Technical Report

UCAM-CL-TR-416
ISSN 1476-2986

Number 416

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

DECLARE: a prototype declarative proof system for higher order logic

Donald Syme

February 1997

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

DECLARE: A Prototype Declarative Proof System for Higher Order Logic

Donald Syme

February 6, 1997

Abstract

This report describes DECLARE, a prototype implementation of a declarative proof system for simple higher order logic. The purpose of DECLARE is to explore mechanisms of specification and proof that may be incorporated into other theorem provers. It has been developed to aid with reasoning about operational descriptions of systems and languages. Proofs in DECLARE are expressed as *proof outlines*, in a language that approximates written mathematics. The proof language includes specialised constructs for (co-)inductive types and relations. The system includes an abstract/article mechanism that provides a way of isolating the process of formalization from what results, and simultaneously allow the efficient separate processing of work units. After describing the system we discuss our approach to two subsidiary issues: automation and the interactive environment provided to the user.

1 Introduction

This technical report describes DECLARE, a prototype implementation of a declarative proof system for simple higher order logic. This system has been developed over the last 9 months as part of a project investigating tools to aid the computerized formulation of results from programming language theory, particularly operational semantics.[1]

The big picture of how DECLARE works is as follows. The user's aim is to develop a set of input files that specify proofs of logical properties expressed in higher order logic. The emphasis is that these files should contain 'declarative' characterizations of specifications and proofs, as distinct from the procedural, tactic-based characterizations often used in theorem provers. The DECLARE user divides his or her work into *theories*, and for each theory gives an *abstract*

[1]This work was carried out under the PhD. program at the Computer Laboratory, University of Cambridge, UK, and under the International Fellowship program at SRI International, Menlo Park, CA.

(or signature) and an *article* (or implementation)[2]. An abstract characterizes a theory from a logical perspective (as a set of constants and postulates), and an article implements a theory as a conservative extension of the logic via a series of definitions and proofs. The use of abstracts and articles enables articles to be implemented and processed independently, and unimportant details of the process of formalization need not escape the bounds of an article.

Besides abstracts and articles, the key feature of DECLARE is the language used to express proofs. Before we discuss this language, we must be clear what we mean by a 'proof' in this context. Harrison [Har97] describes several different uses of the word in the field of automated reasoning. Three of these are of interest here:

1. What is found in a mathematical text book, i.e. a sketch given in a mixture of natural, symbolic and formal languages, sufficient to convince the reader.

2. A script to be presented to a machine for checking. This may be just a sketch, or a program which tells the machine how to construct a formal proof.

3. A formal 'fully expansive' proof in a particular formal system, e.g. a derivation tree of inference rules and axioms.

As with Harrison, we will use the word 'proof' in the second sense. We will also use 'proof outline' to mean proofs (again in the second sense) that are merely sketches, and that require significant reasoning to fill in gaps.

Proofs in DECLARE are expressed as proof outlines, written in a language that approximates written mathematics. As such, DECLARE proofs are meant to express a declarative intent, and not a program. This builds on work done with similar languages by the Mizar group [Muz93] and Harrison [Har96]. Our language is similar to the Mizar language, but has been enhanced with constructs that makes certain kinds of proofs more succinct. We describe the proof language by example in Sections 2 and 3.

The distinction between declarative and procedural descriptions of proofs has been explored in some detail by Harrison [Har97], where he rightly points out that the distinction is very hard to make precise. Along with him we believe that the distinction is useful one, even if only understood in a weak way. The distinction is certainly a common one in computing: Prolog, LaTeX and HTML are examples of languages that aspire to a high declarative content.

1.1 Outline

This report is structured as follows: in the remainder of this section we outline the advantages of DECLARE, and how it is different from related work. Section

[2]This terminology comes from the Mizar system [Muz93]

2 describes a short example of the use of DECLARE's proof language. In Section 3 we give more details of the proof language, in particular specialised constructs for induction and co-induction. In Section 4 we describe the abstract/article mechanism, which is necessary for an efficient implementation of the batch proof checker (and is of independent interest as a light-weight structuring mechanism). In Section 5 we consider issues of automation, as DECLARE proof outlines are only real proofs given the provision of sufficient automation for logical leaps. Section 6 is more speculative: we consider what a 'fully interactive' environment for DECLARE might be like, and introduce the notion of *computer aided proof writing* that complements the style of proof language being used. We look toward future work in Section 7.

1.2 The Advantages of DECLARE

Some of the advantages of using a proof language like DECLARE's have been outlined by Harrison [Har97]. To summarize and extend these:

- Declarative proofs are more readable than tactic proofs.
- Automation is aided by having explicit goals at each stage.
- Proofs may be interpreted without needing to know the behaviour of tactics.
- Proof interpretation always terminates, unlike tactic proof which are expressed in a Turing-complete language.
- Static proof level analyses are feasible, for example it is possible to type-check proofs without running them; perform proof dependency analysis; and find unnecessary proof steps.

We expand on these throughout this report. Three additional, important benefits seem probable but remain to be demonstrated conclusively:

- Declarative proofs are potentially more maintainable.
- Declarative proofs are potentially more portable.
- A declarative style may appeal to a wider class of users, helping to deliver automated reasoning and formal methods to mathematicians and others.

We expand on portability issues in Section 7.2.

DECLARE is not a polished or complete system. The aim of DECLARE is not to supplant theorem provers such as Isabelle, PVS or HOL [Pau90, ORR+96, GM93], but to explore mechanisms of specification and proof that may eventually get incorporated into those systems. These mechanisms can complement rather than compete with existing methodology. We encourage developers and users of other theorem provers to consider the ideas contained in DECLARE with a view to incorporating them in the systems they use.

