Three Tactic Theorem Proving

Don Syme

Microsoft Research Limited, St. George House, 1 Guildhall Street, Cambridge, CB2 3NH, UK

Abstract. We describe the key features of the proof description language of DECLARE, an experimental theorem prover for higher order logic. We take a somewhat radical approach to proof description: proofs are not described with tactics but by using just three expressive outlining constructs. The language is "declarative" because each step specifies its logical consequences, i.e. the constants and formulae that are introduced, independently of the justification of that step. Logical constants and facts are lexically scoped in a style reminiscent of structured programming. The style is also heavily "inferential", because DECLARE relies on an automated prover to eliminate much of the detail normally made explicit in tactic proofs. DECLARE has been partly inspired by Mizar, but provides better automation. The proof language has been designed to take advantage of this, allowing proof steps to be both large and controlled. We assess the costs and benefits of this approach, and describe its impact on three areas of theorem prover design: specification, automated reasoning and interaction.

1 Declarative Theorem Proving

Interactive theorem provers combine aspects of formal specification, manual proof description and automated reasoning, and they allow us to develop machine checked formalizations for problems that do not completely succumb to fully automated techniques. In this paper we take the position that the role of proof description in such a system is relatively simple: it must allow the user to describe how complex problems decompose to simpler ones, which can, we hope, be solved automatically.

This article examines a particular kind of *declarative proof*, which is one technique for describing problem decompositions. The proof description language we present is that of DECLARE, an experimental theorem prover for higher order logic. The language provides the functionality described above via three simple constructs which embody first-order decomposition, second-order proof techniques and automated reasoning. The actual implementation of DECLARE provides additional facilities such as a specification language, an automated reasoning engine, a module system, an interactive development environment (IDE), and other proof language constructs that translate to those described here. We describe these where relevant, but focus on the essence of the outlining constructs.

In this section we describe our view of what constitutes a declarative proof language and look at the pros and cons of a declarative approach. We also make a distinction between "declarative" and "inferential" aspects of proof description, both of which are present in the language we describe. In Section 2 we describe the three constructs used in DECLARE, and present a longer example of proof decomposition, and Section 3 discusses the language used to specify hints. Section 4 compares our proof style with tactic proof, and summarizes related issues such as automated reasoning and the IDE.

Space does not permit extensive case studies to be presented here. However, DECLARE has been applied to a formalization of the semantics of a subset of the Java language and a proof of type soundness for this subset [Sym99]. The purpose of DECLARE is to explore mechanisms of specification, proof and interaction that may eventually be incorporated into other theorem proving systems, and thus complement them.

1.1 Background

This work was inspired by similar languages developed by the Mizar group [Rud92] and Harrison [Har96]. Mizar is a system for formalizing general mathematics, designed and used by mathematicians, and a phenomenal amount of the mathematical corpus has been formalized in this system. The foundation is set theory, which pervades the system, and proofs are expressed using detailed outlines, leaving the machine to fill in the gaps. Once the concrete syntax is stripped away, steps in Mizar proofs are mostly applications of simple deduction rules, e.g. generalization, instantiation, and propositional introduction and elimination.[1] Essentially our work has been to transfer a key Mizar idea (proof outlining) to the setting of higher order logic theorem proving, use extensive automation to increase the size of proof steps, generalize the notion of an outlining construct in a natural way, refine the system based on some large case studies and explore the related issues of specification, automation, interaction. This has led to the three outlining constructs described in this paper.

Some of the other systems that have most influenced our work are HOL [GM93], Isabelle [Pau94], PVS [COR$^+$95], and Nqthm [KM96]. Many of the specification and automation techniques we utilize in DECLARE are derived from ideas found in the above systems. However, we do not use the proof description techniques from these systems (e.g. the HOL tactic language, or PVS strategies).

1.2 Declarative and Inferential Proof Description

For our purposes, we consider a construct to be *declarative* if it states explicitly "what" effect is achieved by the construct. Different declarations may specify different properties of a construct, e.g. type, mode and behavioral specifications in a programming language. A related question is whether the construct describes

[1] Mizar is a poorly documented system, so our observations are based on some sample Mizar scripts and the execution of the Mizar program.

"how" that effect is achieved: we will use *inferential* to describe systems that allow the omission of such details and infer them instead. Many systems are both declarative and inferential, and together they represent an ideal, where a problem statement is given in high level terms and a machine is used to infer a solution. "Inferential" is inevitably a relative notion: one system is more inferential than another if the user need specify fewer operational details. The term *procedural* is often used to describe systems that are not highly inferential, and thus typically not declarative either, i.e. systems where significant detail is needed to express solutions, and a declarative problem statement is not given.[2]

How do "declarative" and "inferential" apply to proof description? For our purposes a declarative style of proof description is one that makes the logical results of a proof step explicit:

> A proof description style is declarative if the results established by a reasoning step are evident without interpreting the justification given for those results.

