductive tf_view (A : Set) : tf_enum -> Set :=
    tf_view_node : A -> nforest A -> tf_view A tf_tree
  | tf_view_nil : tf_view A tf_forest
  | tf_view_cons :
      ntree A ->
      nforest A ->
      tf_view A tf_forest
```

Figure 2.5: A view type using dependent types.

Use of Dependent Types in Programming Languages

Use of dependent types in programming languages has in general been avoided because dependent types lead very rapidly to undecidability of type checking, although several interesting prototype languages that do make extensive use of dependent types are in development. A promising avenue currently being explored is the use of subset types (Sozeau, 2007b) where the domain over which the types may be defined is restricted as with liquid types (Rondon et al., 2008) or the use of dependent types over a restricted domain as in Dependent ML (Xi, 2003, 2007).

We suggest an approach inspired by liquid types for handling the restricted set of dependently typed representations generated by our transformation in a programming language. In this approach we restrict not the domain but rather the form of types by restricting the use of subset types to exactly the enum types. In every case, an enum type is composed only of nullary constructors. Such types are actually simpler than the usual restricted domains which in general include some numeric set to facilitate, e.g., array bounds checking. We allow subset types to be defined only over enum types. Thus, we allow types such as

```
{x : tf_enum &  ((x = tf_tree) * tf A x)}
```

This type packages an argument of `tf_enum` a proof that this argument is equal only to `tf_tree` and an argument in `tf tf_tree` in a single prod-

uct. A function which takes such an argument thus need only match the constructors in type `tf tf_tree`.

Where it is necessary to perform a recursive computation over an element subset types are not generally useful. In general, the result of a recursive computation must include all types, i.e., must be

```
{x : tf_enum & ((x = tf_tree \/ x = tf_forest) * tf A x)}
```

which is the same as `{x : tf_enum & tf A x}`. However, O'Caml programmers are familiar with simple, non-recursive functions which extract some part of an algebraic type. In this case, a type which indicates that the constructors to be matched are only a subset of the constructors in the entire type is valuable for optimization (Xi, 2003).

The development and implementation of such a type system is outside the scope of this thesis. However as interest in automatically inferred subset types and their uses continues to grow such an enhanced type system is worth exploring.

2.3 Implementation of the Transformation

The dependent type transformation can be automated using a simple syntactic approach and taking advantage of the bijections between syntactic components of corresponding types described in Section 2.2.

The first bijection is one between constructors in the two types. Any bijective mapping will do. It is necessary to make sure that the constructor names in the dependently typed definition do not coincide with those in any other definition, as the CoC shares the unique constructor restriction with all the variants of ML.

The second bijection is between the types of the mutually inductive representation and the constructors of the enum definition. Again, the only restriction on the constructors is that they be unique.

The third bijection is between the sets to which constructors may belong. In Section 2.2 we identified these sets by their members; they can also be identified by their type in the case of the mutually recursive representation or by their enum constructor in the case of the dependently typed representation.

Working from the leaves of inductive definitions to their roots, the transformation is accomplished as follows.

type applications All designations of types are transformed to their appropriate counterparts. In the example, `nforest A` must be transformed to `tf A tf_forest` and `ntree A` to `tf A tf_tree`.

constructor names All constructors are transformed to their appropriate counterparts. For example, `node` becomes `tf_node`.

constructor folding All constructor definitions are combined into the body of a single inductive definition.

type annotations The type of the single inductive definition is changed so that the first non-parameter type is the enum type.

type name The name of the inductive definition is changed to the desired name for the dependently typed representation.

Our transformation uses ANTLR (Parr, 2007, 2010). ANTLR is a system of domain specific languages and supporting libraries for building abstract syntax trees from text, for inspecting and transforming them, and for producing structured textual output. We transform the inductive definitions using ANTLR tree matching patterns. Except for the transformation of the type applications the implementation is relatively straightforward since the transformation rules do not depend on the structural properties of the inductive representation.

However, transforming the type applications requires taking into account the *parameters* of the inductive definition. Parameters are arguments that may not change for different constructors. The enum argument must be placed after the parameters but before any other arguments to the type constructor in order to maintain the bijection between the two representations. A purely static tree pattern matcher is inadequate to the task. To handle this we use a two phase approach. First we dynamically generate the type application transformation grammar based on the number of parameters in the inductive type. Then we apply the static and dynamically generated grammars to completely transform the type.

Every constructor in the enum type is nullary and the name of each constructor is available. It is easily generated using a simple template mechanism. Note that its sort is always *Set*, so that individual constructors can be distinguished.

2.4 Structural Induction and Recursion

A dependent representation allows a more flexible approach to structural induction and recursion than a mutually recursive representation. It is possible to view a dependently typed representation as a homogeneous term and also as a term that is partitioned by its enum type. The first view is valuable where the result of the computation is dependent on the structure of the term and not its type. The second approach is essential where the result of the computation is dependent on the type of the term. A mutually recursive representation only allows the second approach.

Our contribution in this section is two-fold. First, we identify what structural induction principles are possible for the two equivalent representations. Second, we describe the advantages of the structural induction principles for a dependent representation over the principles for a mutually recursive representation.

	Unified
Mutually Recursive	sum ntree nforest $\rightarrow$ Type
Dependently Typed	forall (t : tf_enum), tf t $\rightarrow$ Type

Table 2.1: Unified result types for equivalent types.

	Discrete
Mutually Recursive	ntree $\rightarrow$ Type nforest $\rightarrow$ Type
Dependently Typed	forall (t : tf_enum), t = tf_tree $\rightarrow$ tf t $\rightarrow$ Type forall (t : tf_enum), t = tf_forest $\rightarrow$ tf t $\rightarrow$ Type

Table 2.2: Discrete result types for equivalent types.

Structural induction principles can differ in a number of ways. First, the type of the result of the induction can be defined over each discrete type or over the union of all types.

Unified Result versus. Discrete Result

The result of an induction principle may be defined over a unified type or over the discrete types that make up the representation. In a mutually inductive representation this requires using the `sum` type to unify the component types. For a dependently typed representation this requires universal quantification over the enum type. Table 2.1 shows the unified types for both representations and Table 2.2 shows the discrete types for both representations.

Discrete and unified representations of the result type are always interconvertible. It is possible to combine the discrete representation to make a unified representation; it is possible to decompose the unified representation into the discrete representation.

```
(** val tf_rect :
    ('a1 -> 'a1 tf -> 'a2 -> 'a2) -> 'a2 ->
      ('a1 tf -> 'a2 -> 'a1 tf -> 'a2 -> 'a2) ->
      tf_enum -> 'a1 tf -> 'a2 **)

let rec tf_rect f f0 f1 t = function
  | Tf_node (a, t1) -> f a t1
       (tf_rect f f0 f1 Tf_forest t1)
  | Tf_nil -> f0
  | Tf_cons (t1, t2) ->
       f1 t1 (tf_rect f f0 f1 Tf_tree t1)
         t2 (tf_rect f f0 f1 Tf_forest t2)
```

Figure 2.6: The extracted version of a structural recursion principle for `tf`.

Unified Structural Induction

Where the inductive definition is expressed using dependent types a unified traversal, i.e., a traversal that does not take into account distinct types but treats all constructors uniformly, is possible. In fact, this is exactly the structural recursion principle Coq generates automatically from the definition of `tf`. Figure 2.6 shows the extracted version of this recursion principle; the syntactic structure is almost identical to that of the CoC version. The result type of this recursion principle is also unified; discrete result types would not ease development since the whole principle may take an argument of any constructor, hence any type.

Because it is impossible to ignore the types to which constructors belong in a mutually recursive representation a unified traversal is impossible. This is not immediately obvious; a solution seems to be to define a structural recursion principle over the sum of the discrete types. However, such a recursion principle cannot be proved to terminate. Figure 2.7 shows what the extracted body of such a function must look like if it were possible to define it in Coq. At each recursive call it is necessary to pack up the subterms in a `sum` constructor; either `Inl` or `Inr`. This necessity violates the syntactic constraint Coq enforces to ensure that every recursive function

```
(** val ntree_rect :
    ('a1 -> 'a1 nforest -> 'a2 -> 'a2) -> 'a2 ->
      ('a1 ntree -> 'a2 -> 'a1 nforest -> 'a2 -> 'a2) ->
      'a1 (sum ntree nforest) -> 'a2 **)

let rec ntree_rect f f0 f1 = function
  | Inl (Node (a, t1)) -> f a t1
      (ntree_rect f f0 f1 (Inr t1))
  | Inr (Nil) -> f0
  | Inr (Cons (t1, t2)) ->
      f1 t1 (ntree_rect f f0 f1 (Inl t1))
          t2 (ntree_rect f f0 f1 (Inr t1))
```

Figure 2.7: The extracted version of a unified structural recursion principle for `ntree`.

is terminating. Thus, such a function is inadmissible and it is necessary to fall back on some form of general recursion.

Unified structural induction is always possible in the dependently typed representation and never possible in the mutually recursive representation. This is a significant advantage of the dependently typed representation.

With a unified structural recursion principle it is still possible to generate facts that are dependent on the enum type to which the constructor belongs. For example, the mapping from elements in `tf` to elements in `ntree` or `nforest` (Figures 2.3 and 2.4) is dependent on whether the constructor belongs in the set associated with `tf_tree` or `tf_forest`. The function is easily constructed using the standard structural recursion principle for `tf`, `tf_rect` (Figure 2.6).

Discrete Structural Induction

The only sort of induction principle available for a mutually recursive type is discrete induction as unified structural induction is impossible. This means that a separate mutually recursive principle must be defined

```
(** val ntree_r :
    ('a1 -> 'a1 nforest -> 'a3 -> 'a2) -> 'a3 ->
      ('a1 ntree -> 'a2 -> 'a1 nforest -> 'a3 -> 'a3) ->
      'a1 ntree -> 'a2 **)

let ntree_r f f0 f1 n =
  let rec f2 = function
    | Node (a, n1) -> f a n1 (f3 n1)
  and f3 = function
    | Nil -> f0
    | Cons (n1, n2) -> f1 n1 (f2 n1) n2 (f3 n2)
  in f2 n
```

Figure 2.8: The extracted version of a mutually recursive structural recursion principle for `ntree`.

for every discrete type in the recursive type. Moreover, when proving by reduction, it is essential that every principle is identical up to renaming, otherwise reduced terms will appear unequal.

Figure 2.8 shows an extracted version of a mutually recursive structural recursion principle. It has exactly the structure an O'Caml programmer would be accustomed to.

A discrete structural recursion principle is also available for the dependently typed representation. Figure 2.9 shows the extracted version of such a principle. Note that both mutually recursive functions must match all possible constructors. However, where a constructor does not actually belong to the type the phrase `assert false (* absurd case *)` appears. This indicates that this match has been proven to be impossible in Coq, based on the dependent type of the constructor.

2.5 Effect of Representation on Extracted Code

In our discussion so far we have demonstrated that a dependently typed representation has advantages over a mutually inductive representation when developing proofs and program. On the other hand we have also

```
(** val tf_tree_r :
    ('a1 -> 'a1 tf -> 'a3 -> 'a2) -> 'a3->
      ('a1 tf -> 'a2 -> 'a1 tf -> 'a3 -> 'a3) ->
      'a1 tf -> 'a2 **)

let tf_tree_r f f0 f1 t0 =
  let rec f2 = function
    | Tf_node (a, h) -> f a h (f3 h)
    | _ -> assert false (* absurd case *)
  and f3 = function
    | Tf_node (a, h) -> assert false (* absurd case *)
    | Tf_nil -> f0
    | Tf_cons (h', h'') -> f1 h' (f2 h') h'' (f3 h'')
  in f2 t0
```

Figure 2.9: The extracted version of a mutually recursive structural recursion principle for `tf`.

shown that the extracted code has unusual and potentially awkward properties. For example, the extracted version of the `ntree_or_nforest_of_tf` (Figure 2.4) function contains untypable parts inhabited by the `Obj.magic` cast. The discrete traversal structural recursion principles for dependent types (Figure 2.9) contain assertions in cases where a constructor is impossible.

Two philosophical approaches to this situation are possible. In the first place, one may condemn all uses of dependent types in programs. Some users urge an approach where all programs are written in a subset of the CoC which is shared with O'Caml. In this approach, dependent types may appear only in ancillary proofs. Since they form a non-computational part of the proof development they will always be elided by the extraction mechanism and all extracted code will be typable by the O'Caml compiler. An opposite approach, which also has its supporters, is to include the specification of all programs within the type of the program. These two opposing viewpoints are encapsulated in recent work on certified compilers; the CompCert Project (Blazy et al., 2006; Leroy, 2006, 2009) which

takes the former approach and Chlipala's certified compiler (Chlipala, 2007b) which takes the latter approach.

In our opinion either viewpoint is useful, so long as it is adhered to consistently in any single development. However, we would like to draw attention to an area of research that is consistently overlooked in the literature. Coq's extraction mechanism (Letouzey, 2004, 2008) is relatively new. Research efforts relating to extraction have focused on

1. finding representations in Coq so that extracted code is cleaner, i.e., more parts of the program are identified as non-computational and elided (Letouzey and Spitters, 2005)

2. substituting more efficient data structures for less efficient but more functional data structures so long as the more efficient data structures are observationally equivalent (Oury, 2003)

What has been consistently overlooked is the opportunity for optimization of extracted code. Consider a module that makes use of a dependently typed representation as in the ongoing example. Because compilation of modules is modular the actual type can be hidden from external users of the module. Therefore, there are plenty of opportunities for low-level compiler optimizations within the module. For example, the `Obj.magic` cast is proven to never fail. Such a cast can be elided in an optimizing compiler that is aware that it cannot fail. A related approach is used in the implementation of Java Generics where more sophisticated parameterized types are elided during compilation but unnecessary casts are avoided in the generated bytecode (Bracha, 2004; Naftalin and Wadler, 2006). Similarly, where it has been proven that a particular constructor can never be matched the compiler need not check for an assertion failure. Closed polymorphic variants (Hickey, 2008) may represent another alternative target for extraction. A closed polymorphic type is used to describe a type consisting of a subset of the set of constructors belonging to a given type.

Clearly, this describes exactly the situation where the value of an enum argument is known.

Recent work on an extraction to the Glasgow Haskell Compiler's intermediate language(Nanevski et al., 2008) is promising. By an extraction to the intermediate language type information outside the Haskell type system can be preserved. However, the authors report that even their extraction is still forced to discard type information which is inexpressible in the intermediate language.

2.6 Conclusion

In this chapter we have demonstrated that every homogenous mutually inductive datatype may be automatically transformed to an equivalent dependent type with a single inductive definition. We have shown that encoding an inductive definition via a dependent type has advantages when the context is proof development. We have also shown that several opportunities exist in the area of extraction. First, with the growing interest in subset types it is likely that programming languages may be developed that allow subset types of a restricted form, as with liquid types. In that case, it could be possible to target the extraction to an enhanced version of O'Caml that allows subset types. Second, where it has been proven impossible in Coq for a particular constructor to be matched the O'Caml compiler may take advantage of this information to generate optimized code. Third, closed polymorphic variants represent an existing part of the O'Caml type system which provides a promising target for extraction of function types where the function type contains a dependent type constructed using an enum type.

3 AUTOGENERATING UNIFORM FUNCTIONS

3.1 Introduction

The structural recursion principles discussed in previous chapters exist so that a user can develop recursive functions and inductive proofs. Proofs and specifications coexist in the CoC. However, the requirements for a proof and a specification are very different.

A theorem is a type in the CoC; a proof object is any term in the CoC that satisfies that type. The syntactic structure of the term is only important in so far as a simpler term can be built more efficiently. Coq's facilities for automatic proof search are appropriate for proof development since they are type-directed.

On the other hand, the syntactic structure of specifications is important and Coq's automation facilities are not appropriate for the automatic synthesis of specifications. For example, the function `head` defined on `lists` has type

```
forall A : Type, list A -> option A
```

There are an infinite number of functions on lists that satisfy this type. However, the following function

```
head =
fun (A : Type) (l : list A) =>
  match l with
  | List.nil => error
  | x :: _ => value x
  end
     : forall A : Type, list A -> option A
```

implements what we intuitively believe to be the correct semantics, while the function

```
head =
fun (A : Type) (l : list A) => error
    : forall A : Type, list A -> option A
```

does not. Each function can be constructed by a simple tactic script. An automatic search is likely to prefer the second, incorrect, specification since it is smaller.

Certain functions are *uniform*, i.e., they treat each constructor in the same way. In these functions, the body of each principal premise can be defined by the same generic function. In this chapter we demonstrate a way to generate each principal premise automatically by a combination of definitions and tactic scripts. Our approach is robust in that each principal premise will be regenerated correctly even if the inductive definition that defines the type of their argument is changed. Our technique is a novel, tactical approach to datatype-generic programming (Gibbons, 2003, 2007; Hinze et al., 2007; Jeuring and Plasmeijer, 2006).

3.2 Background

In many functions, the body of every principal premise corresponds to some fold operation over the premise's arguments. The list of arguments is described by the regular expression $((ar)|a)*$ where a represents an argument to the principal premise's corresponding constructor and r represents an optional return value associated with the preceding a. The return value may be the result of a recursive call or it may be the result of the application of a principal premise. For example, in Figure 7.3 the list of arguments to the principal premise for the `cons` constructor matches the string $aarar$. It is important to note that the arguments are not syntactically distinct. It is necessary to use semantic information to discover that, e.g., the last argument is a result argument.

Many general recursion principles rely on a *measure*, a function that

maps terms in an inductive definition to some other value. A familiar measure is the number of constructors of the same type in a given term. For each constructor, the body of the corresponding principal premise must be

$$\lambda \mathsf{l}.1 + \text{fold } (\lambda \text{ acc}.\lambda \text{ e. if } e \text{ matches } \mathsf{r} \text{ then } \text{`}(\text{e } + \text{ acc}) \text{ else } \text{`acc}) \text{ `0 } \mathsf{l}$$

For the `Disjunct` function (Figure 6.3) a similar expression

$$\lambda \mathsf{l}.\ \text{fold } (\lambda \text{ acc}.\lambda \text{ e. if } e \text{ matches } \mathsf{r} \text{ then } \text{`}(\text{e } \vee \text{ acc}) \text{ else } \text{`acc}) \text{ `False } \mathsf{l}$$

is appropriate. Note that in each example the parts of the computation marked by a tick are suspended. The operation returns a term in the CoC that evaluates to the desired result, not the result itself. We call fold's first argument the *combine* operation and the second the *initial value*.

Each fold operation is *polytypic* (Gibbons, 2003, 2007; Hinze et al., 2007; Jeuring and Plasmeijer, 2006) in the sense that it is actually datatype-generic. It is therefore reasonable to consider the entire operation of which each fold operation is a component as datatype-generic since it is otherwise only dependent on a structural recursion or induction principle which the Coq system automatically synthesizes for every inductive datatype.

Polytypic specifications are robust in two ways. Being polytypic they are applicable to any inductive datatype. The fold operation for the size function is correct for a data structure such as an `ntree` and for a term in the lambda-calculus. Moreover, they are robust to changes in the underlying datatype for which they are defined.

However, Coq does not allow any facilities for reflecting directly on the structure of functions, so it is not possible to specify these fold operations explicitly. At first glance it might seem fairly easy to directly synthesize the body of each principal premise from the specification of its type. However, this approach is beset with pitfalls. It is necessary to construct each function from scratch, deducing the pattern of constructor arguments and

return values. As we have remarked, this cannot be done by a simple syntactic approach but requires a semantic analysis of the recursion principle of which the principal premise is an argument.

We propose instead a mixed approach based on an innovative use of tactics. This approach takes advantage of facilities in the *Ltac* language for interrogating the proof-context and for inferring the types of implicit arguments. The key components of our approach are *marker functions* and *context folding*, which we describe in Section 3.3 and Section 3.4 respectively. We illustrate this approach using the example of the `In` function defined on `ntrees`.

Our approach makes a large class of polytypic functions available to the Coq developer. Some polytypic functions, e.g., measure functions are so generally useful that it is desirable to make their construction an integral part of Coq in the same that generation of equality functions has been incorporated into the Coq system. For other less universally useful functions the tactics and techniques we make available allow the user to build their own polytypic functions with little difficulty.

3.3 Marker Functions

The purpose of a marker function is to distinguish arguments which are the result of a recursive call or of the application of a principal premise from other arguments that may be of the same type. For example, consider constructing a measure function for elements of type `nat`. If the measure function is just the number of constructors then the result of the function will be one more than its argument. A function on `nats` that returns one more than its argument is, of course, easily constructed by hand. However, the goal is to construct this function automatically, using tactics, in a way that is applicable to every inductive definition. The first step is to state the goal:

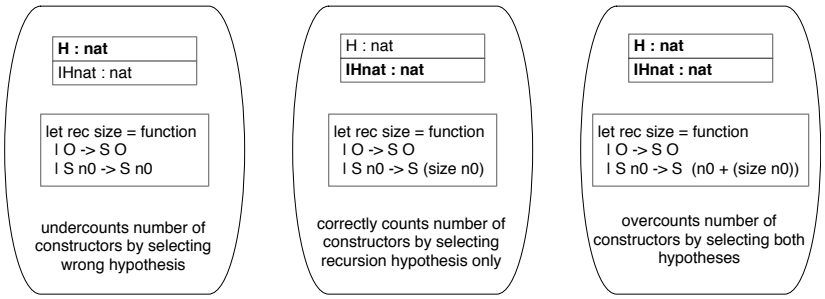


Figure 3.1: Possible combinations of hypotheses for a size function on `nats` and the resulting extracted functions. Hypotheses incorporated into the sum are in bold.

```
nat -> nat
```

That is, the user indicates to the system that a function from `nat` to `nat` is desired. After applying the recursion principle for `nat` the user is presented with two subgoals. The second subgoal, corresponding to the `S` constructor, is confusing. The context contains two hypotheses with type `nat`. One of them is the result of the recursive call; the other the argument to the `S` constructor. If both are combined, the resulting function will overcount the number of constructors in the argument. If the second, but not the first is used, the resulting function will undercount the number of constructors. Figure 3.1 shows the three possible choices of hypotheses and the resulting functions that may arise.

Coq has a mechanism for choosing the names of hypotheses based on type information. In Figure 3.1 the result of the recursive call is `IHnat`. This naming will certainly help a user to pick out the result of the inductive hypothesis from the constructor argument. However, a tactic which relies on a particular naming scheme to identify the results of recursive calls is intolerably brittle.

The Coq system automatically reduces type terms to decide equality of types. So while `nat` and `P n` are syntactically distinct terms the Coq type-

checker judges them equal if `P n` can be reduced to `nat`. On the other hand, Coq will not automatically reduce an application unless such a reduction is explicitly requested. This laziness on the part of Coq's reduction engine is desirable since it facilitates the application of general lemmas. Moreover, it allows tactics to distinguish between the two equivalent, but syntactically distinct, terms.

To obtain a marker function it is necessary to abstract over the argument, yielding a function with the required return type. Thus, in the case of `In`, the marker function, `M`, is just

```
fun (x : list A) => Prop
```

Thus, `M l` = `Prop` for any `l` and so the Coq type-checker identifies them. However, the *Ltac* match statement is able to distinguish `M l` and `Prop` since they are syntactically distinct.

A mutually inductive datatype or a proof that makes use of a heterogeneous induction principle will require more than one marker type. For example, to define a function analogous to the `In` function for `lists` on `ntrees` requires three marker functions, one for `ntree`, one for `nforest`, and one for its argument elements. Each one of the marker functions has the same return type, `Prop`.

Once the induction principle is applied the proof-context for each subgoal is populated with marked terms. Figure 3.2 shows the subgoal corresponding to the `node` constructor. Each hypothesis corresponds to an argument to the principal premise. Two hypotheses are especially marked. `IHn` is marked with the marker function `P_A` indicating that it is the result of an application of the principal premise for `A`. `IHn0` is marked with the marker function `P_nforest` indicating that is is the result of an application of the fixed point function for `nforest`. The goal is marked with `P_ntree` because it is the head type of the principal premise.

These marker functions are specified by the user. They need not be respecified for such trivial changes as the addition or subtraction of a con-

```
H : case node
a : A
IHn : P_A a
n : nforest A
IHn0 : P_nforest n
====================
P_ntree (node A a n)
```

Figure 3.2: A proof-context for a `node` subgoal showing marker types.

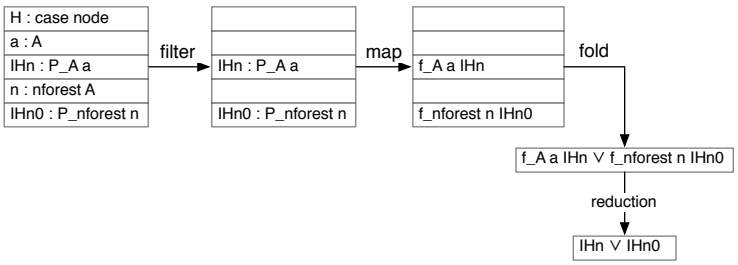


Figure 3.3: A schematic illustrating the operation of constructing a disjunction of marked terms.

structor, the rearrangement of a constructor's arguments, or the reordering of constructors.

The marker functions serve to identify the terms which must be included in the disjunction. To gather the terms up requires an operation shown schematically in Figure 3.3. First, a *filter* operation discards all terms that are unmarked. Second, a *map* operation transforms the remaining terms. The operation is marker dependent, so a different function is applied to the marked term in each case. `f_nforest` is just the identity function so `f_nforest n IHn0` reduces to `IHn0` as does `f_A`. Third, the mapped terms are folded into a single disjunction. The fourth step, reduction, shows the normal form of the disjunction. If the context were available as a list of pairs of terms and types the entire operation would be

entirely simple and obvious. However, the *Ltac* language does not provide any such view of the context. Instead, the user must rely on a tactic that makes subtle use of *Ltac*'s pattern matching and exceptions to achieve the intended result. We discuss how the operation described above can be achieved in Section 3.4.

3.4 Context Folding

The components of the disjunction for any principal premise are easily identified via marker functions. However, accumulating the required disjunction is not straightforward. The *Ltac* language's pattern matching facilities are at the same time sophisticated and primitive. In particular, there is no direct way to perform any sort of fold operation on the proof-context. In the example in Figure 3.2 the appropriate term to satisfy the goal is `IHn` $\vee$ `IHn0`. To construct the term it is necessary to locate both `IHn` and `IHn0` by matching their types and to form the disjunction of the result of the mapping operation on each. This task can only be done by matching every term in the proof-context and investigating its type to determine if it should be included in the disjunction and, if it should, composing it via the `or` operator with the disjunction obtained so far.

Data structures implicitly assume that their elements are not interdependent. The elements may have some relation, but they are not constructed from each other. A phone book can be modeled as a list since it does not record, e.g., the family relationships among different elements in the list but instead orders the elements lexicographically. A program is modeled as an abstract syntax tree to encode the relationships between different syntactic elements; a list is not adequate. A proof-context may appear to be a list but the elements do in fact have dependencies. The proof-context is not arbitrarily reorderable because one hypothesis may be dependent on another. It is really an appropriate topological sort of a

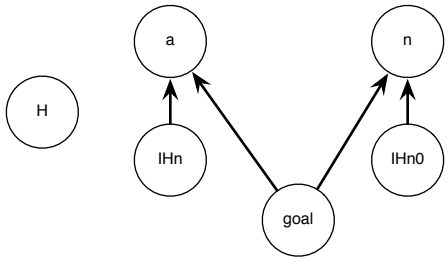


Figure 3.4: The dependencies between terms in the subgoal shown in Figure 3.2.

directed acyclic graph where the edges represent dependencies between terms. Figure 3.4 shows the dependence graph for the context in Figure 3.2. The proof-context can be manipulated; but only in a restricted way. It may only be reordered to another topological sort. Elements can be removed, but only if they have no dependents. Thus, elements can be removed starting at the bottom of the context and moving to the top but never in the opposite order. Elements on which the statement of the goal depends cannot be removed until the goal is satisfied.

A fold operation over elements with dependencies is sensible if the component operation does not take into account the dependencies. For example, a function that counts the elements in a list can be defined via the fold function. Even if the elements are actually interdependent this is irrelevant. The results of recursive calls and applications of principal premises are, by definition, not interdependent. Thus, a fold operation over these is sensible.

However, the implementation of the fold operation is constrained by the dependencies between the terms and by the restrictions of the *Ltac* language. It is not possible to remove a hypothesis once it has been encountered and the result of the map operation incorporated into the disjunction since the hypothesis is required to solve the goal. It is not possible to mark hypotheses as already visited so that they can be skipped. It is not

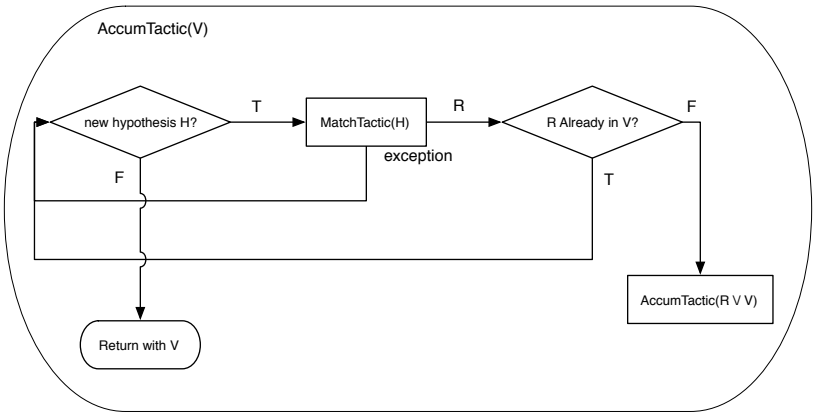


Figure 3.5: A schematic of the operation of the `fold_context` tactic.

possible to select a hypothesis at a certain index, as is usual with lists. The following is a description of the approach we are constrained to take.

Figure 3.5 shows the core structure of the `fold_context` tactic: `AccumTactic`. It in turn relies on `MatchTactic`. Figure 3.6 depicts the operation of `MatchTactic`; Figure 3.7 shows the corresponding *Ltac* source. The tactic is very simple. If the type of the argument `H` is one of the marked types, `MatchTactic H` returns the appropriate CoC term. If it is not one of the marked types, `MatchTactic` throws an exception. In general, it is necessary to have a version of `MatchTactic` for every uniform function. On the other hand, once built it need not change for any minor alterations to the inductive type like moving constructors, changing their arguments, and so forth.

On the other hand, `AccumTactic` is a fully general tactic. In the diagram, subterms are combined using the binary operator `or`, but this is a local variable that is bound to `or` by the enclosing `fold_context` tactic. Figure 3.8 shows the source of the `fold_context` tactic. The arguments `op` and `init` are the combine function and the initial value respectively. `m` is the match tactic.

`AccumTactic` matches hypotheses in order from most to least recent.

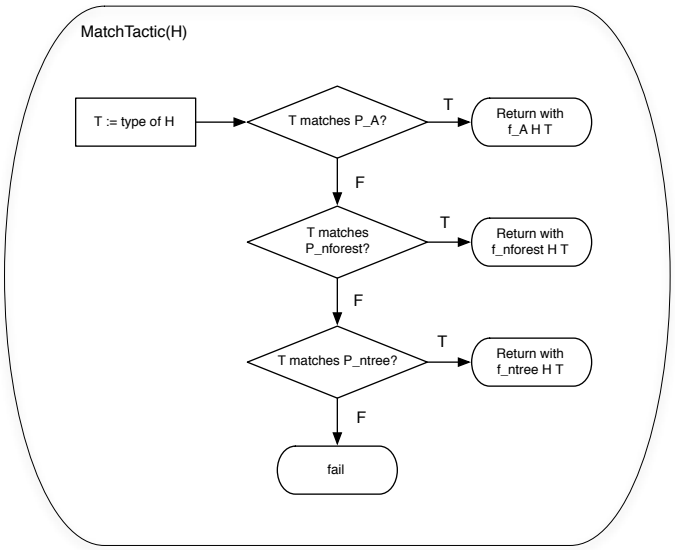


Figure 3.6: A schematic of the operation of the `MatchTactic` tactic.

```
Ltac MatchTactic H :=
  match type of H with
  | P_nforest _ => constr:(f_nforest _ H)
  | P_ntree _ => constr:(f_ntree _ H)
  | P_A _ => constr:(f_A _ H)
  | _ => fail 1
  end.
```

Figure 3.7: The `MatchTactic` tactic.

Each hypothesis matched is passed as an argument to `MatchTactic` which either returns a term in the CoC, `R`, or throws an exception. If it throws an exception, `AccumTactic` proceeds to match a new, less fresh hypothesis. If it returns `R`, `AccumTactic` checks if `R` already appears in the term that represents the final result, `V`. If it does an exception is thrown and a new hypothesis is matched. Otherwise, the new term, `R` $\vee$ `V` is formed and `AccumTactic` is recursively invoked on this term. When all hypotheses are matched unsuccessfully either because they are uninteresting or because

```
Ltac fold_context op init m :=
  let rec AccumTactic V :=
    match goal with
    | H : ?T |- _ =>
      let V' := m H in
      match V with
      | context [V'] => fail 1
      | _ => let P := constr:(op V' V) in AccumTactic P
      end
    | _ => V
    end
in
  let Init := constr:(init) in
  let L := gather Init in L.
```

Figure 3.8: The `fold_context` tactic.

they are already in the term `AccumTactic` returns with the final value. The `fold_context` tactic binds `init` and `op` to its arguments and invokes `AccumTactic` with `init`.

The `fold_context` tactic is considerably less efficient than the standard fold operation on lists, which is $O(n)$ where n is the length of the list. The worst case is one where every hypothesis is interesting. This results in n recursive calls of `AccumTactic`, one for each hypothesis. Since every recursive call must check all hypotheses until encountering a new interesting hypothesis there are $\frac{n^2}{2}$ hypothesis processing steps in all. The best case is one with no interesting hypotheses; this case requires only n hypothesis processing steps.

To make use of the `fold_context` tactic the user must define some simple functions and an appropriate `MatchTactic`. The number of lines necessary is rather small and proportional to the number of types in the inductive definition. The functions for the `ntree` example required about nine lines of setup code.

While these nine lines are more than a user would require when initially developing this function they are a small price to pay for a robust tactic

that does not require manual repair whenever the inductive definition is changed.

If the user requires the order of the hypotheses in the context to correspond to the order of the parameters in each principal premise she must use the *elim* tactic, rather than the *induction* tactic. The *induction* tactic may rearrange the hypotheses introduced into the context. The *elim* tactic performs only the induction step leaving the user to introduce hypotheses into the context or rearrange them as appropriate.

If a tagged structural recursion principle (Chapter 7) is used it is possible to insert a guard into the `fold_context` tactic. This guard detects when the freshest case hypothesis is reached and immediately returns a value. There is some gain in efficiency from this short-circuiting.

Variations

In some circumstances it is preferable to fold the context in reverse order, i.e., from oldest to newest hypothesis rather than the default, which is newest to oldest. For example, the map function requires the appropriate constructor be applied to the arguments of the corresponding principal premise in the order in which they appear. A `fold_context_reverse` tactic can be constructed which matches the hypotheses from oldest to newest rather than newest to oldest, the default.

The examples discussed so far all require that the fold operation be over a homogeneous list. The elements in every disjunction required to build the `In` function are in `Prop`; the elements in every sum required to build the measure function are in `nat`. This convenient homogeneity does not hold in all cases.

In general, it is possible to accumulate any list of heterogeneous terms using the `prod` type shown in Figure 3.9. The unique constructor, `pair`, combines two terms which may be of any type. Using this constructor in place of the `cons` constructor and `Empty_set` in place of the `nil` constructor

```
Inductive prod (A B : Type) : Type :=
    pair : A -> B -> A * B
```

Figure 3.9: The definition of the `prod` type.

it is possible to accumulate a term which is isomorphic to a heterogeneous list structure.

Such a list may be postprocessed using the *Ltac* language in a variety of ways. The disjunctions that comprise the `In` function can, for example, be constructed by building a list using the `pair` constructor and post-processing the resulting term, substituting an `or` constructor for every `pair` constructor and `False` for `Empty_set`. Building a heterogeneous list using `pair` and then post-processing the list using further scripts to generate the desired term is a more general and less structured approach than the homogeneous list technique.

3.5 Properties of the Initial Value

In all the examples in this chapter, the initial value supplied to the fold operation has also been the identity for the combine function. This is the typical case. In most cases it is desirable to simplify the body of the principal premise, eliminating occurrences of the identity where possible. We supply a tactic to simplify the term resulting from an application of the `fold_context` tactic by applying the rules

- `op id T` $\Rightarrow$ `T`
- `op T id` $\Rightarrow$ `T`

in the usual recursive manner.

Because the CoC can express so many sorts of terms and properties the meaning of identity changes based on the choice of the combine function. In the `In` example in Section 3.3 and Section 3.4, the combine function

is `or` and the initial value is `False`. The domain of the `or` constructor is any proposition. Thus, to prove that the initial value is the identity it is necessary to prove:

```
forall (P : Prop), P \/ False <-> P
```

On the other hand, in the measure example the combine function is `plus` and the initial value is `0`. In that case, to prove that `0` is the identity it is necessary to prove:

```
forall (n : nat), n + 0 =  n
```

Because different properties need to be proved for different choices of the combine function and initial value we rely on the user to make sure that the initial value is actually an identity.

3.6 Conclusions and Related Work

With the tactics described in this chapter a user is able to construct any of a large class of polytypic functions for a given inductive definition. The advantage of our approach is that the tactics are robust to small changes in the underlying inductive definitions, changes which would be likely to cause less robust scripts to fail.

Use of the scripts conforms to a simple pattern. For example, the script that builds an `In` function for `ntrees` and one that builds a `size` function are only five lines long and differ by just a few characters. The script to define a size function for `lists` differs from that for `ntrees` by about twenty characters. The initial effort by the user is reasonable; the reward for the effort is considerable whenever the user can expect to be changing underlying inductive definitions or using the same function for many different inductive definitions.

Aydemir et al. make use of a related accumulating technique for a specific application to sets in their metatheoretic developments (Aydemir

et al., 2008). Their tactic is restricted in application and intended to be used in an interactive setting to facilitate their co-finite quantification techniques. It is our contribution to demonstrate that this approach is extensible, generalizable, and useful in an automatic setting. Chlipala discusses heterogeneous lists but does not consider a tactical approach to their construction (Chlipala, 2009).

In this chapter we have described a general approach for building polytypic functions. Though lightweight, this approach is not fully automatic since the user must specify some simple functions and follow an informal tactic pattern to set up automatic synthesis of the function. Certain functions, particularly measure functions, are so generally useful that it is worth the trouble to eliminate even this small overhead by automatically generating the necessary functions and scripts. In Chapter 4 we discuss measure functions generally.

Datatype-generic, i.e. polytypic programming, is an ongoing subject of research (Gibbons and Paterson, 2009; Gibbons, 2007). Generic variants of Haskell (Hinze et al., 2007) have been built and it has been demonstrated that the existing features of Scala support generic programming well (Oliveira and Gibbons, 2008). We have demonstrated a tactic-based approach which is suitable for an automated proof-assistant with tactic support. The approach we have presented requires some initial effort on the part of the developer, who must specify a number of function definition and tactics as described in Section 3.3. We do not believe that this is a fundamental weakness of our approach. The essential operation of every uniform function can be expressed succinctly by a definition of the arguments to the fold function. The definitions that the user is currently required to specify by hand can certainly be automatically generated. We believe that this would prove a suitable direction for further research.

4 AUTOGENERATING MEASURES

4.1 Introduction

Coq supports the automatic generation of a structural induction or recursion principle for every inductive definition. Recently, a facility to add an equality function and some related proofs for every inductive definition has been made available. However, Coq continues to lack any facility for automatically generating a measure for an inductive definition.

This is an unfortunate omission. In Chapter 5 we introduce several general recursion principles which rely on a *measure*. A measure maps objects from the domain of interest to another domain. Typically the other domain is the natural numbers, `nat` in Coq. Since general recursion is so frequently required and since a measure function is one of the essential requirements of a large class of general recursion principles, measure functions should be automatically available.

In this chapter we discuss properties of measures and restrictions on the structure of measures.

4.2 Properties of Measures

The domain of a measure function may in general be quite complex. Coq's `nat` datatype, expressed according to the usual Peano definition of natural numbers, is simple. Moreover, there is considerable tactical support for reasoning about the natural numbers in the Coq standard library. Consequently, `nat` is a convenient and natural choice for the range of a measure function.

A measure may be opaque to the user or it may be transparent. When the user has access to a datatype definition the measure itself is generally transparent. However, Coq support a module system much like that of

ML. The module system intentionally hides the structure of underlying data from users of a module. For example, Coq's standard library provides a collection of set modules. An appropriate measure in this case is the `cardinal` function which the set modules must export. In this case, the `cardinal` function itself is opaque to the user and the user must rely on lemmas which the set modules also export to show necessary properties of `cardinal` in using a general induction principle. An implementation need only export a single fact, that the cardinality of the set is equal to the length of the list of the elements in the set, but from that fact a host of other facts can be derived. Within the module, however, the `cardinal` function must be defined by some form of recursion; the lemmas that the modules provide about induction are proved using recursion.

In the following sections we discuss some of the difficulties of defining an appropriate measure function. Our contribution in this section is to expose and formalize the challenges that arise in the definition of a measure function. In Section 4.2 we discuss the reasons why a measure function may not be defined on a term with a type in *Prop*. In Section 4.2 we discuss the tradeoffs between two alternative definitions of a a measure function defined using structural recursion. The first alternative is to define a mutually recursive function with as many distinct functions as their are mutually inductive datatypes in the inductive definition. The second alternative is to define a measure function with a single function; the type of its parameter is the sum of the distinct types of the inductive datatypes that make up the inductive definition.

Consequences of the Separation between *Prop* and Other Sorts

To support Coq's extraction mechanism a separation between *Prop* and the other sorts is enforced. In particular, it is forbidden for any function to deconstruct a term with a type in *Prop* in order to build a term in *Set* or

```
Inductive lz (n : nat) : nat -> nat -> Prop :=
    lz_n : lz n n 1
  | lz_S : forall m s : nat, lz n m s -> lz n (S m) (S s)
```

Figure 4.1: The definition of the `lz` type, a redefinition of the `le` type (Figure 5.2) that contains the size of any term.

Type.

This rule does not forbid all functions from a type in *Prop* to a type in any other sort. Such functions are allowed; informally, the restriction only forbids deconstructing a proof of some fact in order to build a value. If deconstructing a proof were allowed, the structure of the proof would become part of the function's result, and the distinction between proofs and programs, necessary for the extraction mechanism, would be broken.

This restriction prevents a measure function defined by structural recursion over the terms of the domain. Two ways of overcoming this restriction immediately suggest themselves; it is our contribution to show that both are impossible within the system and that the restriction cannot, in general, be finessed.

Incorporating the Size of the Proof in the Type

The first approach is to change the types of proofs so that they contain an additional term, their size. Figure 4.1 shows the `lz` datatype, a redefinition of the `le` datatype (Figure 5.2) that contains the size of any term. The third argument is the size of the proof. A proof that `n` is less than or equal to `n` is constructed using the `lz_n` constructor and has size 1. A proof that `n` is less than or equal to `S n` has the form

```
lz_S n (S n) 2 (lz_n n n 1)
```

It is possible to define a measure function that returns the size; it does not need to deconstruct the proof, only to extract the size of the proof from its type.

One difficulty with this approach is that the size of the proof is now part of its type. Functions which take a proof constructed using `lz` as an argument will require a universally quantified type, e.g.,

```
forall s, lz 2 3 s
```

or will need to specify the size of a proof that 2 is less than or equal to 3, e.g.,

```
lz 2 3 2
```

It is impossible to construct an argument of the first type since the only possible value for `s` in the example is 2. However, the two numbers to be compared may be arbitrary natural numbers, represented only by variables, e.g., `x` and `y`. In that case, the only way to specify the type completely is to be aware of the relation between `x`, `y` and the size of the proof that `x` is less than `y`. The expression for the size of such a proof constructed using `lz` is `S(y - x)`. However, if the size of a proof can be calculated from its type then it is useless to include the size of a proof in its type.

In some cases it is possible to calculate the size of a proof from its type. The previous example has shown that it is possible to define a measure function on any proof constructed using `le`. This is a special case, as the values in a dependent type may not always indicate the size of every proof that inhabits the type. Thus, in general, it is impossible to encode the size of a proof in its type.

Indiscriminability

The second approach is to define a type in *Prop* that is suitable for the range of the measure function. For example, it is possible to define a type, `natP`, that is isomorphic to `nat` but inhabits *Prop*. Since it is isomorphic to `nat` it must share the properties of `nat`. However, elements with types

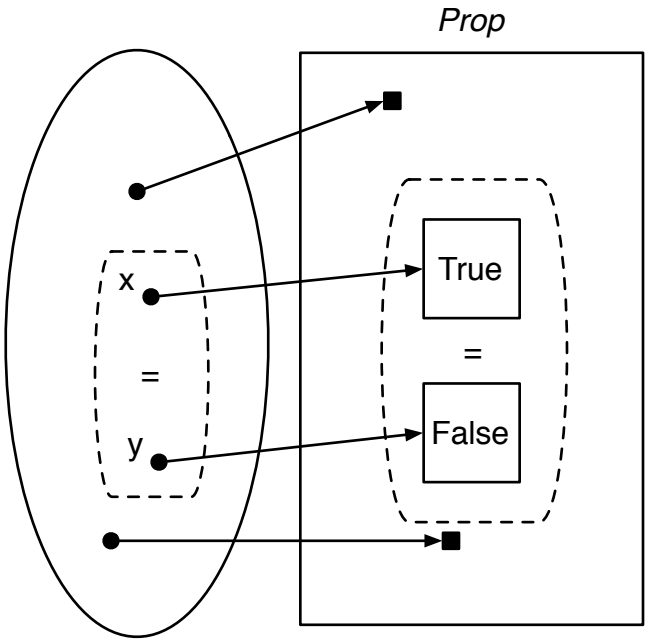


Figure 4.2: A schematic of a proof of inequality showing the discriminating function mapping objects in some type to types in *Prop*.

in *Prop* are *indiscriminable*, i.e., it is impossible to show that two elements with the same type are not equal.

In Coq, the only way to prove that two objects are not equal is to prove that the assumption of their equality proves `False`. A schematic of this approach is shown in Figure 4.2. A discriminating function maps objects in some type to types in *Prop*. Let `x` and `y` be two objects such that the discriminating function maps `x` to the type `True` and `y` to the type `False`. If `x` and `y` are equal it must be the case that `True` and `False` are equal. Therefore, a proof of `True`, which can always be obtained, is also a proof of `False`. Thus, from the hypothesis that two unequal objects are equal it is always possible to prove `False`.

Unfortunately, due to the separation between *Prop* and the other sorts, i.e., *Set* and *Type* the discriminating function is not accepted by Coq where

the domain of the function has a type in *Prop*. Crucially, `True` and `False` are inductive definitions, i.e., they are types, not values. Therefore, it is impossible for them to have types in *Prop*; in fact, they inhabit *Prop*. It is impossible to write a discriminating function with the domain consisting of a type in *Prop* and a range actually in *Prop*. Hence it is impossible to distinguish between two objects of the same type if their type inhabits *Prop*. Consequently, no object with a type in *Prop* is discriminable.

Indiscriminability prevents any proof by induction on the size of any expression with a type in *Prop*. If it is impossible to build a desired proof by structural induction this is a significant obstacle.

Proof-Irrelevance

Coq's type-system prevents discriminating between two proofs of the same type. Often, rather than discriminating between proofs, the user would like two proofs of the same fact to be proven identical. For example, the user may wish to show equality between two instances of a subset type where the values are equal. A subset type is a way of packaging a type and a predicate on that type. An inhabitant of a subset type is a witness and a proof that the desired property holds for that witness.

The equality that is interesting is that between the values, not the proofs. Concretely, let us assume the witness is an element of `nat` and the predicate is that the element must be greater than zero. That is, we wish to show the following goal.

```
forall (n : nat) (H H' : n > 0), exist  n H = exist n H'
```

`exist n H` and `exist n H'` are elided by the extraction mechanism to the identical terms `n` and `n`, so equality always holds in the extracted code. This equality may be essential to the proof we are developing; it is desirable that we can prove it in our development.

Unfortunately, we cannot without resorting to the axiom of proof-irrelevance, which simply states that all proofs of the same property are equal. This axiom is so generally useful that it is defined in the `Logic` module of Coq's standard library. Proof-irrelevance is an axiom, not a lemma. It is impossible to prove in Coq partly because it is false. For example, `or_introl True I` and `or_intror True I` are both proofs of the property `True ∨ True`. On the other hand, they are clearly different, as the one is constructed from `or_introl` and the other from `or_intror`. Clearly, if expressions with types in *Prop* were discriminable then it would be possible to prove that the two expressions were different and at the same time, using the axiom of proof-irrelevance, to prove that they were equal. Thus, proof-discriminability and proof-irrelevance must be avoided in the same proof development in order to prevent an inconsistent system.

To date, proof-irrelevance has prevailed over proof-discriminability. It is included in the Coq standard library and made use of in many proof developments. This seems ironic since proof-irrelevance is false and proof-discriminability is true.

Recently, extensive support for *Setoids* (Barthe et al., 2003) has been included in Coq. Setoids allow the user to define a setoid equality which is not Leibniz equality over elements in a type. Extensive support for setoid equality has been incorporated into tactics which make use of a definition of equality. Setoid equality eliminates some of the motivation for proof-irrelevance. For example, it is possible to define a setoid equality that ignores the values of proofs in a subset type. In that case, the expression

```
exist n H = exist n H'
```

is automatically equal by the setoid equality and there is no need to make use of a proof-irrelevance axiom. Unfortunately, it is still rather difficult to work with setoids. Proof-irrelevance is a good deal simpler and thus continues to be employed in many proofs.

Limited Proof-discriminability

Proof-irrelevance can be asserted with a single axiom. On the other hand, any axiom of proof-discriminability is specific to a particular inductive definition.

For example, suppose `natP` has two constructors, `Op` and `Sp` which are isomorphic to the corresponding constructors in `nat`. In that case, one proof-discriminability axiom must be provided. From the axiom

```
forall (n : natP), Sp n <> Op
```

it is possible to conclude, via the symmetry of equality, that

```
forall (n : natP), Op <> Sp n
```

With this axiom it is possible to re-prove all of the lemmas in the `Peano` module in the Coq standard library for `natP`. We made use of the inequality axiom in proving two facts, that `Op` is not equal to `Sp n` for any `n`, and that `n` is not equal to `Sp n` for any `n`. All other facts were proved identically for `nat` and `natP`. Of course, decidability of equality is easily proved.

The Coq standard libraries provide extensive further support for reasoning about `nat` since its properties are so useful. It would be desirable to identify what properties of `nat` are most valuable for proving well-foundedness of functions. Coq provides a few "power" tactics, e.g., *omega*, which solve particular problems of arithmetic. It would be interesting to see how these tactics can be adapted to `natP`.

Consequences of the Algebraic Representation of Sum Types

In defining a measure function by structural recursion two alternatives present themselves: a mutually recursive function with as many distinct functions as there are distinct datatypes in the inductive definition or a single function which takes the sum type of all datatypes as a parameter.

It is impossible to directly implement the latter approach without already possessing a general recursion principle for the inductive definition for the reasons described in Section 2.4. For this reason, an equivalent dependently typed representation of the inductive definition, as described in Chapter 2, is sometimes preferable.

The alternative is to construct a measure using a discrete recursion principle. Discrete recursion principles for mutually recursive types are described in Section 2.4. A measure which takes a sum type as a parameter may be defined as a wrapper function which deconstructs the sum type until an appropriate ground type is reached and invokes the appropriate one of the discretely typed mutually recursive functions.

4.3 Autogenerating Measures

The two obvious choices for a measure on any inductively defined term are the size of a term, i.e., the number of its constructors, and the depth of a term, i.e., the maximum number of constructors between the root of the term and a leaf. Both measures are instance of the uniform functions described in Chapter 3. Some uniform functions are well-defined only over certain inductive types. For example, a function which calculates the number of free variables in a term is only meaningful for a type which defines the syntax of a language. Functions which calculate the size and depth of a term are universally generic, i.e., they make sense for any inductively defined type.

Such functions are amenable to construction by making use of the approach described in Chapter 3. However, since they are so universally applicable we believe that it would be better to generate them automatically for every inductive definition in the same way that structural induction principles and boolean equality functions are generated.

5 AUTOGENERATING GENERAL RECURSION PRINCIPLES

5.1 Motivation

In this chapter, we describe the *principle* tactic. This tactic populates the proof-context with a general recursion or induction principle specified by the user.

Furthermore, we have developed a language for specifying the general structure of a general recursion or induction principle and an algorithm for expanding the specification to generate a recursion principle for any number of mutually inductive definitions. Consequently, the user may obtain a general recursion principle that is applicable for an induction term in a type defined by any number of mutually inductive types.

We believe that this tactic will be a great convenience to advanced Coq users. It is certain, also, to make the learning curve less steep for novice users. Further, we expect that it will provide a stepping stone for researchers who wish to experiment with the automatic discovery of useful induction principles (Bundy, 2001), either indirectly, by reducing the currently very high cost of experimentation, or more directly, if our specification language is able to serve as an intermediate target for their efforts. We believe that in our own use of this tool we have been able to expose commonalities and properties of different induction principles that would otherwise have been only very dimly perceptible.

In Section 5.2 we discuss the problem of general induction and recursion in more detail.

In Section 5.3 we discuss the structure of a language for specifying the forms of general recursion principles. Our specification language is based on Parameterized Context Free Grammars (PCFGs), our own invention.

In Section 5.4 we discuss the *principle* tactic algorithm. The algorithm is unusual because it must make use of an external Python process and an

```
Inductive False : Prop :=
```

Figure 5.1: The definition of `False`, the simplest uninhabitable inductive type.

external Coq process to synthesize the correct principle.

In Section 5.5 we categorize the different types of general recursion principles that our tactic provides.

In Section 5.6 we describe a number of *measure-based* general induction principles suitable for our specification language.

In Section 5.7 we describe an approach that makes use of a general relation rather than specifying that the relation must be defined by a measure.

In Section 5.8 we demonstrate an alternate approach.

In Section 5.9 we conclude our discussion of this topic.

5.2 Background

Inductive Definitions

In Coq, inductive definitions are the stuff of which finite proofs and values are made. An inductive definition may be uninhabitable; `False`, defined in Figure 5.1, is the most obvious example. Coq also allows co-inductive definitions; these correspond to potentially infinite data structures. Coq does not accept recursive functions on co-inductive values as, due to the potentially infinite nature of these structures, such functions can always be non-terminating.

In Coq, the task of proving is the task of building inhabitants of inductive datatypes or of demonstrating that inhabitants can be built by constructing functions which type-check. When they are in the sort *Prop*, inductive definitions constitute axioms, i.e., statements made without any

```
Inductive le (n : nat) : nat -> Prop :=
    le_n : n <= n
  | le_S : forall m : nat, n <= m -> n <= S m
```

Figure 5.2: The definition of `le`, the inductive type that defines the less than or equal to relation.

```
le_S 0 1 (le_S 0 0 (le_n 0))
```

(a) A Coq proof term.

$$\frac{\dfrac{\dfrac{}{\vdash 0 \leq 0}\ \text{apply le\_n}}{\vdash 0 \leq 1}\ \text{apply le\_S}}{\vdash 0 \leq 2}\ \text{apply le\_S}$$

(b) A Gentzen style representation.

1	$0 \leq 0$	apply le_n	
2	$0 \leq 1$	apply le_S	1
3	$0 \leq 2$	apply le_S	2

(c) A Fitch style representation.

Figure 5.3: Three ways of representing a proof that zero is less than two.

proof. When they are in the sort *Set* inductive definitions are specifications of the structure of values. Figure 5.2 shows the definition of `le`, the inductive type that defines the less than or equal to relation. Figure 5.3 shows three ways of representing a proof that zero is less than or equal to two. They are interconvertible; we show the Gentzen style proof and the Fitch style proof to explain the structure of the proof term.

A more useful proof is one that shows that zero is less than or equal to all natural numbers. Figure 5.4 shows a proof of this fact as a Coq term. Because the proof is about an infinite set, it must be inductive. The

```
good =
fix good (n : nat) : 0 <= n :=
  match n as n0 return (0 <= n0) with
  | 0 => le_n 0
  | S n0 => le_S 0 n0 (good n0)
  end
     : forall n : nat, 0 <= n
```

Figure 5.4: A proof that zero is less than or equal to every natural number.

application `good 14` is a proof that 0 is less than or equal to 14. A proof that 0 is less than or equal to 14 is not actually constructed. Instead, the type-checker verifies that `good 14` has the type `0 <= 14`. It is possible to show the Gentzen or Fitch style representation of such an inductive proof as well. However, the fixed point function shows more clearly than either alternate representation the meaning of the proof.

In the example in Figure 5.4 the proof is a proof by *structural induction*. Structural induction is an instance of well-founded induction (Winskel, 1993) where the well-founded ordering, $\prec$, is the relation "is a subterm of".

Well-founded Induction

A set is well-founded under a particular relation (Winskel, 1993), $\prec$, if the relation allows no infinite descending chains. The well-founded induction principle

$$\forall a, (\forall b, b \prec a \Rightarrow Pb) \Rightarrow Pa$$

allows the user to make use of the hypothesis $P\ b$ for any b that is below a by the relation $\prec$ in order to prove Pa.

In structural induction, the relation $\prec$ is the relation "is a subterm of". This relation is automatically a well-founded relation since inductively defined terms are always finite in Coq. Structural induction principles

```
Inductive Acc (A : Type) (R : A -> A -> Prop) (x : A)
             : Prop :=
    Acc_intro : (forall y : A, R y x -> Acc R y) -> Acc R x
```

(a) The definition of `Acc`.

```
well_founded =
fun (A : Type) (R : A -> A -> Prop) =>
forall a : A, Acc R a
     : forall A : Type, (A -> A -> Prop) -> Prop
```

(b) The definition of `well_founded`.

Figure 5.5: Formalization of well-foundedness in the Coq standard library.

always obey Coq's restrictions to ensure termination. However, the relation "is a subterm of" is very restrictive, i.e., the number of pairs in the relation is relatively small compared to other well-founded relations. For example, the relation "has fewer constructors than" includes all pairs in the "is a subterm of" relation and, in general, infinitely many more. More general forms of recursion make use of different and less restrictive definitions for $\prec$.

Coq's standard library provides a set of inductive definitions which encode well-foundedness. The *accessibility* predicate, `Acc` (Figure 5.5) formalizes a notion of accessibility. A term is accessible by a relation R if all terms that are below it by the relation are also accessible. The predicate `well_founded` generalizes the accessibility property over all inhabitants of a type. Because `Acc` is an inductive type, rather than a co-inductive type, all its inhabitants must be finite. The finiteness of the proof term itself ensures the well-foundedness of the relation.

General recursion principles may make explicit use of well-foundedness by making use of structural induction on the proof of well-foundedness. Figure 5.6 is the structural recursion principle for `Acc`. `Acc_intro` is the unique constructor for `Acc`. Since `Acc` has just one constructor structural recursion as well as structural induction is possible. Since `Acc` is in *Prop* it

```
Acc_rect =
fun (A : Type) (R : A -> A -> Prop)
  (P : A -> Type)
  (f : forall x : A,
       (forall y : A, R y x -> Acc R y) ->
       (forall y : A, R y x -> P y) -> P x) =>
fix F (x : A) (a : Acc R x) {struct a} :
P x :=
  match a with
  | Acc_intro a0 =>
      f x a0 (fun (y : A) (r : R y x) => F y (a0 y r))
  end
     : forall (A : Type)
         (R : A -> A -> Prop)
         (P : A -> Type),
       (forall x : A,
        (forall y : A, R y x -> Acc R y) ->
        (forall y : A, R y x -> P y) -> P x) ->
       forall x : A, Acc R x -> P x
```

Figure 5.6: The definition of `Acc_rect` in the Coq standard library.

```
bad =
fix bad (t : True) : False :=
  match t with
  | I => bad t
  end
     : forall t : True, False
```

Figure 5.7: A proof that `True` implies `False`.

is elided by the extraction mechanism.

Non-termination

Figure 5.7 shows a recursive function that, if it were accepted by Coq, would allow the user to prove that `True`, the always true proposition, implies `False`, the never true proposition. This function is well-typed. It can never "go wrong" in the standard type-theoretic sense.

However, if this function were accepted by Coq then the expression

```
loops =
fix loops (x : nat) : nat :=
  (fun _ : nat => 0) (loops x)
     : nat -> nat
```

Figure 5.8: An example of a recursive function which is not strictly decreasing in its recursive argument. This function is permitted because it terminates under a normal-order evaluation.

`bad I` would have the type `False`. This follows from the usual rules for function application, since `bad` has type `True -> False` and `I` is the unique nullary constructor for `True`. `bad I` would then constitute a proof of `False`. From `False` it is always possible to prove any fact. Consequently, if it were possible for Coq to accept a function like that in Figure 5.7 it would be unusable as a theorem prover.

Therefore, Coq must forbid terms that do not have a normal form by means of some additional restrictions. Since termination is, in general, undecidable Coq must forbid some functions that are terminating as well as all functions that are non-terminating.

Coq's termination checks take into account evaluation order (Charguéraud, 2010). If it can be shown that, under normal-order evaluation, a recursive call will never be evaluated then there are no restrictions on the form of the recursive call. For example, Figure 5.8 shows a function definition that terminates immediately if the outermost function application is evaluated first, but diverges if the recursive call is evaluated first. The form of the recursive call is considered immaterial and is therefore not restricted.

However, if the recursive call will certainly be evaluated then Coq requires that the recursive call be on a subterm of its recursive argument. In the example in Figure 5.7, there is no evaluation order under which the recursive call, `bad t`, will not be evaluated. Consequently, the recursive call is forbidden since `t` is not a subterm of the original argument.

It should be noted that the exceptions to the requirement that every recursive call be on a subterm leads to some surprising properties. Naively, a user would expect that all extracted code would be terminating. But the O'Caml extracted version of the function in Figure 5.8 is actually non-terminating since O'Caml evaluates its arguments eagerly. Moreover, it is possible to specify evaluation order when evaluating terms in Coq itself. The current version of Coq crashes when requested to evaluate the `loops` function eagerly.

General Induction

Due to the restrictions to enforce termination of all recursive programs and the reliance on inductive types to express structures and proofs structural induction is the only form of induction which is an intrinsic part of Coq. When it is defined every inductive type is automatically associated with an induction principle. The induction principle is a higher order function that abstracts the structural induction portion of the recursively defined proof in Figure 5.4. Figure 5.9 shows the structural recursion principle for `nat`, the inductive definition which corresponds to the natural numbers. The arguments `f` and `f0` are the *principal premises* of the function. `f` is the base case; `f0` the inductive step.

Unfortunately, it may be impossible to prove a desired result via structural induction. The insertion sort algorithm is an inefficient sorting algorithm which can be implemented by structural recursion on lists. An empty list is sorted, and a non-empty list can be sorted by inserting the head of the list into the sorted tail of the list. On the other hand, the quicksort algorithm (Hoare, 1962) is an excellent example of a simple algorithm that cannot make use of structural recursion. In the quicksort algorithm, the list is split on each side of a pivot. Each sublist is smaller than its parent list and the algorithm is guaranteed to terminate. However, it is necessary to make use of some form of general recursion since the pivot is not in

```
nat_rect =
fun (P : nat -> Type)
  (f : P 0)
  (f0 : forall n : nat, P n -> P (S n)) =>
fix F (n : nat) : P n :=
  match n as n0 return (P n0) with
  | 0 => f
  | S n0 => f0 n0 (F n0)
  end
     : forall P : nat -> Type,
       P 0 ->
       (forall n : nat, P n -> P (S n)) ->
       forall n : nat, P n
```

Figure 5.9: The structural recursion principle for `nat`.

general located between the head and the tail of the list but may split the list at any point.

That structural induction is not always adequate is a well-recognized problem and various original and diverse proposals for general induction have been proposed (Balaa and Bertot, 2000; Bove and Capretta, 2005; Charguéraud, 2009; Danielsson and Altenkirch, 2009; Megacz, 2007). They all have merit and every one of them should be as accessible to a user of the Coq system as are structural induction and recursion principles. It is unfortunate that none of them are. Our *principle* tactic addresses this problem by providing a specification language for general induction and recursion principles. These principles are *fixed point combinators* that incorporate termination proofs.

Fixed Point Combinators

A *fixed point combinator*, *fix*, is a higher order function that, given a *functional*, *F*, is able to build a recursive function, *f*. fix satisfies the formula

$$f = \mathsf{fix}\, F \Rightarrow fx = Ffx$$

which permits equational reasoning by a one step unfolding of *fix*.

It is a well-known result that *fix* is easily defined in the untyped lambda-calculus but that it is impossible to construct a well-typed definition for a fixed point combinator in the simply typed lambda-calculus (Pierce, 2002). If it were possible to construct such a function then the simply typed lambda-calculus would allow non-terminating functions. However, the simply typed lambda-calculus is, in fact, *strongly normalizing*, i.e., evaluation will eventually reach a normal form under any reduction order.

Clearly, the CoC is not strongly normalizing. In Figure 5.8 we have demonstrated a function that will terminate under one evaluation order but not under another. However, the CoC must be *weakly normalizing*, i.e., must terminate under some evaluation order, in order to be consistent.

Approaches to general recursion are varied. However, a significant subset of these approaches can be classified as the discovery of a fixed point combinator that builds a recursive function from a functional while simultaneously constructing a proof of termination. Such combinators are known as general recursion principles. Our work builds on these efforts by demonstrating a facility for generalizing these terminating fixed point combinators to multiple, mutually recursive, recursion principles by means of a general specification language.

Principle Specifications

Each recursion principle requires a principle specification that designates, in a general way, the form that the principle must follow. General recursion principles are not obvious or easily discovered; every form differs from other forms in significant ways (Balaa and Bertot, 2000; Charguéraud, 2009).

Any inductive datatype may contain several mutually inductive definitions. The greatest difficulty in specifying the general form of a principle is in specifying how the principle varies as the number of mutually inductive

datatypes for which the principle is required varies.

Consider the structural recursion principle, `Acc_rect`, originally described by Balaa and Bertot and defined in the Coq standard library (Figure 5.6) (Balaa and Bertot, 2000). `Acc_rect` is a fixed point combinator. The argument, `f`, is a functional. Note that the functional has a rather lengthy type. Most of the type is involved not with the actual result of the fixed point function but rather with the proof of termination.

Disregard the actual meaning of the principle and consider the type, `A`, over which it is parameterized. `A` can be any type, including the sum of all the inductive datatypes in a mutually inductive definition. If it is the sum, however, `R`, `P`, and `f` must necessarily be quite complicated as each must deconstruct the elements in the sum type into their individual parts. If it is not a sum type, then it is only effective for an inductive definition with just a single inductive datatype. While inductive definitions with just one inductive datatype are typical in simple developments more advanced developments are likely to require mutually inductive datatypes. Consequently, the problem of mutually inductive datatypes must be confronted if more complex developments are to made accessible.

In Section 5.3 we describe our language for specifying the general structure of recursion principles.

5.3 Specification Language

Our observation is that the general induction and recursion principles can have a somewhat fractal-like quality. Each principle may be decomposed into parts which have an identical syntactic structure but vary in the names of particular variables. Each part will vary its structure in a regular way which is dependent upon the number of inductive datatypes in the definition.

We have defined a language which allows the user to indicate the over-

all form of an inductive datatype and also how it varies both in structure and in the names of identifiers.

Our language resembles a context-free grammar (CFG) as it includes the concept of productions, i.e., the replacement of non-terminals with terminals. However, it is also parameterized on the number of inductive datatypes. Moreover, terminals may be subscripted to associate them with their proper inductive datatype. These two things must be achieved by the language specification itself.

Parameterized Context-Free Grammars

A CFG is defined by the following four components: (Fischer et al., 2010)

- A finite terminal alphabet, Σ
- A finite non-terminal alphabet, N
- A start symbol, S, in N
- A finite set of productions, P

Every CFG defines a context-free language (CFL) which is the set of all sequences of terminals that can be generated by the CFG. In specifying an induction principle the user must construct a CFG that defines a singleton set, the form of the principle for a single inductive definition.

It is never necessary to use the full power of a CFG in order to define a singleton set. If the CFG defines a singleton set then the entire CFG can be rewritten as a single production where the left-hand side is the start symbol and the right-hand side is the unique element in the language. The annotations that the user adds to the CFG are significant; they lose their meaning if the CFG is transformed so that it has only one production.

The user annotates the productions in the grammar to define the way in which subterms of the principle are duplicated and altered so that a

$$\begin{array}{lcl} S & \rightarrow & a\overline{X}b \\ X & \rightarrow & x\overline{Y}y \\ Y & \rightarrow & z_0 a_1 \end{array}$$

Figure 5.10: An example of a Parameterized Context-Free Grammar (PCFG).

principle may be built for any number of inductive definitions. These annotations to the original grammar allow the user to define a language of an infinite number of principles, all similar in structure to the principle for a single inductive definition.

Cloning

For this purpose, we annotate the CFG to show where *cloning* occurs. A non-terminal on the right-hand side of a production may be distinguished by a bar above the symbol, e.g., $S \rightarrow a\overline{A}$. This symbol indicates that the non-terminal must be cloned n times, where n is the numeric parameter of the grammar. Every terminal may have a subscript, which is an index into a list of type parameters. We call a CFG that may be annotated in this way a parameterized context-free grammar (PCFG). The grammar in Figure 5.10 is an example of such a PCFG.

The clones are syntactically identical but semantically distinct. Terminals in the clones have different subscripts. Each clone is assigned a type parameter. The type parameters are drawn from an ordered list of n type parameters. Every terminal's subscript is an index into the list of parameters for the non-terminal on the left-hand side of the production which produces the terminal. Figure 5.11 shows the derivation of the resulting string where n is 2 and the list of parameters is $\{A, B\}$. At the first step the S is expanded. The resulting sentential form contains two X's, one corresponding to the type parameter A and the other corresponding to the type parameter B. At the second step, each X is expanded. The list

$$\begin{aligned} S &\Rightarrow a\ X^{A}\ X^{B}\ b \\ &\Rightarrow a\ x\ Y^{A,A}\ Y^{B,A}\ y\ x\ Y^{A,B}\ Y^{B,B}\ y \\ &\Rightarrow a\ x\ z_{A}\ a_{A}\ z_{B}\ a_{A}\ y\ x\ z_{A}\ a_{B}\ z_{B}\ a_{B}\ y \end{aligned}$$

Figure 5.11: A derivation for the PCFG in Figure 5.10 where $n = 2$ and the list of type parameters is $\{A, B\}$.

$$\overline{a\{x\{\overline{z_0 a_1}\}y\}b}$$

Figure 5.12: A tree representation of the PCFG in Figure 5.10.

of parameters for each Y is extended by an additional parameter. At the third step, each Y is expanded to a pair of terminals, za. z's index is 0, so the zeroth parameter is selected from each Y's list of parameters; a's index is 1, so the first parameter is selected.

An alternate specification of the grammar in Figure 5.10 is given in Figure 5.12. Each non-terminal appears only once in the right-hand side of a production and is the left-hand side of only one production. Consequently, the grammar may be depicted as a tree, where the right-hand side of each production is shown at the location where the left-hand side of each production appears. The bars which indicate that a non-terminal must be cloned are nested as well.

Parameterized Pushdown Automata

Pushdown Automata as Language Generators

There exists a transformation from any CFG to an equivalent pushdown automaton (PDA) (Linz, 2006) that encodes an abstraction of the LL parsing algorithm (Fischer et al., 2010) for that CFG. The purpose of a parser is to recognize members of the language defined by its grammar. However, a parser can also be viewed as a generator. Figure 5.13 is the CFG corresponding to the PCFG of Figure 5.10. Figure 5.14 is the PDA which

$$
\begin{aligned}
S &\rightarrow aXb \\
X &\rightarrow xYy \\
Y &\rightarrow za
\end{aligned}
$$

Figure 5.13: A CFG corresponding to the PCFG in Figure 5.10.

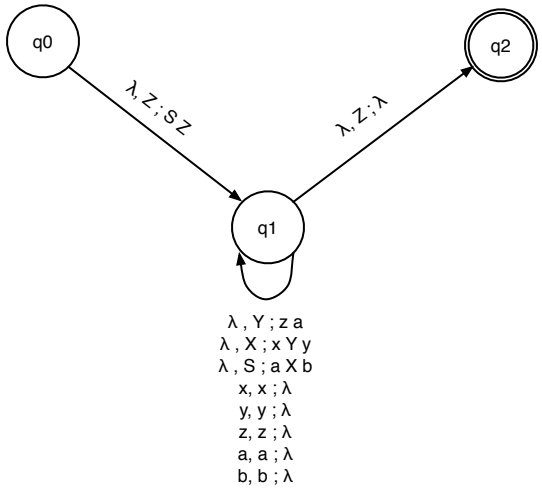


Figure 5.14: A PDA which encodes the abstract LL parsing algorithm for the CFG in Figure 5.13. The transition labels have the format <tape>,<pop>;<push>.

encodes the abstract LL parsing algorithm for this CFG. The transition labels of the PDA have the following format: the symbols on the left of the comma are the symbols to be read off the tape; the symbols on the right of the comma indicate the operations to be performed on the stack. The symbol on the left of the semi-colon indicates the symbol or sequence of symbols to pop off the stack, on the right of the semi-colon are the symbols to push on the stack. The special symbol, Z, is used to indicate the bottom of the stack, and λ means the empty string.

Due to the special properties of the grammar the PDA is deterministic; it recognizes the singleton set of the language specified by the grammar.

The reader can see that it may also be viewed as a generator of the

language. Instead of matching tokens on an input tape it can write them to an output tape. In this interpretation the symbol on the left of the comma is a symbol to be written to the tape, rather than read from the tape. In the example in Figure 5.14 the transition labeled $b, b; \lambda$ indicates that if a b appears on the stack then it may be written to the tape. Similarly, the transition labeled $\lambda, Y; za$ indicates that if a Y is on the top of the stack it may be popped and an a and then a z may be pushed on to the stack while nothing is written to the tape.

Parameterized Generating Pushdown Automata

A PCFG may be transformed into an equivalent PDA in the same manner as a CFG is transformed. The rules which govern the execution of the PDA must be altered to accommodate cloning. We call a PDA which is executed according to this extended set of rules a parameterized generating pushdown automaton (PGPDA).

The execution of the PGPDA is parameterized on n, which must be at least 1. Let I be a list of of n distinct identifiers. I_x indicate the xth identifier for any $1 \leqslant x \leqslant n$.

The stack of the PGPDA is augmented so that each element on the stack is a pair. The first elements of the pair is a terminal or a non-terminal as in a PDA. The second element of the pair is a stack of identifiers. Figure 5.15 shows the PGPDA corresponding to the PDA of Figure 5.14.

In the transition from state 0 to state 1 in Figure 5.14 the label indicates that the bottom symbol, Z, is popped off the stack. Then it is immediately pushed on the stack followed by S, the start symbol of the grammar. In the corresponding PGPDA a pair is pushed onto the stack, the start symbol and an empty stack.

$\lambda, S; aXb$ is a transition label of the PDA in Figure 5.14.

$$\lambda, \{S, st\}; \{a, st\}\{X, [I_1 : st]\} \ldots \{X, [I_n : st]\}\{b, st\}$$

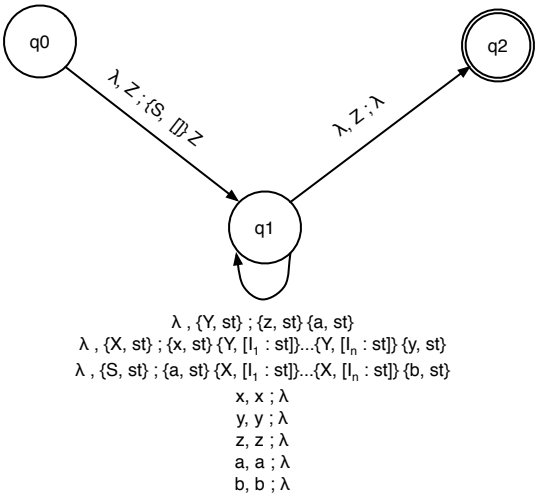


Figure 5.15: A Parameterized Generating Pushdown Automaton(PGPDA) which encodes the abstract generating algorithm for the PCFG in Figure 5.10.

is the corresponding transition label for the PGPDA of Figure 5.15.

This transition label encodes the following rule: If S is on the stack it, and its identifier stack, must be popped, and the following must be pushed onto the stack.

- The pair, $\{b, st\}$, i.e, the symbol b and the stack associated with S
- n pairs of X and an identifier stack. Each identifier stack has a different identifier pushed on the top of the identifier stack associated with S. The identifiers occur in order; I_n is the first identifier to be pushed on the stack, I_1 is the last.
- The pair, $\{a, st\}$, i.e., the symbol a and the stack associated with S.

Practical Specifications

A CFG is a mathematical representation of a concept which finds a practical use in parser generators. In the same way, a PCFG is the mathematical

$$
\begin{array}{lcl}
\text{principle} & \rightarrow & (\text{fragment}|\text{template})* \\
\text{fragment} & \rightarrow & \text{TEMPLATE} \\
\text{template} & \rightarrow & \text{delimiter}(\text{template}|\text{fragment})*
\end{array}
$$

Figure 5.16: A CFG representation of the XML schema for a principle specification.

representation of a concept which find practical use in our specification language and principle generators.

We have chosen an XML representation for our specification. The specification of the XML representation is quite simple and is defined by the grammar in Figure 5.16. This specification allows the user to define a CFG corresponding to the desired principle in the nested fashion illustrated in Figure 5.12. A *template* is a clone non-terminal while a *fragment* is an uncloned non-terminal in the grammar. *principle* corresponds to the start symbol in the grammar. The single terminal in the grammar is *TEMPLATE*. This is a Cheetah (Orr and Rudd, 2005) template string which is filled in appropriately as the derivation is built.

Since it uses templates the representation of the principle is text-based. The alternative is to use some encoding of the abstract syntax tree of an inductive definition in the Calculus of Co-inductive Constructions (CoC). Both are possible and we first explored the alternative approach. However, the alternative approach is less readable and requires the writer of the specification to have a strong knowledge of the abstract syntax corresponding to the concrete syntax to which a normal user is accustomed. Consequently, we believe that the first approach is more accessible and is more likely to gain favor with users.

The algorithm used to construct the principle is that described in Section 5.3. It is implemented in the Python language as Cheetah (Orr and Rudd, 2005) is a Python based template engine. Terminals, i.e., Cheetah templates are filled in using the identifier stack associated with the

enclosing *fragment*.

5.4 The *principle* algorithm

The *principle* tactic makes innovative use of the *external* tactic. This tactic allows the Coq engine to interact with the external environment by invoking an external process and passing arguments to the external process in an XML format. The *external* tactic itself is responsible for packaging Coq terms in XML format suitable for transmission to the external process and for unpacking XML objects into Coq terms. The external process receives the XML formatted data on standard input and returns its response as XML formatted data on standard output which the *external* tactic consumes.

Figure 5.17 illustrates the principle tactic. The dotted rectangles with rounded corners represent Coq processes. The primary Coq process is the process which is computing the proof and for which the induction or recursion principle is required. The secondary Coq process' sole purpose is to transform the generated principle from a Coq vernacular format into an equivalent XML format which can be accepted by the *external* tactic. Dashed arrows indicate the flow of control from one process to another, solid arrows indicate the flow of data from one process to another, and double-headed solid arrows indicate the transfer of data via input and output streams. Squashed rectangles group and serve to further explain the origins of some data. The *principle* tactic itself is a mediator between the *external* tactic and the primary Coq process. On being invoked, it marshals its arguments and passes them to the *external* tactic. These arguments designate the form or principle desired, whether it should be suitable for *Prop*, *Set*, or *Type*, and the number of mutually recursive inductive definitions for which it is to be constructed. The *external* tactic marshals these arguments into an XML format and conveys them to the principle generator.

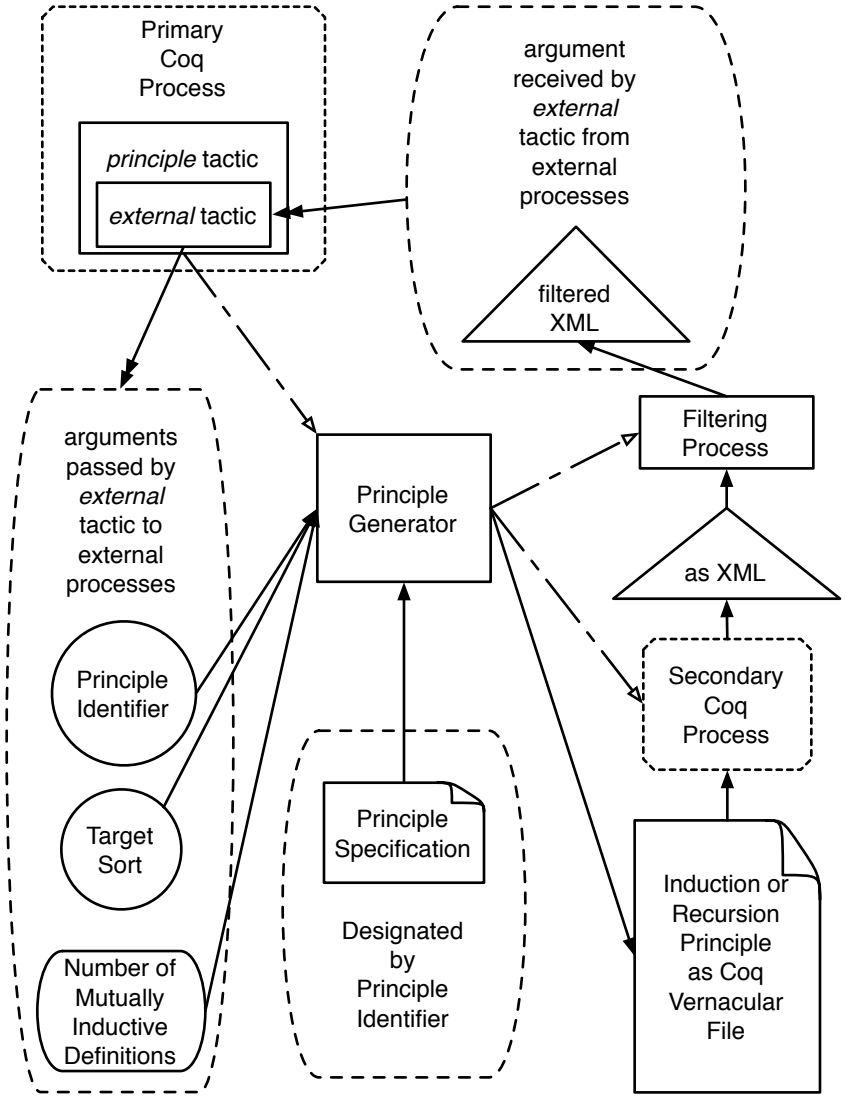


Figure 5.17: A schematic of the operation of the *principle* tactic.

The principle generator unmarshals the arguments and uses the principle identifier to choose among available principle specifications and generate the requested induction principle which it prints to a temporary Coq vernacular file. The secondary Coq process transforms the vernacular file to an XML representation. This XML representation is pruned by the filtering process, so that only the body of the principle is returned to the *external* tactic. The *external* tactic unmarshals the body and the *principle* tactic inserts the principle in the proof-context.

5.5 Types of General Recursion Principles

All principles make use of well-founded induction in some way. The most direct approach is to do structural induction on a proof that some term is accessible under the chosen well-founded relation. Consequently, the proof of accessibility must be larger than the maximum number of steps required for termination of the function. For example, suppose the well-founded relation is the less than relation on the natural numbers. For a natural number, n, the proof of accessibility must have n constructors.

An alternate approach is to use a *bound*. In this approach, induction or recursion is performed on the bound rather than on the actual induction term. The induction or recursion principle may still be general or it may be the structural induction or recursion principle for the type of the bound. Of course, the number of recursive calls required by the algorithm must be less than the number of recursive calls on the bound. The most typical case is to use a *measure* which bounds the term by, e.g., the number of its constructors or its maximum depth. We discuss the specification and types of bounded recursion principles in Section 5.6.

An orthogonal concern is where the burden of proving that the recursive argument is really lower than the previous argument in the well-founded ordering rests. There are two approaches. In the approach