1.3 Related Work

We have already mentioned that DECLARE is inspired by the Mizar system [Muz93], and Harrison's 'Mizar Mode' work. While Mizar is a far more complete and established system, it is particularly limited by the amount of automation that is supported. The aim in DECLARE is to eventually include as much automation as is reasonably possible, thus reducing the size of proof outlines, thus improving the usability of the language.

DECLARE uses a simple higher order logic whereas Mizar uses set theory. From the point of view of mechanization, polymorphic simple type theory seems to occupy a neat, locally optimum position in the spectrum of possible logics. Type checking is decidable and efficient, terms can be represented fairly compactly, and a fair degree of expressiveness is achieved. It is not ideal for all purposes, but is excellent for many.

There are significant differences between the Mizar and DECLARE proof languages. In particular we are keen to add specialised constructs for new problem domains. For example, we provide (co-)induction and case analysis constructs suited for reasoning about (co-)inductive types and relations. In Mizar such reasoning is possible, but clumsy (for instance see the Mizar development of the syntax of ZF terms [Muz93]). Also, DECLARE proofs are 'backward', in the sense that the goal to be proved is always fully specified. Mizar proofs are normally 'forward', though a small backward element is possible through the use of the keyword `thesis`.

In addition to this, a major difference between DECLARE and Harrison's work is that we are working in a document based system. This means we can implement key items of functionality that cannot easily be implemented in Harrison's system, i.e. proof dependency analysis, error recovery and checking individual steps within a proof. Implementing these requires the ability to interpret and manipulate proofs as first class objects.

The use of abstracts and articles as we describe them here is not found in any other system. They are necessary for the efficient implementation of a proof checker for the DECLARE proof language.

2 An Example

In this section we use DECLARE to construct the runtime operational semantics for a toy programming language (a strict untyped, name-carrying lambda calculus), and prove that execution in this language is deterministic. The purpose of this is to demonstrate the use of DECLARE. We choose a problem from operational semantics as this is the domain of application DECLARE has been designed for.

The language we are studying has the following syntax:

$$
\begin{array}{lcll}
exp & = & x & \text{(variable)} \\
 & | & \lambda x.\ exp & \text{(abstraction)} \\
 & | & exp\ exp & \text{(application)}
\end{array}
$$

Evaluation is given by the following rules:

$$\frac{}{\lambda x.\ e \rightarrow \lambda x.\ e}$$

$$\frac{e_1 \rightarrow \lambda x.\ M \quad e_2 \rightarrow v_1 \quad M[x/v_1] \rightarrow v_2}{e_1\ e_2 \rightarrow v_2}$$

Our aim in this section is to develop an outline of a proof that evaluation is deterministic.

2.1 Writing the Article

The article begins with a declaration which specifies the syntax as a recursive type, using ML-like notation. We then declare a constant `subst` that notionally represents the substitution of closed expressions. For simplicity we do not fully define substitution here because the details of its behaviour are irrelevant to the proof we are considering.

```
type exp =
   Var of (Name: string)
 | Lam of (Bvar: string) * (Body: exp)
 | App of (Rator: exp) * (Rand: exp);

const subst : exp * string * exp -> exp;
```

Datatypes have a simple, well-understood interpretation in higher order logic and they represent a conservative (i.e. soundness preserving) extension to the logic [Mel89]. All definitional mechanisms in DECLARE are conservative, up to various proof obligations. DECLARE supports mutually recursive datatypes with nested recursion and possibly infinite branching, though in this case we just have a simple datatype. The declaration above introduces into the logic the type `exp`, constructors `Var`, `Lam` and `App`, selectors `Name`, `Bvar`, `Body`, `Rator` and `Rand`, and discriminators `Var?`, `Lam?` and `App?`.

Next we define the evaluation relation for the language as the least fixed point of a set of rules. The declaration is shown below. Relations defined by least fixed points also have a well-known interpretation in higher order logic [Pau94, CM92].

```
relation EVAL = lfp
  (Lam)        Lam(x,e1) --> Lam(x,e1)

  (App)           e1 --> Lam(x,M)
                & e2 --> v1
                & subst(M,x,v1) --> v2
            -------------------------------
                  App(e1,e2) --> v2;
```

Note that what we are constructing is a document, rather than a script, something along the lines of an Isabelle theory file. All interaction with DECLARE is by constructing documents and submitting them for checking. The batch processor is very quick, so the working environment is essentially interactive.

We are now ready to write the first proof in our article: that evaluation is deterministic. This is a simple fact that we prove in detail to illustrate DECLARE's proof language. The proof begins with a statement of the fact we want to prove:

```
proof of determinism:
    if V1: "e --> v" and V2: "e --> v'" then "v = v'"
...
end
```

Terms of the logic are quoted and formulae may be labelled as in `V1: "e --> v"`. The `if ... then` construct specifies an initial sequent.

The proof is by rule induction on the derivation tree of the evaluation `V1`, using rule case analysis to prove that the evaluation `V2` is forced at each step. The DECLARE proof language contains the `proceed by ...` construct for declaring induction steps:

```
proof of determinism:
    if V1: "e --> v" and V2: "e --> v'" then "v = v'";
proceed by rule induction on V1
  case "Lam x e1 --> Lam x e1"::
    ...
  case "App(e1,e2) --> v"::
    ...
  end
end
```

Each case after the induction has a label. For rule inductions, the labels must match (up to renaming of new variables) the bottom lines of rules in the evaluation relation. If these are ambiguous the rule names can also be used.