Surprisingly, most existing styles of proof description are plainly *not* declarative. For example, typical HOL tactic proofs are certainly not declarative, although automation may allow them to be highly inferential. Consider the following extract from the proof of a lemma taken from Norrish's HOL formalization of the semantics of C [Nor98]:

```
val wf_type_offset = prove
  ``∀smap sn. well_formed_type smap (Struct sn) →
             ∀fld t. lookup_field_info (smap sn) fld t →
                     ∃n. offset smap sn fld n``,
  SIMP_TAC (hol_ss ++ impnorm_set) [offset,
    definition "choltype" "lookup_field_info",
    definition "choltype" "struct_info"] THEN
  REPEAT STRIP_TAC THEN
  IMP_RES_TAC (theorem "choltype" "well_formed_structs") THEN
  FULL_SIMP_TAC hol_ss [well_formed_type_THM] THEN
  FIRST_X_ASSUM SUBST_ALL_TAC THEN
  ...
```

Even given all the appropriate definitions, we would challenge an experienced HOL user to accurately predict the shape of the sequent late in the proof.

In an ideal world we would also like a fully inferential system, i.e. we simply state a property and the machine proves it automatically. For complex properties this is impossible, so we try to decrease the amount of information required to specify a proof. One very helpful way of estimating the amount of information contained in a proof is by looking at the dependencies inherent in it:

> One proof description style is more inferential than another if it reduces the number of dependencies inherent in the justifications for proof steps.

To give a simple concrete example, proofs in interactive theorem provers (e.g. HOL, PVS and Isabelle) are typically sensitive to the order in which subgoals

[2] Declarative and inferential ideas are, of course, very common in computing, e.g. Prolog and LaTeX are examples of languages that aspire to be both declarative and inferential.

are produced by an induction utility. That is, if the $\mathbb{N}$-induction utility suddenly produced the step case before the base case, then most proofs would break. There are many similar examples from existing theorem proving system, enough that proofs in these systems can be extremely fragile, or reliant on a lot of hidden, assumed detail. A major aim of proof description and applied automated reasoning must be to eliminate such dependencies where possible. Other examples of such dependencies include: reliance on the orderings of cases, variables, facts, goals and subgoals; reliance upon one of a number of logically equivalent forms of terms (e.g. $n > 1$ versus $n \geq 2$); and reliance on the under-specified behavior of proof procedures, such as how names are chosen.

2 Three Constructs for Proof Outlining

Proofs in Declare are expressed as outlines, in a language that approximates written mathematics. The constructs themselves are not radical, but our assertion is that most proof outlines can be written in these constructs and their syntactic variants alone. In other words, we assert that for many purposes these constructs are both logically and pragmatically adequate. Perhaps the most surprising thing is that large proof developments can indeed be performed in Declare even though proofs are described in a relatively restricted language.

In this section we shall describe the three primary constructs of the Declare proof language. These are:

- First order decomposition and enrichment;
- Proof by automation;
- Application of second order proof principles.

2.1 Reduced Syntax

A simplified syntax of the proof language is shown below, to demonstrate how small it is at its core. We have omitted aspects that are not relevant for the purposes of this article, including specification constructs for introducing new types and constants with various properties. These include simple definitions, mutually recursive function definitions, mutually recursive fixed point specifications and algebraic datatypes. Several syntactic variations of the proof constructs are translated to the given syntax, as discussed in later sections. Declare naturally performs name resolution and type inference, the latter occurring "in context", taking into account all declarations that are visible where a term occurs. Declare also performs some syntactic reduction and comparison of terms during proof analysis, as described in Section 2.6. We have left some constructs of the language uninterpreted, in particular the language of justifications, which is discussed later in this article. Declarations also include "pragma" specifications that help indicate what various theorems mean and how they may be used by the automated reasoning engine. Finally, the terms and types are those of higher order logic as in the HOL system, extended with pattern matching as described in [Sym99].

```
Article = Decl*
Decl = thm Label "term" proof Proof end
     | ...                (other specification language constructs)
Proof = qed Justification
      | cases Justification Case* end
      | schema Label over Label
             varying Local*
          Case*
Justification = by Hint*
Case = case [Label] Enrichment* : Proof
Enrichment = [locals Local*] Fact*
Local = ident [: type]
Fact =  "term" [Label]
Label = <ident>
```

We consider a semantics describing proof checking for this language in Appendix A.

2.2 An Example

We will use a Declare proof of the following proposition to illustrate the first two constructs of the proof language: "Assume $n \in \mathbb{N}$ is even, and that whenever m is odd, n/m is even, ignoring any remainder. Then the remainder of n/m is always even." We assume that our automated reasoner is powerful enough to do some arithmetic normalization for us, e.g. collecting linear terms and distributing "mod" over + (this is done by rewriting against Declare's standard library of theorems and its built-in arithmetic procedures). We also assume the existence of the following theorems about even, odd and modulo arithmetic.

```
<even> |- even(n) = ∃k. n=2*k
<odd>  |- odd(n) = ∃k. n=2*k+1
<even_or_odd>  |- even(n) ∨ odd(n)
<div_rem_exists>  |- m > 0 → (∃d r. n=d*m+r ∧ r<m)
```

A Declare proof of this property is shown below. The constructs used and their meanings are explained in the following sections.

```
thm <mythm>
  if "m > 0"
     "odd(m) → even(n/m)"  <m>
     "even(n)" <n>
  then "even(n mod m)" <goal>;
proof
  consider d, r st
     "n = d*m + r"
     "r < m"  by <div_rem_exists>;

  have "d = n/m"
       "r = n mod m";
  consider n' st
     "n = 2*n'" by <even>,<n>;

  cases by <even_or_odd> ["m"]
    case "even(m)" :
      consider m' st "m=2*m'" by <even>;
      have "r = 2*(n'-d*m')";
      qed by <even>,<goal>;

    case "odd(m)" :
      consider d' st "r = 2*(n'-d'*m)" by <m>,<even>;
      qed by <even>,<goal>;
  end;
end;
```

2.3 Problem Introduction

A Declare proof begins with the statement of a problem, introduced using some variant of the `thm` declaration. The example from the previous section uses the one shown in Table 1. This variant allows us to begin our proof in a conveniently decomposed form, i.e. without outer universal quantifiers and with facts and goals already named.