```
fixwf =
fun (A : Type) (B : iType) (F : (A -> B) -> A -> B)
  (R : A -> A -> bool) (W : wf R) (x : A) =>
Acc_rect (fun _ : A => B)
  (fun (x0 : A)
     (_ : forall y : A,
          (fun y0 x1 : A => R y0 x1 = true) y x0 ->
          Acc (fun y0 x1 : A => R y0 x1 = true) y)
     (H : forall y : A,
          (fun y0 x1 : A => R y0 x1 = true) y x0 ->
          (fun _ : A => obj_typ B) y) =>
   let f :=
     fun y : A =>
     match Sumbool.sumbool_of_bool (R y x0) with
     | left r => H y r
     | right _ => arbitrary
     end in
   F f x0) (W x)
     : forall (A : Type) (B : iType),
       ((A -> B) -> A -> B) ->
       forall R : A -> A -> bool,
       wf R -> forall x : A, (fun _ : A => obj_typ B) x
```

Figure 5.18: A general recursion principle that dynamically computes a proof that its argument is lower in a well-founded ordering.

of Balaa and Bertot, illustrated by the example in Figure 5.6, the burden rests on the user of the principle (Balaa and Bertot, 2000). The functional that the user supplies cannot make use of the hypothesis

```
forall y : A, R y x -> P y
```

without supplying the proof that `R y x` holds.

Charguéreaud's recursion principle (Charguéraud, 2009) resembles more closely the classic fixed point combinators. Consequently we call it the *explicit combinator* approach. This approach places the burden of the proof on the recursion principle itself. Since the body of the functional is unknown the proof must be calculated dynamically by the principle. Figure 5.18 is an example of this approach. Here, the expression

```
sumbool_of_bool (R y x)
```

calculates either a proof that `y` is below `x` in the relation `R` or a proof that is not. If `y` is below `x`, it supplies the proof in the expression `f y H`. If `y` is not below `x`, it returns an arbitrary value, designated by the keyword `arbitrary`. For this reason, it is necessary that the type, `B`, of `arbitrary` be inhabitable so that the default can be supplied. Note that `fixwf` invokes the recursion principle for `Acc`, `Acc_rect`.

Generally speaking, the user wishes to prove that the arbitrary value is not returned. This is the same as proving that the fixed point equation holds, regardless of the value of the argument. The proof that the equation holds makes use of a *contraction condition*

$$\forall x f_1 f_2, (\forall y, Ryx \Rightarrow f_1 y = f_2 y) => Ff_1 x = Ff_2 x$$

If this condition is satisfied then the fixed point equation may be used to rewrite fx to Ffx.

The approach of Balaa and Bertot places a burden on the developer since she must supply proofs that recursive arguments are decreasing. However, the *Program* tactic (Sozeau, 2007b,a) includes many tactics which are able to automatically satisfy the proof obligations.

Charguéreaud's approach suffers from a similar drawback. A greater part of the burden of the proof is assumed by the accompanying libraries. However, the user must still demonstrate that the contraction condition is satisfied by the functional in order to show that the function terminates normally. Because this approach depends on the approach of Balaa and Bertot we focus on their approach in the next few sections. In Sections 5.6 and 5.7 we discuss several general recursion principles to show their variety and to demonstrate the versatility of our specification language. In Section 5.8 we discuss explicit combinators.

5.6 Recursion Principles with Measures

Bounded Recursion

All induction in Coq is fundamentally structural induction. Where structural induction on the objects in the domain of interest is impossible it may be useful to transfer the structural induction to another domain.

Bounded recursion is a primitive form of general recursion familiar from mathematics. In this approach to general recursion a bound is chosen. A *measure* function (Section 4.2) that maps objects from the domain of interest to the natural numbers is defined. The value of the measure is shown to remain below the bound which itself decreases at every recursive call.

This approach is likely to be the approach that first suggests itself to the user when structural recursion fails. Moreover, it has a very useful property that it shares with its less primitive relative, *measure-based* recursion (Section 5.6); it maps objects which may belong to several mutually recursive datatypes to the homogeneous space of the `nat` datatype.

This mapping is a great convenience. If a well-founded relation must be defined directly on a number of mutually recursive datatypes the complexity of the relation is proportional to the square of their number. In contrast, a measure defined as a mutually recursive function is only linear in the number of mutually recursive datatypes.

Structure and Restrictions

Measures

In Chapter 4 we discuss some restrictions on measures and their properties. In the following discussion we assume that the measure's range is in `nat`. This is by far the most natural approach because the Coq standard library includes considerable support for reasoning about `nats`.

The measure is not restricted to a function which just describes the size of a term. Such a function is easily produced automatically and where the structure of an inductive definition is available will often be the most natural choice. However, Coq incorporates a module system much like that of ML. In the module system, the structure of a type is likely to be hidden from external users of the module. In this case, the measure must be obtained from the module's functions. The most obvious example is the Coq standard library's collection of set modules. These modules define a `cardinal` function which yields the number of elements in a set. The `cardinal` function is an excellent measure function for defining bounded recursion over sets.

Structure of a Function which Implements Bounded Recursion

A function that implements bounded recursion must take at least two arguments: a `nat`, `n` that acts as a bound and the object of interest. The function is structurally recursive on the bound; the object of interest is carried along at each step. To justify each recursive call it is necessary to show that the value of the measure is bounded by `n`. Figure 5.19 shows the recursion principle for this recursion technique for an inductive definition with just one inductive datatype.The formal statement of the structure of this principle is one of the contributions of our work. To use the principle it is necessary to define:

`A` the inductive type for which the recursion principle is desired

`m_A` a measure function

`P_A` a function that yields the result type (which may be dependent)

`F_A` a proof that it is impossible for the measure of any term in `A` to be less than or equal to 0

```
fun (A : Type) (m_A : A -> nat) (P_A : A -> Type)
  (F_A : forall x : A, m_A x <= 0 -> False)
  (S_A : forall (n : nat) (x : A),
         (forall y : A, m_A y <= n -> P_A y) ->
         m_A x <= S n -> P_A x) =>
fix Rec_A (n : nat) (x : A) {struct n} :
  m_A x <= n -> P_A x :=
  match n as n0 return (m_A x <= n0 -> P_A x) with
  | 0 => fun H : m_A x <= 0 => False_rect (P_A x) (F_A x H)
  | S n0 => fun H : m_A x <= S n0 => S_A n0 x (Rec_A n0) H
  end
     : forall (A : Type) (m_A : A -> nat) (P_A : A -> Type),
       (forall x : A, m_A x <= 0 -> False) ->
       (forall (n : nat) (x : A),
        (forall y : A, m_A y <= n -> P_A y) ->
        m_A x <= S n -> P_A x) ->
       forall (n : nat) (x : A), m_A x <= n -> P_A x
```

(a) The principle.

```
(** val rec_A :
    ('a1 -> nat) -> (nat -> 'a1 -> ('a1 -> __ -> 'a2) -> __
    -> 'a2) -> nat -> 'a1 -> 'a2 **)

let rec rec_A m_A s_A n x =
  match n with
    | O -> assert false (* absurd case *)
    | S n0 -> s_A n0 x (fun x0 _ -> rec_A m_A s_A n0 x0) __
```

(b) The principle extracted to O'Caml.

Figure 5.19: A general recursion principle that makes use of a bound.

`S_A` the recursive step, i.e., a proof that if `P_A` holds for every term with measure less than or equal to `n` it holds for any term with size less than or equal to `S n`

The principle is structurally recursive on `n` rather than on `x` although the result of the function, `P_A x` is dependent on `x` and entirely unrelated to `n`. This is strikingly evident in the extracted version of the function where the only *match* expression is a match on the bound.

```
(** val g : (M.t -> M.t) -> M.t -> M.t **)

let g f s =
  let rec f0 n s0 =
    match n with
      | O -> f s0
      | S n0 ->
          let z = f s0 in
          (match Dep.equal z s0 with
             | Left -> z
             | Right -> f0 n0 z)
  in f0 (M.cardinal s) s
```

Figure 5.20: An illustration of the use of a bounded recursion principle. This function recursively applies its function argument, `f`, to the set argument, `s`, until a fixed point is reached. The original bound is the cardinality of the set. Applying `f` to `s` must reduce the cardinality of the remaining set by at least one element.

`S_A` (`s_A` in the extracted version) is itself a function and is likely to contain its own *match* expression which deconstructs the argument, `x`. Figure 5.20 shows the extracted version of a function that recursively applies its argument, `f`, to a set, `s`, until `s` reaches a fixed point.

The outermost case statement in `f0` matches the argument `n`, an upper bound on the cardinality of the set. This obscures the true purpose of the function; `n` was introduced solely to prove termination. It would be much better if only the innermost case statement appeared.

Using the propositional version of `nat`, `natP`, would certainly not solve the problem. It is impossible to deconstruct an element in `natP`, which is in *Prop*, to form an element in `M.t`, which is in *Set*.

Specification

The specification of the principle is as described in Section 5.3 and requires just 38 lines. Figure 5.21 shows the same autogenerated principle for two mutually recursive datatypes. The additional arguments to the principle

```
fun (A B : Type) (m_A : A -> nat) (m_B : B -> nat)
  (P_A : A -> Type) (P_B : B -> Type)
  (F_A : forall x_A : A, m_A x_A <= 0 -> False)
  (F_B : forall x_B : B, m_B x_B <= 0 -> False)
  (S_A : forall (n : nat) (x_A : A),
         (forall y_A : A, m_A y_A <= n -> P_A y_A) ->
         (forall y_B : B, m_B y_B <= n -> P_B y_B) ->
         m_A x_A <= S n -> P_A x_A)
  (S_B : forall (n : nat) (x_B : B),
         (forall y_A : A, m_A y_A <= n -> P_A y_A) ->
         (forall y_B : B, m_B y_B <= n -> P_B y_B) ->
         m_B x_B <= S n -> P_B x_B) =>
fix Rec_A (n : nat) (x_A : A) {struct n} :
  m_A x_A <= n -> P_A x_A :=
  match n as n0 return (m_A x_A <= n0 -> P_A x_A) with
  | 0 => fun H : m_A x_A <= 0 =>
         False_rect (P_A x_A) (F_A x_A H)
  | S n0 => fun H : m_A x_A <= S n0 =>
         S_A n0 x_A (Rec_A n0) (Rec_B n0) H
  end
with Rec_B (n : nat) (x_B : B) {struct n} :
  m_B x_B <= n -> P_B x_B :=
  match n as n0 return (m_B x_B <= n0 -> P_B x_B) with
  | 0 => fun H : m_B x_B <= 0 =>
         False_rect (P_B x_B) (F_B x_B H)
  | S n0 => fun H : m_B x_B <= S n0 =>
         S_B n0 x_B (Rec_A n0) (Rec_B n0) H
  end for Rec_A
     : forall (A B : Type) (m_A : A -> nat)
         (m_B : B -> nat) (P_A : A -> Type)
         (P_B : B -> Type),
       (forall x_A : A, m_A x_A <= 0 -> False) ->
       (forall x_B : B, m_B x_B <= 0 -> False) ->
       (forall (n : nat) (x_A : A),
        (forall y_A : A, m_A y_A <= n -> P_A y_A) ->
        (forall y_B : B, m_B y_B <= n -> P_B y_B) ->
        m_A x_A <= S n -> P_A x_A) ->
       (forall (n : nat) (x_B : B),
        (forall y_A : A, m_A y_A <= n -> P_A y_A) ->
        (forall y_B : B, m_B y_B <= n -> P_B y_B) ->
        m_B x_B <= S n -> P_B x_B) ->
       forall (n : nat) (x_A : A), m_A x_A <= n -> P_A x_A
```

Figure 5.21: A bounded general recursion principle for two mutually recursive datatypes.

are `B`, `m_B`, `P_B`, `F_B`, and `S_B` and must be defined in the same way as the corresponding arguments for the `A` type. The reader will observe that a principle for even as many as two mutually recursive datatypes is lengthy and would doubtless prefer that it be autogenerated. It is also obvious that the principle does have the fractal-like quality that makes it amenable to autogeneration using PGPDAs.

Discussion

In this section we have described a primitive form of general recursion, bounded recursion. While primitive, it is a form of recursion that is intuitive to a student of mathematics. Like measure-based recursion it allows reasoning about the homogeneous space of the type of the measure's domain rather than the in general, heterogeneous space of an inductive definition which may include a number of inductive datatypes. However, extracted code does not elide the bound. Consequently, the bound appears in an extracted function, which is undesirable since it is not an essential part of the computation but only used to prove termination.

We have shown that our specification language is expressive enough to describe the overall structure of bounded recursion and to allow a bounded recursion principle to be autogenerated for any number of mutually inductive definitions. We have constructed a general recursion principle for bounded recursion.

Bounded Recursion With a Dependent Measure

The CoC has a particularly rich type-system. It incorporates dependent types (Altenkirch et al., 2005), i.e., types which include values. In some cases the value is a constant parameter. Figure 5.22 shows the definition of a tree structure where every value in the tree is bounded. When the value is a constant, the general bounded recursion principle described

```
Inductive ltree (n : nat) : Set :=
  lleaf : ltree n
| lnode :
    forall p : nat,
      p <= n -> ltree n -> ltree n -> ltree n
```

Figure 5.22: A definition of a tree datatype where every element in the tree is no greater than `n`.

```
Inductive htree (A : Set) : nat -> Set :=
| hleaf : A -> htree A 1
| hnode :  A ->
    forall n : nat,
      htree A n -> htree A n -> htree A (S n)
```

Figure 5.23: A definition of a tree datatype where the height of the tree is part of the type.

in Section 5.6 is suitable. All constant parameters may be defined using Coq's sectioning mechanism; by this means they are made implicit and do not need to be included explicitly in the recursion principle.

In general, however, the value may vary. Figure 5.23 show the definition of a tree structure where the height of the tree is part of the type. One of the most usual uses of dependent types is to include within the type of a data structure such as a list or a tree some measure of its size. We show that it is possible to write a general bounded recursion principle that uses the size information incorporated in such a dependent type as a measure. Figure 5.24 shows our recursion principle for two mutually recursive datatypes.

We discuss only the arguments for the `A` type; the arguments for the `B` type are defined in the same way. To use the principle it is necessary to define for each inductive definition:

`A` the result type of the inductive type for which the recursion principle is desired

```
fun (A B : nat -> Type) (P_A : forall x : nat, A x -> Type)
  (P_B : forall x : nat, B x -> Type)
  (F_A : forall x : nat, A x -> x <= 0 -> False)
  (F_B : forall x : nat, B x -> x <= 0 -> False)
  (S_A : forall (n x : nat) (x_A : A x),
         (forall (y : nat) (y_A : A y), y <= n -> P_A y y_A) ->
         (forall (y : nat) (y_B : B y), y <= n -> P_B y y_B) ->
         x <= S n -> P_A x x_A)
  (S_B : forall (n x : nat) (x_B : B x),
         (forall (y : nat) (y_A : A y), y <= n -> P_A y y_A) ->
         (forall (y : nat) (y_B : B y), y <= n -> P_B y y_B) ->
         x <= S n -> P_B x x_B) =>
fix Rec_A (n x : nat) (x_A : A x) {struct n} :
  x <= n -> P_A x x_A :=
  match n as n0 return (x <= n0 -> P_A x x_A) with
  | 0 => fun H : x <= 0 =>
         False_rect (P_A x x_A) (F_A x x_A H)
  | S n0 => fun H : x <= S n0 =>
         S_A n0 x x_A (Rec_A n0) (Rec_B n0) H
  end
with Rec_B (n x : nat) (x_B : B x) {struct n} :
  x <= n -> P_B x x_B :=
  match n as n0 return (x <= n0 -> P_B x x_B) with
  | 0 => fun H : x <= 0 =>
         False_rect (P_B x x_B) (F_B x x_B H)
  | S n0 => fun H : x <= S n0 =>
         S_B n0 x x_B (Rec_A n0) (Rec_B n0) H
  end for Rec_A
     : forall (A B : nat -> Type)
         (P_A : forall x : nat, A x -> Type)
         (P_B : forall x : nat, B x -> Type),
       (forall x : nat, A x -> x <= 0 -> False) ->
       (forall x : nat, B x -> x <= 0 -> False) ->
       (forall (n x : nat) (x_A : A x),
        (forall (y : nat) (y_A : A y), y <= n -> P_A y y_A) ->
        (forall (y : nat) (y_B : B y), y <= n -> P_B y y_B) ->
        x <= S n -> P_A x x_A) ->
       (forall (n x : nat) (x_B : B x),
        (forall (y : nat) (y_A : A y), y <= n -> P_A y y_A) ->
        (forall (y : nat) (y_B : B y), y <= n -> P_B y y_B) ->
        x <= S n -> P_B x x_B) ->
       forall (n x : nat) (x_A : A x), x <= n -> P_A x x_A
```

Figure 5.24: A bounded general recursion principle for two mutually recursive datatypes where the measure is contained in the type.

`P_A` a function that yields the result type (which may be dependent)

`F_A` a proof that it is impossible for the measure of any term to be less than or equal to 0

`S_A` the recursive step, i.e., a proof that if `P_A` holds for every term with measure less than or equal to `n` it holds for any term with size less than or equal to `S n`

To our knowledge this is the first description of a bounded general recursion principle which takes advantage of dependent types which themselves contain a measure.

Measure-based General Induction

Measure-based general recursion is similar to bounded recursion (Section 5.6), however, the bound is implicit instead of explicit. Figure 5.25 is a measure-based adaptation of the approach of Balaa and Bertot.

Like bounded-recursion measure-based recursion is powerful because it collapses the space of multiple mutually inductive datatypes into the space of a single inductive datatype. The more general approach is to define a well-founded relation between every pair of the mutually inductive datatypes that make up an inductive definition. This can be burdensome to the programmer where there are many mutually inductive datatypes. Using measure-based recursion the developer can make use of the well-founded relation between the elements in the range of the measure function.

```
fun (A B : Type) (m_A : A -> nat) (m_B : B -> nat)
  (P_A : A -> Type) (P_B : B -> Type)
  (F_A : forall x_A : A,
         (forall y_A : A, m_A y_A < m_A x_A -> P_A y_A) ->
         (forall y_B : B, m_B y_B < m_A x_A -> P_B y_B) ->
         P_A x_A)
  (F_B : forall x_B : B,
         (forall y_A : A, m_A y_A < m_B x_B -> P_A y_A) ->
         (forall y_B : B, m_B y_B < m_B x_B -> P_B y_B) ->
         P_B x_B) =>
fix Rec_A (x_A : A) (r_A : Acc lt (m_A x_A)) {struct r_A} :
  P_A x_A :=
  F_A x_A
    (fun (y_A : A) (H_A : m_A y_A < m_A x_A) =>
     Rec_A y_A (Acc_inv r_A (m_A y_A) H_A))
    (fun (y_B : B) (H_B : m_B y_B < m_A x_A) =>
     Rec_B y_B (Acc_inv r_A (m_B y_B) H_B))
with Rec_B (x_B : B) (r_B : Acc lt (m_B x_B)) {struct r_B} :
  P_B x_B :=
  F_B x_B
    (fun (y_A : A) (H_A : m_A y_A < m_B x_B) =>
     Rec_A y_A (Acc_inv r_B (m_A y_A) H_A))
    (fun (y_B : B) (H_B : m_B y_B < m_B x_B) =>
     Rec_B y_B (Acc_inv r_B (m_B y_B) H_B))
for Rec_A
     : forall (A B : Type) (m_A : A -> nat)
         (m_B : B -> nat) (P_A : A -> Type)
         (P_B : B -> Type),
       (forall x_A : A,
        (forall y_A : A, m_A y_A < m_A x_A -> P_A y_A) ->
        (forall y_B : B, m_B y_B < m_A x_A -> P_B y_B) ->
        P_A x_A) ->
       (forall x_B : B,
        (forall y_A : A, m_A y_A < m_B x_B -> P_A y_A) ->
        (forall y_B : B, m_B y_B < m_B x_B -> P_B y_B) ->
        P_B x_B) ->
       forall x_A : A, Acc lt (m_A x_A) -> P_A x_A
```

Figure 5.25: A measure-based general recursion principle.

5.7 Recursion Principles Using a General Relation

All of the examples in Section 5.6 are specific examples of recursion in which the well-founded relation is defined by means of a measure. There is considerably support for measure-based general recursion in the Coq standard library and measure-based recursion is often preferred. In some cases, the user may prefer to explicitly define a well-founded relation rather than make use of a measure. Figure 5.26 shows a recursion principle appropriate for this case. The principle difference between recursion principles that rely on a general relation and measure-based principles is that, whereas the measure must be divided into multiple recursive components, one for each mutually inductive type, the relation must make use of sum types. In the example, the type of the relation is

```
A + B -> A + B -> Prop
```

Every use of the relation must then form the correct type from the ground type using the `inl` and `inr` constructors of the type. To accommodate this necessity the grammar in Figure 5.16 is extended with an additional *coercion* terminal. The *coercion* tag identifies variables which must be coerced to the correct sum type and the necessary expression is generated programmatically.

5.8 Recursion Principles as Explicit Combinators

The *explicit combinator* approach produces recursion principles with a structure that more closely resembles that of the classic fixed point combinators of the untyped lambda calculus. However, it is necessary to supply an arbitrary value of the correct type. Consequently, their use is restricted

```
fun (A B : Type) (P_A : A -> Type) (P_B : B -> Type)
  (R : A + B -> A + B -> Prop)
  (F_A : forall x_A : A,
         (forall y_A : A,
          R (inl B y_A) (inl B x_A) -> P_A y_A) ->
         (forall y_B : B,
          R (inr A y_B) (inl B x_A) -> P_B y_B) ->
         P_A x_A)
  (F_B : forall x_B : B,
         (forall y_A : A,
          R (inl B y_A) (inr A x_B) -> P_A y_A) ->
         (forall y_B : B,
          R (inr A y_B) (inr A x_B) -> P_B y_B) ->
         P_B x_B) =>
fix Rec_A (x_A : A) (r_A : Acc R (inl B x_A)) {struct r_A} :
  P_A x_A :=
  F_A x_A
    (fun (y_A : A) (H_A : R (inl B y_A) (inl B x_A)) =>
     Rec_A y_A (Acc_inv r_A (inl B y_A) H_A))
    (fun (y_B : B) (H_B : R (inr A y_B) (inl B x_A)) =>
     Rec_B y_B (Acc_inv r_A (inr A y_B) H_B))
with Rec_B (x_B : B) (r_B : Acc R (inr A x_B)) {struct r_B} :
  P_B x_B :=
  F_B x_B
    (fun (y_A : A) (H_A : R (inl B y_A) (inr A x_B)) =>
     Rec_A y_A (Acc_inv r_B (inl B y_A) H_A))
    (fun (y_B : B) (H_B : R (inr A y_B) (inr A x_B)) =>
     Rec_B y_B (Acc_inv r_B (inr A y_B) H_B))
for Rec_A
     : forall (A B : Type) (P_A : A -> Type)
         (P_B : B -> Type) (R : A + B -> A + B -> Prop),
       (forall x_A : A,
        (forall y_A : A,
         R (inl B y_A) (inl B x_A) -> P_A y_A) ->
        (forall y_B : B,
         R (inr A y_B) (inl B x_A) -> P_B y_B) -> P_A x_A) ->
       (forall x_B : B,
        (forall y_A : A,
         R (inl B y_A) (inr A x_B) -> P_A y_A) ->
        (forall y_B : B,
         R (inr A y_B) (inr A x_B) -> P_B y_B) -> P_B x_B) ->
       forall x_A : A, Acc R (inl B x_A) -> P_A x_A
```

Figure 5.26: A recursion principle that make use of a general well-founded relation.

to the definition of functions that have a simple result type, since it is impossible to construct an arbitrary value for a dependently typed result.

Figure 5.27 shows a recursion principle for a mutually inductive type with two types. Note that the combinator is complicated by the necessity that `R`, the relation, be over a some type, but that each functional be for a single type. It is not itself recursive, the recursion is contained in the call to `Acc_rect`.

For each combinator it is also necessary to prove that the fixed point equation holds. Let F_1 and F_2 be the two functionals. Then $f_1 = \mathit{fix2}\ F_1\ F_2$ and $f_2 = \mathit{fix2}\ F_2\ F_1$. It is necessary to show that $f_1 x = F_1 f_1 f_2$ and $f_2 = F_2 f_1 f_2$ if the appropriate contraction condition holds,

Existing proofs of the fixed point equation for combinators for simple types have been constructed using fairly lengthy tactic scripts composed of powerful tactics. We suspect that it will take considerable study to find a way to automatically generate such proofs for automatically constructed combinators for multiple mutually inductive types and that this is a useful direction for future work.

The combinators that are constructed by this method are, in general, not suitable for use by the *induction* tactic. They are more appropriate for explicitly defining recursive specifications (Charguéraud, 2009). A *Combinator* directive which allows the user to define and name combinators for mutually recursive types, akin to the *Scheme* command for specifying structural induction principles would be desirable.

5.9 Discussion and Related Work

In this chapter we have demonstrated a facility for specifying the structure of a provably terminating fixed point combinator, i.e. general recursion principle, so that the principle appropriate for a term that inhabits a type defined by any number of mutually inductive datatypes can be

```
fun (Z : Type) (arbitrary : Z) (A B : Type)
  (R : A + B -> A + B -> Prop)
  (R_dec : forall m n : A + B, {R m n} + {~ R m n})
  (F_A : (A -> Z) -> (B -> Z) -> A -> Z)
  (F_B : (A -> Z) -> (B -> Z) -> B -> Z) (x : A + B)
  (Wf : well_founded R) =>
Acc_rect (fun _ : A + B => Z)
  (fun (x0 : A + B)
     (_ : forall y : A + B, R y x0 -> Acc R y)
     (H : forall y : A + B,
          R y x0 -> (fun _ : A + B => Z) y) =>
   match x0 with
   | inl x_A =>
       F_A
         (fun y_A : A =>
          match R_dec (inl B y_A) x0 with
          | left r => H (inl B y_A) r
          | right _ => arbitrary
          end)
         (fun y_B : B =>
          match R_dec (inr A y_B) x0 with
          | left r => H (inr A y_B) r
          | right _ => arbitrary
          end) x_A
   | inr x_B =>
       F_B
         (fun y_A : A =>
          match R_dec (inl B y_A) x0 with
          | left r => H (inl B y_A) r
          | right _ => arbitrary
          end)
         (fun y_B : B =>
          match R_dec (inr A y_B) x0 with
          | left r => H (inr A y_B) r
          | right _ => arbitrary
          end) x_B
   end) (Wf x)
     : forall Z : Type,
       Z ->
       forall (A B : Type) (R : A + B -> A + B -> Prop),
       (forall m n : A + B, {R m n} + {~ R m n}) ->
       ((A -> Z) -> (B -> Z) -> A -> Z) ->
       ((A -> Z) -> (B -> Z) -> B -> Z) ->
       forall x : A + B,
       well_founded R -> (fun _ : A + B => Z) x
```

Figure 5.27: A fixed point combinator for two types.

automatically generated. We have constructed a tactic that populates the proof-context with the appropriate general recursion principle.

We believe that our facility would be invaluable for gathering together the variety of general recursion principles that have been developed for the Coq proof-assistant and making them all equally accessible to the user. To our knowledge, ours is the first specification language or tactic of its kind.

Our specification approach is suitable for general recursion strategies based on the construction of provably terminating fixed point combinators. Other strategies (Bove and Capretta, 2005; Megacz, 2007) are computation specific, i.e., the recursion strategy must be redeveloped for every computation and are consequently less general. Our specification is unsuitable and unnecessary for these strategies.

The strategy of Bove and Capretta involves constructing an inductive type which has a strong correspondence to the desired computation and is strictly larger than the number of steps required by the computation (Bove and Capretta, 2005). The structural induction or recursion principle of the synthesized type becomes the core of the computation and thus there is no necessity for a general recursion principle

Megacz has developed an approach to general recursion which makes use of a co-inductive computation monad (Megacz, 2007). The return type of the user-specified function is the computation monad which has a result of the desired type. The computation monad is a co-inductive type, i.e., it is a type that allows infinite terms. If the user is able to supply an inductive term that bounds the length of the computation then the computation is terminating.

Both strategies are interesting and potentially valuable. The strategy of Bove et al. is subject to the usual difficulties with extraction since the synthesized type must not be in *Prop* if the result of the computation is not in *Prop*. The synthesized type will not be elided by the extraction

mechanism unless the whole term is elided. Megacz's strategy is novel and appears more compatible with the extraction mechanism.

The *Program* tactic (Sozeau, 2007b,a) and its associated libraries is based on a fixed point combinator approach and makes use of the general technique of Balaa and Bertot. While our approach exposes the structure of a general recursion principle, the *Program* tactic hides the structure of the principle exposing only the computational parts of recursion and not the proof of termination. All parts of the termination proof are expressed in terms of *subset types*, i.e., types where the general type of the term is modified by a predicate which restricts the terms which actually inhabit the type. The subset types are not visible in the specification. The *Program* tactic automatically synthesizes the necessary predicates to form the subset types and presents any lemmas that cannot be automatically satisfied by the associated tactics to the user.

The *Program* tactic is not applicable in the case of mutually inductive definitions. We believe that this is not a fundamental restriction but rather due to an implementation decision. We believe that the *Program* tactic and our specification tools could be combined to make a more general and powerful tactic that is not restricted to a single general recursion principle and to types that are not mutually inductive.

Part II

Extending Structural Induction and Recursion