The first case is the base case of our inductive relation. There are no induction hypotheses, and we just have to prove that `v'` is `Lam(x,e1)`. Although it is implicit in the induction step, we state this explicitly for clarity: (`sts` is short for 'suffices to show')

```
proceed by rule induction on v
  case "Lam(x,e1) --> Lam(x,e1)"::
     sts "v' = Lam(x,e1)"
     ...
```

This can be proved by case analysis on the derivation of `V2`. Case analysis tells us which rules may have generated the fact. Because in this case of the induction $e = \lambda x.\ e1$, only the `Lam` rule could have applied. A `thus` construct introduces new facts, and in this case we use `thus thesis` to indicate the last stated goal. Any `thus` construct generates an associated proof obligation, which in this case is simply `v' = Lam(x,e1)`. All branches of a proof must end with a `thus thesis`, `qed` or `thus contradiction`.

The `by ...` part of the `thus` construct is called a *justification*. The justification must be sufficient to discharge this proof obligation when used in conjunction with an automated routine. After `by` comes a series of parameters to the automated routine. In this case we just have one, a theorem, specified by `rule cases on v'`. Normally theorem specifications are just names of existing facts, though in this case we have used a function `rule cases` on the fact `V2`. The function `rule cases` is a specialized construct of the proof language that automatically determines the relevant inductive relation and produces the cases theorem.

```
proceed by rule induction on v
  case "Lam(x,e1) --> Lam(x,e1)"::
     sts "v' = Lam(x,e1)"
     qed by rule cases on v'
```

This completes the proof of the first case of the induction. In the second case we have three induction hypotheses available because of the induction step. We can assume our proposition to be true for the three sub-evaluations, i.e. that, for some `x`, `M` and `v1`, `e1` uniquely evaluates to `Lam(x,M)`, `e2` to `v'` and `subst(M,x,v1)` to `v2`. Each case may be followed (optionally) by a 'we are given' declaration indicating these.[3] In Section 3.1 we will consider ways to avoid writing out induction hypotheses in full.

[3]The term language uses the following syntax: `!` for $\forall$, `?` for $\exists$, `&` for $\wedge$, `==>` for implication.

```
case "App(e1,e2) --> v2"::
    given x,M,v1 st
         I1: "!v'. e1 --> v' ==> Lam(x,M) = v'"
     and I2: "!v'. e2 --> v' ==> v1 = v'"
     and I3: "!v'. subst(M,x,v1) --> v' ==> v2 = v'";
    sts "v2 = v'";
    ...
```

Like the first case, the second case also uses rule case analysis. In this case we do not complete the proof in one step, instead we use a 'now consider' declaration, which is similar to `thus ...` except new local constants may be also introduced. Rule case analysis tells us that rule App must have been used to derive V1, and so intermediate values x', M' and v1' with the appropriate properties must exist.

```
    consider x',M',v1' st
         L1: "e1 --> Lam(x',M')"
      and L2: "e2 --> v1'"
      and L3: "subst(M',x',v1') --> v'"
      by rule cases on v';
```

This gives rise to the following proof obligation that automatic prover will be able to discharge based on the cases theorem:

$$\exists.x'\ M'\ v_1'.\ e_1 \rightarrow \lambda\, x'.\ M' \wedge\ e_2 \rightarrow v_1' \wedge\ M'[x'/v_1'] \rightarrow v'$$

We now use the uniqueness results from the induction hypotheses to complete the proof, finishing with qed. The word hence automatically includes immediately preceding facts in the next justification clause.

```
    thus "x = x'" and "M = M'" by I1,L1;
    hence "v1 = v1'" by I2,L2;
    hence qed by I3,L3;
```

The proof outline is now complete, and the complete article is shown in Appendix A.

2.2 Checking the article

Once the article has been written, it may be checked as follows. For illustrative purposes, we show the output if I1 is omitted from the third last line of the proof.

```
> decl lang.art
DECLARE v. 0.1a
```

```
parsing...done
importing and merging abstracts...done
type checking...done
establishing type "exp"...done
establishing constant "subst"...done
checking proof of "determinism"...

lang.art:23:30:step unjustified: simplification produced these goals:
  + e1 --> Lam(x',M')
  - x = x'

  + e1 --> Lam(x',M')
  - M = M'
```

The feedback shown is from the default automated routine used to discharge proof obligations. Note that 'error recovery' is possible: the rest of the proof after the error is checked, on the assumption that the facts stated were indeed derivable. Error recovery improves productivity and proof maintainability.

The article may be checked incrementally as it is being written. For example, it is possible to omit a case from the induction proof: this will be flagged by a warning. If a proof obligation cannot be discharged by the automatic prover, a warning will be given and the fact will be assumed for the remainder of the proof. The article checker is very quick, and so the working environment tends to mimic an interactive one (this is also the case with Mizar). We discuss the pros and cons of the proof language and 'batch mode' from the perspective of interaction in Section 6, and consider what a fully interactive environment for DECLARE might be like.

2.3 A shorter proof

A declarative proof language 'scales' as we add more automation to the system. For instance, the built-in prover, which is similar to Isabelle's `auto_tac`, can perform all the steps in the example proof automatically (apart from the induction step). Thus the proof could be as short as:

```
proof of determinism:
    if v: "e --> v" and v': "e --> v'" then "v = v'"
proceed by rule induction on v
  default:: qed by rule cases on v'
end;
```

Note the `default` construct specifies a proof outline for all unhandled cases of an induction proof.