External Form	Internal Form
`thm` *label* `if` *facts* `then` *goals*	`thm` *label* "$\forall vars.\ (\bigwedge facts) \rightarrow (\bigvee goals)$"
`proof`	`proof`
main-proof	`cases by <goal>`
`end`	`case locals` *vs*
	facts
(simplified problem introduction)	$goals^{-1}$:
	main-proof
	`end`
	`end`
	where *vars* = free symbols in *facts*,*goals*
	and $goals^{-1}$ = *goals* with each term negated

Table 1. Syntactic variation for *Decl* with the equivalent primitive form.

2.4 Construct 1: First Order Decomposition and Enrichment

Enrichment is the process of adding facts, goals and local constants to an environment in a logically sound fashion. Most steps in vernacular proofs are enrichment steps, e.g. "consider d and r such that $n = d * m + r$ and $r < m$." The example above illustrates how this translates into Declare's syntax. An enrichment step has a corresponding proof obligation that constants exist with the given properties. The obligation for this step is "$\exists d\ r.$ `n=`d`*m+`r $\wedge$ r `< m`".

This kind of enrichment is *forward reasoning*. When goals are treated as negated facts, *backward reasoning* also corresponds to enrichment. For example if our goal is $\forall x.(\exists b. x = 4b) \rightarrow \mathsf{even}(x)$ then the vernacular "given b and x such

External Form	Internal Form

`consider` *vars* `st` *facts justification*;
main-proof
 	
(inline introduction)
	`cases` *justification*
	  `case locals` *vars*
	      facts `:`
	    main-proof
	`end`

`have` *facts justification*;
main-proof
 	
(inline assertion)
	`cases` *justification*
	  `case` *facts* `:`
	    main-proof
	`end`

`let` *id* `= "`*term*`";`
main-proof
 	
(inline definition)
	`cases`
	  `case locals` *id*
	      `"`*id* `=` *term*`" :`
	    main-proof
	`end;`

`sts` *goal justification*;
main-proof
 	
(inline backward reasoning)
	`cases` *justification*
	  `case` $goal^{-1}$ `:` *main-proof*
	`end;`
	
	where $goal^{-1}$ = *goal* with the term negated

Table 2. Syntactic variations for *Proof* with equivalent primitive forms.

that $x = 4b$ then by the definition of even it suffices to show $\exists c.2 * c = x$" is an enrichment step (this example is not taken from the larger example above). Based on an existing goal, we add two new local constants b and x, a new goal $\exists c.2 * c = x$ and a new fact $x = 4b$. In DECLARE we can use +/- to indicate new facts/goals respectively (goals are treated as negated facts), and we have:

```
consider b,x such that
   + "x = 4*b"
   - "∃c. 2*c = x"
  by <goal>;        // obligation "∃b x. x=4*b ∧ ∄c. 2*c=x"
```

Decomposition is the process of splitting a proof into several cases. We combine decomposition and enrichment via the `cases` construct, and an example can be seen in Section 2.2. For each decomposition/enrichment there is a proof obligation that corresponds to the "default" case of the split, where we may assume each other case does not apply. Syntactically, the `locals` declaration for each enrichment can be omitted, as new symbols are assumed to be new local constants. The construct is very general, and some highly useful variants are translated to it as shown in Table 2, including assertion, abbreviation, and the linear forms of enrichment seen above. These forms assume the automated prover can, as a minimum, decide the trivial forms of first order equational problems that arise as proof obligations in the translations. For example, $\exists v.v = t$ is the proof obligation for the `let` construct, where v is not free in t.

General specification constructs could also be admitted within enrichments, e.g. to define local constants by fixed points. DECLARE does not implement these within proofs.

2.5 Construct 2: Appeals to Automation

At the tips of a problem decomposition we find appeals to automated reasoning to "fill in the gaps" of an argument, denoted by `qed` in the proof language. A set of “hints” (also called a *justification*) is provided to the engine. We shall discuss the justification language of hints in the Section 3. The automated reasoning engine is treated as an oracle, though of course the intention is that it is sound with respect to the axioms of higher order logic.

2.6 Construct 3: Second Order Schema Application

In principle, decomposition/enriching and automated proof with justifications are sufficient to describe any proof in higher order logic, assuming a modicum of power from the automated engine (e.g. that it implements the 8 primitive rules of higher order logic described by Gordon and Melham [GM93], and can decide propositional logic). However, we have found it useful to add one further construct for inductive arguments. The general form we have adopted is *second-order schema application*, which can encompass structural, rule and well-founded induction and other techniques.