```
ntree_rect =
fun (A : Set) (P : ntree A -> Type)
  (f : forall (a : A) (n : nforest A), P (node A a n))
  (n : ntree A) =>
match n as n0 return (P n0) with
| node x x0 => f x x0
end
     : forall (A : Set) (P : ntree A -> Type),
       (forall (a : A) (n : nforest A), P (node A a n)) ->
       forall n : ntree A, P n
```

Figure 5.28: The automatically generated recursion principle for `ntree`.

Induction is fundamental to proof development in Coq. The *induction* tactic's basic function is application of the appropriate induction or recursion principle to the current goal; it also performs some preprocessing of the current goal and some postprocessing of generated subgoals. Variants of the *induction* tactic allow the user to specify alternate induction principles. The default principle is the one that Coq automatically generates when processing an inductive definition.

No facilities exist in Coq for generating alternate principles. Even the default autogenerated recursion principle is usually not the one that is desired when the definition is mutually inductive. Figure 5.28 shows the default structural recursion principle generated by Coq for `ntree`. Although `ntree` and `nforest` are mutually inductive the default recursion principles generated for `ntree` and `nforest` are not mutually recursive.

To obtain a mutually recursive induction principle the user must make use of the *Scheme* command. Figure 5.29 shows the mutually recursive principle obtained by the command

```
Scheme ntree_r := Induction for ntree Sort Type with
nforest_r := Induction for nforest Sort Type.
```

The fundamental difference between the default principle and that generated by the *Scheme* command is that the default principle's unique

```
ntree_r =
fun (A : Set)
  (P_ntree : ntree A -> Type)
  (P_nforest : nforest A -> Type)
  (f_node : forall (a : A) (n : nforest A),
       P_nforest n -> P_ntree (node A a n))
  (f_nil : P_nforest (nil A))
  (f_cons : forall n : ntree A,
        P_ntree n ->
        forall n0 : nforest A, P_nforest n0 ->
        P_nforest (cons A n n0)) =>
fix F_ntree (n : ntree A) : P_ntree n :=
  match n as n0 return (P_ntree n0) with
  | node a n0 => f_node a n0 (F_nforest n0)
  end
with F_nforest (n : nforest A) : P_nforest n :=
  match n as n0 return (P_nforest n0) with
  | nil => f_nil
  | cons n0 n1 => f_cons n0 (F_ntree n0) n1 (F_nforest n1)
  end
for F_ntree
     : forall (A : Set) (P_ntree : ntree A -> Type)
         (P_nforest : nforest A -> Type),
       (forall (a : A) (n : nforest A),
        P_nforest n -> P_ntree (node A a n)) ->
       P_nforest (nil A) ->
       (forall n : ntree A,
        P_ntree n ->
        forall n0 : nforest A, P_nforest n0 ->
        P_nforest (cons A n n0)) ->
       forall n : ntree A, P_ntree n
```

Figure 5.29: The recursion principle generated by the *Scheme* command for `ntree`. `A` is the function *parameter*. `P_ntree` and `P_nforest` are *statements of conclusion*. `f_node`, `f_nil`, and `f_cons` are *principal premises*.

principal premise, `f`, does not provide the user with any useful induction hypothesis to build the induction step. In fact, the default principle is not even a fixed point. To prove any fact about `ntrees` the user must prove that a fact holds for any `ntree` constructed from a `node` constructor. Since that is every `ntree` the default recursion principle is worthless.

This situation has two unfortunate drawbacks. First, the novice user will learn to use the *induction* tactic without understanding its underlying mechanism. When the autogenerated recursion principle is inadequate, which will always happen with a mutually inductive definition, the novice user is stymied. Second, even a knowledgeable user will prefer to use the default principle whenever it is adequate, since it is the only one that is automatically generated. All other principles the user must write out and update whenever the inductive definition is changed. This preference is likely to constrain the user to prefer a less general solution that can be implemented automatically to a more general solution that requires more labor.

We argue that, rather than restricting the user to a single default induction principle, it is better to supply the user with a family of principles appropriate to different tasks and the means of specifying the default principle for the particular task at hand. This will assist the novice user by making clear the functionality of the *induction* tactic and the importance of choosing the correct principle. It will assist the more advanced user by reducing the labor of constructing and maintaining an alternate recursion principle as the inductive definition for which the principle is defined evolves.

The structural induction or recursion principle generated automatically by Coq is always homogeneous, i.e, it encodes an inductive definition over objects of one type. This is true regardless of whether it is generated as a side-effect of processing its inductive definition or explicitly via the *Scheme* command. For example, the structural recursion principle for

`ntree`, `ntree_r` (Figure 5.29) encodes an inductive definition exclusively over elements constructed from `ntree` and `nforest` constructors. A homogeneous principle has the basic syntactic structure from which other principles can be syntactically derived.

In Chapter 6 we describe a syntactic transformation from the automatically generated homogeneous structural recursion principle to a heterogeneous structural recursion principle. This transformation is applicable when the inductive definition is parameterized. We show that the heterogeneous principle allows the user to generalize the property of inclusion in a list to any element-wise property over lists. We demonstrate that most properties of inclusion that are proved in the `List` module of the Coq standard library can be generalized analogously.

In Chapter 7 we address a different problem. When a user makes use of a structural recursion principle a new subgoal is generated, one for every constructor in the type of the induction term. There is no information within each subgoal to indicate which constructor each new subgoal corresponds to. Consequently, a user or an automated tactic must rely on a mapping between the position of the subgoal and the location of the constructor within the inductive definition of the type. The user is likely to become confused while automated tactics are likely to fail if the position of a constructor within the inductive definition is changed. We propose extending structural induction and recursion principles with a `case` hypothesis which allows the user and automated tactics to identify the case to which every subgoal corresponds.

The CoC does not allow dependently typed variadic types. Consequently, it is necessary to use a `list` to specify a variadic type for a constructor in an inductive definition. However, Coq does not place a variadic interpretation on `list` when it is used in this way. Instead, it treats the list as opaque and consequently generates a structural induction or recursion principle which is in general useless because it contains no induction hy-

pothesis for the variadic portions of the type. In Chapter 8 we introduce a programmer's idiom for expressing variadic types via the `list` type. We demonstrate a tactic that can automatically generate the variadic portions of any recursion principle. We describe an experiment which demonstrates that variadic structural recursion principles can be employed in a disciplined development with minimal readjustment.

In this section we demonstrate three useful extensions to the structural recursion principles that Coq automatically generates. We envision a more powerful version of Coq's *induction* tactic, like the *principle* tactic described in Chapter 5, that allows the user to specify the extension or combination of extensions desired. We believe that such an extensions to Coq's automatic facilities for structural recursion will facilitate development in Coq by experts and novices alike.

6.1 Introduction

In this chapter, we discuss the synthesis and use of a heterogeneous structural principle from a homogeneous structural principle.

A heterogeneous form of structural recursion is possible where the inductive definition is parameterized. `ntree` is parameterized on `A`, the type of the elements of a tree and hence of the elements of the trees in a forest (Figure 2.1(a)). Figure 6.1 shows the extracted version of such a heterogeneous structural recursion principle. Note that the only difference between this recursion principle and the less general principle for `ntree` in Figure 5.29 is the `f0` call, which produces a value for `a`, the node's element. Of course, where `f0` is the identity, the heterogeneous recursion principle of Figure 6.1 reduces to the homogeneous principle of Figure 5.29.

The heterogeneous recursion principle is useful where it is desired to abstract a property or computation over the elements in the tree. Consider a function that calculates the number of elements in a tree. The number

```
(** val ntree_rect_het :
    ('a1 -> 'a1 nforest -> 'a2 -> 'a4 -> 'a3) ->
    'a4 -> ('a1 ntree -> 'a3 -> 'a1 nforest ->
    'a4 -> 'a4) -> ('a1 -> 'a2) -> 'a1 ntree ->
    'a3 **)

let ntree_rect_het f1 f2 f3 f0 n =
  let rec f = function
    | Node (a, n1) -> f1 a n1 (f0 a) (f4 n1)
  and f4 = function
    | Nil -> f2
    | Cons (n1, n2) -> f3 n1 (f n1) n2 (f4 n2)
  in f n
```

Figure 6.1: The extracted version of a heterogeneous structural recursion principle for `ntree`.

```
In =
fun (A : Type) (a : A) (l : list A) =>
list_rect (fun _ : list A => Prop) False
  (fun (b : A) (_ : list A) (H : Prop) => a = b \/ H) l
     : forall A : Type, A -> list A -> Prop
```

(a) With invocation of structural recursion principle

```
In =
fun (A : Type) (a : A) =>
fix In (l : list A) {struct l} : Prop :=
  match l with
  | nil => False
  | b :: m => b = a \/ In m
  end
     : forall A : Type, A -> list A -> Prop
```

(b) With structural recursion principle inlined

Figure 6.2: The `In` function on lists.

of elements can be defined in many ways. For example, suppose that `A` is `nat`. In that case, it may be reasonable to let `f0` be

```
fun (n : nat) => 1
```

so that each natural number is viewed as a single element. On the other hand, suppose that `A` is `list nat`. In that case, it may be reasonable to let `f0` be a recursive function that calculates the number of elements in the list.

Another instructive example is the function `In` over lists shown in Figure 6.2. Note that the proposition, `a = b` must be asserted explicitly in the second principal premise since `list_rect` is a homogeneous structural recursion principle. `In` is a recursive function that computes a proposition from a list argument. The proposition can be proved exactly when the element `a` is in the list `l`. For example, assume `l` is

```
2 :: 3 :: nil
```

Then `In 4 l` is

```
Disjunct =
fun (A : Type) (P_A : A -> Prop) =>
fix Disjunct (l : list A) : Prop :=
  match l with
  | nil => False
  | cons b m => P_A b \/ Disjunct m
  end
     : forall A : Type, (A -> Prop) -> list A -> Prop
```

Figure 6.3: The `Disjunct` function on lists.

```
2 = 4 \/ (3 = 4 \/ False)
```

which it is clearly impossible to prove. Similarly, `In 2 l` is

```
2 = 2 \/ (3 = 2 \/ False)
```

which can certainly be proved. The definition of `In` is accompanied by many ancillary proofs since containment in a list is an essential property of a list. Unfortunately, this definition of `In` assumes that `A` is both the type of the element being searched for and the type of the elements in the list and so do all the ancillary proofs. So, if `l` is instead

```
(2::3::nil)::nil
```

it is impossible to search for 2 within the list.

The proposition `a = b` could be abstracted if the structural recursion principle were heterogeneous yielding the function `Disjunct` (Figure 6.3) which is more generally useful than the `In` function. For example,

```
Disjunct (In 2) ((2::3::nil)::nil)
```

evaluates to

```
(2 = 2 \/ 3 = 2 \/ False) \/ False
```

which is obviously provable.

In the rest of the chapter we discuss our mechanism for generating heterogeneous structural recursion principles. We demonstrate the utility of heterogeneous principles by showing that they can be used to construct a `Disjunct` function which generalizes an `In` function in a valuable way. We show that many properties of `In` proved in the `List` module can be generalized and that this generalization makes a module more useful to external users. We also demonstrate that such a generalization is applicable to other data structures.

6.2 Implementation

Inductive definitions in the CoC can be sophisticated. Structural induction principles are correspondingly complex. Rather than re-implement the work required to generate these principles we make use of a source-to-source translation.

We assume that the source of the inductive definition has a canonical form; in particular, the universal quantification constructor is used only when an argument name is needed. For example, the types

```
forall (t : ntree bool), nat
```

and

```
ntree bool -> nat
```

are identical. The second form is the canonical form; since the identifier `t` is not used in the result type, i.e., `nat`, it is unnecessary and the equivalent expression that uses the arrow constructor is preferred.

The transformation has two principle components: adding additional parameters to the list of conclusions and principal premises and modifying existing principal premises and function bodies to make use of the additional parameters.

Identifying Syntactic Elements

To perform a transformation on the structural recursion principle it is necessary to be able to identify and group syntactic elements. Figure 5.29 shows the homogeneous principle for `ntree`. By convention the arguments to the recursion principle are assembled in the following order:

parameters these may have any type

statements of conclusion the head type of every statement of conclusion is a sort, i.e., `Prop`, `Set`, or `Type`

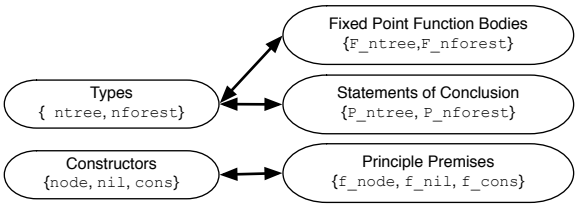
principal premises the head type of every principal premise is an application of a statement of conclusion

Thus, every principal premise can be identified by its type. By examining the head type of every principal premise it is possible to extract the name of every statement of conclusion. By this, every argument to the recursion principle can be placed in its correct category.

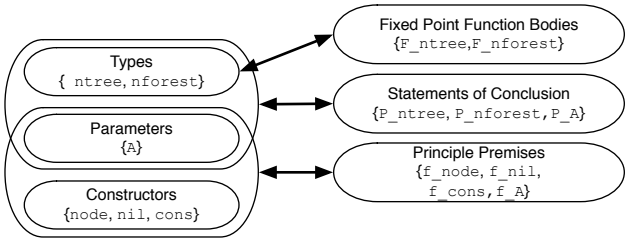
Adding Parameters

The syntactic structure of a homogeneous structural recursion principle includes several significant bijections. These are represented graphically in Figure 6.4. The names in the sets in Figure 6.4 identify the corresponding terms in Figure 5.29. Every type has a corresponding conclusion and every constructor has a corresponding principal premise. The type of the statement of conclusion and principal premise is always the same. The statement of conclusion is always a simple function from the parameter type to the appropriate sort. The principal premise is always a function from the parameter type to the conclusion.

Figure 6.4(b) shows graphically the bijections in a heterogeneous structural recursion principle. Note that the set of parameters plays a role in



(a) Homogeneous structural recursion principle



(b) Heterogeneous structural recursion principle

Figure 6.4: The bijections among the sets of types, constructors, fixed point functions, statements of conclusion and principal premises for homogenous and heterogeneous structural recursion principles.

the bijections for a heterogeneous recursion principle. There is a bijection between the set of principal premises and the union of the set of constructors and parameters and between the set of statements of conclusion and the union of the set of types and the set of parameters. Figure 6.5 shows the corresponding heterogeneous recursion principle. The additional conclusions and principal premises are inserted last to facilitate currying.

Transforming Principal Premises and Function Bodies

The type of a principal premise is synthesized from the type of its corresponding constructor by a simple syntactic approach. Figure 6.6 shows the abstract syntax tree (AST) for the type of `node` and for the body of the invocation of the principal premise in the appropriate case of the *match* expression. The labels next to nodes in the type diagram are the pattern

```
ntree_r_het =
fun (A : Set) (P_ntree : ntree A -> Type)
  (P_nforest : nforest A -> Type)
  (P_A : A -> Type)
  (f_node : forall (a : A) (n : nforest A),
       P_A a -> P_nforest n -> P_ntree (node A a n))
  (f_nil : P_nforest (nil A))
  (f_cons : forall n : ntree A,
        P_ntree n ->
        forall n0 : nforest A, P_nforest n0 ->
        P_nforest (cons A n n0))
  (f_A : forall a : A, P_A a) =>
fix F_ntree (n : ntree A) : P_ntree n :=
  match n as n0 return (P_ntree n0) with
  | node a n0 => f_node a (f_A a) n0 (F_nforest n0)
  end
with F_nforest (n : nforest A) : P_nforest n :=
  match n as n0 return (P_nforest n0) with
  | nil => f_nil
  | cons n0 n1 => f_cons n0 (F_ntree n0) n1 (F_nforest n1)
  end
for F_ntree
     : forall (A : Set) (P_ntree : ntree A -> Type)
         (P_nforest : nforest A -> Type)
         (P_A : A -> Type),
       (forall a : A,
        P_A a ->
        forall n : nforest A,
        P_nforest n -> P_ntree (node A a n)) ->
       P_nforest (nil A) ->
       (forall n : ntree A,
        P_ntree n ->
        forall n0 : nforest A, P_nforest n0 ->
        P_nforest (cons A n n0)) ->
       (forall a : A, P_A a) ->
       forall n : ntree A, P_ntree n
```

Figure 6.5: A heterogeneous structural recursion principle for `ntree`.

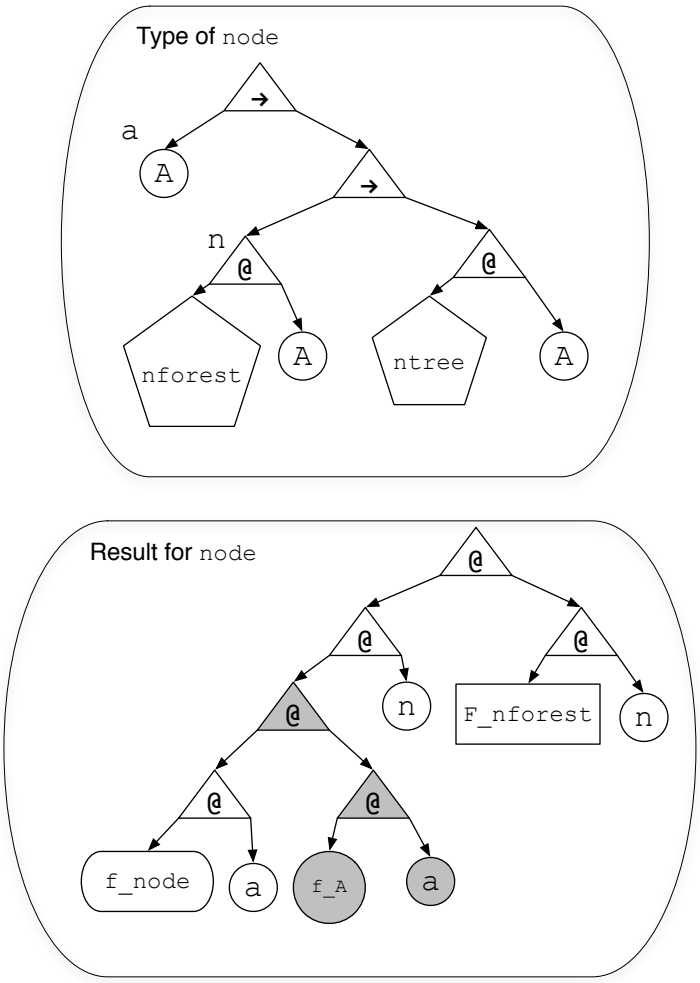


Figure 6.6: An AST showing the type of `node` and the body of the principal premise application.

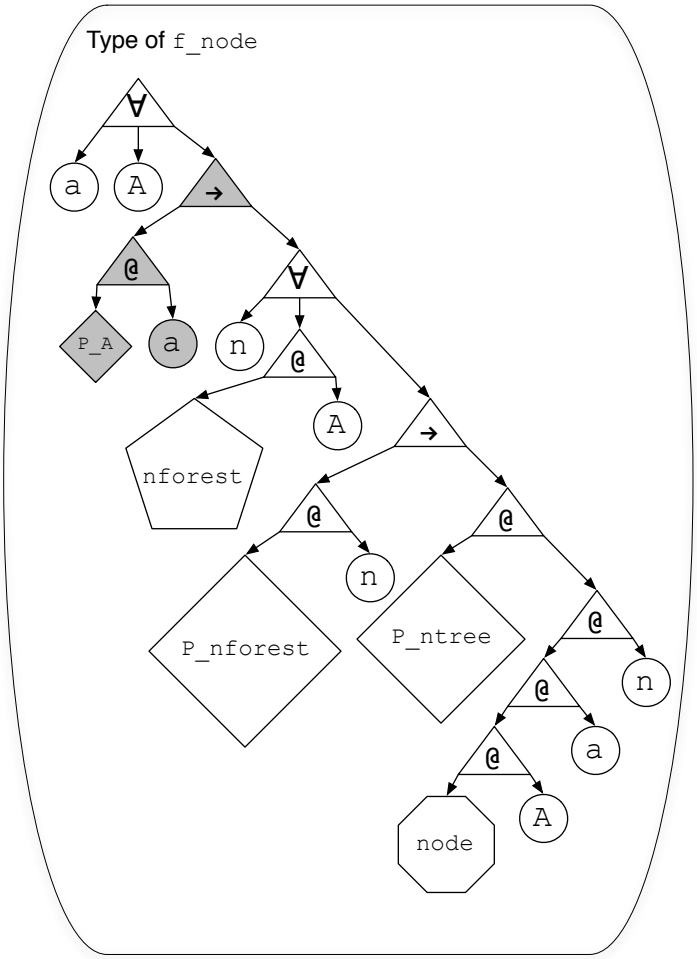


Figure 6.7: An AST showing the type of `f_node`.

variables bound in a match statement and used in the body. Figure 6.7 shows the AST for the type of the principal premise for `node`. Figure 6.8 gives the meaning of the different shapes used in the diagrams. Types or values that are added by our transformation are shaded.

Although the synthesized type may be quite complex its skeleton is easily discovered from the skeleton of the constructor's type. The type

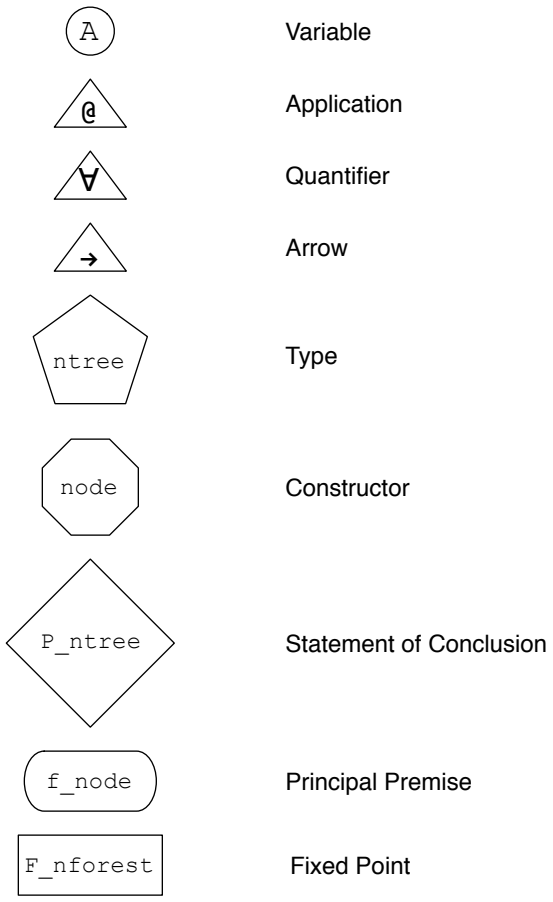


Figure 6.8: AST node legend.

synthesis algorithm travels along the spine of the constructor's type synthesizing subtrees from each left child. Each constructor on the spine is either a universal quantification constructor or an arrow constructor.

If it is a universal quantification constructor then its bound variable must be used later in the type. The bound variable can never be the sole argument to a function; it may be one of several arguments where the principle argument's value depends on it. Hence the constructor and its left subtree are copied without change to the synthesized type.

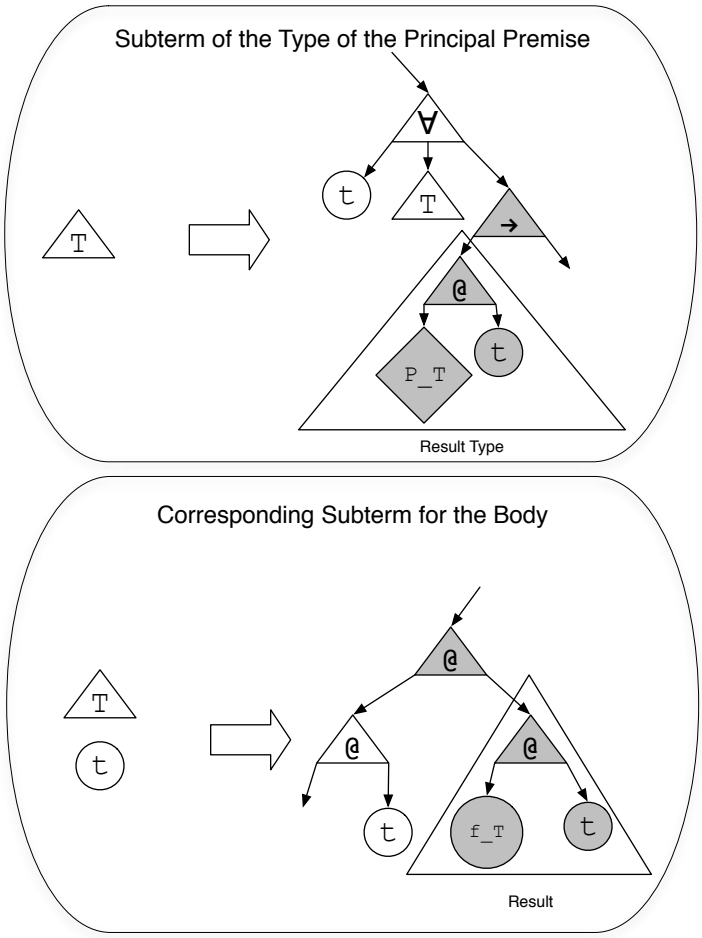


Figure 6.9: A schematic showing the synthesis of subtrees for the type of the principal premise and its corresponding body from a type term.

If it is an arrow constructor then a new subtree must be constructed. Figure 6.9 shows the construction. `t` is a new name for the argument of type `T`. It is bound using the universal quantification constructor. The bound value is used to construct the result type, the triangle labeled "Result Type" in the figure.

In general, a result type may be quite complex. However, if the bound

value is a parameter, the result type is `P_T t` where `P_T` is the appropriate statement of conclusion for the type `T`. This simple result type is the one shown inside the triangle in the figure. The incoming and outgoing arrows show where the subtree is connected to its enclosing type. The right child of the innermost arrow type is treated differently; the generated subtree is the result of applying the appropriate statement of conclusion to a term synthesized from the appropriate constructor.

The body of the application of the principal premise is synthesized in a related fashion. The term already constructed is applied to the match variable, `t`, and then to the result of a function call, the triangle labeled "Result" in the figure. As before, an actual result term may be quite complex. However, where `t` is a parameter the result is always the application of the principal premise for `T` to the bound variable. This simple term is the one shown inside the triangle in the figure and corresponds to the result type above.

If `T` is a parameter the shaded portions are omitted in the homogeneous principle but must be included in the heterogeneous principle.

A constructor's type determines the shape of the AST for the type of the corresponding principal premise and for its application for both homogeneous and heterogeneous principles. Thus, for any constructor, it is possible to define a transformation from a homogeneous to a heterogeneous form. We implement the transformation as a two-phase, tree-to-tree transformation using ANTLR (Parr, 2007, 2010). In the first phase, the inductive definition is read and the appropriate tree matching grammar is generated via ANTLR templates. In the second phase, the homogeneous structural recursion principle is parsed and the generated tree matching grammar is applied yielding the transformed, heterogeneous structural recursion principle.

6.3 Generalized Properties over Lists

An examination of the lemmas proved about `In` in the `List` module shows that most of the proofs are not about inclusion specifically but rather about any property composed from a disjunction of a property over the individual elements of a list. Generally speaking, inclusion is not the only property on lists that it may be desirable to prove. Numerous other properties can be expressed via a disjunction of an element-wise property. We argue that since this is in fact the case it is better to state the more abstract property and prove more general lemmas thereby making these more general lemmas available to users of the module.

This generalization makes use of the `Disjunct` function (Figure 6.3) which generalizes the structure of the `In` function. Note that in the `nil` case the `Disjunct` function returns `False`. The `False` proposition can never be proved. This is the correct choice for a property over lists defined by the disjunction of an element-wise property. It requires that a proof be found for at least one element in the list. If there are no elements in the list, then the property cannot hold. In our revised version of the `List` module `In` is defined in terms of the `Disjunct` function.

Our contribution in this section is to demonstrate how readily lemmas about `In` can be abstracted to more general lemmas. We thereby demonstrate that these lemmas have nothing to do with the semantics of `In` but are exclusively about its structure. We also point out a few cases where the lemmas cannot be abstracted; in these cases the semantics of `In` is an essential part of the lemma.

We have generalized and re-proved every lemma about `In` which requires a simple substitution of the general property `P_A` for equality. In most cases, reproving the lemmas was simple though tedious, merely requiring small syntactic changes to the proof scripts. In general, the overall structure of the ungeneralized and generalized proofs was identical.

In the following, we give several examples of lemmas about `In` that we

generalized. For example, the statement

```
forall (A : Type) (a : A) (l : list A), In a (a :: l)
```

becomes

```
forall (a : A) (l : list A), P_A a -> D (a :: l)
```

where `P_A` is

```
fun (x : A) => x = a
```

and `D` is

```
Disjunct A P_A
```

The property of equality to `a`, implicitly invoked by the repeated use of the variable `a`, is abstracted to the general property `P_A` and the disjunction of that property over every element of the list, i.e., `D`. In a similar vein, the statement

```
In a nil -> False
```

can be generalized to

```
D nil -> False
```

Since both `In a nil` and `D nil` evaluate to `False` the conclusion follows immediately. An arbitrary disjunctive property is preserved across list operations like reversal and permutation; we generalized lemmas about the `rev` function and the `Permutation` predicate with little difficulty.

Some operations, like map, require enriching the specification of some other player. For example,

```
forall (A B : Type) (f : A -> B)
  (l : list A) (x : A),
  In x l -> In (f x) (map f l)
```

requires that the specification of `f` be enriched. That is, if it can be shown that `f` yields a result for which `P_B` holds whenever it takes an argument for which `P_A` holds then it is possible to generalize the above statement to

```
forall (A B : Type) (f : A -> B)
  (P_A : A -> Prop) (P_B : B -> Prop),
  (forall a : A, P_A a -> P_B (f a)) ->
  forall l : list A, D_A l -> D_B (map f l)
```

where `D_A` and `D_B` are `Disjunct A P_A` and `Disjunct B P_B` respectively. If `P_A` is

```
fun (a : A) => a = x
```

and `P_B` is

```
fun (b : B) => b = f (x)
```

the generalized version is equivalent to the original.

A few lemmas about `In` resist generalization in this way. However these lemmas are specifically about list inclusion. For example the `NoDup` predicate holds exactly when there are no duplicates in a list. So the lemma relating `NoDup` to `In`

```
forall (A : Type) (x : A) (l : list A),
  (In x l -> False) -> NoDup l -> NoDup (x :: l)
```

is not generalizable. Similarly

```
forall (A : Type) (n : nat) (l : list A) (d : A),
  {In (nth n l d) l} + {nth n l d = d}
```

which states that the function `nth` returns an element actually in the list or else the default is about inclusion and therefore cannot be generalized.

In this section we have demonstrated that almost all the lemmas about `In` in the `List` module of the Coq standard library can be generalized to any disjunctive property over lists and that of these lemmas many can be

generalized and re-proved without any large changes. We believe that the `List` module would be significantly more valuable if these lemmas had been stated and proved in their more general form.

Other Data Structures

Many lemmas about the `In` property over lists can be generalized to any disjunctive property. The `In` property can be defined just as readily over data structures other than lists. It follows that the `In` property can be abstracted to a more general property. Figure 6.10 shows two equivalent versions of the `Disjunct` function for `ntree`. The top version is that constructed from the heterogeneous recursion principle. The bottom version is the same, except that the recursion principle has been inlined. Inspection of the principal premises in the first version shows that each is constructed by forming the disjunction of the results of all recursive calls and also of invocations of `P_A` just as with the version of `Disjunct` for lists. We argue that there should exist facilities for automatically generating the `Disjunct` function for any datatype. We discuss general techniques for making use of the heterogeneous structural recursion principle to automatically build `Disjunct` in Chapter 3.

6.4 Conclusions and Related Work

In this chapter we have demonstrated that a heterogeneous structural recursion principle can be automatically synthesized from an inductive definition just as a homogeneous principle can. We have demonstrated its utility by showing that it can be used to express properties and functions which take into account terms which are not in the type for which the recursion principle is constructed. A particularly powerful example is the `In` predicate on lists. We have shown that most lemmas about `In` defined in the `List` module are not about inclusion of an element in a list; but hold

```
Disjunct =
fun (A : Set) (P_A : A -> Prop) =>
ntree_rect_het A (fun _ : A => Prop) (fun _ : ntree A => Prop)
  (fun _ : nforest A => Prop)
  (fun (_ : A) (_ : nforest A) (X X0 : Prop) => X \/ X0)
  False
  (fun (_ : ntree A) (X : Prop) (_ : nforest A) (X0 : Prop) =>
    X \/ X0)
  (fun a : A => P_A a)
     : forall A : Set, (A -> Prop) -> ntree A -> Prop
```

(a) With invocation of heterogeneous structural recursion principle

```
Disjunct =
fun (A : Set) (P_A : A -> Prop) =>
fix F (n : ntree A) : Prop :=
  match n with
  | node a n0 => P_A a \/ F0 n0
  end
with F0 (n : nforest A) : Prop :=
  match n with
  | nil => False
  | cons n0 n1 => F n0 \/ F0 n1
  end
for F
     : forall A : Set, (A -> Prop) -> ntree A -> Prop
```

(b) With heterogeneous structural recursion principle inlined

Figure 6.10: The `Disjunct` function for `ntree`.

about any property defined as the disjunction of an element-wise property. We have argued that a version of the `List` module that proved these more general properties would be more valuable to a client of the module. We have further shown that the `In` predicate can be defined over other data structures by means of a heterogeneous structural recursion principle. We argue that it is in general desirable to generalize the `In` predicate and the lemmas about it to a more general `Disjunct` predicate.

Setoids (Barthe et al., 2003) are a way of defining an equivalence relation in Coq where an equality relation does not exist. If an equivalence relation is defined as a setoid Coq allows identity elimination even where terms

are not equal. The `List` module has been rewritten as `SetoidList` where all lemmas which include equality have been redefined to make use of setoid equality. For example,

```
forall (A : Type) (a : A) (l : list A), In a (a :: l)
```

has been redefined as

```
forall l x y, eqA x y -> InA x l -> InA y l
```

where equality has been rewritten to a setoid equality, i.e., `eqA`.

This rewriting is another kind of abstraction and thus has a similarity to the work presented in this chapter. However, the motivation and consequences are very different. The setoid rewrite relies on the extensive changes to Coq's substitution and rewriting mechanism which allow the user to take advantage of a setoid equality in the absence of Leibniz equality. The work presented in this chapter demonstrates that equality of any sort is not fundamental to many of the lemmas about `In` and that a user of the module would prefer that these lemmas be generalized. The contributions are entirely orthogonal. In fact, had the `List` module been originally written in the more general fashion that we recommend the rewriting work for `SetoidList` would not have been so extensive as all properties about `Disjunct` would continue to hold and properties specifically about `In` would simply have required specializing `Disjunct` with a function constructed from the setoid equality rather than the more basic Leibniz equality.

7 TAGGED STRUCTURAL RECURSION

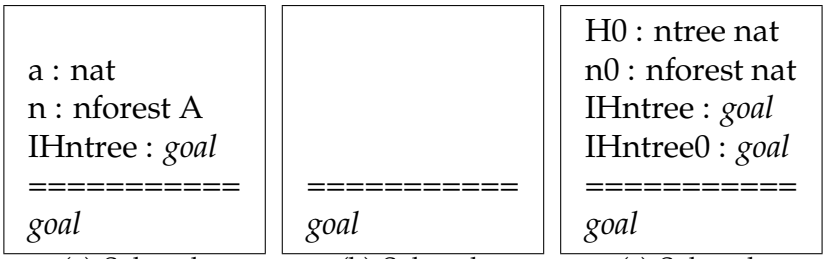
7.1 Introduction

In Chapter 6 we discussed a transformation on structural recursion principles to increase their generality. In this chapter, we discuss a transformation to make them more informative.

Figure 5.29 shows the structure of a homogeneous structural recursion principle for `ntree` (Figure 2.1(a)). Application of the induction principle results in the generation of three subgoals (Figure 7.1), one for each principal premise. In many cases, the choice of tactics is dependent on the principal premise's constructor. However, the proof-context fails to indicate what constructor each subgoal corresponds to.

The subgoals are always ordered in the same order as the principal premises are ordered in the induction principle which is the same order as the constructors occur in the inductive definition. For that reason the tactic language, *Ltac* (Delahaye, 2000), has a "General Sequence" operator which allows the user to specify a tactic for each subgoal resulting from the previous application of a tactic. The tactic expression

$$\mathtt{tac}_0;[\mathtt{tac}_1|\ldots|\mathtt{tac}_n]$$



(a) Subgoal (b) Subgoal (c) Subgoal

Figure 7.1: Subgoals resulting from application of `ntree`'s induction principle.

causes $\mathtt{tac}_0$ to be executed on the current subgoal; $\mathtt{tac}_1$ is applied to the first subgoal generated, $\mathtt{tac}_2$ to the second subgoal and so forth through the nth subgoal. Tactics composed from this operator are brittle since they are dependent on the order and number of subgoals generated by the prior tactic. If the inductive definition is changed the tactic may fail. Alternatively, $\mathtt{tac}_0$, may be made stronger by some addition to a hints database or other change so that it eliminates one subgoal. In that case, the succeeding list of tactics will fail since the number of subgoals on which they must operate does not match the number of tactics. Additionally, a user may become confused as to which principal premise she is endeavoring to satisfy.

We propose to alter the structural recursion principle so that each principal premise includes a tag term indicating its corresponding constructor. We also show how the "General Sequence" *Ltac* operator may be replaced by an *Ltac* match statement wherever it immediately follows an induction step thereby yielding more robust tactic scripts.

We expect that the introduction of the tag term in each principal premise will allow the construction of more robust tactics and reduce user frustration and confusion.

7.2 Structure and Use of the Tagged Structural Recursion Principle

Structure

Our goal is to tag every proof-context with its corresponding case. To achieve this, every principal premise must be transformed so that its first parameter is a tag indicating the associated constructor or type to which it corresponds. This tag is in `Prop` so that it can be elided by the extraction mechanism. Figure 7.2 shows a suitable inductive definition, `case`. `case`

```
Inductive case : forall A : Type, A -> Prop :=
  Case : forall (A : Type) (a : A), case a
```

Figure 7.2: The definition of the `case` type.

has just one constructor which can be used to instantiate a proof of `case` for any type and value. Figure 7.3 shows a tagged version of the heterogeneous structural recursion principle for `ntree`. Note that the first parameter of every principal premise is a value in type `case`. In each case, the type is dependent upon the constructor. The appropriate values are constructed anonymously where required in the body of each function.

The induction principle can be used in the normal way using the *induction* tactic. It will always insert an additional hypothesis in the proof-context of type `case`. This hypothesis can be used by an *Ltac* tactic script to decide the appropriate action.

Figure 7.4 shows subgoals introduced by a tagged structural recursion principle. Each subgoal is clearly marked by its `case` tag, shown in bold.

Making Use of `case` Hypotheses in Tactic Scripts

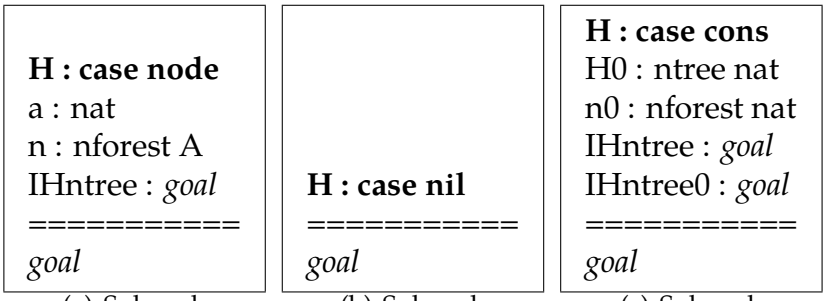
`case` hypotheses are not expected to play an active part in the proof, i.e., no `case` hypothesis is a necessary premise for any conclusion. Their purpose is solely to guide a tactic script in selecting the correct tactic to apply based on the particular principal premise that is being proved.

The *Ltac* tactic language includes a *match* expression which allows the user to pattern match against the proof-context. Figure 7.5 is a simple *Ltac* tactic expression which matches the most recently generated `case` hypothesis and prints the matched value.

Although the *Ltac match* expression superficially resembles a CoC or O'Caml *match* expression it has a very different semantics. If a particular case fails, either because there was no match or because the associated tactic failed backtracking will occur. Alternate matches for the same pat-

```
ntree_tag =
fun (A : Set) (P : ntree A -> Type)
  (P0 : nforest A -> Type)
  (f : case node ->
       forall (a : A) (n : nforest A),
       P0 n -> P (node A a n))
  (f0 : case nil -> P0 (nil A))
  (f1 : case cons ->
        forall n : ntree A,
        P n ->
        forall n0 : nforest A, P0 n0 -> P0 (cons A n n0)) =>
fix F (n : ntree A) : P n :=
  match n as n0 return (P n0) with
  | node a n0 => f (Case node) a n0 (F0 n0)
  end
with F0 (n : nforest A) : P0 n :=
  match n as n0 return (P0 n0) with
  | nil => f0 (Case nil)
  | cons n0 n1 => f1 (Case cons) n0 (F n0) n1 (F0 n1)
  end
for F
     : forall (A : Set) (P : ntree A -> Type)
         (P0 : nforest A -> Type),
       (case node ->
        forall (a : A) (n : nforest A),
        P0 n -> P (node A a n)) ->
       (case nil -> P0 (nil A)) ->
       (case cons ->
        forall n : ntree A,
        P n ->
        forall n0 : nforest A, P0 n0 -> P0 (cons A n n0)) ->
       forall n : ntree A, P n
```

Figure 7.3: A tagged structural recursion principle for `ntree`.



(a) Subgoal (b) Subgoal (c) Subgoal

Figure 7.4: Subgoals resulting from application of `ntree`'s tagged structural induction principle. Note that each subgoal is identified by a hypothesis with the appropriate dependently typed `case` type.

