On Formal Specification of a Proof Tool*

R.D. Arthan
ICL Secure Systems,
Eskdale Road,
Winnersh,
Berks. RG11 5TT

1 Introduction

1.1 Background

Tools and methods for the specification and design of computer systems are increasing in sophistication. Much current research and development is attempting to exploit this sophistication to improve the effectiveness of systems development practices. It is becoming feasible to offer much higher assurance than hitherto that systems meet critical requirements, e.g. concerning safety or security. Standards such as [7] are evolving to demand the use of formal specification *and verification* of designs (and, one day, perhaps implementations). Thus, tools giving cost-effective means for providing formal proofs of critical requirements are of increasing importance. ICL Secure Systems, as part of its role as lead partner in the DTI-sponsored FST project, is attempting to improve the technology base for formal verification.

The main enabling technology with which ICL is concerned is the HOL theorem proving system, [1, 8]. A public domain version of HOL has been distributed by Cambridge University and has been used with some success both in academia and in industry. ICL plans to offer an industrial quality implementation of HOL and to use it to provide proof support for other formalisms such as Z. An experimental Z proof tool based on a prototype reimplementation of HOL has recently been produced for use by ICL and its collaborators.

This paper gives a simplified case study, in Z, illustrating some of the techniques being used. The case study is concerned with two main themes. The first theme is concerned with the integrity of the proof tool, the second is concerned with the consistency of the specifications about which we wish to reason and with the extension mechanisms which the logics used should support. The work on integrity is fairly recent work carried out within the FST project. The treatment of consistency is based on much earlier work of ICL on using HOL to reason about Z specifications which was first described in an ICL internal document, [6] and which resulted in the inclusion of a new facility supporting loose specifications in the Cambridge HOL system.

1.2 Integrity

One issue in industrialising HOL which is felt to be of particular importance is its integrity, i.e., the level of confidence that users and their customers can have in the correctness of the proof tool itself. While research continues to ameliorate the problem, the production of fully formal proofs is a difficult and time-consuming activity. The expense of producing such proofs is not justifiable unless one can be confident that they are correct. Commercial proofs involve millions of primitive inference steps (mostly automatically performed, one hopes!); human checking of each step is inconceivable.

*Published in *VDM '91 Formal Software Development Methods*, LNCS 551, Springer Verlag 1991.

The basic approach to the integrity problem is to specify formally the logic used in the tool and to give an abstract specification of the critical requirements for the tool. The proof tools we are concerned with follow the LCF paradigm, described in [2], which encapsulates all critical code inside an abstract data type. In our approach the implementation of this critical kernel is based on a formal design which, we assert, meets the critical requirements. This assertion may be rigorously formalised and so is susceptible both to informal analysis and to fully formal proof. Finding the proofs is eased by constructing the design to facilitate a top-down decomposition of the critical requirements into requirements on its subsystems. The main goal of all of this is to minimise the amount of proof work which must be carried out. The critical requirements deliberately fall short of a fully formal functional specification (the design is much closer to that), since what we wish to prove is not that the tool does everything the user wants correctly, but that it cannot be used to prove 'false'. A proof of full functional correctness would be both unnecessary and unfeasible. We believe that this method of reducing the complexity of a high-assurance problem by concentrating effort on what is critical is a very important part of making formal verification possible in real situations.

An important aspect of the LCF paradigm is that, once we have the kernel of the proof tool, development of facilities to make the tool effective to use can be produced freely without prejudicing the integrity of the system. So, for example, an automatic proof procedure may be transferred from a research and development environment into actual industrial use without requiring more extensive verification of its correctness than would be required for any other software engineering tool — infelicitous behaviour of the proof procedure cannot compromise the integrity of the system. The construction of these higher level facilities is outside the scope of this paper.

1.3 Consistency of Specifications

We wish to use the proof tool to reason about specifications written in languages such as Z. Such specification languages include features which at first sight may compromise the integrity of the proof tool. In this paper we describe a practical solution to one such problem which arises with Z.

Informal proof work with Z usually treats the specification as a collection of axioms. This approach has the inherent disadvantage that one may prove anything on the basis of an inconsistent set of axioms. It is therefore necessary to relate proof to specification in such a way that an incorrect specification does not give rise to this problem.