Why is this construct needed? We consider a typical proposition proved by inducting over the structure of a particular set. Assume *typ list* ⊢ *exp* `hastype` *typ* is an inductive relation defined by the four rules over a term structure as shown in Appendix B. Our example theorem states that substitution of well-typed values preserves types (we omit the definition of substitution):

```
thm <subst_safe>
if "[] ⊢ v hastype xty"  <v_hastype>
   "len(E) = n"        <n>
   "(E@[xty]) ⊢ e hastype ty" <typing>
then "E ⊢ (subst n e v) hastype ty";
```

The induction predicate that we desire is:

```
P = λE e ty. ∀n. len E = n → E ⊢ (subst n e v) hastype ty
```

One of our aims is to provide a mechanism to specify the induction predicate in a natural way. Note it is essential that n be universally quantified, because it is necessary to instantiate it with different values in different cases of the induction. Likewise E, e and *ty* also “vary”. Furthermore, because v and *xty* do not vary, it is better to leave `<v_hastype>` out of the induction predicate to avoid extra antecedents to the induction hypothesis.

It is possible to use decomposition along with an explicit instantiation to express an inductive decomposition.

```
thm <subst_safe>
if "[] ⊢ v hastype xty"  <v_hastype>
then "∀E e ty.
        (E @ [xty]) ⊢ e hastype ty ∧
        len E = n →
          E ⊢ (subst n e v) hastype ty" <goal>
proof
  let "ihyp E e ty =
        ∀n. len E = n → E ⊢ (subst n e v) hastype ty";
  cases by <hastype.induct> ["ihyp"], <goal>
    case + "ihyp ([dty]@(E@[xty])) bod rty" <ihyp>
         - "ihyp E (Lam dty bod) (FUN dty rty)" :
     ...
    case + "e = App f a"
         + "ihyp (E@[xty]) f (FUN dty ty)" <ihyp1>
         + "ihyp (E@[xty]) a dty" <ihyp2> :
         + "len E = n"
         - "E ⊢ (subst n e v) hastype ty" :
     ...
  end;
end;
```

The induction theorem has been *explicitly instantiated*, a mechanism available in the language of justifications discussed in Section 3. Two trivial cases of the proof have been subsumed in the decomposition itself (see Section 2.8 — the cases in question correspond to the rules `Int` and `Var` in Appendix B). For the other two cases we have listed the available induction hypotheses explicitly, at two different depths of expansion — in the second case we have revealed more of the structure of the goal.

This approach is sometimes acceptable. Its advantages include flexibility, because simple cases may be omitted altogether; control, because we name the facts and constants introduced on each branch of the induction; and explicitness, which can be helpful for readability and the tool environment. Its disadvantages are an unnatural formulation of the original problem; the unnecessary repetition of induction hypotheses; a relatively complex proof obligation; and poor feedback because it is non-trivial to provide good feedback if the user makes a mistake when recording the hypotheses.

We now show how the proof appears using the `schema` construct of the Declare proof language.

```
thm <subst_safe>
if "[] ⊢ v hastype xty"
   "len(E) = n"
    "(E@[xty]) ⊢ e hastype ty" <typing>
then "E ⊢ (subst n e v) hastype ty";
proof
  schema <hastype.induct> over <typing> varying n,E,ty,e
    case <Int>: ...
    case <Var>: ...
    case <Lam>
      "e = Lam dty bod"
      "ty = FUN dty rty"
      "ihyp ([dty]@(E@[xty])) bod rty"  <ihyp> :
      ...
    case <App>
      "e = App f a"
      "ihyp (E@[xty]) f (FUN dty ty)" <f_ihyp>
      "ihyp (E@[xty]) a dty"          <a_ihyp> :
```

```
      ...
  end;
end;
```

Actually, a little simpler is the `induct` variant of the `schema` construct, which chooses a default induction principle based on the predicate used to define the inductive set. The first line of the proof could have been written:

```
  induct over <typing> varying n,E,ty,e
```

Thus DECLARE provides one very general construct for decomposing problems along syntactic lines based on a second-order proof principle, along with some simple variants. The induction predicate is determined automatically by indicating those local constants V that "vary" during the induction. Effectively we tell DECLARE to reformulate the problem so some local "constants" become universally quantified, and then apply the induction principle. The induction hypothesis is thus the conjunction of all the axioms in the current logical context that contain a member of V. This gives a declarative specification of the induction predicate without contorting the initial specification of the problem.

The schema must be a fact in the logical environment of the form:

$$(\forall \bar{v}_1 .\, ihyps_1 \rightarrow P \bar{v}_1) \;\wedge\; \ldots \;\wedge\; (\forall \bar{v}_n .\, ihyps_n \rightarrow P \bar{v}_n) \rightarrow (\forall \bar{v}.\, R[\bar{v}] \rightarrow P \bar{v})$$

Equational constraints are encoded in the induction hypotheses, and the fact denoted using `over` must be an instance $R[\bar{t}]$ of $R[\bar{v}]$ for some $\bar{t}$. If $\mathbb{N}$ is an inductive subset of a type for $\mathbb{Z}$, then the schema would be:

$$(\forall i.\ i\texttt{=0} \rightarrow P\ i) \wedge (\forall i.\ (\exists k.\ i\texttt{=}k\texttt{+1} \wedge P\ k) \rightarrow P\ i) \rightarrow (\forall i.\ i \in \mathbb{N} \rightarrow P\ i)$$

The `induct` form where the schema is implicit from a term or fact is most common, however the general mechanism above allows the user to prove and use new induction principles for constructs that were not explicitly defined inductively, and allows several proof principles to be declared for the same logical construct.