```
match goal with
| I : case ?c |- _ =>
  match c with
  | node |- _ => first [idtac "node" | fail 2]
  | nil |- _ => first [idtac "nil" | fail 2]
  | cons |- _ => first [idtac "cons" | fail 2]
  | _ => fail 2 "Unexpected case"
  end
end
```

Figure 7.5: An *Ltac match* expression illustrating matching of `case` hypotheses.

tern will be sought and the associated tactic will be retried. Matches are by default tried in order from freshest to least fresh. If all matches fail, execution will proceed to the next pattern.

In this context, backtracking is actually undesirable. The goal is to match the freshest `case` hypothesis and apply the appropriate tactic. Should the freshest `case` hypothesis correspond to a constructor for which there is no associated action then the entire tactic should fail immediately. The desired semantics is achieved through careful placement of exceptions, i.e., invocations of the *fail* tactic.

The *fail* tactic takes a natural number argument. The default value is 0,

```
match case with
| node => idtac "node"
| nil => idtac "nil"
| cons => idtac "cons"
| _ => fail 1 "Unexpected case"
end
```

Figure 7.6: A proposed extended syntax for matching cases.

which allows all backtracking. A value of 1 terminates all backtracking within the nearest enclosing *match* expression. The value is decremented as *match* expressions are exited. Thus, the choice of 2 terminates all backtracking with the tactic in Figure 7.5.

The nested *match* expressions are necessary to achieve the desired semantics. A single *match* expression could not prevent backtracking and rematching a less fresh `case` hypothesis.

The *match* expression in Figure 7.5 follows a simple tactic pattern. If the syntax of the *Ltac* tactic language were sufficiently extensible it would be possible to define a new syntax for this tactic pattern and make use of it as in Figure 7.6. In this example, all the repeated elements of the design pattern have been elided, leaving only the semantics that a user must specify. This *match* expression is distinguished from the one in Figure 7.5 by its use of the keyword `case` instead of `goal`. Unfortunately, the *Ltac* tactic language is not so extensible; it allows abbreviations for syntactic fragments only.

We can address part of this problem by introducing new syntax for each case in the match statement. Figure 7.7 shows a simple tactic abbreviation that allows the user to specify the action for each case. The tactic in Figure 7.8 which makes use of the `for_case` notation defined in Figure 7.7 and the *Ltac* `first` tactical, is semantically equivalent to the tactic in Figure 7.5 The `for_case` notation makes the purpose and semantics of the tactic more clear. The ordering of the arguments to the `first` tactic is

```
Tactic Notation "for_case" constr(c) "do" tactic(t) :=
  match goal with
  | I : case ?C |- _ =>
    match C with
    | c => first [ t | fail 2 ]
    | _ => fail 2
    end
  end
```

Figure 7.7: The `for_case` tactic. `c` captures the constructor which forms part of the type of the `case` hypothesis. `t` is the tactic to execute if the constructor is matched.

```
first [ for_case node do (idtac "node") |
        for_case nil do (idtac "nil") |
        for_case cons do (idtac "cons") ]
```

Figure 7.8: An *Ltac match* expression using `for_case` notation.

irrelevant and the entire tactic will fail if none of the cases are matched.

Tagged structural recursion principles and the `for_case` tactic notation can facilitate proof development and comprehension. First, by matching against cases, reliance on the order in which constructors occur in an inductive definition becomes entirely unnecessary. Second, a tactic which matches cases can be made to always fail if a particular case is not found. This means that the location of the error in the script is precisely identified.

Third, the user may make use of the tactic to progressively refine a super tactic. For example, initially the user may find that Coq's standard automation is able to discharge several subgoals. Only a few may require special handling. The user may wish to defer discovering the necessary automation that will prove these subgoals while focusing on some other aspect of the proof, temporarily including the necessary tactic scripts to solve each recalcitrant subgoal. These scripts serve as a record of each subgoal's solution and a useful reference when returning to the same proof to consider how to improve and generalize the whole tactic. If this

refactoring is successful the user may then use the untagged structural recursion principle as the context information which the `case` hypotheses provide has been made unnecessary.

Excluding the `case` hypothesis from the resulting proof

It is not the intention that the user should actually incorporate a `case` hypothesis into the final proof. The only appearance of the `case` hypothesis within the generated code should be restricted to the tagged structural recursion principle.

Since the `case` hypothesis has the same status as any other hypothesis it is impossible to prevent the user somehow making use of it in a proof. However, the `case` type is so simple as to be almost meaningless, and is in that respect just like the `True` type which is easily constructed, is useless as a hypothesis, and immediately provable as a goal.

The `for_case` tactic notation that we have supplied does not pass the value of the matched constructor to the selected tactic. This is deliberate, since the tactic is not supposed to make use of the value it matches. However, the constructor, being a constant, is always available .

7.3 Implementation

We implement the construction of a tagged structural recursion principle via a syntactic source-to-source transformation. We interpose the appropriate `case` hypothesis as the first argument of each principal premise. The hypothesis is easily constructed by applying the unique constructor for `case`, `Case`, to the constructor for the *match* case.

It is necessary to change the type of each principal premise to correspond with the change of its use. However, the type of each argument of every principal premise is known, so that the Coq type-inference engine is able to reconstruct the type of each principal premise without having to

```
(** val ntree_tag :
    (__ -> 'a1 -> 'a1 nforest -> 'a3 -> 'a2) ->
    (__ -> 'a3) -> (__ -> 'a1 ntree -> 'a2 -> 'a1
    nforest -> 'a3 -> 'a3) -> 'a1 ntree -> 'a2 **)

let ntree_tag f f0 f1 n =
  let rec f2 = function
    | Node (a, n1) -> f __ a n1 (f3 n1)
  and f3 = function
    | Nil -> f0 __
    | Cons (n1, n2) ->
        f1 __ n1 (f2 n1) n2 (f3 n2)
  in f2 n
```

Figure 7.9: An extracted version of a tagged structural recursion principle for `ntree`.

fall back on user supplied annotations. The output stage of our transformation simply strips all types from the function header, leaving only the parameter names.

7.4 Results of Extraction and Evaluation

Figure 7.9 shows the extracted version of the tagged structural recursion principle. The first argument to every principal premise has type `__`. Similarly, the first argument passed to each principal premise is `__`. `case` is in *Prop*, so it should be entirely elided by the extraction mechanism. Unfortunately, due to technical reasons it remains as an anonymous placeholder.

However, since it is unused in proofs it is discarded even by the Coq evaluation mechanism. Figure 7.10 shows a function that calculates the number of elements in an `ntree`. The recursion principle has been inlined and each `case` hypothesis has been removed.

```
fun H : ntree A =>
       (fix F (n : ntree A) : nat :=
          match n with
          | node _ n0 => 1 + F0 n0
          end
        with F0 (n : nforest A) : nat :=
          match n with
          | nil => 0
          | cons n0 n1 => F0 n1 + F n0
          end
        for F) H
     : ntree A -> nat
```

Figure 7.10: A function illustrating the elimination of `case` hypotheses by inlining of the tagged structural recursion principle.

7.5 Conclusion and Related Work

Coq includes a number of other tactics besides the *induction* tactic which generate a partial *match* expression. The *inversion* tactic, described in Chapter 9 is one such. The *case* tactic is similar to the induction tactic except that it provides no induction hypothesis within the proof-context. Neither of these tactics makes use of the structural recursion principle. However, since a `case` hypothesis can always be satisfied it would be relatively easy to extend these tactics to introduce the appropriate `case` hypotheses in the generated subgoals.

Aydemir et al. included a small set of tactics for annotating proof-contexts with their corresponding case to accompany one version of their library for supporting metatheoretic reasoning (Aydemir et al., 2008). This library is based on the *Case* tactic which assert an equality fact and introduces that fact into the proof-context. However, the *Case* tactic is used after the fact; the user may make use of the tactic to, in effect, attach a label to the current subgoal. The tactic itself gives no guarantee that the label is the correct one.

The developers suggest that the tactic be incorporated into a labeling

tactic as in the following brief tactic script.

```
induction n; [ Case O | Case S ].
```

If `n` is a `nat` the usual induction step is performed and the first subgoal, which corresponds to the `O` constructor is annotated with a label for `O` while the second subgoal, which corresponds to the `S` constructor, is annotated with a label for `S`.

This approach is inferior to ours in two ways. First, the script shown above may be used to do induction on a term of any type. For example, if `n` is a `list` instead of a `nat` then the induction step will occur as before, the `nil` case be annotated with `O` and the `cons` case with `S`. In our approach the labels are part of the structural recursion principle. So long as our algorithm transforms the structural recursion principle correctly then incorrect labels cannot be generated. Second, the script is brittle. The definition of Peano numbers is stable but other inductive definitions are likely to change during development. In that case a script that makes use of the *Case* tactic in the same way as in the example must be updated whenever its corresponding inductive definition is changed. Since our transformation of the structural recursion principle is automatic our approach is more robust.

Asperti et al. have proposed a new type for tactics (Asperti et al., 2009). Lacking from their type is any way to explicitly provide case information. However, their type explicitly maintains the structure of a partial proof as each tactic is executed. It is possible that their type could be expanded to include the case context. Alternatively, it may be possible to discover the case context by inspection of the partial proof term. These approaches are heavyweight, requiring alteration of fundamental aspects of the Matita proof engine. Our solution is a robust and lightweight alternative to these approaches.

8.1 Introduction

Variadic Functions

Many languages allow *variadic* functions, i.e., functions that take an arbitrary number of arguments. The family of `printf` functions in C (Kernighan and Ritchie, 1988) are an instance of a dependently typed variadic function; the number of arguments depends on the value of the format string argument. Because dependent types are a fundamental part of the CoC functions where the number of subsequent arguments is dependent on the value of a previous argument are easily defined (Weirich and Casinghino, 2010). In this, the CoC is unlike its programming language relatives ML and Haskell, which, lacking dependent types, are not variadic in this way (Blume et al., 2008) and more like recently developed dependently typed programming languages such as Agda (Bove et al., 2009) and Epigram (Chapman et al., 2006).

In Java, an ellipsis is used to indicate an arbitrary number of arguments (Naftalin and Wadler, 2006) of one type. The ellipsis is syntactic sugar for an array of elements. This approach is easily expressed in the CoC using parameterized lists or arrays.

Constructors of an inductive definition in the CoC are both functions and tags. When applied to an argument they serve as functions; when used in a *match* expression they serve as tags that discriminate between the different inhabitants of their type. The `S` constructor of the natural numbers illustrates these different uses. In the context of the statement

```
S O + O
```

the `S` constructor is a function which takes an argument of type `nat` and returns a value tagged with `S`. In the context of the *match* expression

```
genType =
fun (V : Type) (R : Type) =>
fix F (n : nat) : Type :=
   match n with
   | 0 => R
   | S n0 => V -> F n0
   end
      : nat -> Type
```

(a) Definition of `genType`

```
AND =
fix F (n : nat) : genType bool bool n :=
  match n as n0 return (genType bool bool n0) with
  | 0 => false
  | S n0 => fun x : bool => if x then G n0 else F n0
  end
with G (n : nat) : genType n :=
  match n as n0 return (genType bool bool n0) with
  | 0 => true
  | S n0 => fun _ : bool => G n0
  end
for F
```

(b) Definition of `AND`

Figure 8.1: A definition of a dependently typed `AND` function. The first argument, `n`, indicates the number of `bool` arguments the function should expect. The function is curried; the auxiliary `genType` function calculates the type of the `AND` function from the first argument. `V` is the variadic type. `R` is the return type of the function.

```
match n with S => 1 | O => O end
```

`S` is a tag that distinguishes terms in `nat` constructed using the `S` function from terms in `nat` constructed using the nullary `O` function.

As an example of a dependently typed function, we define the function `AND` that takes any number of `bool`s and returns the conjunction of all of them. Figure 8.1 shows the function, `AND`, and its type generating function, `genType`. The expression

```
AND 2 false false
```

```
Inductive illegal : Set :=
  | Illegal : forall n, genType illegal illegal n
```

Figure 8.2: An illegal inductive definition. The `genType` invocation is harmless in itself but if Coq were to allow it a loophole would be created through which it would be possible to construct non-terminating computations.

evaluates to `false` while the expression

```
AND 2 true
```

evaluates to the function

```
fun _ : bool => true
```

and the expression

```
AND 1 true false
```

is ill-typed.

Because constructors are a special kind of function that builds data it is forbidden to make them recursively dependently typed. Figure 8.2 shows an illegally defined inductive type. The type `illegal` is passed as an argument to the the `genType` function. If this were accepted by Coq, such terms as

```
Illegal 2 (Illegal 0) (Illegal 0)
```

would be acceptable. Clearly, the definition is harmless in this case. But, if it were allowed the Coq system would be unable to prevent certain inductive definitions that could lead to non-terminating computations and thereby introduce inconsistency into the system. Consequently, to define a constructor with a variadic recursive type it is necessary to use a list in a way analogous to the implementation of variadic functions in Java.

```
Inductive exp : Set :=
    var : atom -> exp
  | var_ind : atom -> exp
  | abs : exp -> exp
  | app : exp -> exp -> exp
  | record : list exp -> exp
```

Figure 8.3: The lambda-calculus with records.

The variadic interpretation of the `list` type constructor overloads its default interpretation, that of a data structure. When defining a function, the user may choose the interpretation that is correct. However, structural recursion principles are defined automatically. Without guidance Coq will use the default interpretation. The principle will be useless where the variadic interpretation is intended.

The fundamental problem that needs to be resolved is the confusion between the two meanings of the use of the `list` datatype. The interpretation the Coq system puts on the use of `list` is "a list data structure"; the structural recursion principle it generates is the correct one for that interpretation. We present an alterative structural recursion principle appropriate to the variadic interpretation of `list`. We describe how such a principle can be generated from the principle that Coq automatically generates. We describe a case study that demonstrates that the use of the principle need not require significant changes to the underlying tactical infrastructure. Our experiments indicate that integrating first-class variadic types and appropriate structural recursion principles into Coq would facilitate proof development.

The Default Interpretation of `list`

Figure 8.3 is an inductive definition of the syntactic structure of the untyped lambda-calculus. The last constructor, `record`, defines a record expression as a `list` of expressions. The intuitive notion of a record (Pierce,

```
exp_rect =
fun (P_exp : exp -> Type)
  (f_var : forall n : atom, P_exp (var n))
  (f_var_ind : forall n : atom, P_exp (var_ind n))
  (f_abs : forall e : exp, P_exp e -> P_exp (abs e))
  (f_app : forall e : exp,
        P_exp e ->
        forall e0 : exp, P_exp e0 -> P_exp (app e e0))
  (f_record : forall l : list exp, P_exp (record l)) =>
fix F_exp (e : exp) : P_exp e :=
  match e as e0 return (P_exp e0) with
  | var n => f_var n
  | var_ind n => f_var_ind n
  | abs e0 => f_abs e0 (F_exp e0)
  | app e0 e1 => f_app e0 (F_exp e0) e1 (F_exp e1)
  | record l => f_record l
  end
```

Figure 8.4: The automatically generated structural recursion principle for `exp` illustrating non-variadic interpretation of the `list` datatype.

2002) is fundamentally but subtly different; a record is an expression containing any number of subexpressions. The example requires a variadic interpretation of `list`.

However, since there is no way to indicate that the intended meaning is variadic Coq generates a structural recursion principle appropriate to the interpretation of `list` as a data structure. Figure 8.4 shows the principle Coq automatically generates. The principal premises and terms for the `record` case are identical in structure to those for the `var` and `var_ind` case. There is no recursive call in the corresponding result for the *match* expression; none of these principal premises contains an inductive hypothesis.

This is consistent with the interpretation of `list` as a data structure. However, it is unsuitable for the variadic interpretation. It is impossible to make use of this recursion principle to build any of the function or proofs that one would naturally desire to build.

For example, consider a function that calculates the number of free variables in an expression. This is a uniform function like those discussed in Chapter 3. The function that describes the fold operation is

$$\lambda l.\ \text{fold}\ (\lambda\ \text{acc}.\lambda\ e.\ \text{if}\ e\ \text{matches}\ r\ \text{then}\ `(e\ \cup\ \text{acc})\ \text{else}\ `\text{acc})\ `\emptyset\ l$$

However, if the recursion principle in Figure 8.4 is used the resulting subgoal for `record` is the useless subgoal containing just a single hypothesis with type `list exp`. A useful subgoal would contain as many additional hypotheses as there are are elements in the list, each one representing the result of the application of the fixed point function to an element in the list. Without these additional hypotheses it is impossible to calculate the number of free variables in the record. An analogous situation holds for every proof of any property and for any specification on expressions which uses the automatically generated recursion or induction principles.

The addition of records to a specification of the lambda-calculus ought to be a simple problem. Records are a fairly simple theoretical notion; they can easily be viewed as generalization of products (Pierce, 2002). Products are easily represented in Coq because the number of arguments is the constant two. The extension to records has proven difficult. It is notable that publicized solutions to the POPLmark challenge (Aydemir et al., 2005) are not readily extensible to any syntactic form which may contain an unbounded number of elements. Examples and corresponding solutions to this problem are noticeably absent from published results of formal proofs using the Coq proof assistant.

The accepted workaround is to make use of a mutually recursive datatype as in Figure 8.5. This approach obscures the intended variadic meaning and adds additional overhead due to the use of the mutually recursive definition. The second problem could be ameliorated by transforming the mutually inductive type to a single dependent type as described in Chapter 2. However, the intended variadic use would be as obscure in the dependent type as in the original mutually recursive type.

```
Inductive exp : Set :=
    var : atom -> exp
  | var_ind : atom -> exp
  | abs : exp -> exp
  | app : exp -> exp -> exp
  | record : explist -> exp
with explist : Set :=
  | explistnil : explist
  | explistcons : exp -> explist -> explist
```

Figure 8.5: An alternative way to express an arbitrary number of subexpressions with a mutually recursive datatype.

In the rest of the chapter we demonstrate an alternate approach. Rather than deform our inductive definition so that the automatically generated structural recursion principle is useful we keep the most natural inductive definition that Coq allows and modify the structural recursion principle so that it is appropriate to the variadic interpretation of `list`.

8.2 Structure of the Recursion Principle

The structural recursion principle shown in Figure 8.6 is appropriate to the variadic interpretation. The text in italics indicates the additional terms that must be added to the principle to make it appropriate to the variadic interpretation of `list`. `f_record` takes an additional variadic hypothesis,

```
forall e : exp, SubtermT e l -> P_exp e
```

The variadic hypothesis defines the type of a function that demonstrates that for any expression, `e`, in the record, `P_exp e` holds. The body of the variadic function is supplied in the body of the structural recursion principle.

We make use of two functions, `SubtermP` and `SubtermT`, to define the type of the variadic function. `SubtermP` is identical to the `In` function over lists (Figure 6.2). `SubtermT` is similar, but makes use of the `sumor`

```
exp_rect =
fun (P_exp : exp -> Type)
  (f_var : forall n : atom, P_exp (var n))
  (f_var_ind : forall n : atom, P_exp (var_ind n))
  (f_abs : forall e : exp, P_exp e -> P_exp (abs e))
  (f_app : forall e : exp,
        P_exp e ->
        forall e0 : exp, P_exp e0 -> P_exp (app e e0))
  (f_record : forall l : list exp,
        (forall e : exp, SubtermT e l -> P_exp e) ->
        P_exp (record l)) =>
fix F_exp (e : exp) : P_exp e :=
  match e as e0 return (P_exp e0) with
  | var n => f_var n
  | var_ind n => f_var_ind n
  | abs e0 => f_abs e0 (F_exp e0)
  | app e0 e1 => f_app e0 (F_exp e0) e1 (F_exp e1)
  | record l =>
      f_record l
        (fun e0 : exp =>
         (fix F (l0 : list exp) : SubtermT e0 l0 -> P_exp e0 :=
            match
              l0 as l1 return (SubtermT e0 l1 -> P_exp e0)
            with
            | nil =>
                fun H : SubtermT e0 nil => False_rect (P_exp e0) H
            | h :: t =>
                fun H : SubtermT e0 (h :: t) =>
                match H with
                | inleft H0 => F t H0
                | inright H0 =>
                    eq_rect_r (fun e1 : exp => P_exp e1)
                      (F_exp h) H0
                end
            end) l)
  end
```

Figure 8.6: A structural recursion principle for `exp` consistent with the variadic interpretation of `list`.

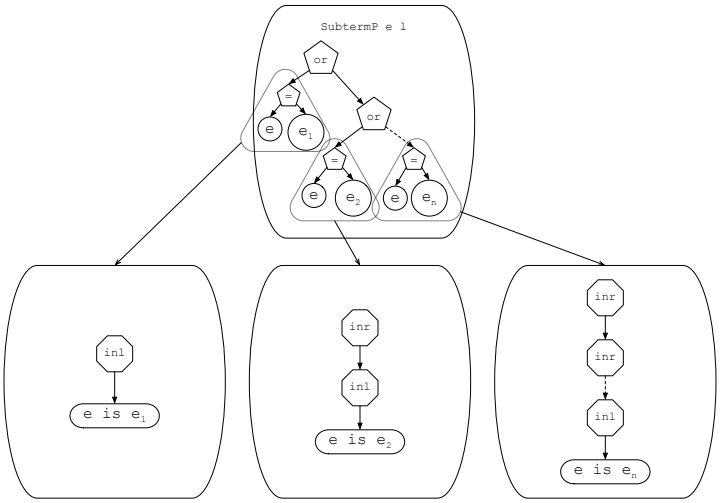


Figure 8.7: The type of `SubtermP e l` for a list with `n` elements.

type instead of the `or` type. The types are isomorphic in structure but have different kinds. `sumor` is appropriate for recursion principles since it inhabits *Type*; `or` is appropriate for induction principles since it inhabits *Prop*.

Figure 8.7 shows the type that `SubtermP e l` builds as well as three possible inhabitants of that type. The dashed lines in each figure indicate an elided portion of the term. There are as many possible inhabitants of the type as there are elements in the list. Each inhabitant corresponds to a particular element in the list. Each contains one proof of equality between `e` and one element in the list. The lines between the subterms of the type and each inhabitant of the type indicate which proof of equality it contains. The variadic function must recursively deconstruct this term until it discovers the equality proof at its core. It then applies `F_exp` to the appropriate element of the list. Because the value of `SubtermP e l` is dependent on `l` the variadic function must proceed by structural recursion over `l`,

```
Inductive lc_e : exp -> Prop :=
    lc_e_var : forall x,
      lc_e (var x)
  | lc_e_abs : forall e,
      (forall x : atom, lc_e (open_ex (var x) e)) ->
      lc_e (abs e)
  | lc_e_app : forall e e1,
      lc_e e ->
      lc_e e1 ->
      lc_e (app e e1)
  | lc_e_record : forall l,
      (forall e, SubtermP e l -> lc_e e) ->
      lc_e (record l)
```

Figure 8.8: A definition of local closure that makes explicit use of list inclusion.

8.3 Use of a Variadic Type

Defining Properties and Functions on Variadic Types

A variadic type makes use of the `list` constructor under a variadic interpretation. Consequently, when defining properties on a variadic type it is necessary to make use of list inclusion. Figure 8.8 shows the definition of local closure (Aydemir et al., 2008) for the definition of `exp` in Figure 8.3. An `app` is locally closed if its subexpressions are locally closed. The same is true for a `record`; the hypothesis

```
forall e, SubtermP e l -> lc_e e
```

expresses the fact that all subexpressions of `record` must be locally closed. Since `lc_e` is in *Prop*, `SubtermP`, which is an alias for the `List.In` function (Figure 6.2), is used instead of `SubtermT`.

Similarly, in defining recursive functions on a variadic type it is necessary to proceed by recursion over the list of subterms. Figure 8.9 shows the definition of a function that counts the number of constructors in an expression. `fold_subterms` is an alias for the `fold_right` function defined

```
size_e =
fix size_e (e : exp) : nat :=
  match e with
  | var _ => 1
  | var_ind _ => 1
  | abs e1 => 1 + size_e e1
  | app e1 e2 => 1 + size_e e1 + size_e e2
  | record l =>
      1 +
      fold_subterms
      (fun (x : exp) (acc : atom) => size_e x + acc) 0 l
  end
```

Figure 8.9: A definition of `size_e` showing use of `fold_right` for the `record` case.

in the `List` module of the Coq standard library. This function is identical in structure to the canonical fold operation on lists.

Many functional language programmers have been taught to prefer the complementary left fold operation due to its greater efficiency. The canonical fold operation is superior in the variadic context because a single recursive call exposes an operation on the first element in the list allowing a proof by induction to proceed naturally.

Proving with Variadic Types

A strategy for making relatively robust proof developments is to divide a development into lemmas such that each lemma can be proved in three stages:

- setup
- induction
- automation

Coq's *Ltac* language allows the user to define some very general tactics. However, these tactics can also prove brittle and are hard to maintain. In

general, the proof script is more robust if the automation consists solely of decomposition into distinct cases and repeated uses of modus ponens using lemmas drawn from an appropriately constructed hint database.

A variadic term includes a nested term, the `list` of subterms. To traverse this `list` requires an additional induction step. Consequently it may seem that the use of a variadic type and the appropriate structural recursion principle will necessarily break the relatively robust tactic strategy we have described since it will require two induction steps within a single lemma. We will show that, by the automatic construction of sublemmas appropriate to the variadic use of list, this necessity can be avoided.

Example

Consider a proof of

```
forall (e : exp), size_e (g e) = size_e e
```

`size_e` is defined in Figure 8.9. It is not necessary to know the definition of `g` only that it makes use of a fold operation over the subexpressions of the `record` constructor and that it does not change the number of constructors in its argument.

Proceeding by induction on `e` the user is presented with five subgoals, one for each constructor. Figure 8.10 shows the subgoals for the `app` constructor and for the `record` constructor. To save space, `s` is used as an abbreviation for `size_e` and `fold` as an abbreviation for `fold_right`. At stage 1 the subgoals are exactly as they have been left by application of the induction principle. The context for the `app` constructor contains two induction hypotheses, one for each subexpression. Since the `record` constructor contains any number of hypotheses the context includes a single induction hypothesis that covers all subterms. The goals are identical in structure.

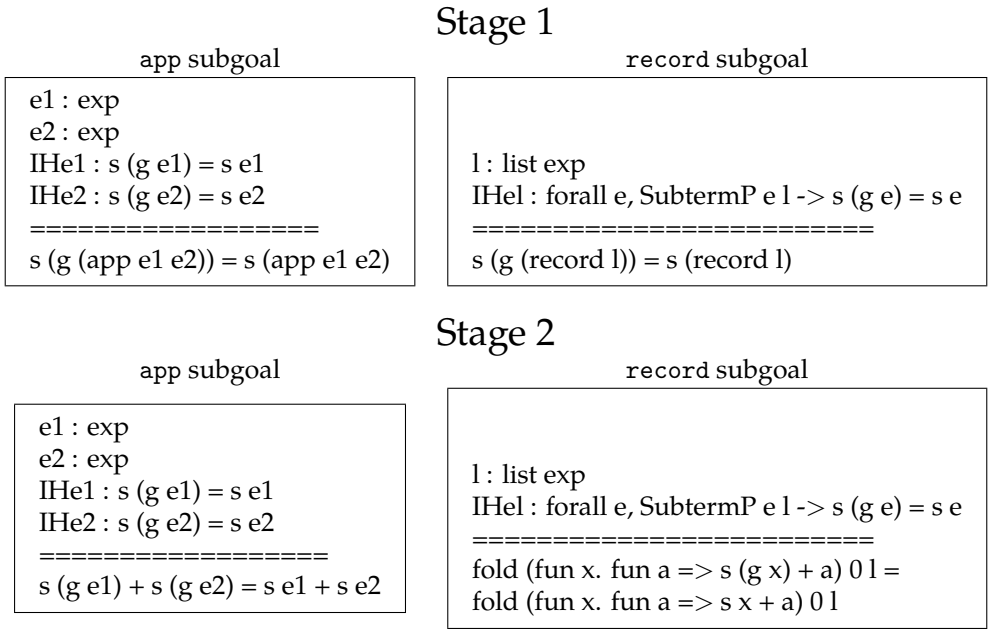


Figure 8.10: Progress of subgoals after initial induction step.

The goals at stage 2 have been modified by computation using Coq's internal engine. They are now very different in character. In both cases, the constructor has disappeared because one step of computation moves the `s` and `g` calls below each constructor. However, the goal corresponding to `app` has been thoroughly simplified and a simple substitution using the equality hypotheses in the context will complete the proof. The goal corresponding to `record` has been simplified by a fusion transformation (Section 8.3) but the goal term is hidden in a the fold operation. No direct application of the equality hypothesis in the context will solve the goal.

The `record` subgoal is obviously true to anyone familiar with the fold operation. Another induction step, this time on `l`, will solve the goal. This additional induction step is awkward. The user is forced to stage a second induction step at some ideal point after the original induction step when the subgoal has been transformed so that it is in a state where induction will succeed. Ideally, the necessary induction step should be abstracted

```
size_e_variadic =
fun (f : exp -> nat) (l : list exp) =>
fold_subterms (fun (x : exp) (acc : nat) => f x + acc) 0 l
     : (exp -> nat) -> list exp -> nat

size_e =
fix size_e (e : exp) : nat :=
  match e with
  | var _ => 1
  | var_ind _ => 1
  | abs e1 => 1 + size_e e1
  | app e1 e2 => 1 + size_e e1 + size_e e2
  | record l => 1 + size_e_variadic size_e l
  end
     : exp -> nat
```

Figure 8.11: An abstraction of the variadic computation in `size_e`.

into its own lemma.

Abstracting Computation on a Variadic Type

The first step in this abstraction is to observe that computation over the variadic portion of a term can be abstracted. For example, the function `size_e` (Figure 8.9) adds the result of its recursive arguments in every context. If the definition of `exp` were extended with another variadic constructor the computation over its `list` argument would be almost certainly identical. The `size_e` function can be restated in terms of the abstracted function. Figure 8.11 shows the abstracted version. Note that they are not mutually recursive. `size_e_variadic` expects a function parameter, `f`. The function that is passed as an argument is `size_e`.

Abstracting Proofs on a Variadic Type

It is clearly possible to abstract the computation on the variadic portion of a term so that it can be defined before the recursive computation over

```
forall l,
    (forall e,
       SubtermP e l ->
       size_e (g e) = size_e e) ->
    (size_e_variadic size_e (g_variadic g l) =
     size_e_variadic size_e l
```

Figure 8.12: A variadic sublemma for the proof that `size_e (g 3) = size_e e` for all `e`.

the whole term. The same approach can be applied to the statements of lemmas for the variadic portion of a type.

Continuing with our ongoing example, we wish to abstract the goal

```
forall(e: exp), size_e (g e)=size_e e
```

so that it expresses an appropriate sublemma that will encapsulate the induction step over the variadic portion of any term in the `exp` type. Just as with the abstraction of `size_e` the sublemma must take an argument which represents the abstracted lemma. The type of the sublemma is shown in Figure 8.12. The head type is a restatement of the goal in terms of the variadic operations. The first argument, `l`, is the list of subterms. The second argument is a restatement of the original goal with the additional hypothesis that every term must be in `l`. This additional hypothesis is necessary to preserve well-foundedness. This sublemma is provable by induction on `l` followed by the same automation that proves the original lemma. Once proved, this sublemma can be inserted into the hint database and is thus available to the automation step used to prove the original lemma.

To restate the goal in terms of the variadic operations it is necessary to discover the inductive variable. In the example there is only one variable, `e`. However, in general, there may be many variables, most of which are held constant. The goal must be abstracted over the inductive variable. Wherever the inductive variable appears its enclosing context must be

$$\begin{array}{lcl} f[\![e \mapsto l]\!] & \Rightarrow & [\![fe \mapsto f_v l]\!] \\ P[\![e \mapsto l]\!] & \Rightarrow & [\![Pe \mapsto \forall e, e \in l \rightarrow Pe]\!] \end{array}$$

Figure 8.13: Rewrite rules for the conversion to a variadic statement of a goal. f is the original function. f_v is its variadic counterpart. P is a property.

transformed so that it is appropriate to a `list` rather than the original type. Figure 8.13 shows the appropriate transformations. e and l represent arguments of the base type and of the variadic type respectively. `f` is a function which takes an argument of the base type. `f_v` is the corresponding variadic function. `P` is a property of the base type. The transformation is in general recursive; for example, the term $P(fe)$ may be rewritten to $P(f_v l)$ and then to $\forall e, e \in (f_v l) \rightarrow Pe$.

It is obvious that the term $\forall e, e \in (f_v l) \rightarrow Pe$ is equivalent to the term $\forall e, e \in l \rightarrow P(fl)$ which is more amenable to automation. The terms are equivalent, but they are not equal. Since they are equivalent the more complex term can always be replaced by the simpler term regardless of whether the term is a hypothesis or a conclusion. Since they are not equal, the lemma that states their equivalence is not compatible with Coq's rewriting mechanism. Since the lemma is parameterized on `P` the lemma cannot be inserted into Coq's hints database. Consequently, it is necessary to write a specialized tactic to make use of the lemma.

This restriction exposes a weakness not in the technique but in Coq's automation facilities. We expect that in a future version of Coq the rewriting mechanisms will be enhanced to make use of equivalences as well as of equalities.

The Fusion Property

The *fusion* property (Figure 8.14) defines the conditions that must hold so that the composition of an arbitrary function and a fold operation can be

$$\begin{array}{lcl} h\,w & = & v \\ h\,(g\,x\,y) & = & f\,x\,(h\,y) \end{array} \quad \Rightarrow \quad h \cdot (\mathtt{fold}\ g\ w) \ = \ \mathtt{fold}\ f\ v$$

Figure 8.14: The fusion property for the *fold* operation on lists.

rewritten as a single fold operation (Hutton, 1999). The fusion property is easily proved by induction. Many standard functions on lists, e.g. map, can be defined in terms of the fold function. Consequently, the fusion property is very useful for proving properties of these functions.

A proof of the fusion property is not useful as a rewriting mechanism since the value of f cannot be calculated directly from the conditions. The following assumptions restrict the generality of the fusion property and make it more useful for rewriting in the variadic context.

***h* is itself a fold operation** This situation arises quite naturally when calculating the result of the composition of several functions on an expression.

the structure of *g* is known The expected implementation of g under the variadic interpretation is $\lambda x.\lambda a.(s\,x) :: a$ where s corresponds to some function defined recursively over all elements.

Under these assumptions the fusion property defines *f*. Figure 8.15 shows the derivation of a less general property that is useful as a rewrite rule. f is shown to be equal to $\lambda x.\lambda a.h\,(s\,x)\,a$. Under this assumption, a consequence of the assumptions about g and h the conclusion of the fusion property can be rewritten as

$$(\mathtt{fold}\ h\ v) \cdot (\mathtt{fold}\ (\lambda x.\lambda a.(s\,x) :: a)\ w) = \mathtt{fold}\ (\lambda x.\lambda a.h\,(s\,x)\,a)\ v$$

A rewrite rule can be applied to a goal with the same form as the left hand side of the equation without difficulty, since all variables on the right hand side of the equation occur on the left hand side. An application of this rewrite rule leaves the premises

$$
\begin{array}{ll}
 & h\ (g\ x\ y) = f\ x\ (h\ y) \\
 & \text{Rule: } g \to \lambda x.\lambda a.(s\ x) :: a \\
\Rightarrow & h\ ((s\ x) :: y) = f\ x\ (h\ y) \\
 & \text{Rule: } h \to \mathtt{fold}\ h\ v \\
\Rightarrow & \mathtt{fold}\ h\ v\ ((s\ x) :: y) = f\ x\ (\mathtt{fold}\ h\ v\ y) \\
 & \text{Rule: by evaluation} \\
\Rightarrow & h\ (s\ x)(\mathtt{fold}\ h\ v\ y) = f\ x\ (\mathtt{fold}\ h\ v\ y) \\
 & \text{Rule: by abstraction} \\
\Rightarrow & h\ (s\ x)(\mathtt{fold}\ h\ v\ y) = (\lambda x.\lambda a.h\ (s\ x)\ a)\ x\ (\mathtt{fold}\ h\ v\ y)
\end{array}
$$

Figure 8.15: A derivation for the body of f under assumptions about g and h.

$$
\begin{array}{lcl}
\mathtt{fold}\ h\ v\ w & = & v \\
\mathtt{fold}\ h\ v\ ((s\ x) :; y) & = & h\ (s\ x)\ (\mathtt{fold}\ h\ v\ y))
\end{array}
$$

which follow easily. The example in Figure 8.10 is an illustration of the application of this less general fusion property.

8.4 Implementation

The recursion principle itself can be synthesized from the default recursion principle by a combination of syntactic transformations from the automatically generated induction principle and tactics. Every use of a variadic list is identified in the automatically generated principle by the `star` alias. For every occurrence of `star` in the type of a principal premise an additional parameter must be added to specify that the required property holds for every item in the list. For `exp` that parameter is