3 The Proof Language: More Details

The previous section introduced most of the proof outlining constructs of DECLARE's proof language:

- The initial problem statement using `proof of` *sequent.*
- Introducing facts, goals and variables using:
  - `have` *facts* (fact introduction);
  - `consider` *vars* `st` *facts* (variable/fact introduction);
  - `sts` *goal* (goal introduction);
  - `for` *vars* `sts` *goal* (variable/goal introduction).
- Declaring inductions using `proceed by` *schema.*
- Giving justifications for proof obligations using `by` clauses.

There are three other proof outlining constructs that we will illustrate in Section 3.2 below. These are:

- Case analysis by disjunctive splitting using `per cases ...`.
- Strengthening the sequent using `stronger is ...`.
- Instantiating existential and universal quantifiers using `take ...`.

Apart from schema application, each proof outlining construct gives rise to an implicit *proof obligation.* For example, introducing a fact with `have` demands that the facts be proved true in the given context. These proof obligations can be discharged either by a subproof, or by an automated routine in conjunction with a justification clause. We deal with justification clauses and proof automation in the Section 5.

3.1 Easy Induction Hypotheses

Writing out induction hypotheses in detail can be informative, but also time-consuming and error-prone. Two mechanisms are available to help with this. First, the cases of the induction can be generated automatically by DECLARE. We consider this in detail in Section 6. Secondly, the shorthand `ihyp for ...` can be used within a `proceed by ...` construct to indicate the implicit induction property. Applying this to our first example yields:

```
proceed by rule induction on v
 case "App(e1,e2) --> v2"::
     given x,M,v1 st
          I1: ihyp for "e1 --> Lam(x,M)"
      and I2: ihyp for "e2 --> v1"
      and I3: ihyp for "subst(M,x,v1) --> v2"
     sts "v2 = v'"
     ...
```

The `ihyp` construct has a similar status in the proof language as `thesis` and `rule cases`, as it is a special purpose construct for particular kinds of proofs. In Section 7.1 we consider future options for making the language extensible as new constructs are needed.

What is the meaning of `ihyp`? This requires an understanding of the working of the `proceed by` construct, and in particular an appreciation for the induction predicate that is implicit in the sequent preceding an induction step. The induction predicate is determined by packaging the sequent as a single formula, omitting facts and goals that only contain variables that are constant throughout the induction. In the above example, the implicit two-place induction predicate is

$$\lambda(x,y).\ \forall v.\ x \dashrightarrow v \Longrightarrow y = v$$

Of course there is a disadvantage to using `ihyp`: it is less clear what fact is being stated, because the information is not 'up front'. The successful use of `ihyp` clearly relies on a strong intuitive understanding of the induction predicate. It is possible this could be addressed in an interactive environment for DECLARE, where a switch could be available to toggle between contracted and expanded visual representations.

3.2 Specialized constructs for co-induction

Co-inductive constructs are a natural dual to traditional induction theory, and co-inductive constructs are particularly useful when reasoning about infinite behaviour in operational semantics. For this reason, DECLARE includes support for co-induction. To illustrate this we consider the co-inductive definition of divergence for some transition relation $\rightsquigarrow$: a diverges if and only if there exists some b such that $a \rightsquigarrow b$ and b diverges. Such a recursive definition makes sense only if we interpret it 'permissively', not inductively, by allowing anything into the relation unless it is excluded by the definition.

In DECLARE, divergence for such a transition relation is defined by:

```
relation Divergent = gfp
(Step)                Divergent(a)
              --------------------------
               a --> b & Divergent(b)
```

To illustrate the use of co-induction in the proof language, let us presume that $W_1 \rightsquigarrow W_2$ and $W_2 \rightsquigarrow W_1$. If we want to prove that W1 is divergent, we must use co-induction over an appropriately strengthened goal:[4]

```
proof of example:
   if  W12: "W1 --> W2"
   and W21: "W2 --> W1"
   then "Divergent(W1)"
stronger is
   if W12 and W21
   and L1: "x IN {W1,W2}"
   then G: "Divergent(x)"
proceed by rule coinduction on G holding W1,W2 constant
  case Step::
     sts "?b. x --> b & b IN {W1,W2}"
     per cases by L1
        case "x = W1":: take b = "W2"; qed by W12
        case "x = W2":: take b = "W1"; qed by W21
     end
  end
end
```

Not surprisingly, the proof is very similar to an inductive proof, though we quote a goal rather than a fact as the support for the co-induction.

Induction and co-induction are good examples of constructs that require special treatment in the proof language if an acceptable level of declarative proof is to be reached.

3.3 Enhancing the language for the PVS logic

A prototype implementation of the DECLARE proof language has been performed for the PVS [ORR+96] proof system. Although we will not give full details here, it is worth noting how the proof language must change when we move to a more powerful logic.

The most important difference between simple higher order logic and the PVS logic is that terms must be proved to be well-formed. Normally this is simple

[4]Note that for a co-inductive relation, we must use co-induction to demonstrate membership of the set, and rules to indicate non-membership. This is the opposite way around to an inductive proof, as we would expect.

(Milner's W type inference algorithm decides this problem for simple higher order logic), but for the PVS logic the problem is in general undecidable, and so type correctness may involve the proof of various 'type correctness conditions' (TCCs).