One reason for allowing what are apparently arbitrary axiomatic extensions in Z, is to allow the user to make loose specifications. A variable can be defined by stating a property which does not have to define the variable uniquely. This feature helps us to avoid cluttering specifications with irrelevant details and to make our specifications more general.

The underlying logical mechanism we propose to solve this problem for the axiomatic and generic boxes in Z is a rule of *conservative extension* allowing new objects to be defined provided an appropriate consistency proposition has been proved. The apparent disadvantage with this is that we must interleave our specification activity with proof work to supply the necessary consistency propositions. Fortunately some fairly straightforward automated proof techniques allows us to defer the proof obligations (essentially by approximating a loose specification by one which can automatically be proved consistent and which is equivalent to the original one if it can be proved consistent).

In fact, to ease this problem further, ICL have developed machinery for HOL which can automatically discharge the consistency proof obligations for quite a useful class of specification idioms. Work is in progress on extending this machinery to the prototype Z proof tool.

1.4 Overview of the Case Study

The bulk of the sequel comprises a formal specification in Z of the following:

1. the language and deductive system of a simple logic (section 2);

2. an abstract model of a proof tool for the logic and a statement of its critical properties (section 3);
3. a design for the kernel of a proof tool for the logic, believed to satisfy the critical properties (section 4).

Section 5.1 discusses how one might informally or formally verify the "belief" mentioned in item 3 above; Sections 5.2 and 5.3 consider how such a kernel might be implemented and discuss its use to support specification activities.

The Z specification is presented in definition-before-use order. Those who prefer to read top-down are invited to read section 2.1 first, to set the scene, and then sections 3 and 4, in that order, skipping back to section 2 when necessary. An index to the specification may be found at the end of the paper.

Our use of the Z notation is intended to follow [5]. The forms of Z paragraph such as free type definitions which do not come in boxes are high-lighted by a bar in the left margin. Defining occurrences of global variables are shown in **bold** type.

The source text of this document is in fact a script from which the Z paragraphs can be extracted for machine processing. The type-correctness of the specification has been checked using the prototype Z proof support tool referred to in section 1.1 above. No formal proof work has been carried out on the material in this document, however proof work has begun on the HOL specification of the proof tool for HOL on which this paper has been based.

2 The Logic

The logic used in our example will be classical first order logic. The treatment is fairly close to that which may be found in [3], the main departure being that we envisage variable names and the like being character strings rather than single letters with superscripts and subscripts. Also since we are interested in tools which help a user to build particular theories of interest, later on we are very explicit about mechanisms with which axioms are introduced.

2.1 Language

Names The language with which we shall work contains names whose structure we do not wish to specify here. In an implementation these might be character strings. For the specification, we introduce the set of names as a given set.

z

$$[\mathbf{name}]$$

Terms The terms of our language are variables or are formed from simpler terms by function application. It is technically convenient to treat constants as functions with no arguments.

z

$$\begin{array}{lll} \mathbf{term} & ::= & \mathbf{var} \langle\langle\ name\ \rangle\rangle \\ & | & \mathbf{app} \langle\langle\ name \times seq\ term\ \rangle\rangle \end{array}$$

Formulae A formula is either an application of an atomic predicate to a list of formulae or is formed from other formulae via negation, implication or universal quantification.

z

$$\begin{array}{lll} \mathbf{form} & ::= & \mathbf{prd} \langle\langle\ name \times seq\ term\ \rangle\rangle \\ & | & \mathbf{neg} \langle\langle\ form\ \rangle\rangle \\ & | & \mathbf{imp} \langle\langle\ form \times form\ \rangle\rangle \\ & | & \mathbf{all} \langle\langle\ name \times form\ \rangle\rangle \end{array}$$

Theories A theory is just a set of formulae. We will think of the theory as specifying the functions and predicates which appear in it.

z

$$\mathbf{theory} == \mathbb{P}\ \mathit{form}$$

Well-Formedness of Terms We will say that a formula is well-formed with respect to a theory if all the functions and predicates it contains appear in the theory. Note that this is a rather different use of terminology from [3], where the idea of a wff just corresponds to our representation of the syntax as a free type.