```
forall e : exp, SubtermT e l -> P_exp e
```

for a recursion principle and

```
forall e : exp, SubtermP e l -> P_exp e
```

```
refine (
fun P_exp f_var f_var_ind f_abs f_app f_record =>
fix F_exp (e : exp) : P_exp e :=
  match e as e0 return (P_exp e0) with
  | var n => f_var n
  | var_ind n => f_var_ind n
  | abs e0 => f_abs e0 (F_exp e0)
  | app e0 e1 => f_app e0 (F_exp e0) e1 (F_exp e1)
  | record l => f_record l _
  end);
variadic_for_type F_exp
```

Figure 8.16: A script to build a variadic recursion principle for a definition of the lambda-calculus with records. The script makes use of the *refine* tactic. All holes are guaranteed to be filled in by the *Ltac* tactic script `variadic_for_type`.

for an induction principle. All parameters must follow the same overall pattern. If the type of the elements of the list is dependent then `P_exp` must take additional arguments. However, the names of these arguments can be extracted from the specification of the type of the elements of `star`.

Figure 8.6 shows the body of the variadic recursion principle. The function which inhabits the additional parameter is recursive and complicated. However, because it has a regular structure it is easily constructed by a tactic. Our approach makes use of the `refine` tactic, which allows the user to specify a partial term in the CoC where dashes represent the parts of the term that must be filled in by tactics executed subsequently. The preliminary step is to modify the body of the recursion principle so that dashes are introduced wherever an additional parameter has been introduced into a principal premise and to apply this modified body to the goal using the `refine` tactic. Figure 8.16 shows the script which builds the principle. `variadic_for_type` is an *Ltac* tactic script which constructs a term that inhabits the type

```
forall e : exp, SubtermT e l -> P_exp e
```

Because this function has a regular structure it is possible to write a simple tactic that will complete any recursion principle constructed in this way.

8.5 Experimental Results

We experimented with the principle by extending an example associated with Aydemir et al. (2009) to include a record datatype. Our goal was to discover how difficult it was to make use of a variadic principle in this context. We hoped that the addition of a record type and the use of a variadic recursion principle would not require significant changes to the tactic scripts that made up the development.

The development has three parts. The first part is a default solver consisting of about 150 lines of tactics. The solver uses generally applicable tactics to simplify the goal and solve subgoals. It makes use of Coq's *auto* tactic, which implements a Prolog-like decision procedure, to solve any remaining subgoals. The final step is domain-specific since the databases on which *auto* relies are filled with lemmas from the development. The second part is a set of libraries that support the locally-nameless encoding of variables (Aydemir et al., 2008). The third part is an application which generates definitions, lemmas, and tactic scripts based on a specification of the syntax of the language for which the lemmas are defined. The relationship of these definitions, lemmas, and scripts to the syntax of the language are embedded in the application. The tactic scripts are intended to be small and general; most tactic scripts commence with an induction step and the remaining subgoals are solved by the default solver of the first part. The application is accompanied by several test languages.

We modified a test language to include a record type and thus to require a variadic recursion principle. Our experiment would have been a failure if it had proved necessary to substantially change the existing tactic scripts to accommodate the variadic recursion principle.

```
Lemma size_e_close_ex_rec_mutual_variadic :
  forall l,
  (forall e x n,
    SubtermP e l ->
    size_e (close_ex_rec n x e) = size_e e) ->
  (forall x n,
    size_e_variadic size_e
      (close_ex_rec_variadic (close_ex_rec n x) l) =
    size_e_variadic size_e l).
Proof.
induction l; default_simp_variadic.
Qed.
```

(a) The variadic sublemma.

```
Lemma size_e_close_ex_rec_mutual :
(forall e x n,
  size_e (close_ex_rec n x e) = size_e e).
Proof.
apply_mutual_ind exp_mutind;
default_simp.
Qed.
```

(b) The original lemma.

Figure 8.17: A representative lemma and its mechanically generated variadic sublemma. The sublemma is placed before its lemma in the development so that it can be used by automatic tactics.

The development for the original test language contained 93 lemmas. Of these, none had to be substantially altered to accommodate the variadic recursion principle. In 28 cases it was necessary to add a variadic sublemma. Each sublemma was mechanically generated by hand using the method described in Section 8.3 and easily solved using the default solver. Figure 8.17 shows an original lemma and its corresponding sublemma generated in this way. The variadic sublemma makes use of a tactic, `default_simp_variadic`. This tactic is the same as the the `default_simp` tactic except that it includes some instructions to explicitly unfold the definitions of variadic functions.

One lemma required a sublemma of an unusual kind. We discuss this

lemma further in Section 8.6.

As a further test, we modified the language again, adding a constructor that took two arguments: a variadic use of list and a single additional argument in `exp`. Our experiment would have been a failure if we had found that we had to modify any of the tactic scripts from the previous experiment in any way; we did not. Our experiment exposed a small bug in the original scripts which we fixed in both versions.

Finally, we reverted all previous definitions to their original form, but left all lemmas and tactic scripts the same as in our first experiment and second experiment. Our experiment would have been a failure if any of the tactic scripts had failed. None did, indicating that any minor changes we had made to the tactic scripts and the presence of the 28 variadic sublemmas were benign.

8.6 Restrictions on the Use of Variadic Types

Autogenerating Uniform Variadic Functions

The `size_e` function (Figure 8.9) is an example of a uniform function. It would be desirable to show that the technique for generating uniform functions described in Chapter 3 is easily extended to variadic types. It is not; however this is due not to a weakness of our technique but rather to Coq's too stringent syntactic rules for ensuring termination. This unfortunate situation could be obviated by making variadic types in Coq first-class.

The *compose* argument of the `fold_right` function takes an element of the list argument and the value accumulated so far and returns a new value. However, the signature of the *compose* function does not include the fact that its element argument is in the list to which `fold_right` is applied. This fact is certainly implicit in the specification of `fold_right` as the *compose* function is never applied to elements in some other list or arbitrary elements that happen to have the same type as the list elements.

The technique for autogenerating uniform functions makes use of a recursion step which proceeds by application of the appropriate recursion principle (Figure 8.6). The application of this principle generates the hypothesis

```
forall e : exp, SubtermT e l -> P_exp e
```

where `l` is the list of subexpressions of the `record` constructor. This hypothesis is incompatible with the `fold_right` function, since its signature explicitly includes the fact that its argument is in the list.

It is necessary to define a more precise version of `fold_right` with a *compose* argument that is restricted so that it applies only to the elements in the list. This restricted version of `fold_right` is compatible with the hypothesis generated by the variadic recursion principle. It has the same structure as the canonical `fold_right` function; the extracted versions of the functions are identical modulo eta-conversion and reordering of function arguments. Since Coq accepts the canonical version in the context of the `size_e` function it should accept the more restricted version. Unfortunately, it rejects the restricted version, because it is unable to perform the necessary eta-conversions to expose the simple recursive structure. Consequently, the technique described in Chapter 3 is ineffective; it may become effective in later versions of Coq if the syntactic restrictions that ensure termination are refined appropriately.

This problem could be eliminated by making variadic types first-class since it arises due to the explicit inclusion phrase in the hypothesis which first-class variadic types make unnecessary.

Extending Properties of Binary Functions to their Variadic Counterparts

The `size_e` function is uniform. Proven properties relating the natural numbers are easily applied to the non-variadic parts of the `size_e` function.

For example, the property

```
le_plus_l: forall n m : nat, n <= n + m
```

is easily used to show that the size of the left hand expression in an `app` expression is necessarily less than or equal to the size of the whole `app` expression.

A proof that the size of any expression in a variadic list is necessarily less than or equal to the size of the whole list does not follow so automatically due to the need to perform induction over the elements of the list. We are not suggesting that such a proof is difficult to construct, the problem is that it is not available without extra work by the developer.

We suggest that a valuable extension of the current work would be to automatically discover necessary lemmas based on the structure of the *compose* argument to each function in a way analogous to the approach describe in Section 8.3. In the case of `size_e` the *compose* argument is

```
fun x => fun acc => (size_e x) + acc
```

The essential composition operation is done by the `plus` operator. If we search for already existing lemmas about `plus` we will immediately discover the lemma `le_plus_l` shown above. An informal argument for the synthesis of a lemma regarding variadic lists is as follows.

`n + m` is an expression joined by a `plus`. Therefore, it corresponds to the result of applying the `size_e` function to a variadic list of expression, which we call `l`. On the other hand, `n` appears on the left hand side of the inequality alone, i.e., not in a `plus` expression. Therefore, `n` represents a single expression, which we call `e`. Since it also appears in the plus expression we include the hypothesis `SubtermP e l`, i.e., `e` is some element of `l`. From these observations we conclude that

```
forall l,
  (forall e,
    SubtermP e l ->
    size_e e <= size_e_variadic size_e l)
```

is a useful lemma that we may wish to include in our development. In fact, this is the one not wholly mechanical lemma that we needed to include in the experiment described in Section 8.5.

8.7 Conclusion and Related Work

Our main contribution is the observation that the variadic use of the `list` type is distinct from its use as a data structure. These distinct uses require different structural recursion principles. Coq provides a recursion principle appropriate for the data structure interpretation; we describe the structure, construction, and use of a recursion principle appropriate to the variadic interpretation.

We have conducted a case study, demonstrating that the use of the variadic structural recursion principle can be integrated easily and mechanically into an existing corpus of proof developments.

Unlike variadic functions, variadic constructors are virtually undiscussed in the literature. Recent work (Petit, 2009) gives a formal theoretical treatment. Coq requires that its proof language have certain properties, i.e., weak normalization, in order to ensure consistency. The preliminary results indicate that an extension of the polymorphic lambda-calculus with variadic constructors preserves the desired properties.

An implementation of variadic constructors must necessarily deal with the practical problems of matching an unbounded number of fields. Operations which distinguish among the individual fields must be prohibited, since it is unknown at compile time how many fields are matched. Consequently, the implementation itself is likely to store the matched fields

in an array or list and to perform operations using list functions like *map* and *fold*. We expect that eventually variadic constructors will appear in high-level languages that make use of algebraic types. In the meantime, however, the programmer or proof developer must continue to encode variadic constructors by overloading the meaning of list in the way we have described in this chapter.

We have described a number of restrictions and difficulties in the use of the variadic principle in Section 8.6. We believe that the restrictions we have discovered should guide the language designer in integrating variadic types into an existing language for a proof assistant. It would be desirable if both of the main difficulties we have described in Section 8.6 were obviated by the language design itself.

Part III

Lemma Extraction and Proof Analysis

Certified programs require more skill to develop than uncertified programs and the process is generally more time-consuming. It is generally far more difficult to understand the purpose and intended use of the lemmas in a proof development than it is to understand the purpose and intended use of, e.g., the methods that a Java class provides. It is necessary to use automation to prove all but the simplest and smallest lemmas. The automation may be implemented by means of sophisticated *Ltac* scripts or by making use of Coq's internal proof-search mechanisms. In either case, it is very difficult to predict the proof that results.

Coq exports compiled proofs in an XML format. Matita, a more recently developed proof-assistant, maintains the proofs it compiles in the same XML format. It is possible for both proof-assistants to share the same format since they both make use of the CoC.

Proofs in the XML format can be inspected by means of standard XML tools like XPATH. Such tools have the advantage that they are not closely tied to the implementation of either proof-assistant.

In Chapter 9 we describe the *canonical_inversion* tactic. This tactic is strictly more powerful than Coq's *inversion* tactic and can be used as a drop-in replacement. The *canonical_inversion* tactic generates an anonymous inversion lemma and memoizes it in Coq's XML format so that it may be reused later.

In Chapter 10 we present two tools that also make use of the XML format. The first tool is an impact analysis tool. To our knowledge it is the first impact analysis tool for the Coq system. The second tool is a tool for visualizing the structure of a module by a graphical representation of the dependencies between lemmas and definitions.

All the work described in this chapter makes use of the external XML representation for compiled proofs. In this way, we demonstrate that it is possible to construct useful tools that are not wholly tied to the implementation of any proof-assistant. The *canonical_inversion* tactic automatically

generates lemmas which are stored in the XML format. This approach is particularly well adapted for use in Matita, which has considerable infrastructure for automatically managing an indexed database of compiled proofs and definitions.

9 CANONICAL INVERSION

The separation between *Prop* and *Set* makes it impossible to deconstruct an object in sort *Prop* to build an object in sort *Set*. The *inversion* tactic, which extends the more primitive *case* tactic by deriving contradictions in as many subcases as possible, is therefore inapplicable in this situation. We have implemented a more powerful tactic, *canonical_inversion*. The *canonical_inversion* tactic advances a proof in the same way as the *inversion* tactic but succeeds in contexts where the *inversion* tactic fails.

9.1 Introduction

Coq's *inversion* tactic (The Coq Development Team, 2008) implements the intuitive notion of a proof by cases. Given a hypothesis, it decomposes the current goal into as many subgoals as there are ways to prove the hypothesis. If each subgoal is proved, then the current goal is proved.

Proofs and proof objects in Coq are specified in the CoC(Calculus of Co-inductive Constructions) (Paulin-Mohring, 1993). The CoC maintains multiple sorts in order to support Coq's extraction mechanism (Letouzey, 2003, 2008). Types in the sort *Prop* define properties and are elided by the extraction mechanism while types in the other sorts are retained by the extraction mechanism. Consequently, a term with a type in *Prop* can not be decomposed to form a term with a type in any other sort. Coq's type-checker forbids a *match* expression where the matched expression has a type in *Prop* but the type of the *match* expression itself is in some other sort. If such a *match* expression were allowed it would create a paradoxical situation; the structure of a term retained by the extraction mechanism would be dependent on the structure of a term elided by the extraction mechanism.

The *inversion* tactic works by constructing a partial *match* expression and

automatically completing some of the generated subgoals. Consequently it always fails when its hypothesis is in *Prop* but the goal is in another sort.

Where the hypothesis has a *canonical form*, i.e., its constructor is entirely determined by its type, it should succeed, regardless of the sorts of the hypothesis and the goal.

In the following we describe a situation using the familiar definition of evenness of natural numbers in which the *inversion* tactic fails. In this situation, the evenness hypothesis has a canonical form, and it would be correct for the *inversion* tactic to succeed. We describe an algorithm for a strictly more powerful tactic, *canonical_inversion*, which will always succeed in the case where an hypothesis has a canonical form. We make use of a novel *sandboxing* technique in the algorithm. The sandbox is a subgoal within the proof that is not an essential part of the proof, but rather a way to explore the possibility of generating a particular lemma. It should be elided in a declarative representation of the proof (Kaliszyk and Wiedijk, 2009; Sacerdoti Coen, 2010).

Asperti et al. describe a new type for tactics which allows more flexibility than the LCF type (Asperti et al., 2009). This type does not account for sandbox subgoals which should be removed once they have served their purpose. However, we suspect that the type could be augmented to accommodate sandbox subgoals.

Example

Let us assume that we have some hypothesis, `H`, which is a proof that some number, `n`, is even. The definition of `even` in the Coq standard library is shown in Figure 9.1. From our hypothesis we wish to prove some goal and it turns out that we must proceed by dividing the proof into different cases corresponding to the several possible ways in which `even n` can be proved. We naturally make use of the *inversion* tactic, which extends the more primitive *case* tactic by deriving contradictions in as many subcases

```
Inductive even : nat -> Prop :=
    even_O : even 0
  | even_S : forall n : nat, odd n -> even (S n)
  with odd : nat -> Prop :=
    odd_S : forall n : nat, even n -> odd (S n)
```

Figure 9.1: The definition of the `even` type.

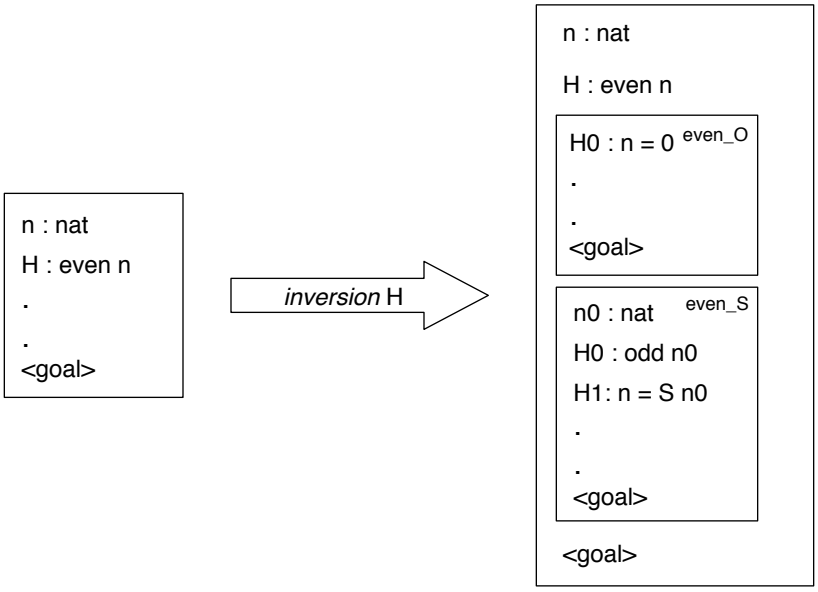


Figure 9.2: A schematic of the effect of the *inversion* tactic.

as possible. Figure 9.2 shows the effect of the *inversion* tactic. Two subgoals, corresponding to the two constructors `even_O` and `even_S` are generated. To prove the goal for the whole proof it is necessary to prove the goal in the context of each subgoal. Each newly generated subgoal contains the premises required to build a proof that `n` is even using its associated constructor.

However, suppose that the goal must be proved not for an arbitrary number `n` but for some number that is known to have the form `S n`. Figure 9.3 shows the effect of the inversion tactic in this case. The hypothesis

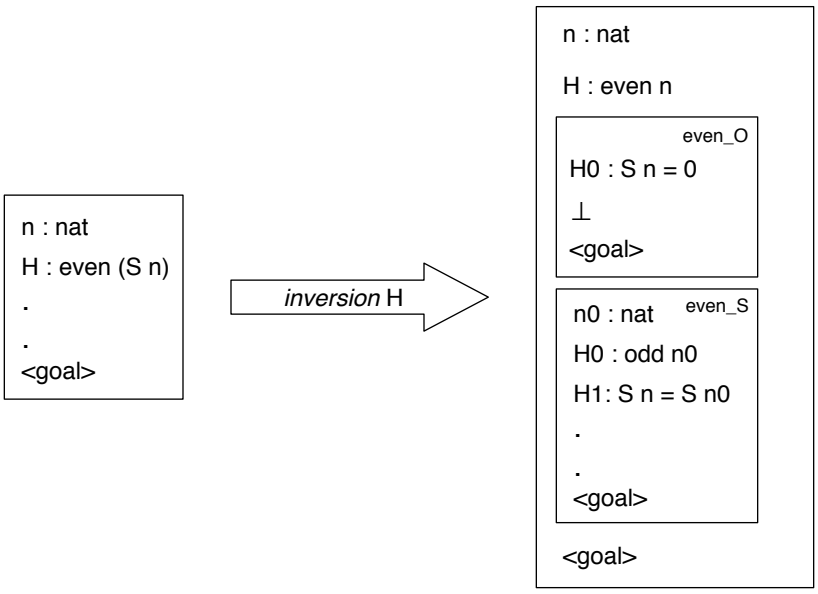


Figure 9.3: A schematic of the effect of the *inversion* tactic when only one subgoal does not lead to a contradiction. The goal corresponding to the `even_O` constructor is eliminated since the assumption that `S n` is equal to `O` leads to a contradiction.

that `S n` is equal to `O` leads to an immediate contradiction. Consequently, the subgoal corresponding to `even_O` is immediately proved by the *inversion* tactic. The subgoal corresponding to the constructor `even_S` is the only subgoal remaining. Because all but one subgoal is eliminated it is possible to conclude from the original hypothesis `H` the two premises corresponding to the `even_S` constructor, that the fresh variable, `n0`, is odd, and that `S n0` is equal to `S n`.

Figure 9.4 shows an abstract representation of the effect of the *inversion* tactic. `H`, which inhabits some inductive type `I` is inverted yielding n subgoals corresponding to `I`'s n constructors. In this case, all subgoals lead to a contradiction except for the kth subgoal.

When `I` is in *Prop* and the goal is not the *inversion* tactic will fail because

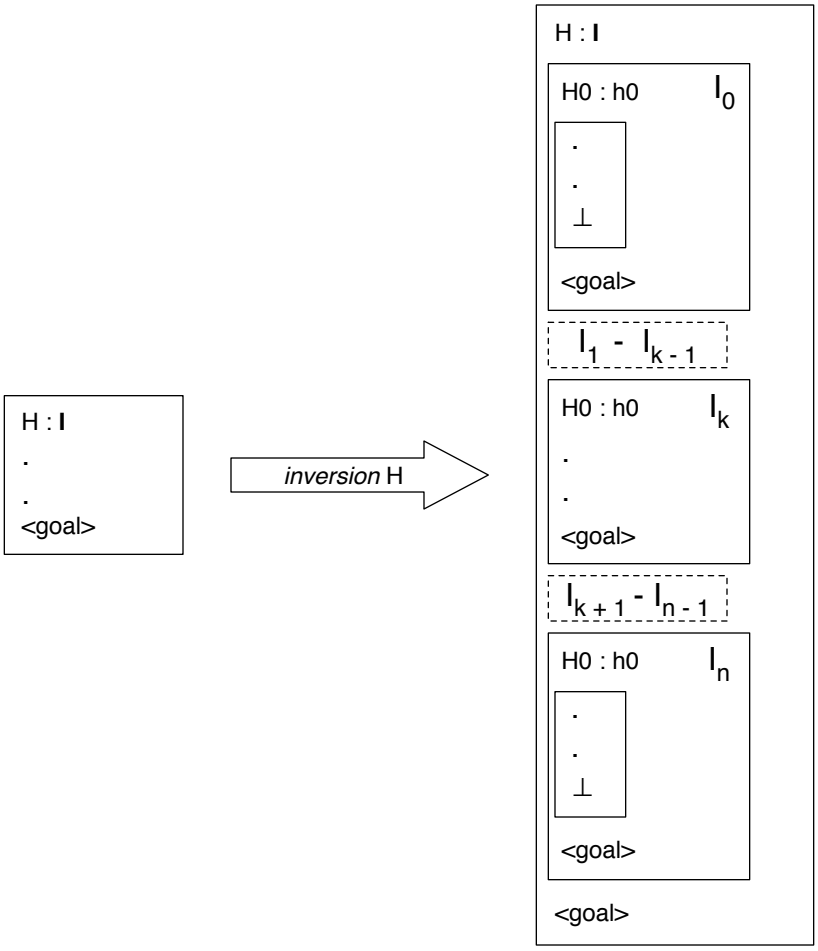


Figure 9.4: A schematic of the effect of the *inversion* tactic when the inverted term has the canonical form property. A new subgoal is generated for every constructor of `I`. Every subgoal but that corresponding to the *k*th constructor leads to a contradiction. Recall that the *inversion* tactic cannot proceed when `I` is in *Prop* and the goal is not.

it must decompose the hypothesis into cases and this is forbidden when building a goal that is not in *Prop*. This is an unnecessary restriction when only one subgoal is possible. In that case, it is possible to encapsulate the inversion in a sublemma. For the proof shown in Figure 9.3 the sublemma must have the form

```
forall n, even (S n) -> forall n0, S n = S n0 -> odd n0
```

or, simplifying the equality,