In the PVS version of the proof language, proofs of well-formedness may be given with `wellformed` (or `wf`) clauses wherever terms with TCCs appear. For example:

```
proof of example:
  if A:"x=3*y" and B:"y>0"
  then "x/y = 3" (wf by B);
  qed by A;
end;
```

In PVS logic, the division operator has type $real \times non\_zero \rightarrow real$, and so a TCC arises: we must show $y \neq 0$.[5]

Once all well-formedness conditions have been proved, the interpretation of the language is essentially the same as for the simple type theory case.

4 Abstracts and Articles

In the previous two sections we have concentrated on DECLARE's proof language. In this section we step up a level to describe DECLARE's abstract/article mechanism. The purpose of this mechanism is to incorporate the benefits of modularization and separate processing into a specification system in a light weight and easily accessible fashion.

The abstract/article mechanism is a *necessary* complement to the proof language described in the last two sections. This is because batch processing requires the ability to create an efficient working environment where the exact *context* of a proof can be constructed quickly. By 'context' we mean all available resources, including those for the syntax, type checking and proof. Every time the batch checker is started the context must be recreated. By choosing the correct mechanism, efficient caching of these contexts by separate compilation can be achieved.

It may seem strange that we are concerned with adding an extra level of abstraction to our 'specification' language. Shouldn't the specification language be adequate to provide articles that are free from irrelevant detail? The problem is that we are adopting the philosophy of 'conservative extension', where constructs are not axiomatized, but defined by mechanisms that are known to be sound. This often leads to extra detail in the process of formalization that is irrelevant to the rest of the system. Abstracts provide a way of isolating the

[5]Note that logical context is accumulated left to right in all circumstances in PVS, and thus the fact `B` is accessible to discharge this obligation.

Figure 1: Part of an abstract: The Natural Numbers

```
import bool classical combin;
type 0 nat;
const "+" ": nat * nat -> nat";
const "*" ": nat * nat -> nat";

thm nat_induct [scheme] "P(0) & (!n. P(n) ==> P(n+1)) ==> !n. P(n)";
thm nat_wf [scheme] "(!n. (!m. m < n ==> P(m)) ==> P(n)) ==> !n. P(n)";
thm nat_recax [recax] "?!fn. fn(0) = e & (!n. fn(n + 1) = f (fn(n),n)";

thm ADD_CLAUSES [eqn] "0 + n = n & m + 0 = m";
thm ADD_AC [ac] "m + n = n + m & m + (n + p) = (m + n) + p";

thm MULT_ZERO [eqn] "0 * n = 0 & m * 0 = 0'';
thm MULT_ZERO [eqn] "1 * n = n & m * 1 = m";
thm MULT_AC [ac] "m * n = n * m & m * (n * p) = (m * n) * p";

thm LEFT_ADD_DISTRIB [distrib] "m * (n + p) = (m * n) + (m * p)";
thm RIGHT_ADD_DISTRIB [distrib] "(m + n) * p = (m * p) + (n * p)";
thm MULT_EQ_0 [eqn] "m * n = 0 <=> m = 0 | n = 0";
```

process of definition from what results, and simultaneously allow the efficient separate processing of work units. This is analogous to signatures, structures and separate compilation in a language like Standard ML. Abstracts may also be a first step to a more sophisticated parameterized modules system.

4.1 An Example

We give an example abstract for a fragment of the theory of natural numbers. Abstracts and articles can share files of notation that specify details of concrete syntax. The following is part of the notation file `nat.ntn`:

```
infix "+" 16 right;
infix "-" 18 left;
infix "*" 20 right;
```

Other notational devices are also supported, though they have not yet been finalized.

Figure 1 shows part of the abstract for the same theory. Note that the abstract does not specify *how* natural numbers are constructed as a conservative

extension of the logic, and nor does it specify how addition or multiplication are defined (they are in fact defined as primitive recursive functions, though they could equally be defined by some other method). Abstracts do not have to specify how a construct is defined, they just specify resulting properties of a construct.

One advantage of abstracts is that they provide a natural place to specify the *external* view of a theory, i.e. how the theory looks from the theory user's perspective. This is the purpose of the tags `eqn`, `scheme`, `distrib` and `ac` in the example above. These tags give crucial declarative information about the interpretation of theorems and the abstract properties they represent, and can be utilised by proof routines to good effect. For example, an automated routine may interpret an `ac` (associative/commutative) tag in different ways:

- It may cause ordered rewriting to be enabled;
- It may cause AC matching to be enabled during rewriting;
- It may cause AC unification to be enabled;
- It may be ignored.

Another advantage of abstracts is that they may be compiled and used without reference to an actual 'implementation' of the theory. This has both conceptual and practical benefits:

- It is easier to divide work into separate units that may be developed independently, which is especially an advantage in group projects.
- The theorem prover becomes more efficient, because creating an in-core representation of a 'working context' just involves combining several precompiled abstracts.
- Abstracts provide a light-weight mechanism for implementing delayed proof obligations.

4.2 Contrasting Abstracts with Other Mechanisms

Readers familiar with Isabelle's theory mechanism should note that abstracts are not the same as Isabelle theory files, since those files are concerned with the process of definition, rather than the properties that result. Isabelle's theory files are really more like articles than abstracts. Similarly, abstracts differ from PVS theory files. Compiled abstracts do bear a passing similarity to HOL theory files.

PVS supports the notion of proof obligations and ensures security through 'proof chain analysis'. This mechanism is more expensive in comparison to the use of abstracts, since an in core representation of theorem dependency information is kept all the time a user is working. In our scheme, dependency analysis is only done as a final phase.