If a function (or predicate) name appears in two places in a formula or a theory with differing numbers of arguments we think of the different instances as being distinct functions (or predicates). (We might expect a proof tool to protect the user from getting into this situation, lest it be confusing in practice).

We use the following auxiliary definitions to define well-formedness:

z

$$\begin{array}{l}
\mathbf{term_funcs} : \mathit{term} \rightarrow \mathbb{F}\ (\mathit{name} \times \mathbb{N});\\
\mathbf{form_funcs} : \mathit{form} \rightarrow \mathbb{F}\ (\mathit{name} \times \mathbb{N});\\
\mathbf{form_preds} : \mathit{form} \rightarrow \mathbb{F}\ (\mathit{name} \times \mathbb{N})
\end{array}$$

$$\begin{array}{ll}
\multicolumn{2}{l}{\forall n{:}\mathit{name};\ ts{:}\mathit{seq\ term};\ p,\ q{:}\mathit{form} \bullet}\\
 & \mathit{term_funcs}\ (\mathit{var}\ n) = \{\}\\
\wedge & \mathit{term_funcs}\ (\mathit{app}(n,\ ts)) = \{(n,\ \#\ ts)\} \cup \bigcup(\mathit{ran}\ (\mathit{term_funcs}\ o\ ts))\\
\wedge & \\
 & \mathit{form_funcs}\ (\mathit{prd}(n,\ ts)) = \bigcup(\mathit{ran}\ (\mathit{term_funcs}\ o\ ts))\\
\wedge & \mathit{form_funcs}\ (\mathit{neg}\ p) = \mathit{form_funcs}\ p\\
\wedge & \mathit{form_funcs}\ (\mathit{imp}(p,\ q)) = \mathit{form_funcs}\ p \cup \mathit{form_funcs}\ q\\
\wedge & \mathit{form_funcs}\ (\mathit{all}(n,\ p)) = \mathit{form_funcs}\ p\\
\wedge & \\
 & \mathit{form_preds}(\mathit{prd}(n,\ ts)) = \{(n,\ \#\ ts)\}\\
\wedge & \mathit{form_preds}\ (\mathit{neg}\ p) = \mathit{form_preds}\ p\\
\wedge & \mathit{form_preds}\ (\mathit{imp}(p,\ q)) = \mathit{form_preds}\ p \cup \mathit{form_preds}\ q\\
\wedge & \mathit{form_preds}\ (\mathit{all}(n,\ p)) = \mathit{form_preds}\ p
\end{array}$$

Now we can define *wff*, the function which assigns to a theory the set of formulae which are well-formed with respect to it.

z

$$\mathbf{wff} : \mathit{theory} \rightarrow \mathbb{P}\ \mathit{form}$$

$$\begin{array}{lll}
\multicolumn{3}{l}{\forall \mathit{thy}{:}\mathit{theory};\ p{:}\mathit{form} \bullet}\\
 & \multicolumn{2}{l}{p \in \mathit{wff}\ \mathit{thy}}\\
\Leftrightarrow & (& \mathit{form_funcs}\ p \subseteq \bigcup(\mathit{form_funcs}(\!|\mathit{thy}|\!))\\
 & \wedge & \mathit{form_preds}\ p \subseteq \bigcup(\mathit{form_preds}(\!|\mathit{thy}|\!)))
\end{array}$$

2.2 Operations on Syntax

This section contains the definitions of certain operations on syntax which we shall need. The operations are extraction of free variables, substitution of terms for variables and some derived formula constructors. We define substitution in terms of a matching operation. This section contains the definitions of these operations.

Free Variables The functions which extract the free variables of terms and formulae are defined as follows:

z

$$\mathbf{term\_frees} : term \to \mathbb{F}\ name;$$
$$\mathbf{form\_frees} : form \to \mathbb{F}\ name$$

$$\begin{array}{ll}
\forall n{:}name;\ ts{:}seq\ term;\ p,\ q{:}form \bullet & \\
 & term_frees\ (var\ n) = \{n\} \\
\land & term_frees(app(n,\ ts)) = \bigcup(ran\ (term_frees\ o\ ts)) \\
\land & \\
 & form_frees(prd(n,\ ts)) = \bigcup(ran\ (term_frees\ o\ ts)) \\
\land & form_frees\ (neg\ p) = form_frees\ p \\
\land & form_frees\ (imp(p,\ q)) = form_frees\ p \cup form_frees\ q \\
\land & form_frees\ (all(n,\ p)) = form_frees\ p \setminus \{n\}
\end{array}$$

Matching We will need to define the notion of substituting a term for a variable in a formula. This is an idea which is frequently defined vaguely or incorrectly and so it is worth specifying formally. To do this in a reasonably abstract way, we first of all define a notion of matching. The definition is quite technical and readers are invited to skip this section, if they wish.