Each antecedent of the inductive schema generates one new branch of the proof, so no subsumption is possible. For each case:

– If no facts are given, then the actual hypotheses (i.e. those specified in the schema) are left implicit: they become "automatic" unlabelled facts used by the automated prover.
– If facts are given, they are interpreted as "purported hypotheses" and syntactically checked to ensure they correspond to the actual hypotheses (see [Sym99] for details).

The semantics of the construct are described in full in Appendix A.

2.7 Issues Relating to Second Order Schema Application

Writing out the induction predicate is time-consuming and error-prone. The macro `ihyp` can be used to stand for the induction predicate — the user does not have to define this predicate explicitly.

It is often necessary to strengthen a goal or weaken some assumptions before using induction. This can often be done simply by stating the original goal in this way, but in a nested proof we typically prove that the stronger goal is sufficient (this is usually trivial), and before we perform an induction we purge the environment of the now irrelevant original goal, to avoid unnecessary conditions being included in the induction predicate. This means adding a "discarding" construct to the proof language. Discarding facts breaks the monotonicity proof language, so to minimize its use we have chosen to make it part of the induction construct. Our case studies suggest it is only required when significant reasoning is performed before the induction step of a proof, which is rare.

A final twist on the whole proof language that comes when describing mutually recursive inductive proofs is described in [Sym99]. Essentially we need to modify the language to accommodate multiple (conjoined) goals, if the style of the proof language is to be preserved.

2.8 A Longer Example of Decomposition/Enrichment

We now look at a longer example of the use of enrichment/decomposition, to demonstrate the flexibility of this construct. The example is similar to several that arose in our case studies, but has been modified to demonstrate several points. Assume:

- The inductive relation $c \rightsquigarrow c'$ is defined by many rules (say 40).
- c takes a particular form $(\mathtt{A}(a,b),s)$ at the current point in our proof.
- Only 8 of the rules apply when c is of this form, and of these, 5 represent "exceptional transitions" $c \rightsquigarrow (\mathtt{E}(\mathit{val}), s)$. The last 3 possible transitions are given by:

$$\frac{(a,s) \rightsquigarrow (v,s') \ \vee \ (b,s) \rightsquigarrow (v,s')}{(\mathtt{A}(a,b),s) \rightsquigarrow (v,s')} \qquad \frac{}{(\mathtt{A}(a,b),s) \rightsquigarrow (a,s)} \qquad \frac{}{(\mathtt{A}(a,b),s) \rightsquigarrow (b,s)}$$

We are trying to prove that the predicate cfg_ok is an invariant of $\rightsquigarrow$:

```
type exp = A of exp * exp | E of string
thm <cfg_ok> "cfg_ok (t,s) ↔ match t with
                                 A(x,y) -> term_ok(s,x) ∧ state_ok(s)
                               | E(str) -> state_ok(s)";
thm <cfg_ok-invariant>
if  "c ⇝ c'" <trans>
    "c = (A(a,b),s)"
    "cfg_ok c"
then "cfg_ok c'";
```

Note the proof will be trivial in the case of the exceptional transitions, since the state is unchanged. So, how do we formulate the case analysis? Do we have to write all 40 cases? Or even all 8 which apply syntactically? No - we need specify only the interesting cases, and let the automated reasoner deduce that the other cases are trivial:

```
cases by <⇝.cases> [<trans>], <cfg_ok>, <goal>
  case "c' = (v, s')"
       "t = a ∨ t = b"
       "(t,s) ---> c'" :
    rest of proof;
  case "c' = (t, s)"
       "t = a ∨ t = b" :
    rest of proof;
end;
```

The hints given to the automated reasoner are explained further in Section 3. The key point is that the structure of the decomposition does *not* have to match the structure inherent in the theorems used to justify it (i.e. the structure of the rules). There must, of course, be a logical match that can be discovered by the automated engine, but the user is given a substantial amount of flexibility in how the cases are arranged. He/she can:

– *Subsume trivial cases.* 37 of the 40 cases inherent in the definition of $\leadsto$ can be subsumed in justification of the split.
– *Maintain disjunctive cases.* Many interactive splitting tools would have generated two cases for the first rule shown above, by automatically splitting the disjunct. However, the proof may be basically identical for these cases, up to the choice of t.
– *Subsume similar cases.* Structurally similar cases may be subsumed into one branch of the proof by using disjuncts, as in the second case. This is, in a sense, a form of factorization. As in arithmetic, helpful factorizations are hard for the machine to predict, but relatively easy to check.

The user can use such techniques to split the proof into chunks that are of approximately equal difficulty, or to dispose of many trivial lines of reasoning, much as in written mathematics.

3 Justifications and Automated Reasoning

Our language separates *proof outlining* from *automated reasoning.* We adopt the principle that these are separate activities and that the proof outline should be independent of complicated routines such as simplification. The link between the two is provided by *justifications.* A spectrum of justification languages is possible. For example, we might have no language at all, which would assume the automated engine can draw useful logical conclusions efficiently when given nothing but the entire logical environment. Alternatively we might have a language that spells out deductions in great detail, e.g. the forward inference rules of an LCF-like theorem prover. It may also be useful to have domain specific constructs, such as variable orderings for model checking.