```
forall n, even (S n) -> odd n
```

The deconstruction of the `even (S n)` hypothesis occurs within the sublemma. Since both the hypothesis and the conclusion are in the same sort, this decomposition is permitted.

The result type of such a lemma is, in general, a conjunction of types. This conjunction may always be decomposed into its individual terms, regardless of the sort of the goal.

Canonical form property

If the type of a term fully determines the constructor for the term then it is said to possess the *canonical form* property. In that case, our *canonical_inversion* tactic will always succeed. All terms in the `and` type have the canonical form property since `and` has only the unique constructor `conj`. Other terms, particularly those that inhabit dependent types like `lc_e`, are also likely to have the canonical form property.

However, whether or not a term has the canonical form property is dependent not only on its type but also on how precisely its arguments are known. For example, the term `lc_e e` for any `e` does not have the canonical form property. `e` might be any expression, so the proof of local closure might be constructed from any of the three constructors for `lc_e`.

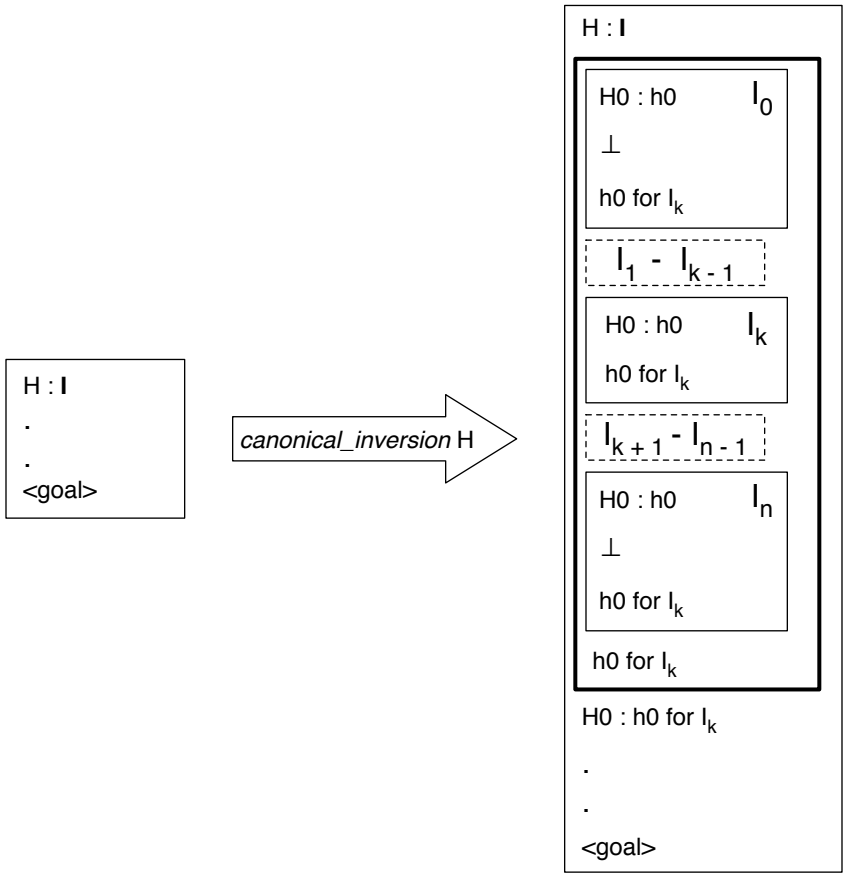


Figure 9.5: A schematic of the effect of the *canonical_inversion* tactic. The automatically generated inversion sublemma is outlined in bold.

Figure 9.5 shows the effect of our *canonical_inversion* tactic in the case where the inverted term possesses the canonical form property. The *canonical_inversion* tactic automatically constructs the necessary inversion lemma, outlined in bold, and applies it to generate the appropriate inversion hypothesis. Because the *match* expression is encapsulated, the *canonical_inversion* tactic can proceed regardless of the sort of the goal. Note that the subgoal generated by the *canonical_inversion* tactic is identical to the one that would be generated by the *inversion* tactic (Figure 9.4) if it were

able to succeed, even though it does not occupy the same position among the generated subgoals.

Currently, to achieve the effect of the *canonical_inversion* tactic the user must manually define a sublemma for every term that she anticipates inverting and which possesses the canonical form property. This is certainly tedious. Moreover it is brittle; should the inductive definition for which the lemmas have been constructed be altered then so must the lemmas themselves be altered. In contrast, the *canonical_inversion* tactic allows the user to automatically construct the necessary sublemmas whenever possible. It may be seamlessly substituted for the *inversion* tactic; it is strictly more powerful.

9.2 The *canonical_inversion* algorithm

The *canonical_inversion* tactic is an extension of the *inversion* tactic. In fact, it always attempts the *inversion* tactic; and only if that tactic fails does it attempt to build the necessary sublemma.

Should the *inversion* tactic fail the *canonical_inversion* tactic proceeds to attempt to build an anonymous sublemma of the correct type. Where the inductive definition is a canonical form it will always succeed. However, it can discover this only by exploratory tests as the structure of the inductive definition is not available to the *Ltac* language. For this reason, it builds a proof *sandbox* in which to run these tests. Figure 9.6 is a diagram of the tactic. Each box represents a subgoal. The goal, G, appears below the bar, while the proof-context is above. H is the hypothesis to be inverted. We assume that T, the type of H is in *Prop* while G, the proof goal, is not in *Prop*. Only relevant hypotheses are shown in the diagram. The solid arrow shows the new subgoal produced by the *canonical_inversion* tactic. The dotted arrows show occurrences of subgoal generation during the execution of the tactic. The dashed arrows show how information flows

from the sandbox to the main part of the proof to complete the inversion.

A proof sandbox can be made easily by asserting the intention to prove `True`. `True` is always provable; moreover, since it is a proposition a proof of `True` will be elided in any extracted code. By this assertion (Marker 1) the current goal is split into two subgoals; the first subgoal has all the hypotheses of the original goal but a goal of `True`, the second subgoal has all the hypotheses of the original goal as well as the new hypothesis, `True`, and a goal identical to that of the original goal. Clearly, including a hypothesis of `True` in the proof-context does not substantially change the goal; as nothing can be deduced from `True`. Thus the second subgoal is, in all important respects, identical to the original goal; removing the hypothesis `True` (Marker 5) simply tidies up the subgoal. The first subgoal is our sandbox.

Following this step, the proof-context of the sandbox is cleared of all hypotheses except the hypothesis to be inverted (Marker 2). Note that the hypothesis to be inverted may be dependently typed; any hypotheses on which it depends must also remain in the subgoal. The hypothesis is then inverted using the *inversion* tactic (Marker 3). The *inversion* tactic will always make progress, since the goal, `True`, is in *Prop*. The application of the *inversion* tactic may result in zero, one, or more remaining subgoals.

If there is more than one remaining subgoal then the inductive definition does not have the canonical form property. The only thing to do is clean up the sandbox and allow the tactic to fail, yielding an error indicating that the inductive definition is unsuitable. This cleanup procedure is not shown in the diagram.

If there is just one remaining subgoal then it is certain that the inductive definition has the canonical form property. In that case, the proof-context will have been populated with new hypotheses. The conjunction of these new hypotheses is exactly the result type of the sublemma to be be constructed, while the type of the original hypothesis is exactly the parameter

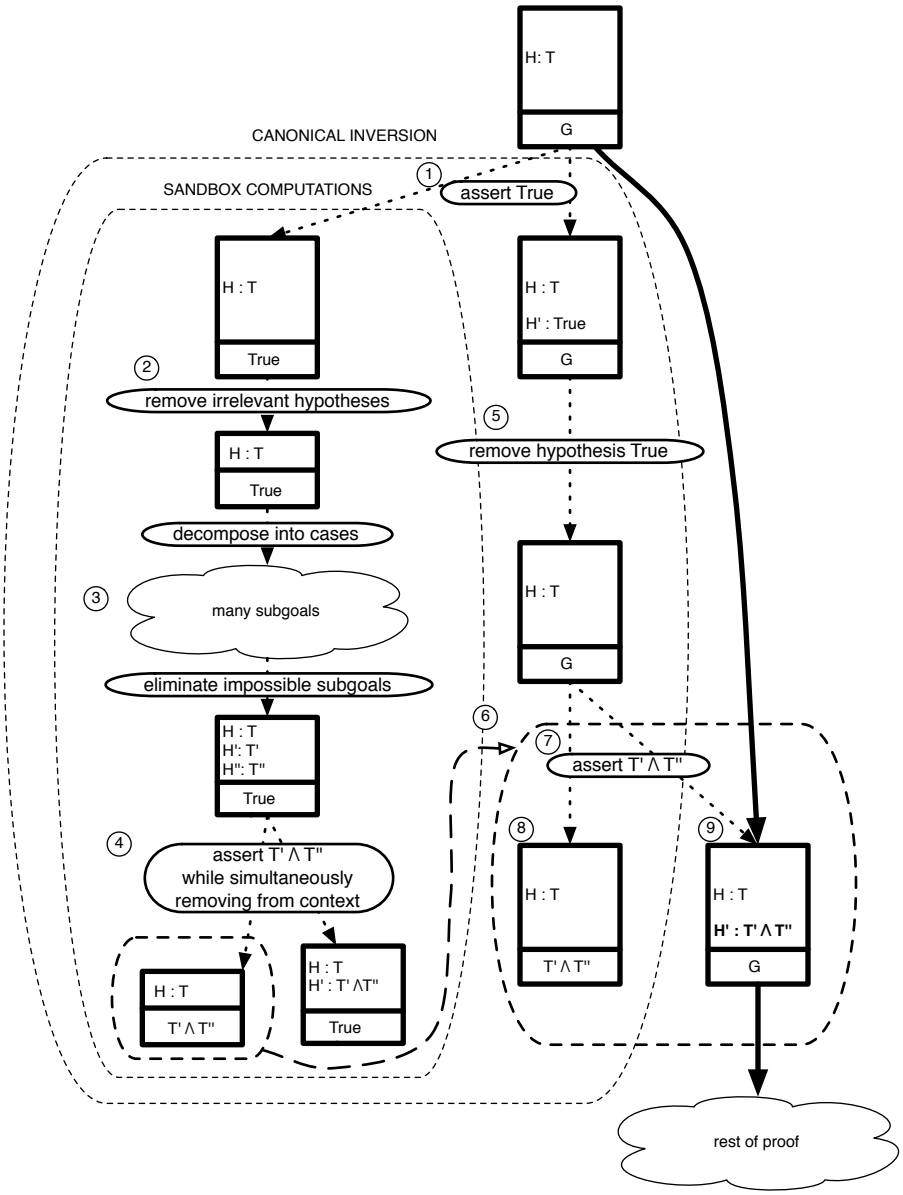


Figure 9.6: A schematic of the operation of the *canonical_inversion* tactic.

type. It only remains to construct a proof of the sublemma.

The tactic does that by asserting the intention to prove the result type from the original hypothesis. This is easily done by removing the hypotheses generated by the *inversion* tactic from the proof-context in the sandbox and asserting their conjunction (Marker 4). The sandbox is itself duplicated by this action, splitting into two sandboxes, both of which can certainly be proved. The first sandbox has only the original hypothesis in its proof-context; the goal is the conjunction of the new hypotheses generated by the *inversion* tactic. We can use the *abstract* tactic to simultaneously prove the goal and retain the generated proof. This proof is exactly the sublemma we sought in the first place.

It only remains to return to the original goal, sublemma in hand (Marker 6), and introduce the conclusion of the sublemma into the proof-context. This is easily accomplished by asserting the intention to prove the subgoal (Marker 7). This splits the goal in the same way as the assertion at Marker 1; this time, the goal of the first subgoal is the conclusion of our synthesized sublemma. The first subgoal is immediately proved by applying our sublemma (Marker 8).

We are left with just one subgoal; the proof-context of our original goal has been extended with the new hypotheses generated by the inversion lemma (Marker 9). This is exactly the subgoal that would have resulted from the direct application of the *inversion* tactic if the goal, G, had been in *Prop* or if the type of the hypothesis, T had not been in *Prop*.

When H is an impossible hypothesis

It may be the case that inverting the hypothesis H would yield no subgoals, i.e., the *inversion* tactic would derive a contradiction from every subgoal. If that is the case, then the goal can be eliminated entirely. Our *canonical_inversion* tactic explores this special case first; before investigating the more general case described above.

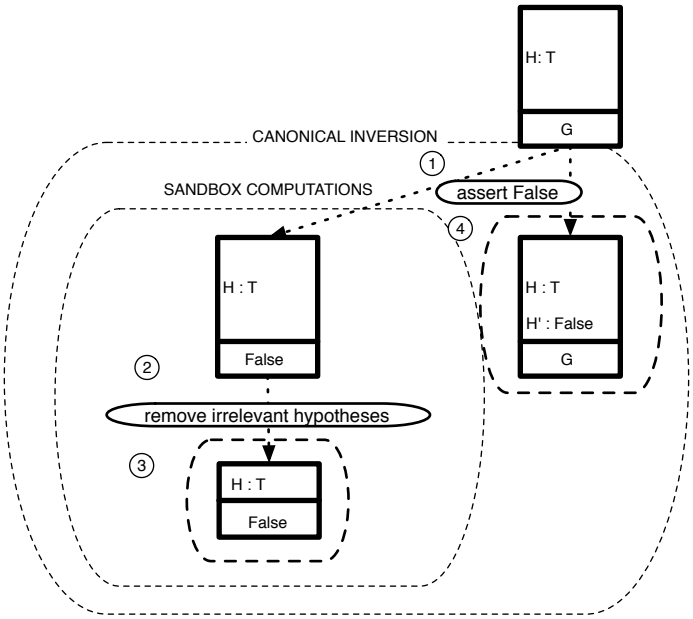


Figure 9.7: A schematic of the operation of the *canonical_inversion* tactic in the case where H yields a contradiction.

Figure 9.7 is a diagram of this portion of the tactic. Here, we begin by asserting the intention to prove a contradiction (Marker 1). As in the more general portion of the tactic, we clear the generated subgoal of irrelevant hypotheses. If we succeed in proving a contradiction (Marker 3) the subgoal in the sandbox disappears leaving only the right hand subgoal (Marker 4) which differs from the original goal only in that the proof-context has been extended with the hypothesis `False`. `False`, like `and`, has the canonical form property; in fact, it has zero constructors. We may invert the hypothesis H' and the subgoal is eliminated.

9.3 Memoization

Creation of the sandbox and exploration of the resulting subgoal consumes resources. However, once the sublemma has been constructed through

sandboxing and exploration it may be reused in any similar situation.

The *canonical_inversion* tactic has been extended with a memoizing capability. To memoize the sublemma we take advantage of Coq's *external* tactic which allows Coq to make external calls via an XML encoding. The *canonical_inversion* tactic communicates with an external database via the *external* tactic retrieving the appropriate sublemma if it has already been inserted the database and exploring via the sandbox only when a sublemma does not already exist.

The sublemma has no free variables. The task of marshalling the sublemma from its CoC form to an XML representation to send it to the database and unmarshaling it when it is extracted from the database and inserted into an ongoing proof presents little theoretical difficulty. Much more difficult is the question of how to index the sublemma within the database so that it can be extracted. Indexing on the name of the variable being inverted is manifestly useless. However, indexing on the type presents problems as well. Many types will contain variable names and these names are in general going to differ in different proof-contexts. We are really interested in indexing each sublemma so that we can retrieve the appropriate lemma regardless of superficial changes in variable names.

There are no facilities within the *Ltac* language to canonicalize terms in the CoC. It is, of course, possible to construct a facility to canonicalize the terms using facilities external to Coq. However, this would be a kind of reimplementation of a small portion of the Coq engine; we are philosophically opposed to this, preferring to leverage existing Coq facilities. For this reason, we index on the exact text of the type in XML format. This is useful in the particular case where a user must rerun a proof; perhaps to replay a tactic for proof understanding. In that case, the type of the proof to be inverted is quite likely to have the same textual representation and an advantage is gained from memoization.

By rerunning the identical proof twice the user may avoid sandboxing

entirely, taking complete advantage of memoization. On the first pass, since the sublemmas have not been constructed, each use of the *canonical_inversion* tactic may require a sandbox. On the second pass, all sublemmas will have been memoized and can be used directly in each proof without any sandboxing. Thus, by using two passes, a proof optimization may be achieved without any additional user intervention.

Extending tactics to allow memoization

We have proposed an external form of memoization. The advantage of this approach is that it does not require any change to the existing Coq infrastructure. However, we believe that our *canonical_inversion* tactic would be more powerful and useful if it were able to make use of an internal memoization facility.

Tactics in Coq, like *auto*, which implement a Prolog-like decision procedure, have an extended syntax which allows the user to specify a list of databases containing lemmas to be used in proof search. We believe that the *canonical_inversion* tactic could be usefully extended with the same facility. Unlike the *auto* tactic and all other tactics in Coq which make use of hints databases, the *canonical_inversion* tactic should insert into as well as retrieve lemmas from the specified databases. Thus, on a previous execution of the tactic, it might make use of the sandbox to build the type and body of the appropriate lemma and insert it into the specified hints database, while on a subsequent execution it might simply look up the lemma in the database and make full use of Coq's automation facilities in choosing the correct lemma.

Matita (Asperti et al., 2006, 2007; Sacerdoti Coen et al., 2007) is a proof-assistant which resembles Coq to an unusually large degree as it is based on the CoC and makes explicit use of the Curry-Howard isomorphism. Unlike Coq, Matita places considerable emphasis on automatic management and regeneration of libraries, tracking dependencies and regenerating

lemmas that have been invalidated by a change in some definition (Asperti et al., 2004). In Coq, definitions are only available to the user if they have been explicitly included in the development. In Matita, however, the development environment maintains a database of all definitions that have ever been included during any development. If the user wishes to make use of a module in the current development then she must still explicitly include the module in her development. However, if she wishes to search for a definition that matches some criterion the search will not be limited to those module that she has explicitly loaded. Another interesting aspect of Matita is that lemma names are required to bear a relationship to their type (Asperti et al., 2007).

Consequently, we believe that Matita is better able to support the form of internal memoization that we suggest. We propose that it extend its current system so that it is able to maintain a database of tactic-generated lemmas as well as of lemmas explicitly defined by the user. We propose the generation of pseudo-modules for these lemmas so that Matita can maintain dependencies between definitions in the identical manner that it uses for explicitly defined lemmas. In keeping with Matita's requirement for theorem names that bear a relationship to their types we suggest that the module names bear a relationship to their generating tactic.

Memoizing in other contexts

The *canonical_inversion* tactic fails when the hypothesis to be inverted does not have the canonical form property or when the *inversion* tactic would itself fail. The reader may wonder why it is not useful to build sublemmas in the case where a term lacks the canonical form property. It is certainly possible to build such a sublemma; the return type must be a disjunction of conjunctions with one subterm for every subgoal that the *inversion* tactic failed to eliminate. The `or` type lacks any canonical form; thus it is impossible to decompose a disjunction into subterms wherever the goal

is not in *Prop*. Consequently, the result of application of the sublemma must be impossible to decompose in exactly the same situations where the *inversion* tactic would be unable to proceed. To allow such sublemmas would not advance the current goal of strengthening and improving the inversion tactic.

The memoization that we have described constitutes a user-assisted form of *procedure extraction* (Komondoor and Horwitz, 2000, 2003), which is a recognized goal of software engineering. We would hesitate to maintain that it is only the *inversion* tactic that can make use of memoization. The ingenuity of tactic developers is likely to discover existing tactics or to invent new ones where memoization of automatically generated lemmas is appropriate. The introduction of a memoizing capability into an existing proof-assistant as described in Section 9.3 is likely to precipitate the development of such tactics.

9.4 Discussion

We have implemented the *canonical_inversion* tactic, which is strictly more powerful than the *inversion* tactic. We have made use of the *Ltac* tactic language in a novel way. To our knowledge, the use of sandboxing and of memoization in Coq are both original with us.

The tactic is always correct because it use the *Ltac* tactic language to generate sublemmas which are checked by the Coq type-checker. It may fail if it is impossible to construct the necessary sublemmas. Consider the following scenario. The user has defined an inductive definition and written multiple proofs using the *canonical_inversion* tactic all of which succeed. Subsequently, the user changes the inductive definition and the *canonical_inversion* tactic fails. This can have but one cause; the user has changed the inductive definition so that inverted terms no longer have the canonical form property they had previously. Thus, our *canonical_inversion*

tactic provides the user with useful feedback, i.e., an indication that the user has broken a property that most likely is intended to hold.

Coq provides a command, *Derive Inversion*, for deriving specialized inversion lemmas (The Coq Development Team, 2008). The primary motivation is one of efficiency since inversion lemmas can be substantial. The command fails when the term to be inverted is in *Prop* and the result type is in any other sort in the same way and for the same reason that the *inversion* tactic fails. We propose that the *Derive Inversion* command be made stronger so that it succeeds where the *canonical_inversion* tactic succeeds.

The algorithm implemented is certainly intricate; however we believe that it will remain useable through multiple versions of Coq since it relies solely on an interface to the Coq engine rather than on details of its implementation. Asperti et al. propose a new type for tactics to address some of the limitations of the LCF tactic type (Asperti et al., 2009). The sandbox which we make use of in our algorithm represents a different kind of subgoal which the type does not allow for. We believe that this type could be adapted to allow sandbox goals. Moreover, we believe that the new type for tactics could reduce the complexity of the currently implemented algorithm.

Our database-based memoization is a restricted substitute for memoization implemented within the system. Tactics which implement an automatic proof search in Coq, e.g., *auto*, may specify a list of hint databases which contain lemmas to be tried during proof search. The syntax of the *canonical_inversion* tactic could be extended to allow specifying hints databases in the same way. Unlike the *auto* tactic, however, the *canonical_inversion* tactic would insert as well as look up lemmas. The proof-assistant Matita (Asperti et al., 2006, 2007; Sacerdoti Coen et al., 2007) has extensive support for managing the dependencies between definitions. We believe that it could be extended with facilities for a general memoization

capability and that lemmas generated as a side effect of tactic execution could coexist with lemmas explicitly asserted and proved by the user.

10.1 Introduction

Tools for software analysis (Anderson, 2004; Anderson and Teitelbaum, 2001) and for automated software refactoring (Fowler, 1999; Opdyke, 1992) have been available for many years. Similar tools for Coq (Bertot et al., 2000; Pons et al., 1998; Pons, 1997, 1999, 2000b,a, 2002), though developed in the last decade, have not endured to the present version.

This is unfortunate. As proofs grow larger and more ubiquitous the need for such tools will only increase. We believe that these tools have failed to endure in some cases because they were too specific and in other cases because they were too general. Moreover, the tools were closely coupled to Coq's internal representation and rapidly became obsolete as Coq evolved.

We propose two tools that suffer from none of these problems. Neither tool will be rendered obsolete by a new version since each makes use of Coq's XML extraction facility (Asperti et al., 2004, 2000b,a, 2001; Sacerdoti Coen, 2003) which remains relatively stable between versions.

The first tool is an impact analysis tool (Arnold, 1996). Such tools are familiar in an object-oriented setting (Ren et al., 2005; Ryder and Tip, 2001) where dynamic dispatch makes the effect of modifying a class hierarchy difficult to predict. In programs in object-oriented languages changes in one part of a program may have significant non-local effects. It may be the case in some version of a program that only objects of a certain class can reach a certain method call-site. A developer may modify the program so that in the new version objects of a different class now flow to the method call-site. Due to virtual method invocation, a different method may be invoked at the call-site, causing the behavior of the program to change in unanticipated ways. An impact analysis tool for object-oriented

languages makes use of static and dynamic analyses to infer the often widely distributed affects of what may appear to be a minor code change.

The type-system of the CoC includes the sorts *Prop*, *Set*, and *Type*. Inductive definitions in *Set*, e.g., `nat` represent values, those in *Prop*, e.g., `even`, represent properties of values. The type-system prevents deconstructing a term with a type in *Prop* to build a term with a type in *Set*. In Coq, a superficially benign change in the sort of an inductive datatype may make a previously well-typed proof development ill-typed. A naive developer might expect that if the name and type of every constructor of every inductive type remains the same and if the syntactic structure of every proof remains the same then every proof should type-check as before. However, when the sort of an inductive definition is transferred from *Prop* to *Set* some *match* expressions may become ill-typed because they now deconstruct a term in *Prop* to build a term in *Set* or *Type*. These *match* expressions are often hidden in large and complex proof terms and are virtually invisible.

Our tool allows the user to predict the effect on an existing proof development of transferring an inductive definition from *Prop* to *Set* or *Type*. It discovers *match* expressions that will become ill-typed if the inductive definition in question is transferred from *Prop* to another sort. Furthermore, it identifies the dependencies between inductive types that result in the ill-typing and the proof terms that contain those dependencies.

The second tool is a comprehension tool. It is generally the case that a particular module is constructed using tactics. Lemmas defined earlier in the module may be used in lemmas defined later in the module. The developer makes use of Coq's automation facilities to apply the correct lemmas to complete the proof. Unlike a developer in a standard programming language, she has no knowledge of which functions are applied in what contexts in the final proof. The user of such a module is likely to be oblivious to the dependencies between lemmas and may fail to under-

stand the purpose of certain lemmas. We study graphical methods to help the developer and user of the module understand the hidden structure of modules both for better comprehension and to assist modular compilation.

Both tools make use of Coq's XML extraction mechanism. The extraction mechanism was developed for use in the HELM project at the University of Bologna. The general purpose of the HELM project (Asperti et al., 2004, 2001; Sacerdoti Coen, 2003) is to serve as a searchable repository for formal proofs developed in any theorem prover or proof-assistant. Every proof-assistant and automated theorem prover has a different representation and often a different underlying logic (Wiedijk, 2006). Consequently, a proof extracted from Coq and a proof of the same fact extracted from another theorem prover may have profoundly different XML representations. Automatic transformation to a declarative specification (Sacerdoti Coen, 2010) may demonstrate that the proofs are logically similar or expose their fundamental logical differences.

It is our insight that the XML representation of a Coq proof development can serve as a substrate for proof analysis. Proof terms in an XML format are easily inspected using standard tools such as XPATH. Since the syntax of the CoC is quite simple it is fairly straightforward to devise XPATH queries which extract important facts about the relationships between different parts of a proof. By making use of the XML representation of proof developments the analysis can be be uncoupled from Coq's internal representation. The tools described in this chapter are just two possible analyses that can be performed on the XML representation of a Coq proof development.

Besides the relatively loose coupling to the Coq implementation each tool has another advantage. Our impact analysis tool is more specific than preceding tools; it addresses a precisely defined problem that faces any developer who has discovered a need to change the sort of an existing inductive definition. Our second tool is more general; unlike tools such

as the Interactive Derivation Tool (Trac et al., 2007) which allows users to navigate a single proof analyzing dependencies in detail our tool allows users to see at a glance the dependencies between many lemmas (Ball and Eick, 1996).

Matita (Asperti et al., 2006, 2007) is a recently developed proof-assistant with a strong resemblance to Coq. It makes explicit use of the Curry-Howard isomorphism and is based on the CoC(Calculus of Co-inductive Constructions) (Paulin-Mohring, 1993). Unlike Coq, which only produces an XML representation of terms when explicitly requested, Matita performs an extraction to an XML representation as a side-effect of compilation. Because the underlying language of proof-terms is almost the same the techniques described in this chapter should be easily adapted to use with Matita.

10.2 Impact Analysis

Introduction

Software refactoring (Fowler, 1999; Opdyke, 1992) is a well-established field of study. Modern IDEs incorporate powerful refactoring tools (Dig et al., 2007; Tip et al., 2004, 2003). No such tools exist for Coq or, to our knowledge, any other theorem prover or proof assistant.

Our tool predicts the impact of transferring an inductive datatype from *Prop* to *Set* or *Type*. If one inductive datatype is translated from *Prop* to *Set* or *Type* then it may be necessary to translate other inductive datatypes that depend on it to *Set* or *Type* as well. The impact may be transitive, forcing a migration of many inductive datatypes from *Prop*.

In general, inductive datatypes inhabit *Prop* if they represent properties rather than computational content. However, a developer may consider transferring an inductive definition to *Set* or *Type* for a number of reasons. For example, the automatically generated recursion principle associated

with the inductive definition may be especially convenient; it is for this reason that the inductive definition that defines local closure is in *Set* in Aydemir et al. (2009) rather than *Prop* as in Aydemir et al. (2008).

Dependencies between inductive datatypes are generated by *match* expressions. If *I* is the type of the expression matched and *J* is the type of the *match* expression then *J* depends on *I*. If *J* is transferred from *Prop* to *Set* or *Type* then the proof will be untypable unless *I* is in *Set* or *Type*. If *I* inhabits *Prop* then it must be transferred to *Set* or *Type*.

Currently, to discover these dependencies a developer must transfer an inductive definition from *Prop* and then rerun proof scripts to see whether or not a dependency exists. If a dependency does exist, the script will break. A novice developer may be stymied by this problem; an experienced developer will understand the difficulty. However, the only remedy is to transfer the inductive definition that caused the script to break from *Prop* to another sort. This may happen many times as the developer iteratively migrates one inductive definition after another from *Prop*.

The remedy is to identify these dependencies and their causes and thereby estimate the impact of the change before implementing it. We do this by exploring the bodies of proofs in a development, identifying dependencies between inductive definitions. To our knowledge, ours is the first tool to analyze and automatically predict the impact of any change in a Coq development.

Implementation

The dependencies we have discussed are caused solely by *match* expressions. In theory, the task is simply to find all *match* expressions. If every *match* expression in a proof term, *T*, has a type constructed from the inductive definition *J* and its expression has a type constructed from the inductive definition *I* then we can store the triple, *(J,I,T)*, indicating that *I* depends on *J* through the lemma *T*. The developer can then search the

database of dependencies to identify triples where J is the inductive definition that she wishes to translate. If she discovers some triple where I is in *Prop* then she will know that she must either transfer I from *Prop* or jettison the proof T which contains the dependency.

However, proof terms contain function calls which may themselves contain *match* expressions. Induction and recursion principles are all functions of this type and they are ubiquitous. No XML query is powerful enough to identify these functions, dynamically inline them, and discover if a dependency exists.

The obvious solution to this problem is to inline all functions. This approach is impracticable due to the large size of Coq proofs. It may take as much as half an hour to extract a single proof with all functions inlined to an XML format. Moreover, we have observed that the XML extraction mechanism is likely to reach machine-imposed limits in far less than half an hour.

The only practical solution is to inline only those functions that are relevant. Consequently, the whole strategy must change. Rather than find all pairs, we consider a strategy where we investigate queries about dependencies between just two inductive definitions. We anticipate that the user will select some inductive definition J which she wishes to change and will construct a query to discover whether there is a dependency between it and some likely inductive definition I.

The first task is therefore to locate all functions that deconstruct expressions in I. To locate the functions we examine the bodies of every expression in a development. Every *match* expression is identified in the extracted XML by the `MUTCASE` tag; the type of the matched expression is identified by the `uriType` attribute. Thus every function or lemma that has a body where the `uriType` attribute corresponds to I can be easily identified. It is often the case that there are only a handful of such functions.

Every identified function must be inlined in the body of every lemma

in which it appears. This also presents a challenge. In general, Coq expects that all lemmas will be terminated by the vernacular command `Qed`. This command renders the body of each lemma *opaque*, i.e., it is impossible to perform any reductions within the body of the lemma or to replace the occurrence of the lemma with its body in any other proof. Making proofs opaque enforces proof-irrelevance and allows more efficient automation.

Clearly, delta-reduction, i.e., inlining of functions is exactly the operation necessary for the analysis. To make this possible, we terminate every lemma in the original development with the vernacular command `Defined` rather than `Qed` making each lemma *transparent*. Delta-reductions are performed within the body of the lemma as appropriate. The dependency analysis is performed on the XML representation extracted from the reduced versions of the lemmas.

The dependency analysis locates all *match* expressions with a type in J where the matched expression has a type in `I` and reports the lemma where the dependency exists. If a dependency does exist between J and I the developer may search for an additional dependency between I and some other inductive definition.

We implemented our analysis in Python and stored our results in an SQLITE database making use of existing Python libraries for constructing and executing XPATH and SQL queries. Our analysis takes about fifteen minutes on a moderately sized development with about one hundred lemmas. We believe that our analysis is linear in the size of proof terms and expect that it will scale well to larger developments. Since dependencies are recorded in an SQLITE database queries do not need to be rerun unless the development is changed.

Discussion and Related Work

We have presented the task of finding dependencies as an iterative procedure. However, it is only the practical limits of the Coq extraction

mechanism and memory that force us to constrain our approach in this way. Consequently, other approaches that do not place too great a strain on the extraction mechanism are just as good.

For example, the developer may really be interested in the dependencies among a small set of inductive definitions. The algorithm can be adapted easily to inline all functions with *match* expressions where the matched expression has the type of any of the inductive definitions the developer is interested in. The analysis that discovers the dependencies can analyze all *match* expressions with a type in any of these interesting definitions.

Our algorithm is sound, but not complete. Any dependency that it identifies is a real dependency; however, it may not identify all dependencies. The analysis is hampered by the CoC's sophisticated type system which makes it possible to compute types from values. If the function that computes a type is itself a fixed-point function on a term that may be of unbounded size no reduction step can yield the result type. Consequently, our analysis will not discover dependencies where the types are generated in this way.

Our analysis is still valuable. The recursive calculation of types from values is part of a particular proof style which is enthusiastically embraced by some and sedulously avoided by others. Our analysis will be most effective for those in the latter group. If our algorithm is able to identify just one dependency among several this information is still valuable; the developer is made aware that she must either change the sort of the dependent inductive definition or omit the lemma which contains the dependency.

Pons et. al (Pons et al., 1998; Bertot et al., 2000) formalized the distinction between *transparent* and *opaque* dependencies. A transparent dependency is one which arises through a proof term; an opaque dependency is one which arises through the statement of a proof. The dependen-

cies identified by our analysis are all transparent dependencies. It is our own observation that when changing the sort of an inductive definition transparent dependencies must be considered.

Pons et al. developed some tools to graphically represent the dependencies between lemmas. Unfortunately, these tools have been rendered obsolete by newer versions of Coq. Moreover, the tools they developed were not as focused on a particular application as our impact analysis tool but rather required the user to navigate an often large and complicated graphical representation.

Matita includes a mechanism for invalidating and regenerating lemmas when a definition is changed (Asperti et al., 2007). Such an automatic mechanism is also potentially awkward as the user may inadvertently trigger a massive invalidation by an apparently innocuous change. We believe that our tool could nicely supplement Matita's invalidation mechanism by warning a user when they are about to make changes that are likely to have a global effect.

10.3 Proof Segmentation

Introduction

In a sophisticated development which makes extensive use of automation the dependencies between lemmas may be only imperfectly understood by the developer and nearly invisible to the user of a module. For the developer this inhibits modular compilation; for the user it limits proof comprehension.

Almost all languages today make use of modular compilation. A large development is separated into individual files. Each file contains a logically coherent subpart of the entire development. Often, the interface and the implementation are distinguished. If the implementation remains true to the interface parts of the development that depend on the changed

implementation do not need to be recompiled.

Coq developments are not as modular. Partly this is because Coq is a proof-assistant. In languages like O'Caml the top-level and the compiler are largely separate; in Coq the compiler is simply a wrapper for the top-level. Coq does not make use of interface files. If an implementation is changed everything that depends on the implementation must be recompiled regardless of whether the interface has changed.

Frequently, a developer may wish to tinker with some lemma in a file. If the lemma is toward the end of the file she will have to compile all lemmas before the one in which she is interested. CoqIDE facilitates interaction, allowing a user to back up in a file or advance to a point of interest. However, even this can be quite cumbersome since the Coq interpreter must preserve state. Moving back just one line in CoqIDE may take considerable time.

Standard build tools like *make* and *ant* operate at the file level. A file is recompiled if the files on which it depends are newer. This coarse granularity is reasonable for a standard programming language. A disciplined programmer places logically separate components in separate files and the benefits of modular compilation are realized.

In automated theorem proving, however, there is a mismatch between the logical size and the compilation time. While a compiler for a standard programming language must compile only source code a proof assistant like Coq must often do time consuming proof search and build very large proof terms. The ratio of compile time to text size is much greater for a proof assistant than for a standard compiler. Consequently, it is desirable to subdivide a logical development into smaller parts that can be compiled separately. Since the bodies of proofs are really large and may contain many implicit dependencies on other proofs this is a difficult task.

It is relatively easy to extract the dependencies between lemmas from Coq's XML representation. However, if the file truly does form a logical

unit a graph of these dependencies is likely to be large and confusing with many interconnected nodes. The graph alone cannot assist the developer to decide how to separate the file into compilable units.

Often a single file will contain a number of core definitions that are not dependent on anything else in the file. Simple lemmas that define simple properties about the core definitions will follow. More sophisticated lemmas make use of the simpler lemmas and so forth.

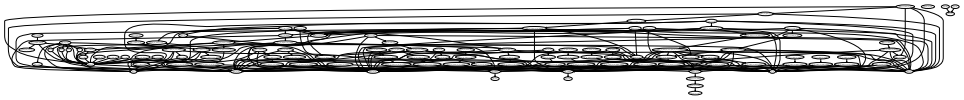
If this stratification exists, it is possible to discover it by post-processing the dependence graph, merging nodes with identical out-edges. This approach can decrease the number of distinct nodes by an order of magnitude and expose the stratified structure. The developer can make use of this simplified graph to subdivide a single logical unit into multiple compilation units.

The same graph can assist a user of the module to comprehend its overall structure. Because it encodes the stratified structure of the module the user is able to single out fundamental definitions, simple lemmas, and super lemmas that connect many parts of the development.

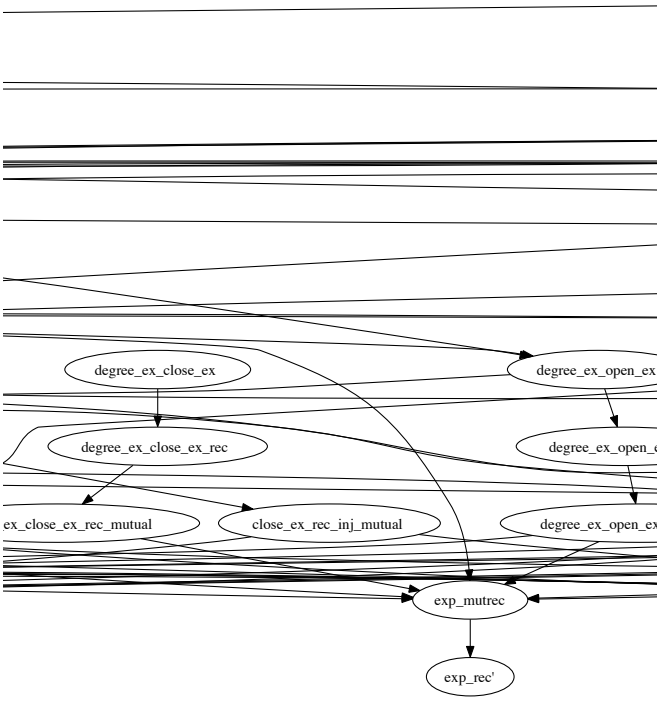
Implementation

A simple set of XPATH queries identifies all constant names, i.e., lemmas or functions, in any function body. This information is easily output in the dot format (Gansner and North, 2000), suitable for display. However, in its unmodified form the graphical representation is too complex to convey much information for all but the simplest developments. Figure 10.1 shows a graph of the dependencies in a development associated with Aydemir et al. (2009). The whole graph is large and complicated. The representative graph detail shows a mess of intersecting lines. The graph does not convey the overall structure of the module at all.