4.3 An Outline of the Semantics of Abstracts

The notion of an article 'implementing' an abstract can be made formal: an article implements an abstract if, given articles for all imported abstracts, the article represents a conservative extension of the logic, and that extension contains all theorems, constants and types specified in the abstract. The article must provide outlines of proofs of all theorems in the abstract, and these proof outlines must suffice to construct fully expansive proofs when appropriate automation tools are added.

Abstracts can be given a simple semantics by saying that they are just placeholders for real, conservative extensions to the core logic. That is, an abstract is given a meaning by replacing it with an article that implements the abstract as a conservative extension. Once articles have been provided for all abstracts, then all articles can then be rechecked in an 'open environment'. Under these semantics, abstracts are just a useful, rather than a formal, device. Combined with the use of an LCF style proof checker like HOL or Isabelle in the final phase, this is adequate to establish the soundness of a development.

These semantics are inelegant, but can serve as an orthodoxy test for alternative semantics. It may be possible to give a semantics where an abstract corresponds to the formal logical assumption that constants and types with the specified properties do exist. This would be fine in set theory, but is non-trivial in polymorphic higher order logic, as this involves quantification over over polymorphic constants and type functions. Such a semantics may be given in future work.

To summarize: under the simplest semantics abstracts represent the informal assumption that the constants, types and theorems described in them may be derived by a conservative extension of the base logic. This assumption is discharged by providing an article for the abstract.

5 Automation and Justifications in DECLARE

DECLARE proofs are just proof outlines, and are not unambiguous, fully expansive proofs. Proof automation is essential to the success of the approach to proof outlining that we have described in this report. Without adequate automation, proofs scripts would be far too tedious to write.[6] In this section we consider two things:

- What information is available to automated routines.
- The default prover in the current implementation of DECLARE.

[6] Curiously, the proof language approach we have chosen can also aid automation, because unfettered goal-directed automation can be superior to the automation achieved by interactive tactics.

thm ::=	*thmname*	(quoting an existing theorem)
	thm[*terms*]	(explicitly instantiating a theorem)
	`rule cases on` *thm*	(rule case analysis)
	`structural cases on` *thm*	(structural case analysis)

Table 1: Theorem specifications in justification clauses

5.1 Information available to automated strategies

Justification clauses in DECLARE contain information about how the associated obligation might be proved automatically. They specify two things:

- the proof strategy to be used;
- a set of parameters to this strategy.

One of the main aims of DECLARE is to *minimize* the amount of strategy-specific information that needs to be supplied to automated reasoning routines. In particular, we want to avoid the 'knobs and dials' that often plague the use of automated reasoning routines, such as weightings, variable orderings and traversal flags. Presently parameters can only be theorem names, or short specifications of theorems using forms such as `rule cases on ...` and explicit instantiations. The exact forms allowed are shown in Table 5.1. The theorem specification language forms a little forward-proof language. In future versions it will be possible to add new forms here.

Strategies, then, must work with theorems as their main source of guidance. A central question is the following: how does a strategy know how to make effective use of a theorem? The declarative tags from abstracts described in Section 4 are used here. This is following the approach taken by Isabelle, where theorems (especially higher order theorems) plus interpretative guidelines are used as the central form of input to automated routines.

In summary, proof strategies have two kinds of information to guide them:

- a set of theorems, some of which are global theorems, and others local assumptions;
- a set tags giving interpretative information about these theorems.

A proof strategy is free to use this information as it pleases.

5.2 The Default Prover

In principle the nature of the proof outlining language is quite independent of the proof strategies adopted, and thus other automated provers, or specially written HOL or Isabelle strategies could be substituted. The present implementation of DECLARE includes a default prover somewhat similar to Isabelle's `auto_tac`,

though less extensible. In future work we hope to use Isabelle as a back-end for DECLARE.

The default prover works by:

- repeatedly simplifying the sequent using higher-order conditional/contextual rewriting. Presuming the tags on input theorems induce a terminating rewrite set, this always terminates.
- Applying generalized single point rules for eliminating constants from existential and universal formulae, for example:

$$\forall x.\, T_1[x] \wedge x = t \rightarrow T_2[x] \quad \rightsquigarrow \quad T1[t] \rightarrow T_2[t]$$

$$\exists x.\, T_1[x] \wedge x = t \wedge T_2[x] \quad \rightsquigarrow \quad T_1[t] \wedge T_2[t]$$

- Collecting like terms of linear arithmetic.
- Applying decision procedures for linear arithmetic, utilising contextual facts in the process.
- The prover then chooses a case-splitting method and repeats the process above. Potential case splits are specified by higher-order case splitting theorems like those used in the Isabelle simplifier [Pau90].
- Finally, when simplification and case-splitting has run its course, a simple tableau prover is called on the resulting goals.

The default proof checker of DECLARE is hard-coded into the system. We would prefer to use an LCF proof system to be certain about soundness, but this is not yet the focus of DECLARE. However, it should be possible to implement the strategies used in DECLARE in any LCF system.

6 Toward a Fully Interactive Environment for DECLARE

In previous sections we have considered the most important aspects of DECLARE: the proof language; the abstract/article mechanism and issues of automation. We now step up a level to consider an aspect of the interactive environment we eventually plan to deliver with the system.

6.1 Computer Aided Proof Writing

The current implementation of DECLARE makes the user work in 'batch mode', by repeatedly presenting the system with abstracts and articles for checking. Usually this is managed by a `make` script. Although 'batch mode' may seem

anathema to people used to using 'interactive' theorem provers, it is worth noting that this method of interaction is conceptually simple, easy to implement, well-understood and is the standard amongst compilation and information processing tools. It allows for reasonably quick 'context switching', and for integration with existing tools.