DECLARE provides a small set of general justification constructs that were adequate for our case studies. The constructs allow the user to:

– Highlight facts from the logical environment that are particularly relevant to the justification;
– Specify explicit instantiations and resolutions;
– Specify explicit case-splits;

These constructs are quite declarative and correspond to constructs found in vernacular proofs. Facts are *highlighted* in two ways:

– By quoting their label
– By never giving them a label in the first place, as unlabelled facts are treated as if they were highlighted in every subsequent proof step.

The exact interpretation of highlighting is determined by the automated engine, but the general idea is that highlighted facts must be used by the automated engine for the purposes of rewriting, decision procedures, first order search and so on.

"Difficult" proofs often become tractable by automation if a few *explicit instantiations* of first order theorems are given. Furthermore, this is an essential debugging technique when problems are not immediately solvable: providing instantiations usually simplifies the feedback provided by the automated reasoning engine. In a declarative proof language the instantiations are usually easy to write, because terms are parsed in-context and convenient abbreviations are often available. Formal parameters of the instantiations can be either type directed of explicitly named, and instantiations can be given in any order. For example, consider the theorem `<subst_safe>` from Section 2.6. When using this theorem a suitable instantiation directive may be:

```
  qed by <subst_safe> ["[]", "0", "xty"/xty];
```

We have one named and two type-directed instantiations. After processing the named instantiation five instantiable slots remain: e,v,E,n and ty. Types give the instantiations $E \to$ `[]` and $n \to$ `0` and the final fact:

$\vdash \forall e\ v\ ty.$ `[]` $\vdash v$ `hastype xty` $\wedge$ `len [] = 0` $\wedge$ `([]@[xty])` $\vdash e$ `hastype` ty
$\quad\quad \to$ `[]` $\vdash$ `(subst 0` $e\ v$`) hastype` ty

Explicit resolution is a mechanism similar in spirit to explicit instantiation. It combines instantiation and resolution and allows a fact to eliminate an appropriate unifying instance of a literal of opposite polarity in another fact. We might have:

```
  have "[] ⊢ e2 hastype xty" <e2_types> by ...
  qed by <subst_safe> ["0", <e2_types>];
```

The justification on the last line gives rise to the hint:

$\vdash \forall e\ v\ ty.$ `true` $\wedge$ `len [] = 0` $\wedge$ `([]@[xty])` $\vdash e$ `hastype` ty
$\quad\quad \to$ `[]` $\vdash$ `(subst 0` $e\ v$`) hastype` ty

DECLARE checks that there is only one possible resolutions. One problem with this mechanism is that, as it stands in DECLARE, unification takes no account of ground equations available in the logical context, and thus some resolutions do not succeed where we would expect them to.

Explicit case splits can be provided by *instantiating a disjunctive fact*, *rule case analysis*, or *structural case analysis*. Rule case analysis accepts a fact indicating membership of an inductive relation, and generates a fact that specifies the possible rules that might have been used to derive this fact. Structural case analysis acts on a term belonging to a free algebra (i.e. any type with an abstract datatype axiom): we generate a disjunctive fact corresponding to case analysis on the construction of the term.

4 Assessment

We now look at the properties of the proof language we have described and compare it with other methods of proof description. The language is essentially based on *decomposing* and *enriching* the logical environment. This means the environment is *monotonically increasing* along any particular branch of the proof That is, once a fact becomes available, it remains available.[3] The user manages the environment by labelling facts and goals, and by specifying meaningful names for local constants. This allows coherent reasoning within a complicated logical context.

Mechanisms for brevity are essential within declarative proofs, since a relatively large number of terms must be quoted. DECLARE attempts to provide mechanisms so that the user need never quote a particular term more than once with a proof. For example one difficulty is when a formula must be quoted in both a positive and a negative sense (e.g. as both a fact and an antecedent to a fact): this happens with induction hypotheses, and thus we introduced `ihyp` macros. Other mechanisms include local definitions; type checking in context; and stating problems in sequent form.

When using the proof language, the user often declares an enrichment or decomposition, giving the logical state he/she wants to reach, and only states "how to get there" in high level terms. The user does not specify the syntactic manipulations required to get there, except for some hints provided in the justification, via mechanisms we have tried to make as declarative as possible. Often the justification is simply a set of theorem names.

4.1 Comparison

Existing theorem provers with strong automation, such as Boyer-Moore [RJ79], effectively support a kind of declarative/inferential proof at the top level — the user conjectures a goal and the system tries to prove it. If the system fails, then the user adds more details to the justification and tries again. DECLARE extends this approach to allow declarative decompositions and lemmas in the internals of a proof, thus giving the benefits of scope and locality.

[3] There is an exception to this rule: see Section 2.6

One traditional form of proof description uses "tactics" [MRC79]. In principle tactics simply decompose a problem in a logically sound fashion. In practice tactic collections embody an interactive style of proof that proceeds by syntactic manipulation of the sequent and existing top level theorems. The user issues proof commands like "simplify the current goal", "do induction on the first universally quantified variable" or "do a case split on the second disjunctive formula in the assumptions". A major disadvantage is that the sequent quickly becomes unwieldy, and the style discourages the use of abbreviations and complex case decompositions. A potential advantage of tactics is programmability, but in reality user-defined tactics are often examples of arcane *adhoc* programming in the extreme.

Finally, many declarative systems allow access to a procedural level when necessary. One might certainly allow this in a declarative theorem proving system, e.g. via an LCF-like programmable interface. It would be good to avoid extending the proof language itself, but one could imagine adding new plug-in procedures to the automated reasoning engine and justification language via such techniques.