We wish to define a partial function *form_match*. Given two formulae, p_1 and p_2, say, such that p_2 is a substitution instance of p_1 under some assignment, *match*, of terms to the free variables of p_1, *form_match* should return *match*. Here we wish to allow renaming of bound variables in order to avoid variable capture problems. So for example, under the assignment which sends y to $x + 1$, we wish $\forall x \bullet x = y$ to match $\forall x' \bullet x' = x + 1$ (but not $\forall x \bullet x = x + 1$).

First of all we define matching for terms and sequences of terms. It turns out that matching is easiest to specify if we introduce an auxiliary argument which records the correspondence between bound variables in the two terms.

z

$$
\begin{array}{l}
\quad \mathbf{term\_match} : (term \times term \times (name \twoheadrightarrow term)) \nrightarrow (name \twoheadrightarrow term); \\
\quad \mathbf{seq\_term\_match} : (seq\ term \times seq\ term \times (name \twoheadrightarrow term)) \nrightarrow (name \twoheadrightarrow term) \\
\hline
\quad (\forall t1\ ,\ t2 \colon term;\ env,\ match \colon name \twoheadrightarrow term \bullet \\
\qquad\quad (t1,\ t2,\ env) \mapsto match \in term\_match \Leftrightarrow \\
\qquad\qquad\quad (\exists n \colon name \bullet t1 = var\ n \\
\qquad\qquad\quad \wedge \qquad (env\ n = t2 \vee n \notin dom\ env) \\
\qquad\qquad\quad \wedge \qquad match = \{n \mapsto t2\}) \\
\\
\qquad\quad \vee \qquad (\exists n \colon name;\ ts1,\ ts2 \colon seq\ term \bullet t1 = app(n,\ ts1) \wedge t2 = app(n,\ ts2) \\
\qquad\qquad\quad \wedge \qquad (ts1,\ ts2,\ env) \mapsto match \in seq\_term\_match)) \\
\wedge \\
\quad (\forall ts1,\ ts2 \colon seq\ term;\ env,\ match \colon name \twoheadrightarrow term \bullet \\
\qquad\quad (ts1,\ ts2,\ env) \mapsto match \in seq\_term\_match \Leftrightarrow \\
\qquad\qquad\qquad\qquad dom\ ts1 = dom\ ts2 \\
\qquad\qquad\quad \wedge \qquad (\exists matches : seq(name \twoheadrightarrow term) \bullet \\
\qquad\qquad\qquad\qquad \forall i : dom\ ts1 \bullet \\
\qquad\qquad\qquad\qquad\qquad (ts1\ i,\ ts2\ i,\ env) \mapsto matches\ i \in term\_match \\
\qquad\qquad\qquad\qquad \wedge \qquad disjoint\ matches \wedge match = \bigcup(ran\ matches)))
\end{array}
$$

form_match is now defined as follows:

z

$$
\begin{array}{l}
\quad \mathbf{form\_match} : (form \times form \times (name \twoheadrightarrow term)) \nrightarrow (name \twoheadrightarrow term) \\
\hline
\forall p1\ ,\ p2 \colon form;\ env,\ match \colon name \twoheadrightarrow term \bullet \\
\quad (p1,\ p2,\ env) \mapsto match \in form\_match \\
\Leftrightarrow \\
\qquad\quad (\exists n \colon name;\ ts1,\ ts2 \colon seq\ term;\ matches : seq\ (name \twoheadrightarrow term) \bullet \\
\qquad\qquad\quad p1 = prd(n,\ ts1) \wedge p2 = prd(n,\ ts2) \\
\qquad\quad \wedge \qquad (ts1,\ ts2,\ env) \mapsto match \in seq\_term\_match) \\
\quad \vee \qquad (\exists q1,\ q2 \colon form \bullet \\
\qquad\qquad\quad p1 = neg\ q1 \wedge p2 = neg\ q2 \\
\qquad\quad \wedge \qquad (q1,\ q2,\ env) \mapsto match \in form\_match) \\
\quad \vee \qquad (\exists q1,\ r1,\ q2,\ r2 \colon form;\ matchq,\ matchr \colon name \twoheadrightarrow term \bullet \\
\qquad\qquad\quad p1 = imp(q1,\ r1) \wedge p2 = imp(q2,\ r2) \\
\qquad\quad \wedge \qquad (q1,\ q2,\ env) \mapsto matchq \in form\_match \\
\qquad\quad \wedge \qquad (r1,\ r2,\ env) \mapsto matchr \in form\_match \\
\qquad\quad \wedge \qquad disjoint \langle matchq,\ matchr \rangle \\
\qquad\quad \wedge \qquad match = matchq \cup matchr) \\
\quad \vee \qquad (\exists n1,\ n2 \colon name;\ q1,\ q2 \colon form \bullet \\
\qquad\qquad\quad p1 = all(n1,\ q1) \wedge p2 = all(n2,\ q2) \\
\qquad\quad \wedge \qquad (q1,\ q2,\ env \oplus \{n1 \mapsto var\ n2\}) \mapsto match \in form\_match)
\end{array}
$$

Substitution This is now easy to specify in terms of matching. Note that the fact that we make it a total function implies that the given set *name* must be infinite (so that the supply of names for use in renaming bound variables is never exhausted).

z

$$\textbf{subst} : (name \nrightarrow term) \times form \rightarrow form$$

$$\forall subs\colon name \nrightarrow term;\ p{:}form \bullet form\_match(p,\ subst(subs,\ p),\ \varnothing) = subs$$

Derived Formula Constructors To define our rule of conservative extension, we require two derived formula constructors ('derived' as opposed to the 'primitive' constructors, *prd*, *neg*, *imp* and *all*).

z

$$\textbf{exists} : (name \times form) \rightarrow form$$

$$\forall n{:}name;\ p{:}form \bullet exists(n,\ p) = neg(all(n,\ neg\ p))$$

We also need a function to form the universal quantification of a formula over a list of variables:

z

$$\textbf{list\_all} : seq\ name \times form \rightarrow form$$

$$\begin{aligned} &\forall p{:}form;\ n{:}name;\ ns\colon seq\ name \bullet \\ &\quad list\_all(\langle\rangle,\ p) = p \\ &\wedge\quad list\_all(\langle n\rangle \frown ns,\ p) = all(n,\ list\_all(ns,\ p)) \end{aligned}$$

2.3 Inference

The two inference rules are exactly as in [3]. They are the rule of *modus ponens,* and the rule of generalisation. We formalise these and the axioms of first order logic in this section and also define the notion of derivability.

Modus Ponens This rule says that from $p \Rightarrow q$ and p we may infer q:

z

$$\textbf{mp} : (form \times form) \nrightarrow form$$

$$\begin{aligned} &dom\ mp = \{p,\ q{:}form \mid true \bullet (imp(p,\ q),\ p)\} \\ &\wedge\quad (\forall p,\ q{:}form \bullet mp(imp(p,\ q),\ p) = q) \end{aligned}$$

Generalisation This rule says that from p we may infer $\forall x \bullet p$ for any variable x.

z

$$\textbf{gen} : (name \times form) \rightarrow form$$

$$\forall n{:}name;\ p{:}form \bullet gen(n,\ p) = all(n,\ p)$$

Logical Axioms We also need the logical axioms for first order logic with equality[1]. See [3] for a description of these. Since their formalisation is not particularly illuminating we omit the details here.

[1] Building in the equality axioms makes it easier to pretend that the proof tool we are presenting would be of practical use, since we do not supply a mechanism by which the user could introduce an infinite set of axioms. In a proof system such as HOL, polymorphism and the ability to define higher order functions allow most, if not all, theories of practical interest to be finitely axiomatised and the introduction of infinite axiom schemes is not required.

z

$$\textbf{logical\_axioms} : \mathbb{P}\ \mathit{form}$$

.. ..