We have implemented an algorithm to coalesce nodes in the graph that have identical dependencies. Figure 10.2 shows the graph that results



(a) All Dependencies



(b) Detail (relative scale 24:1)

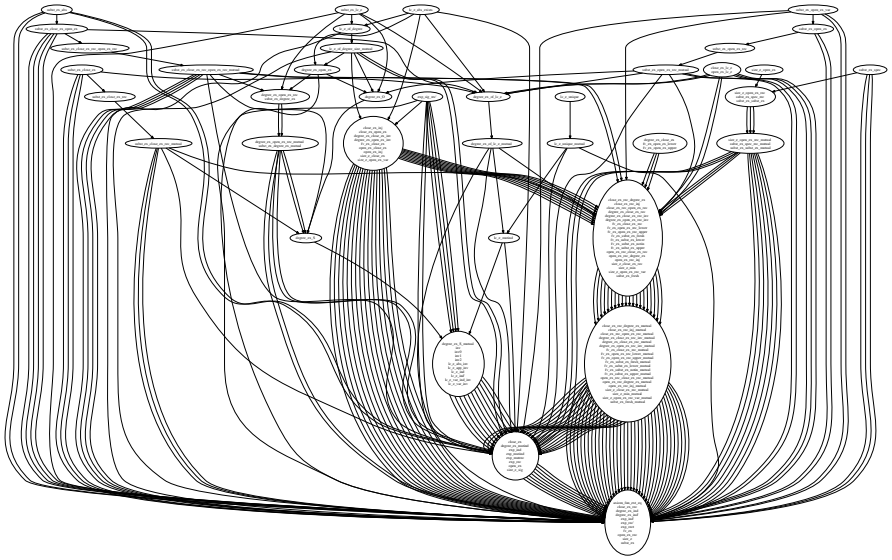
Figure 10.1: Dependencies among lemmas in a metatheoretic development for the untyped lambda-calculus.

from applying our algorithm to the graph in Figure 10.1. The graph is still large but its complexity is greatly reduced and its overall structure is now apparent. The single node at the very bottom contains all fundamental definitions, i.e. definitions that do not depend on any other definitions in the module. It is easy to identify the set of simple lemmas of basic facts about the fundamental definitions and so forth.

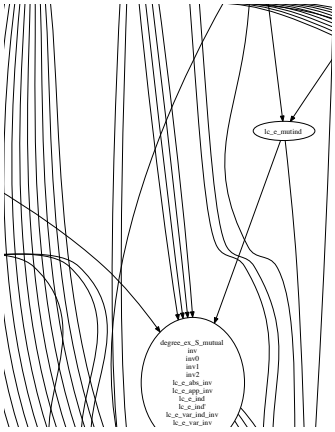
About thirty lemmas have dependencies that make them unique. These lemmas are the most powerful and most interesting. This interestingness is not made obvious from the complexity of the tactic scripts. For example, `lc_e_abs_exists` is a proof that if the body of a lambda-calculus term is locally closed under a certain operation, then so is the whole term. The tactic script is a single line. The simplified graph shows that this is a significant theorem. No other theorem depends on it; so clearly the intention is to expose it to users of the library. It depends on several other unmergeable lemmas. By singling out significant lemmas our tool can assist the user in making use of the module. She is less likely to needlessly make use of simpler lemmas to re-prove a more complicated lemma that the module already makes available to her.

The simplified graph also allows a developer to divide the logically coherent development into smaller parts more suitable for modular compilation. Each of the supernodes has a set of very simple dependencies. Each of these nodes could be placed in a single compilation unit and thereby be recompiled only when the compilation units on which it depends are recompiled. This can make for a significant time savings; even the simple language examples distributed with the abstracting syntax development take several minutes to compile.

We implemented our node coalescing algorithm in JGraphT, an open source Java graphical library. Our algorithm iteratively identifies and coalesces pairs of nodes with identical dependencies. It is non-deterministic since it may identify pairs in any order but it will always converge to the



(a) All Dependencies



(b) Detail (relative scale 4:1)

Figure 10.2: Dependencies among lemmas in a metatheoretic development for the untyped lambda-calculus. Nodes with identical dependencies are coalesced.

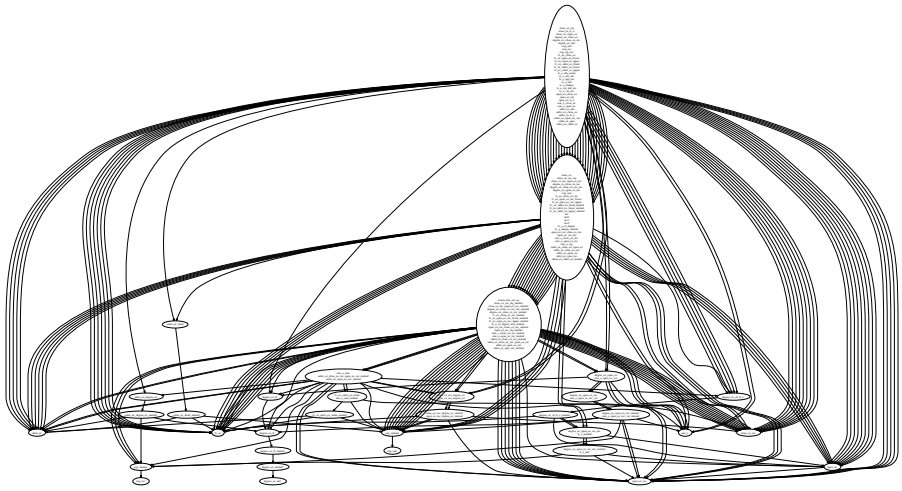


Figure 10.3: Dependencies among lemmas in a metatheoretic development for the untyped lambda-calculus. Nodes with identical dependents are coalesced.

same solution regardless of the order in which nodes are coalesced.

Discussion and Related Work

In the graph in Figure 10.2 nodes are coalesced if they depend on the same nodes. The node at the bottom, with no outgoing edges, represents all definitions that do not depend on any other definitions in the module. Figure 10.3 shows a graph derived from the same development where nodes have been coalesced if they are depended on by the same nodes. The node at the top, with no incoming edges, represents all definitions that are not depended on by any other definitions in the module. We anticipate that such a graph would be useful in the situation where a developer realizes that there is a general lemma which subsumes a number of previous lemmas that she has proved and used in her development. She may wish to recompile her development to discover whether the new general lemma is substituted for the set of less general lemmas. If her

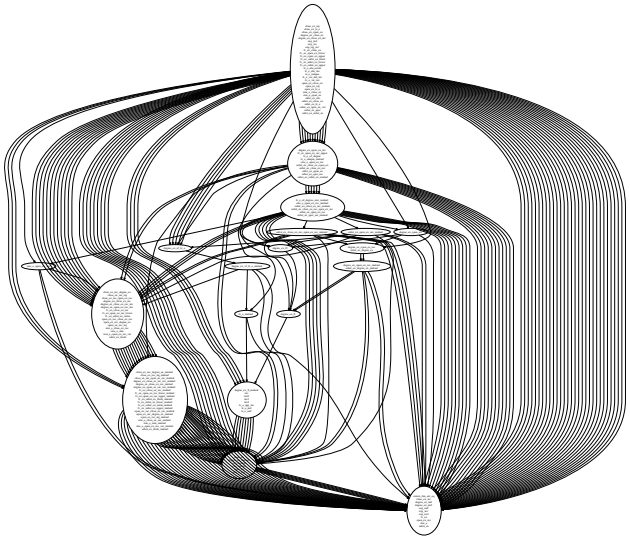
changes have been successful she will observe that the general lemma is depended on by many other lemmas but that the less general lemmas it replaced are unused and have therefore migrated to the top node.

It is possible to combine the two coalescing strategies so that nodes are coalesced if their dependents are the same or if the nodes they depend on are the same. However, this computation will converge to different solutions depending on the order in which nodes are coalesced. Figure 10.4 shows the outcome of two executions of an algorithm which coalesces nodes using both criteria. The graphs are distinct but similar and retain much of the structure of the graph in Figures 10.2 and 10.3.

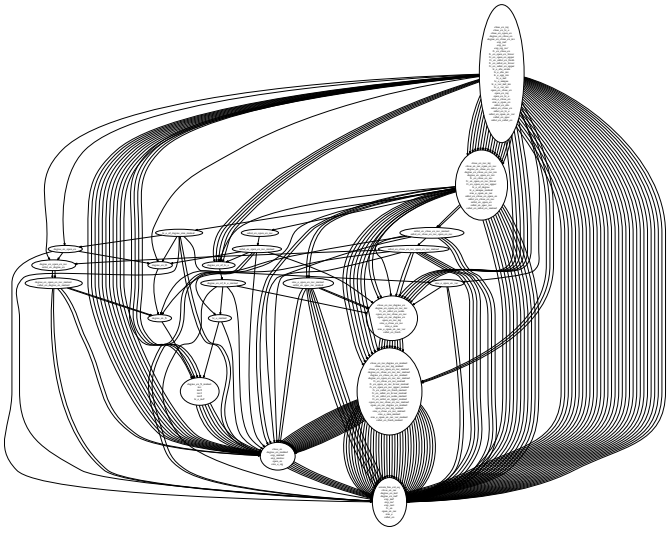
Pons et al. (Bertot et al., 2000; Pons et al., 1998; Pons, 1997, 1999, 2000b,a, 2002) examined a number of graphical proof visualization techniques. Unfortunately, these tools were closely tied to the Coq implementation. Without the time to build a strong user community they were made obsolete by the next version of Coq.

Pons also experimented with reducing a graph by omitting explicit dependencies that were also transitive. This reduced the edges in the graph and made the graph appear less busy. We have implemented the same procedure (Figure 10.5) and have found that, while the graph does appear less cluttered, the overall structure of the module remains unclear. Coalescing graph nodes to show the stratified structure of a development is entirely our own idea.

We have experimented with coalescing the nodes of a graph with all transitive edges eliminated. Figure 10.6 shows the result of coalescing the graph in Figure 10.5 based on shared dependents and dependencies. Figure 10.7 shows the result of coalescing according to shared dependencies and Figure 10.8 shows the result of coalescing according to shared dependents. The reader will observe that the structure of the graphs is quite similar to the corresponding graphs with transitive edges shown in Figures 10.4, 10.2 and 10.3 respectively. The moiré effect is lessened



(a) One execution



(b) Another execution

Figure 10.4: Dependencies among lemmas in a metatheoretic development for the untyped lambda-calculus. Nodes with identical dependencies and dependents are coalesced.

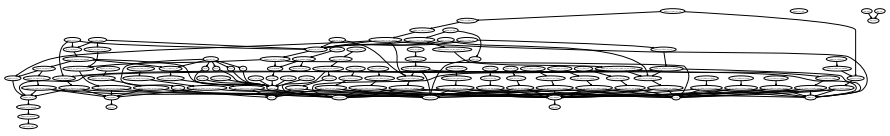


Figure 10.5: Dependencies among lemmas in a metatheoretic development for the untyped lambda-calculus. Transitive edges have been eliminated.

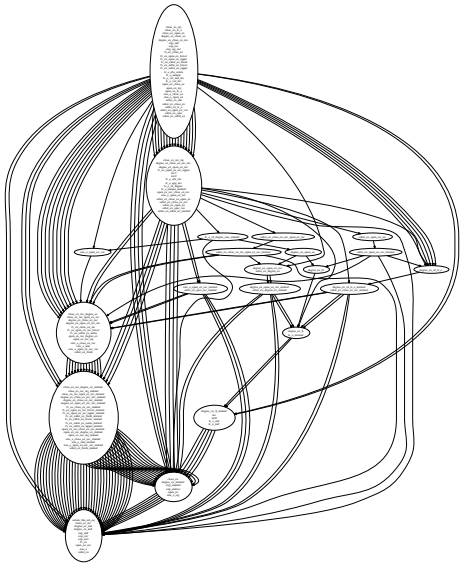
and many of the edges between the nodes at the top and at the bottom of graph have disappeared entirely.

Matita (Asperti et al., 2007) makes use of dependency information in invalidation and regeneration of lemmas. While its invalidation mechanism is fine grained, operating at the level of individual definitions, its regeneration mechanism is much coarser, operating at the level of individual files. At this time, it does not display dependency information, or make use of the information to suggest possible segmentation of files. Thus it is likely to pay the same costs of regeneration as Coq and to reap similar benefits from decomposition of a development into smaller files.

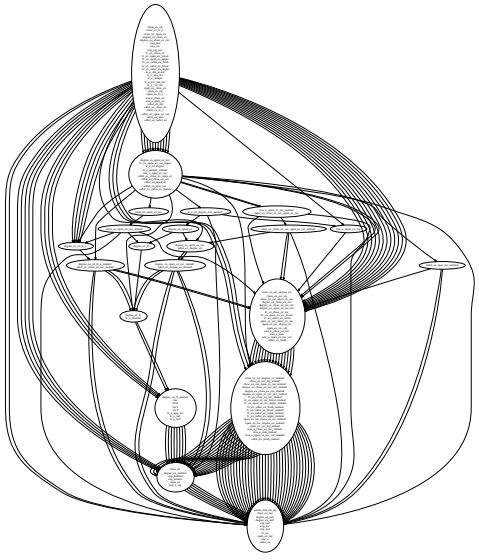
Ball and Eick experimented with visualizations for software engineering (Ball and Eick, 1996). Mulhern et al. suggested related visualizations for proof engineering (Mulhern et al., 2006). Sacerdoti Coen developed a facility for extracting declarative specifications of a proof from the proof terms generated by Coq (Sacerdoti Coen, 2010).

10.4 Conclusion

In this chapter we have demonstrated an impact analysis tool and a comprehension tool. Both are novel in that they make use of Coq's XML representation as a basis for the analysis rather than relying on Coq's internal representations. We thereby illustrate that the barriers to this analysis are not too high. Rather than requiring understanding of Coq's internal structure these analyses require the ability to construct XPATH



(a) One execution



(b) Another execution

Figure 10.6: Dependencies among lemmas in a metatheoretic development for the untyped lambda-calculus. Non-transitive edges are eliminated and nodes with identical dependencies and dependents are coalesced.

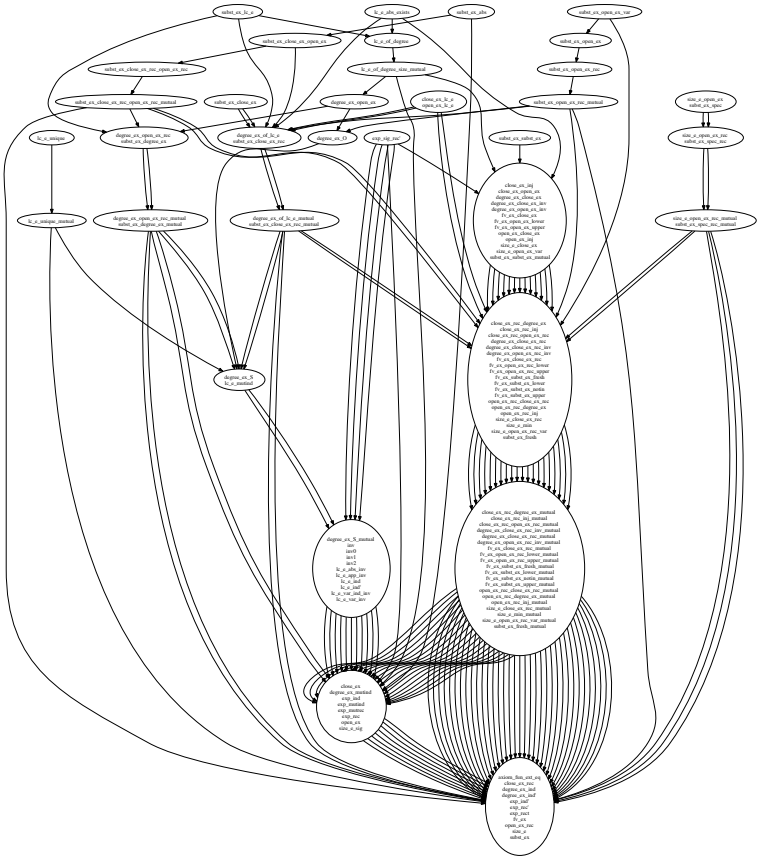


Figure 10.7: Dependencies among lemmas in a metatheoretic development for the untyped lambda-calculus. Non-transitive edges are eliminated and nodes with identical dependencies are coalesced.

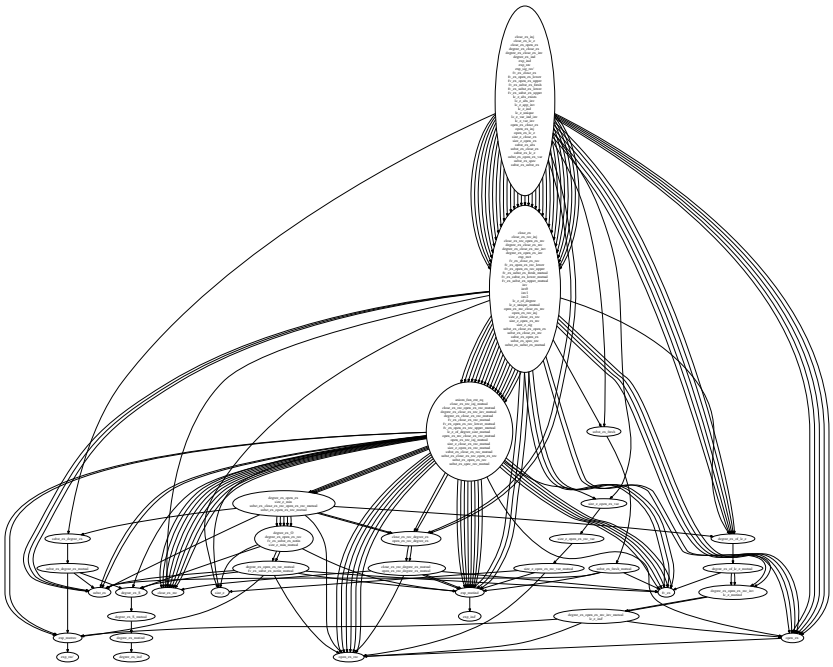


Figure 10.8: Dependencies among lemmas in a metatheoretic development for the untyped lambda-calculus. Nontransitive nodes are eliminated and nodes with identical dependents are coalesced.

queries and to be able to relate the extracted XML to the actual structure of proof terms. Of course, the second task is difficult, however it is made easier by the CoC's simple syntax and by the ease with which diagnostic XPATH queries can be constructed.

Coq's user community has grown in the last ten years. The use of automation has grown much more sophisticated and developments are larger. There is increasing need for tools for proof comprehension. By making use of the XML representation of Coq terms we show that such tools can be developed by someone who is not closely associated with the Coq development team and that they can be robust to changes in Coq's implementation. Our analyses address current problems. We believe that we have demonstrated that ordinary users can develop tools to address the

problems with which they are confronted; problems which the developers of Coq may not have anticipated.

We anticipate a powerful IDE where our impact analysis tool is run automatically when the sort of an inductive definition is changed from *Prop* to another sort in order to predict the necessary changes. We have pointed out that the graph constructed from all dependencies between lemmas is really too complicated for a human being to understand. On the other hand, a sophisticated compiler could make use of this structure by separately compiling every lemma.

Matita tracks dependencies between lemmas, invalidating and regenerating lemmas as appropriate. We anticipate that a tool for more precise analysis of dependencies may be useful in evolving its regeneration mechanism. The current mechanism relies entirely on re-running scripts. In some cases it might be preferable to operate directly on the proof-object originally generated by the script, making modifications as suggested by Mulhern (Mulhern, 2006).

More philosophically, we note that both our tools are made necessary in part by Coq's awkward dual role. We have pointed out that Coq's facilities for interactive theorem proving are highly developed. However, Coq's batch compilation is just a wrapper around its interactive component. A tremendous amount of ingenuity has gone into working within the existing framework to make proofs more modular, more fully automatic, and more regular. We hope for an eventual paradigm change; where batch compilation is facilitated by much more sophisticated tools. We believe that the tools we have developed may prove useful in facilitating this change.

Part IV

Conclusion

Formal methods (Hoare, 1969) for the specification and verification of programs have been in existence for several decades. Automated theorem proving (Moore, 2004a) , one part of formal methods, is just as old.

However, while the academic community continues to refine and improve formal methods, their use is still not widely known in the software engineering community nor is exposure to formal methods a common experience for students in a computer science curriculum.

This situation must be remedied. The complexity of software continues to grow, as does our reliance on it. The gap between formal methods in academia and in the worlds of software engineering and of computer science education is likely to have increasingly deleterious consequences.

Our thesis works addresses this problem by seeking to make working with the automated proof-assistant Coq less difficult both for the novice and for the experienced user.

11.1 Generalizing General Induction and Recursion

We go about this in several ways. First, we address the problem of general induction and recursion. In many cases, structural induction or recursion is insufficient for the particular specification or proof required. A familiar example is the quicksort algorithm (Hoare, 1962). This algorithm is indeed recursive, but rather than working recursively on the subparts of a list, i.e., its head and tail, it divides the list into two sublists around a pivot element. Developing a specification of the quicksort algorithm in Coq requires some expertise since the developer must make use of general recursion to prove that the specification is terminating.

This problem is well known. Consequently, much effort has been directed toward developing novel approaches to general induction and recursion. To our knowledge, however, only our work addresses the specific problem of induction and recursion over CoC terms defined using types with mutually inductive definitions. Many introductory examples will not require such definitions. However, more advanced developments, e.g., real compilers, certainly do. We present two mutually exclusive approaches to the problem of general recursion on mutually inductive datatypes. The first approach is to eliminate mutually inductive datatypes by converting any such type to a dependently typed definition. This approach is described in Chapter 2. Alternatively, we provide a facility that allows a developer to specify the general structure of a general induction or recursion principle and a tactic that allows the user to request an automatically generated principle suitable for the number of mutually inductive types that the induction term inhabits. This work is described in Chapter 5. Many general induction principles make use of a *measure*. We discuss the structure and domain of measures more fully in Chapter 4. Measures are an example of a particular form of recursively defined function that we call *uniform* functions. We discuss ways to automatically generate such functions in Chapter 3.

11.2 Extending Structural Induction and Recursion

The automatically generated structural induction and recursion principles that Coq provides are themselves sometimes inadequate.

Heterogeneous Structural Recursion

Coq's automatically generated recursion principles are *homogeneous*, that is they allow induction only over elements of a single type. It is often desirable to perform induction over a structure composed of elements of several types. In Chapter 6 we discuss how a heterogeneous principle may be constructed from a homogeneous principle and demonstrate how it can be used to generalize the lemmas about the `list` type that are defined in the Coq `List` module.

Tagged Structural Recursion

Moreover, the principles contain no information that allows the user or a tactic to identify the principal premise or constructor to which the current subgoal corresponds. We show how each principal premise can be augmented with an irrefutable `case` hypothesis. This allows the user to orient herself within the proof and allows automatic tactics to proceed based on the case to which the subgoal corresponds.

Variadic Types

The CoC does not allow the specification of variadic constructors, i.e., constructors which may take any number of arguments, by means of dependent types. Instead the user must make use of the `list` datatype for this purpose. Coq's automatically generated structural recursion principle does not place a variadic interpretation on the `list` type constructor. Instead, the `list` argument is treated as opaque. Consequently, the principal premise for that constructor is in general unsatisfiable since it provides no inductive hypothesis.

In Chapter 8 we discuss the automatic generation of a structural recursion principle that puts a variadic interpretation on the `list` type when that interpretation is specified by the special alias for `list`, `star`. We

demonstrate a tactic that will always succeed in constructing the variadic portions of the structural recursion principle. We demonstrate that, in spite of the fact that it incorporates nested recursion, a variadic principle works very well within a real development.

11.3 Lemma Extraction and Proof Analysis

Lemma Extraction

Coq's *inversion* tactic fails when the hypothesis to be inverted is in *Prop* but the goal is in some other sort. We show that if the hypothesis has a *canonical form*, i.e., its structure is determined by its type, that this restriction is unnecessary. In this case, it is possible to hide the inversion step within an automatically generated sublemma. Our *canonical_inversion* tactic makes use of a *sandboxing* technique to discover whether the canonical form property holds and, if so, to automatically generate the necessary sublemma. This tactic is strictly stronger than the *inversion* tactic and can be used as a drop in replacement for that tactic.

Proof Analysis

We make use of Coq's XML extraction facility to develop what is, to our knowledge, the first ever impact analysis for Coq. This analysis identifies *match* expression dependencies where the transfer of an inductive definition from *Prop* to another sort will break a proof development.

We make use of the same XML extraction facility to build graphical representations for the dependencies in a proof development. We show that we can post-process the graphs to expose important aspects of the structure of a proof development. This structure can assist a developer to separate a proof into smaller, more rapidly compiled components. It

can assist a user to understand the overall structure of the development as well as the purpose of individual lemmas.

Our impact analysis tool and our dependency analysis tool were both generated entirely from Coq's XML representation of proofs. Consequently, neither has a strong dependence on Coq's implementation and each is likely to survive through multiple Coq versions.

11.4 Coq and Matita

Coq is a well-established proof-assistant. Consequently, it has both the flaws and strengths of software project that has been undergoing constant development for over a decade. While it is very powerful it contains many oddities and some conflicting features and work-arounds.

Matita is a much newer proof-assistant being developed at the University of Bologna. It resembles Coq quite closely as it it also makes explicit use of the Curry-Howard isomorphism and its proof language is a variant of the CoC. However, since it is much newer, its design remains more flexible than that of Coq. Our approaches are suitable for Coq but may be more easily adopted by this newer, more adaptable, proof-assistant.

Part V

References

Altenkirch, Thorsten, Conor McBride, and James McKinna. 2005. Why dependent types matter. Manuscript, available online.

Anderson, Paul. 2004. CodeSurfer/Path Inspector. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, 508. Washington, DC, USA: IEEE Computer Society.

Anderson, Paul, and Tim Teitelbaum. 2001. Software inspection using Codesurfer. In *WISE '01: Proceedings of the Workshop on Inspection in Software Engineering*. Paris, France.

Arnold, Robert S. 1996. *Software change impact analysis*. Los Alamitos, CA, USA: IEEE Computer Society Press.

Asperti, Andrea, Herman Geuvers, Iris Loeb, Lionel Elie Mamane, and Claudio Sacerdoti Coen. 2005. An interactive algebra course with formalised proofs and definitions. In *MKM '05: Revised Selected Papers from the 4th International Conference on Mathematical Knowledge Management*, ed. Michael Kohlhase, vol. 3863 of *Lecture Notes in Computer Science*, 315–329. Springer.

Asperti, Andrea, Ferruccio Guidi, Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli. 2004. A content based mathematical search engine: Whelp. In *TYPES '04: Revised Selected Papers from the International Workshop on Types for Proofs and Programs*, ed. Jean-Christophe Filliâtre, Christine Paulin-Mohring, and Benjamin Werner, vol. 3839 of *Lecture Notes in Computer Science*, 17–32. Jouy-en-Josas, France: Springer.

Asperti, Andrea, Luca Padovani, Claudio Sacerdoti Coen, and Irene Schena. 2000a. Towards a library of formal mathematics. In *TPHOLs '00: Panel session of the 13th International Conference on Theorem Proving in Higher Order Logics*. London, UK.

———. 2000b. XML, stylesheets and the re-mathematization of formal content. In *Proceedings of Extreme Markup Languages 2001 Conference*, 3.

———. 2001. HELM and the semantic math-web. In *TPHOLs '01: Proceedings of the 14th International Conference on Theorem Proving in Higher Order Logics*, 59–74. London, UK: Springer-Verlag.

Asperti, Andrea, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. 2009. A new type for tactics. Tech. Rep. UBLCS-2009-14, University of Bologna.

Asperti, Andrea, Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli. 2006. Crafting a proof assistant. In *TYPES '06: Proceeding of the International Workshop on Types for Proofs and Programs*, ed. Thorsten Altenkirch and Conor McBride, vol. 4502 of *Lecture Notes in Computer Science*, 18–32. Springer.

———. 2007. User interaction with the Matita proof assistant. *Journal of Automated Reasoning* 39(2):109–139.

Aydemir, Brian, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. 2008. Engineering formal metatheory. In *POPL '08: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 3–15. New York, NY, USA: ACM.

Aydemir, Brian, Stephanie Weirich, and Steve Zdancewic. 2009. Abstracting syntax. Tech. Rep. MS-CIS-09-06, University of Pennsylvania.

Aydemir, Brian E., Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. 2005. Mechanized metatheory for the masses: The POPLmark Challenge. In *TPHOLs '05: Proceedings of the 18th International Conference on Theorem Proving in Higher*

Order Logics, ed. Joe Hurd and Thomas F. Melham, vol. 3603 of *Lecture Notes in Computer Science*. Oxford, UK: Springer.

Balaa, Antonia, and Yves Bertot. 2000. Fix-point equations for well-founded recursion in type theory. In *TPHOLs '00: Proceedings of the 13th International Conference on Theorem Proving in Higher Order Logics*, 1–16. London, UK: Springer-Verlag.

Ball, Thomas, and Stephen G. Eick. 1996. Software visualization in the large. *IEEE Computer* 29(4):33–43.

Barnes, David J., and Michael Kolling. 2006. *Objects first with Java: A practical introduction using BlueJ*. 3rd ed. Upper Saddle River, NJ, USA: Prentice-Hall, Inc.

Barthe, Gilles, Venanzio Capretta, and Olivier Pons. 2003. Setoids in type theory. *Journal of Functional Programming* 13(2):261–293.

Bertot, Yves, and Pierre Casteran. 2004. *Interactive theorem proving and program development : Coq'art: The calculus of inductive constructions*, vol. XXV of *Texts in Theoretical Computer Science*. Springer.

Bertot, Yves, Olivier Pons, and Loïc Pottier. 2000. Dependency graphs for interactive theorem provers. Tech. Rep. RR-4052, INRIA.

Blazy, Sandrine, Zaynah Dargaye, and Xavier Leroy. 2006. Formal verification of a C compiler front-end. In *FM '06: Proceedings of the 14th International Symposium on Formal Methods*, ed. Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, vol. 4085 of *Lecture Notes in Computer Science*, 460–475. Hamilton, Canada: Springer.

Blume, Matthias, Michael Rainey, and John Reppy. 2008. Calling variadic functions from a strongly-typed language. In *ML '08: Proceedings of the 2008 ACM SIGPLAN Workshop on ML*, 47–58. New York, NY, USA: ACM.

Bornat, Richard. 2005. *Proof and disproof in formal logic: An introduction for programmers*. Oxford Texts in Logic, Oxford University Press, USA.

Bove, Ana, and Venanzio Capretta. 2005. Modelling general recursion in type theory. *Mathematical Structures in Computer Science* 15(4):671–708.

Bove, Ana, Peter Dybjer, and Ulf Norell. 2009. A brief overview of Agda — a functional language with dependent types. In *TPHOLs '09: Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, 73–78. Berlin, Heidelberg: Springer-Verlag.

Bracha, Gilad. 2004. Generics in the Java programming language.

Bundy, Alan. 2001. The automation of proof by mathematical induction. In *Handbook of Automated Reasoning*, ed. John Alan Robinson and Andrei Voronkov, 845–911. Elsevier and MIT Press.

Chapman, James, Thorsten Altenkirch, and Conor McBride. 2006. Epigram reloaded: A standalone typechecker for ETT. In *Trends in functional programming*, ed. Marko van Eekelen. Intellect.

Charguéraud, Arthur. 2009. Proof pearl: A practical fixed point combinator for type theory. Unpublished. http://arthur.chargueraud.org/research/2009/fixwf/.

———. 2010. The optimal fixed point combinator. In *ITP '10: Proceedings of the International Conference on Interactive Theorem Proving*. To appear. http://arthur.chargueraud.org/research/2010/fix/.

Chlipala, Adam. 2006. Modular development of certified program verifiers with a proof assistant. In *ICFP '06: Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming*, 160–171. New York, NY, USA: ACM.

———. 2007a. A certified type-preserving compiler from lambda calculus to assembly language. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 54–65. New York, NY, USA: ACM.

———. 2007b. Implementing certified programming language tools in dependent type theory. Ph.D. thesis, EECS Department, University of California, Berkeley.

———. 2009. Certified programming with dependent types. `http://adam.chlipala.net/cpdt/`.

———. 2010. A verified compiler for an impure functional language. In *POPL '10: Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 93–106. New York, NY, USA: ACM.

Cross, James H., and T. Dean Hendrix. 2007. jGRASP: an integrated development environment with visualizations for teaching Java in CS1, CS2, and beyond. *Journal of Computing in Small Colleges* 23(2):170–172.

Danielsson, Nils Anders, and Thorsten Altenkirch. 2009. Mixing induction and coinduction. `http://www.cs.nott.ac.uk/~nad/publications/danielsson-altenkirch-mixing.html`.

Dann, Wanda P., Stephen Cooper, and Randy Pausch. 2008. *Learning to program with Alice*. Upper Saddle River, NJ, USA: Prentice Hall Press.

Delahaye, David. 2000. A tactic language for the system Coq. In *LPAR '00: Proceedings of the 7th International Conference on Logic for Programming and Automated Reasoning*, ed. Michel Parigot and Andrei Voronkov, vol. 1955 of *Lecture Notes in Computer Science /Lecture Notes in Artificial Intelligence*, 85–95. Reunion Island, France: Springer.

Dig, Danny, Ralph Johnson, Frank Tip, Oege de Moor, Jan Becicka, William G. Griswold, and Markus Keller. 2007. Refactoring tools. In *ECOOP Workshops*, ed. Michael Cebulla, vol. 4906 of *Lecture Notes in Computer Science*, 193–202. Springer.

Dillinger, Peter C., Panagiotis Manolios, Daron Vroon, and J. Strother Moore. 2007. ACL2s: The ACL2 sedan. In *UITP '06: Proceedings of the 7th Workshop on User Interfaces for Theorem Provers*, ed. Serge Autexier and Christoph Benzmüller, vol. 174 of *Electronic Notes in Theoretical Computer Science*, 3–18. Seattle, Washington: The 2006 Federated Logic Conference.

Fischer, Charles, Richard LeBlanc, and Ronald Cytron. 2010. *Crafting a compiler*. Addison Wesley.

Fowler, Martin. 1999. *Refactoring: Improving the design of existing code*. Boston, MA, USA: Addison-Wesley.

Gansner, Emden R., and Stephen C. North. 2000. An open graph visualization system and its applications to software engineering. *Software: Practice and Experience* 30(11):1203–1233.

Gibbons, Jeremy. 2003. Origami programming. In *The fun of programming*, ed. Jeremy Gibbons and Oege de Moor, 41–60. Cornerstones in Computing, Palgrave.

———. 2007. Datatype-generic programming. In *SSDGP '06: Revised Lectures from the International Spring School on Datatype-Generic Programming*, ed. Roland Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring, vol. 4719 of *Lecture Notes in Computer Science*. Nottingham, UK: Springer.

Gibbons, Jeremy, and Ross Paterson. 2009. Parametric datatype-genericity. In *WGP '09: Proceedings of the 2009 ACM SIGPLAN Workshop on Generic Programming*, 85–93. New York, NY, USA: ACM.

Gonthier, Georges. 2005. A computer-checked proof of the Four Color Theorem.

Hasker, Robert W. 1995. The replay of program derivations. Ph.D. thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA.

Hickey, Jason. 2008. *Introduction to Objective Caml*. Cambridge University Press. Forthcoming. An older version of the book is available at `http://files.metaprl.org/doc/ocaml-book.pdf`.

Hinze, Ralf, Johan Jeuring, and Andres Löh. 2007. Comparing approaches to generic programming. In *SSDGP '06: Revised Lectures from the International Spring School on Datatype-Generic Programming*, ed. Roland Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring, vol. 4719 of *Lecture Notes in Computer Science*. Nottingham, UK: Springer.

Hoare, C. A. R. 1962. Quicksort. *The Computer Journal* 5(1):10–15.

———. 1969. An axiomatic basis for computer programming. *Communications of the ACM* 12(10):576–580.

Hutton, Graham. 1999. A tutorial on the universality and expressiveness of fold. *Journal of Functional Programming* 9(4):355–372.

Jeuring, Johan, and Rinus Plasmeijer. 2006. Generic programming for software evolution. Tech. Rep. UU-CS-2006-024, Department of Information and Computing Sciences, Utrecht University.

Kaliszyk, Cezary, and Freek Wiedijk. 2009. Merging procedural and declarative proof. *TYPES '08: Revised Selected Papers from the Proceedings of the 8th International Workshop on Types for Proofs and Programs* 203–219.

Kalman, John Arnold. 2001. *Automated reasoning with Otter*. Rinton Press, Incorporated.

Kernighan, Brian W., and Dennis M. Ritchie. 1988. *The C programming language*. Prentice Hall Professional Technical Reference.

Komondoor, Raghavan, and Susan Horwitz. 2000. Semantics-preserving procedure extraction. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 155–169. New York, NY, USA: ACM.

———. 2003. Effective, automatic procedure extraction. In *IWPC '03: Proceedings of the 11th IEEE International Workshop on Program Comprehension*, 33. Washington, DC, USA: IEEE Computer Society.

Leroy, Xavier. 2006. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *POPL '06: Proceedings of the 33rd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 42–54. Charleston, SC: ACM Press.

———. 2009. Formal verification of a realistic compiler. *Communications of the ACM* 52(7):107–115.

Letouzey, P. 2004. Programmation fonctionnelle certifiée – l'extraction de programmes dans l'assistant Coq. Ph.D. thesis, Université Paris-Sud.

———. 2008. Coq extraction, an overview. In *Logic and Theory of Algorithms, Fourth Conference on Computability in Europe, CiE 2008*, ed. A. Beckmann, C. Dimitracopoulos, and B. Löwe, vol. 5028 of *Lecture Notes in Computer Science*. Athens, Greece: Springer-Verlag.

Letouzey, Pierre. 2003. A new extraction for Coq. In *TYPES '02: Proceeding of the 2nd International Workshop on Types for Proofs and Programs*, ed. Herman Geuvers and Freek Wiedijk, vol. 2646 of *Lecture Notes in Computer Science*. Berg en Dal, The Netherlands: Springer-Verlag.

Letouzey, Pierre, and Bas Spitters. 2005. Implicit and noncomputational arguments using monads. Tech. Rep., HAL-CCSD.

Linz, Peter. 2006. *An introduction to formal languages and automata*. 4th ed. Jones and Bartlett Publishers.

Megacz, Adam. 2007. A coinductive monad for prop-bounded recursion. In *PLPV '07: Proceedings of the 2nd Workshop on Programming Languages Meets Program Verification*, 11–20. New York, NY, USA: ACM.

Moore, J. Strother. 1999. Proving theorems about Java-like byte code. In *Correct System Design, Recent Insight and Advances, (to Hans Langmaack on the occasion of his retirement from his professorship at the University of Kiel)*, 139–162. Springer-Verlag.

Moore, J Strother. 2004a. How to prove theorems formally. `http://www.cs.utexas.edu/~moore/publications/how-to-prove-thms/main.ps`.

Moore, J. Strother. 2004b. On the adoption of formal methods by industry: The ACL2 experience. In *ICFEM '04: Proceeding of the 6th International Conference on Formal Methods and Software Engineering*, ed. Jim Davies, Wolfram Schulte, and Michael Barnett, vol. 3308 of *Lecture Notes in Computer Science*, 13. Springer.

Mulhern, Anne. 2006. Proof weaving. In *WMM '06: Proceedings of the 1st Informal ACM SIGPLAN Workshop on Mechanizing Metatheory*. Portland, Oregon: The Eleventh ACM SIGPLAN International Conference on Functional Programming.

———. 2009. Coq talks lecture notes. `http://www.cs.wisc.edu/~mulhern/coqtalks`.

Mulhern, Anne, Charles Fischer, and Ben Liblit. 2006. Tool support for proof engineering. In *UITP '06: Proceedings of the 7th Workshop on User Interfaces for Theorem Provers*, ed. Serge Autexier and Christoph Benzmüller. The 2006 Federated Logic Conference, Seattle, Washington: Electronic Notes in Theoretical Computer Science.

Naftalin, Maurice, and Philip Wadler. 2006. *Java generics and collections*. O'Reilly Media, Inc.

Nanevski, Aleksandar, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. 2008. Ynot: Reasoning with the awkward squad. In *ICFP '08: Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*. Victoria, British Columbia.

Nipkow, Tobias, L. C. Paulson, and Markus Wenzel. 2002. *Isabelle/HOL: A proof assistant for higher-order logic*. Lecture Notes in Computer Science 2283, Springer.

Oliveira, Bruno C.d.S., and Jeremy Gibbons. 2008. Scala for generic programmers. In *WGP '08: Proceedings of the ACM SIGPLAN Workshop on Generic Programming*, 25–36. New York, NY, USA: ACM.

Opdyke, William F. 1992. Refactoring object-oriented frameworks. Ph.D. thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA.

Orr, Mike, and Travis Rudd. 2005. *The Cheetah users' guide, release 0.9.17rc1*.

Oury, Nicolas. 2003. Observational equivalence and program extraction in the Coq proof assistant. In *TLCA '03: Proceedings of the International Conference on Typed Lambda Calculi and Applications*, ed. Martin Hofmann, vol. 2701 of *Lecture Notes in Computer Science*, 271–285. Valencia, Spain: Springer-Verlag.

Parr, Terence. 2007. *The definitive ANTLR reference: Building domain-specific languages*. Pragmatic Bookshelf.

———. 2010. *Language implementation patterns: Create your own domain-specific and general programming languages*. The Pragmatic Bookshelf.

Paulin-Mohring, Christine. 1993. Inductive definitions in the system Coq - rules and properties. In *TLCA '93: Proceedings of the International*

Conference on Typed Lambda Calculi and Applications, 328–345. London, UK: Springer-Verlag.

Petit, Barbara. 2009. A polymorphic type system for the lambda-calculus with constructors. In *TLCA '09: Proceedings of the 9th International Conference on Typed Lambda Calculi and Applications*, 234–248. Berlin, Heidelberg: Springer-Verlag.

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

———. 2008. Using a proof assistant to teach programming language foundations, or, Lambda, the ultimate TA. White paper.

Pons, Olivier. 1997. Undoing and managing a proof. In *UITP '97: Electronic Proceedings of the 3rd Workshop on User Interfaces for Theorem Provers*. Sophia-Antipolis, France.

———. 1999. Conception et réalisation d'outils d'aide au développement de grosses théories dans les systèmes de preuves interactifs. Ph.D. thesis, Conservatoire National des Arts et Métiers.

———. 2000a. Proof engineering. Rapport de Recherche, aboratoire CEDRIC-CNRAM.

———. 2000b. Proof generalization and proof reuse. In *Supplementary Proceedings of the 13th International Conference on Theorem Proving in Higher Order Logics: Poster session TPHOLs'00*, ed. J. Harrison M. Aagaard and T. Schubert. Portland U.S.: Oregon Graduate Institute Technical Report CSE 00-009.

———. 2002. Generalization in type theory based proof assistants. In *TYPES '00: Selected Papers from the 1st International Workshop on Types for Proofs and Programs*, 217–232. London, UK: Springer-Verlag.

Pons, Olivier, Yves Bertot, and Laurence Rideau. 1998. Notions of dependency in proof assistants. In *UITP '98: Electronic Proceedings of the 4th Workshop on User Interfaces for Theorem Provers*. Sophia-Antipolis, France.

Ren, Xiaoxia, Barbara G. Ryder, Maximilian Stoerzer, and Frank Tip. 2005. Chianti: a change impact analysis tool for Java programs. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, 664–665. New York, NY, USA: ACM.

Resnick, Mitchel, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. 2009. Scratch: Programming for all. *Communications of the ACM* 52(11):60–67.

Rondon, Patrick M., Ming Kawaguci, and Ranjit Jhala. 2008. Liquid types. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 159–169. New York, NY, USA: ACM.

Ryder, Barbara G., and Frank Tip. 2001. Change impact analysis for object-oriented programs. In *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, 46–53. New York, NY, USA: ACM.

Sacerdoti Coen, Claudio. 2003. From proof-assistants to distributed libraries of mathematics: Tips and pitfalls. In *MKM '03: Proceedings of the 2nd International Conference on Mathematical Knowledge Management*, ed. Andrea Asperti, Bruno Buchberger, and James H. Davenport, vol. 2594 of *Lecture Notes in Computer Science*, 30–44. Springer.

———. 2010. Declarative representation of proof terms. *Journal of Automated Reasoning* 44(1-2):25–52.

Sacerdoti Coen, Claudio, Enrico Tassi, and Stefano Zacchiroli. 2007. Tinycals: Step by step tacticals. In *UITP '06: Proceedings of the 7th Workshop on User Interfaces for Theorem Provers*, ed. Serge Autexier and Christoph Benzmüller, vol. 174 of *Electronic Notes in Theoretical Computer Science*, 125–142. Seattle, Washington: The 2006 Federated Logic Conference.

Sozeau, Matthieu. 2007a. Program-ing finger trees in Coq. In *ICFP '07: Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, 13–24. New York, NY, USA: ACM.

———. 2007b. Subset coercions in Coq. In *TYPES'06: Selected Papers from the 6th International Workshop on Types for Proofs and Programs*, vol. 4502 of *Lecture Notes in Computer Science*, 237–252. Nottingham, UK: Springer.

Swords, Sol, and William R. Cook. 2006. Soundness of the simply typed lambda calculus in ACL2. In *ACL2 '06: Proceedings of the Sixth International Workshop on the ACL2 Theorem Prover and its Applications*, 35–39. New York, NY, USA: ACM Press.

The Coq Development Team. 2008. *The Coq proof assistant reference manual, version 8.2*.

Tip, Frank, Robert Fuhrer, Julian Dolby, and Adam Kieżun. 2004. Refactoring techniques for migrating applications to generic Java container classes. IBM Research Report RC 23238, IBM T.J. Watson Research Center, Yorktown Heights, NY, USA.

Tip, Frank, Adam Kieżun, and Dirk Bäumer. 2003. Refactoring for generalization using type constraints. In *OOPSLA '03: Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, 13–26. Anaheim, CA, USA.

Trac, Steven, Yury Puzis, and Geoff Sutcliffe. 2007. An interactive derivation viewer. In *UITP '06: Proceedings of the 7th Workshop on User Interfaces*

for Theorem Provers, ed. Serge Autexier and Christoph Benzmüller, vol. 174 of *Electronic Notes in Theoretical Computer Science*, 109–123. The 2006 Federated Logic Conferences, Seattle, Washington: Elsevier Science Publishers B. V.

Wadler, P. 1987. Views: a way for pattern matching to cohabit with data abstraction. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 307–313. New York, NY, USA: ACM.

Wadler, Philip. 2000. Proofs are programs: 19th century logic and 21st century computing. *Report, Avaya Labs*.

Weirich, Stephanie, and Chris Casinghino. 2010. Arity-generic datatype-generic programming. In *PLPV '10: Proceedings of the 4th Workshop on Programming Languages Meets Program Verification*. Madrid, Spain.

Wiedijk, Freek. 2006. *The seventeen provers of the world: Foreword by Dana S. Scott (Lecture Notes in Computer Science / Lecture Notes in Artificial Intelligence)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc.

Winskel, Glynn. 1993. *The formal semantics of programming languages: an introduction*. Cambridge, MA, USA: MIT Press.

Xi, Hongwei. 2003. Dependently typed pattern matching. In *Proceedings of Simposio Brasileiro de Linguagens de Programacao*, 149–165. Ouro Preto, Brazil.

———. 2007. Dependent ML: an approach to practical programming with dependent types. *Journal of Functional Programming* 17(2):215–286.

Xu, Guoqing, Matthew Arnold, Nick Mitchell, Atanas Rountev, and Gary Sevitsky. 2009. Go with the flow: profiling copies to find runtime bloat. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 419–430. New York, NY, USA: ACM.