4.2 Pros and Cons

Some benefits of a declarative approach are:

- *Simplicity.* Proofs are described using only a small number of simple constructs, and the obligations can be generated without knowing the behavior of a large number of tactics.
- *Readability.* The declarative outline allows readers to skip sections they aren't interested in, but still understand what is achieved in those sections.
- *Re-usability.* Declarative content can often be re-used in a similar setting, e.g. the same proof decomposition structure can, in principle, be used with many different automated reasoning engines.
- *Tool Support.* An explicit characterization of the logical effect of a construct can often be exploited by tools, e.g. for error recovery and the interactive debugging environment. See [Sym98] for a description of an interactive environment for DECLARE, and a detailed explanation of how a declarative proof style allows proof navigation, potentially leading to more efficient proof debugging.

A declarative outline does not, of course, come for free. In particular, new facts must be stated explicitly. Procedurally, one might describe the syntactic manipulations (modus-ponens, specialization etc.) that lead to the production of those facts as theorems, and this may be more succinct in some cases. This is the primary drawback of declarative proof.

DECLARE proofs are not only declarative, but also highly inferential, as the automated prover is left to prove many obligations that arise. The benefits of a highly inferential system are also clear: brevity, readability, re-usability and robustness. The cost associated with an inferential system is, of course, that the computer must work out all the details that have been omitted, e.g. the syntactic manipulations required to justify a step deductively. This is costly both in terms of machine time and the complexity of the implementation.

Proofs in DECLARE are relatively independent of a number of factors that are traditional sources of dependency in tactic proofs. For example, Isabelle, HOL and PVS proofs frequently contain references to assumption or subgoal numbers, i.e. indexes into lists of each. The proofs are sensitive to many changes in problem specification where corresponding DECLARE proofs will not be. In DECLARE such changes will alter the proof obligations generated, but often the obligations will still be discharged by the same justifications.

To summarize, declarative theorem proving is about making the logical effect of proof steps explicit. Inferential theorem proving is about strong automated reasoning and simple justifications. These do not come for free, but in the balance we aim to achieve benefits that can only arise from a declarative/inferential approach.

References

[COR+95] Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, and Mandayam Srivas. A tutorial introduction to PVS. In *Proceedings of the Workshop on Industrial-Strength Formal Specification Techniques*, Baco Raton, Florida, 1995.

[GM93] M.J.C Gordon and T.F Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.

[Har96] J. Harrison. A Mizar Mode for HOL. In J. Von Wright, J. Grundy, and J. Harrison, editors, *Ninth international Conference on Theorem Proving in Higher Order Logics TPHOL*, volume 1125 of *Lecture Notes in Computer Science*, pages 203–220, Turku, Finland, August 1996. Springer Verlag.

[KM96] Matt Kaufmann and J. Strother Moore. ACL2: An industrial strength version of Nqthm. *COMPASS — Proceedings of the Annual Conference on Computer Assurance*, pages 23–34, 1996. IEEE catalog number 96CH35960.

[MRC79] M.J.C. Gordon, R. Milner, and C.P. Wadsworth. *A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science.* Springer-Verlag, New York, 1979.

[Nor98] Michael Norrish. *C Formalized in HOL.* PhD thesis, University of Cambridge, August 1998.

[Pau94] L.C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science.* Springer Verlag, 1994.

[RJ79] R.S. Boyer and J.S. Moore. *A Computational Logic Handbook.* Academic Press, 1979.

[Rud92] P. Rudnicki. An overview of the MIZAR project, 1992. Unpublished; available by anonymous FTP from `menaik.cs.ualberta.ca` as `pub/Mizar/Mizar_Over.tar.Z`.

[Sym98] Don Syme. Interaction for Declarative Theorem Proving, December 1998. Available from `http://research.microsoft.com/users/dsyme`.

[Sym99] Don Syme. *Declarative Theorem Proving for Operational Semantics.* PhD thesis, University of Cambridge, Computer Laboratory, January 1999. Available from `http://research.microsoft.com/users/dsyme`.

A A Semantics

A *logical environment* or *theory* Γ contains:

- A signature of type and term constants;
- A set of axioms, each of which are closed higher order logic terms.

Logical environments must always be wellformed: i.e. all their terms must typecheck with respect to their signature. Axioms are named ($label \mapsto prop$). We can add ($\oplus$) a fragment of a logical environment to another environment. These fragments specify new types, constants and axioms. We assume the existence of a logical environment Γ_0 containing the theory of all standard propositional and first order connectives, and other axioms of higher order logic.

The judgment $Decls \vdash \Gamma$ indicates that the given declarations establish Γ as a conservative extension of a minimal theory of higher order logic. The judgment $\Gamma \vdash Decl : \Gamma_{frag}$ is used to elaborate single declarations.

$$\frac{}{[] \vdash \Gamma_0} \qquad \frac{\begin{array}{c} Decls \vdash \Gamma \\ \Gamma \vdash Decl : \Gamma_{frag} \end{array}}{Decls; Decl \vdash \Gamma \oplus \Gamma_{frag}} \qquad \frac{\begin{array}{c} prop \text{ is a closed term of type } bool \\ \Gamma \oplus (\texttt{"goal"} \mapsto \neg prop) \vdash proof\checkmark \end{array}}{\Gamma \vdash \texttt{thm <}lab\texttt{>}\ prop\ \ proof : (lab \mapsto prop)}$$

The label "goal" is used to represent the obligation: in reality problems are specified with a derived construct in decomposed form, so this label is not used.