Direct Derivability We now wish to say formally how the inference rules and the logical axioms are used to construct the consequences of a set of formulae. This notion is defined using the following idea of a direct consequence.

z

$$\textbf{direct\_consequences} : \mathbb{P}\ \mathit{form} \rightarrow \mathbb{P}\ \mathit{form}$$

$$\begin{array}{ll}
\forall \mathit{hyps}{:}\mathbb{P}\ \mathit{form};\ p{:}\mathit{form}\bullet p \in \mathit{direct_consequences}\ \mathit{hyps} \Leftrightarrow & \\
\quad & p \in \mathit{logical_axioms} \\
\vee & p \in \mathit{hyps} \\
\vee & (\exists q,\ r{:}\mathit{hyps}\bullet p = \mathit{mp}(q,\ r)) \\
\vee & (\exists n{:}\mathit{name};\ q : \mathit{hyps}\bullet p = \mathit{gen}(n,\ q))
\end{array}$$

Derivability The consequences of a set, *hyps* say, of formulae comprise the smallest set containing *hyps* which is closed under taking direct consequences.

z

$$\textbf{consequences} : \mathbb{P}\ \mathit{form} \rightarrow \mathbb{P}\ \mathit{form}$$

$$\begin{array}{l}
\forall \mathit{hyps}{:}\mathbb{P}\ \mathit{form}\bullet \\
\qquad \mathit{consequences}\ \mathit{hyps} = \\
\qquad \bigcap\{\mathit{ps} : \mathbb{P}\ \mathit{form} \mid \mathit{hyps} \subseteq \mathit{ps} \wedge \mathit{direct_consequences}\ \mathit{ps} \subseteq \mathit{ps}\}
\end{array}$$

We shall often use the term *theorem* to describe a formula which is a consequence of a particular theory under discussion.

2.4 Extending Theories

In this section we define two mechanisms for extending a theory. The first mechanism, called *new_axiom*, supports addition of an arbitrary formula as an axiom. The second mechanism, called *new_specification*, is parameterised by a formula of a particular form, which must be a theorem of the theory we are extending. This theorem constitutes a proof of the consistency of an implicit specification of a new function. Given such a theorem, *new_specification* adds a defining axiom for the new function which, in fact, constitutes a *conservative* extension of the theory.

Examples of definitions in actual use are very frequently conservative. For example, all of the definitions in this document are intended to be conservative over the sort of axiom system one imagines should be provided for Z (allowing Z to be viewed as a many-sorted variant of Zermelo set theory).

2.4.1 Axiomatic Extension

An arbitrary formula may be introduced into a theory by the function *new_axiom*:

z

$$\textbf{new\_axiom} : \mathit{theory} \rightarrow \mathit{form} \rightarrow \mathit{theory}$$

$$\forall \mathit{thy}{:}\mathit{theory};\ p{:}\mathit{form}\bullet \mathit{new\_axiom}\ \mathit{thy}\ p = \mathit{thy} \cup \{p\}$$

We take the view that, in practice, the user will wish to work within a fixed set of axioms, and part of the critical properties we identify for the kernel of the proof system asserts that axioms introduced with *new_axiom* are clearly distinguished from the axioms introduced by *new_specification*.

2.4.2 Conservative Extension

Our conservative extension mechanism allows us to introduce a new function satisfying a specified property. Assume, for example, that we were working in the theory of real numbers and that we wished to define the ceiling functions. That is to say we wish to define a function, *ceil*, say, with one parameter satisfying the property

Example

$$\forall x \bullet (ceil\ x \in \mathbb{Z} \ \wedge \ ceil\ x \geq x \ \wedge \ (\forall m \bullet (m < ceil\ x \ \wedge \ m \in \mathbb{Z}) \Rightarrow m < x))$$

To introduce such a function we must first demonstrate that the above definition is conservative (and hence, *a fortiori* consistent). To do this we would first prove the theorem:

Example

$$\forall x \bullet \exists ceil \bullet (ceil \in \mathbb{Z} \ \wedge \ ceil \geq x \ \wedge \ (\forall m \bullet (m < ceil \ \wedge \ m \in \mathbb{Z}) \Rightarrow m < x))$$

It is fairly easy to see that the theoremhood, and hence the truth, of the above assertion implies that for any element, x, of any model of the theory, there is an element, *ceil*, such that the above holds. It follows that in any model we can find an interpretation for the desired new function symbol *ceil* (provided the set theory in which we do the model theory has the axiom of choice). Note that the argument relies on the fact that the assertion contains no free variables.