Despite this, in the long run it is desirable to understand how we might build a more interactive environment for a system like DECLARE. Actually building such an environment would need significant investment of resources, and is beyond the scope of this present report, but methodically considering the options is the first step to take.

The PVS [ORR+96] project has demonstrated that a superior interactive environment is achieved when a document processing style of interaction is adopted in a theorem prover. In this section we present one idea that would be possible to add to such an environment, related to the use of a declarative proof language.

The new feature of our planned interactive environment is what we call *computer aided proof writing* (CAPW). CAPW supports the process of constructing a proof document by getting the computer to actually write parts of the document for you. In this section we consider what we mean by this term and describe a primitive implementation of some CAPW functionality.

6.2 An example of CAPW

Consider the following interaction, which we will use as our example of CAPW: the user places the cursor on a `proceed by` construct in their proof document, and asks the computer to layout the cases of the induction step with the induction hypotheses and goals indicated in each inductive case. This is an excellent example of one of many possible situations where the human wants assistance in constructing a document, and the computer can easily be programmed to give it.

To make the example concrete, assume the buffer contains the article described in Section 2 with the cursor on the `proceed by` line. If the user now enters the command `expand-induct` then the computer writes the induction cases of the document for him or her, as shown in Figure 6.2. Underneath, this implemented by parsing and interpreting the proof script to the given point, generating the cases internally, and inserting the relevant steps in the proof document. The parsing and interpretation is done by a batch process, which generates the text to be inserted.

CAPW clearly shares similarities of intent with existing tactic mechanisms, though is fundamentally different because parts of the generated goal states are included in the document. This demands higher standards of output, because the generated text forms the only record of the action performed.

The mechanism above can be further enhanced. For example, the computer can adjust purported proofs to take into account changes in the generated cases.

Figure 2: The part of the proof written by DECLARE

```
case "Lam(x,e1) --> Lam(x,e1)"::
   sts "Lam(x,e1) = v'"

case "App(e1,e2) --> v2"::
   given x,M,v1 st
        I1: "!v'. e1 --> v' ==> Lam(x,M) = v'"
    and I2: "!v'. e2 --> v' ==> v1 = v'"
    and I3: "!v'. subst(M,x,v1) --> v' ==> v2 = v'"
   sts "v2 = v'";
```

If a case has disappeared, the user can be queried if he or she wants the case removed. The future challenge is to generalise and simplify the range of CAPW features that are provided in a system.

Some other potential instances of CAPW are:

- Presenting the user with plausible instantiations of theorems, and requesting the user to make a choice. The choice would be recorded as an entry in the proof document.
- Presenting the user with expansions and simplifications of existing formulae, again recording the results in the proof document.
- Performing proof planning [Bun91] and inserting the resulting proof plan as a part of the proof outline.

The example CAPW mechanism described above has been implemented and is available in the current version of the system.

6.3 CAPW contrasted with other mechanisms

CAPW differs from existing functionality in tactic based theorem provers, where the computer performs symbolic manipulations that result in new goal-states. These are then presented to the user, but are not recorded in the user's proof document. Such tactics frequently require arbitrary choices, such as names for new variables. These are choices that should be under the control of the user, and that should be recorded in the proof document. CAPW is a mechanism that will allow for this kind of user input, without polluting the core of the automated routines with devices for interaction.

CAPW bears some similarity to the use of templates to construct documents, which is common in applications such as word processors. However, templates are a 'dumb' mechanism, but CAPW actions depend on interpreting the logical

content of a document. The computer cannot sensibly layout the cases of an induction without interpreting the document to the point of interest.

Some instances of CAPW have been implemented in CHOL and TkHOL [The92, Sym95], where fragments of tactic scripts are produced by interaction in a structure editor. Dropping tactics, and moving to a declarative proof language greatly improves the possibilities for CAPW.

7 Summary and Future Work

In this report we have presented a prototype declarative proof system for higher order logic. Many of the ideas presented are immediately transferable to other systems.

In this section we consider two aspects that must be dealt with in future work: extensibility and proof portability.

7.1 Extensibility

Systems such as HOL and Isabelle have demonstrated the value of extensibility in theorem proving systems. Unlike these systems, DECLARE does not rely on a single programmable meta-language for extensibility. This is very deliberate: we want proofs expressed in a dedicated proof-outlining language, and not in a general Turing complete language.

A 'full strength' version of DECLARE would include programmability in several targeted ways by allowing:

- Plug-in back end provers that implement different automated strategies.
- Additional functions similar to `rule cases` in the forward proof language.
- Additional tags similar to `eqn` and `scheme` in the theorem interpretation language.
- Additional abbreviated forms `ihyp` and `thesis` in the term language.
- Additional forms in the definition language.
- Additional `proceed by` forms for tactic-like procedural proof constructs (we discuss this below).

All of these will involve some degree of programming in DECLARE's implementation language (presently Standard ML). Security would be ensured by LCF methods: new back end provers may only use a high-level secure set of primitives provided by DECLARE.

7.1.1 Tactics and `proceed by`

The astute reader will observe that schema application corresponds closely to the traditional notion of a tactic, because it applies a proof procedure that generates new goals. Thus tactics do have a place in the language, but their behaviour must be fully specified, declarative and predictable. A future version of DECLARE will include the ability to add new tactic-like decomposition algorithms to the `proceed by ...` construct.