The relation $\Gamma \vdash proof\checkmark$ indicates that the proof establishes a contradiction from the information in Γ, as given by the three rules below. However first some definitions are required:

- *Enriching an environment.*

$$\begin{array}{c}\Gamma \oplus (\texttt{locals}\ c_1 \dots c_i;\ fact_1\ \texttt{<}lab_1\texttt{>} \dots fact_j\ \texttt{<}lab_j\texttt{>}) = \\ \Gamma \oplus c_1 \dots c_i \oplus (lab_1 \mapsto fact_1), \dots, (lab_j \mapsto fact_j)\end{array}$$

  There may be no free variables in $fact_1 \dots fact_j$ besides $c_1 \dots c_i$.
- *The obligation for an enrichment to be valid.*

$$\mathsf{oblig}(\texttt{locals}\ c_1 \dots c_i;\ fact_1\ \texttt{<}lab_1\texttt{>} \dots fact_j\ \texttt{<}lab_j\texttt{>}) = \exists c_1 \dots c_i.\ fact_1\ \wedge\ \dots\ \wedge\ fact_j$$

  In the r.h.s, each use of a symbol c_i becomes a variable bound by the $\exists$ quantification.
- *Discarding.* $\Gamma - labels = \Gamma$ without axioms specified by *labels*
- *Factorizing.* $\Gamma/V =$ the conjunction of all axioms in Γ involving any of the locals specified in V. When the construct is used below, each use of a local in V becomes a variable bound by the $\forall$ quantification that encompasses the resulting formula.

Decomposition/Enrichment

$$\frac{\begin{array}{l} proof = \texttt{cases}\ proof_0 \\ \qquad\qquad \texttt{case}\ lab_1\ enrich_1\ proof_1 \\ \qquad\qquad \dots \\ \qquad\qquad \texttt{case}\ lab_n\ enrich_n\ proof_n \\ \qquad\quad \texttt{end} \\ \Gamma \oplus (lab_1 \mapsto \neg\mathsf{oblig}(enrich_1)), \dots, (lab_n \mapsto \neg\mathsf{oblig}(enrich_n) \vdash proof_0\checkmark \\ \forall i < n.\ \Gamma \oplus enrich_i \vdash proof_i\checkmark \end{array}}{\Gamma \vdash proof\checkmark}$$

Automation

$$\frac{\mathsf{prover}(\Gamma, \mathit{hints}(\Gamma)) \text{ returns ``yes''}}{\Gamma \vdash \texttt{qed by } \mathit{hints} \checkmark}$$

Schemas

```
proof = schema schema-label over fact-label
            varying V discarding discards
          case lab_1 enrich_1 : proof_1
          ...
          case lab_n enrich_n : proof_n
        end
```

$\Gamma' = \Gamma - \mathit{discards}$

$\Gamma'(\mathit{schema\text{-}label}) = \forall P. (\forall \bar{v}.\mathit{ihyps}_1 \to P(\bar{v}))$
$\quad \ldots$
$\quad (\forall \bar{v}.\mathit{ihyps}_n \to P(\bar{v}))$
$\quad \to (\forall \bar{v}.Q(\bar{v}) \to P(\bar{v}))$

$\Gamma'(\mathit{fact\text{-}label}) = Q(\bar{t})$

$\mathit{ipred} = \text{``}\lambda\bar{v}.\forall V.(\bigwedge(\bar{v} = \bar{t})) \to \Gamma'/V\text{''}$

static matching determines that

$\quad \forall \bar{v}. \bigwedge(\bar{v} = \bar{t}) \;\wedge\; \mathit{ihyps}_i[\mathit{ipred}/P] \to \mathsf{oblig}(\mathit{enrich}_i) \quad (\forall i.1 \le i \le n))$

$$\frac{\Gamma' \oplus \mathit{enrich}_i \vdash \mathit{proof}_i \checkmark \qquad (\forall 1 \le i \le n)}{\Gamma \vdash \mathit{proof} \checkmark}$$

The conditions specify that:

- The proof being considered is a second-order schema application of some form;
- The given axioms are discarded from the environment (to simplify the induction predicate);
- *schema-label* specifies a schema in the current logical context of the correct form;
- *fact-label* specifies an instance of the inductive relation specified in the schema for some terms $\bar{t}$. These terms may involve both locals in V and other constants.;
- The induction predicate is that part of the logical environment specified by the variance. If the terms $\bar{t}$ involve locals in the variance V then they become bound variables in this formula.
- Matching: the generated hypotheses must imply the purported hypotheses.
- Each sub-proof must check correctly.

B Typing Rules for the Induction Example

```
<Int>
          -----------------------------
            "E ⊢ (Int i) hastype INT"

<Var>       "i < len(E) ∧ ty = el(i)(E)"
          --------------------------------
            "E ⊢ (Var i) hastype ty"

<Lam>       "[dty]@E ⊢ bod hastype rty"
        ---------------------------------------------
        "E ⊢ (Lam dty bod) hastype (FUN dty rty)"

<App>    "E ⊢ f hastype (FUN dty rty) ∧ E ⊢ a hastype dty"
        --------------------------------------------------------
                 "E ⊢ (f % a) hastype rty";
```