7.2 Proof Portability

In this section we consider the prospect for 'porting' DECLARE proofs to other provers. Proof portability is a desirable property, especially in projects where the proofs themselves are of long term importance, such as projects to formalize large bodies of mathematics.

Because proof outlines are declarative, it may be possible to interpret them in other logics. At a certain conceptual level in mathematics, it matters little what underlying formalism is being used, as long as a certain minimum strength is assumed. A system like DECLARE allows proofs to be expressed at this level, and then translated into 'fully expansive' proofs in a particular deductive framework as needed. None of the proofs described in this report require more than a few second-order features, and all of them have an obvious translation into ZF set theory.

Remember that proofs in DECLARE are just outlines: the proof outline specifies a fully expansive proof *assuming a certain amount of automation.* The kind of automation required is normally left implicit, as most obligations are discharged by the default prover.

The question then is as follows: when translating to another logic or 'target system', where should the automation reside? Two options are available to us, both of which will be explored in future work.

- The automation resides in the target prover. DECLARE can output the proof outline in some simple, compiled format, without giving details of the proofs of proof obligations. The target prover can then translate the proof outline into its own logic and deductive framework and discharge the obligations using its own automation.

- The automation resides in DECLARE. DECLARE can output the proof as a 'proof object' in higher order logic, fully reduced to primitive inferences. This may then be translated to the appropriate calculus.

Two threats exist to proof portability. First, proofs may rely on quirks of automated strategies that allow large logical leaps in an argument. To avoid this, it may even be useful to *weaken* the power of the default prover, to ensure that proofs remain portable! The aim of portability is thus in conflict with the

aim of a 'minimal' script. Shortening a script via automation is tempting, but it will make it acceptable to less systems, just as cutting pieces out of a journal proof may make it comprehensible to fewer readers.

Secondly, proofs may rely on aspects of the logic that are not interpretable in other logics. This can only be avoided by restricting the use of exotic features of a logic.

Acknowledgments

Many thanks to Tobias Nipkow, John Rushby, Natarajan Shankar, Mike Gordon, Florian Kamuller, Michael Norrish, Chris Pratten, Mark Staples and Paul Curzon for the stimulating discussions I have had with them about this work, and for encouraging me to document it. I am also deeply appreciative of the support of SRI International through their International Fellowship program.

A A Complete Article

The complete article explained in Section 2 is shown below:

```
datatype exp =
   Var of (Name: string)
 | Lam of (Bvar: string) * (Body: exp)
 | App of (Rator: exp) * (Rand: exp);

const subst : exp * string * exp -> exp;

relation EVAL = lfp
  (Lam)        Lam(x,e1) --> Lam(x,e1)

  (App)           e1 --> Lam(x,M)
                & e2 --> v1
                & subst(M,x,v1) --> v2
             -------------------------------
                 App(e1,e2) --> v2;

proof of determinism:
    if V1:"e --> v" and V2: "e --> v'" then "v = v'";
proceed by rule induction on V1
  case "Lam(x,e1) --> Lam(x,e1)"::
     sts "Lam(x,e1) = v'";
     qed by rule cases on V2;
  case "App(e1,e2) --> v2"::
     given x,M,v1 st
          I1: "!v'. e1 --> v' ==> Lam(x,M) = v'"
      and I2: "!v'. e2 --> v' ==> v1 = v'"
```

```
      and I3: "!v'. subst(M,x,v1) --> v' ==> v2 = v'";
    consider x',M',v1' st
         L1: "e1 --> Lam(x',M')"
      and L2: "e2 --> v1'"
      and L3: "subst(M',x',v1') --> v'"
      by rule cases on V2';
    thus "x = x'" and "M = M'" by I1,L1;
    hence "v1 = v1'" by I2,L2;
    hence thesis by I3,L3;
  end;
end;
```

References

[Bun91] A. Bundy. A science of reasoning. In J-L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 178–198. MIT Press, 1991.

[CM92] J. Camilleri and T.F Melham. Reasoning with inductively defined relations in the HOL theorem prover. Technical Report 265, University of Cambridge Computer Laboratory, August 1992.

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

[Har96] J. R. Harrison. A Mizar Mode for HOL. In J. Grundy J. von Wright and J. Harrison, editors, *Theorem Proving in Higher Order Logics: 9th International Conference*, Lecture Notes in Computer Science, pages 203–230. Springer-Verlag, August 1996.

[Har97] J. R. Harrison. Proof style. Technical Report 410, University of Cambridge Computer Laboratory, January 1997.

[Mel89] T.F Melham. Automating recursive type definitions in higher-order logic. In G. Birtwhistle and P.A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 341–386. Springer-Verlag, 1989.

[Muz93] M. Muzalewski. *An Outline of PC Mizar.* Foundation Philippe le Hodey, 1993.

[ORR+96] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining specification, proof checking, and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*,

pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[Pau90] L. C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–385. Academic Press, 1990.

[Pau94] L. C. Paulson. A fixed point approach to implementing (co)inductive definitions. In A. Bundy, editor, *12th International Conference on Automated Deduction*, pages 148–161. Springer, 1994.

[Sym95] D. R. Syme. A new interface for HOL - ideas, issues and implementation. In J. Alves-Foss, editor, *Higher Order Logic Theorem Proving and its Applications*, LNCS, pages 17–31. Springer-Verlag, September 1995.

[The92] Laurent Thery. Real theorem provers deserve real interfaces. In *Software Engineering Notes*, volume 17. ACM Press, 1992.