Thus, in essence, our rule says that given a theorem of the form:

Example

$$\forall x1 \bullet \forall x2 \bullet \ ... \ \forall xk \bullet \exists c \bullet P$$

where the x_i are distinct variables, we may introduce a new function, f say, of k arguments, with the defining axiom

Example

$$\forall x1 \bullet \forall x2 \bullet \ ... \ \forall xk \bullet P[f(x1,\ x2,\ ...,\ xk)/c]$$

where the notation in the square brackets denotes substitution.

Our principle of definition is then given by the following function, *new_specification*, in which we assume, for simplicity that the name of the new function is the same as the existentially quantified variable in the theorem:

Z

$$\textbf{new\_specification} : theory \rightarrow form \nrightarrow theory$$

$$\begin{array}{llll}
\forall thy{:}theory \bullet & dom\ (new_specification\ thy) & & \\
= & \{ & xs{:}\ seq\ name;\ c{:}name;\ p{:}wff\ thy & \\
 & | & & form_frees\ p \subseteq \{c\} \cup ran\ xs \\
 & & \wedge & xs \in \mathbb{N} \rightarrowtail\!\!\!\!\!\twoheadrightarrow name \\
 & & \wedge & (c,\ \#xs) \notin \bigcup(form_funcs(\!|thy|\!)) \\
 & & \wedge & list_all(xs,\ exists(c,\ p)) \in consequences\ thy \\
 & \bullet & list_all(xs,\ exists(c,\ p)) & \quad \} \\
\wedge & (\forall\ xs{:}\ seq\ name;\ c{:}name;\ p{:}wff\ thy & & \\
 & | & list_all(xs,\ exists(c,\ p)) \in dom\ (new_specification\ thy) & \\
 & \bullet & new_specification\ thy\ (list_all(xs,\ exists(c,\ p))) & \\
 & = & thy \cup \{subst(\{c \mapsto app(c,\ var\ o\ xs)\},\ p)\}) &
\end{array}$$

3 Critical Properties of the Proof Tool

We envisage a tool which assists the user in building theories and proving theorems. We now give a rather abstract model of such a tool. It is intended to make visible only those features which are necessary to state the critical requirements on such a tool. For simplicity, we assume that the tool works with a single theory. Extending the present specification to cater for a tool managing a collection of named theories is straightforward.

States The state of the proof tool is a triple, $(thy, defs, thms)$ say. *thy* gives the set of non-logical axioms which have been introduced using one of the two extension mechanisms. *defs* gives the subset of *thy* comprising the axioms introduced using *new_specification*. *thms* records the theorems which have been proved by the user. The following defines the allowable states of the abstract proof tool, which we sometimes refer to as 'abstract states':

z

STATE

$$\begin{array}{ll} thy & : theory; \\ defs & : \mathbb{F}\ form; \\ thms & : \mathbb{F}\ form \end{array}$$

$$defs \subseteq thy \land thms \subseteq wff\ \ thy$$

The Kernel Let us say that a 'kernel' is a transition function equipped with an interpretation function allowing us to view its state space as an abstract state. We formalise this property by the following definition (which is generic in the state space, ST, inputs, IP, and outputs, OP, of the transition function):

z

KERNEL$[ST, IP, OP]$

$$\begin{array}{ll} tr\_f & : (IP \times ST) \to (OP \times ST); \\ int & : ST \to STATE \end{array}$$

We may now formulate two critical properties which we would like the proof system to have.

Critical Property 1 The first critical property is intended to ensure that the tool contains a correct implementation of the rules of inference and to place some constraints on the mechanisms which modify the theory (e.g. it would prevent an operation which deleted an arbitrary axiom). It says that states in which all the alleged theorems are indeed consequences of the axioms are mapped to states with the same property:

z

$[ST, IP, OP]$

derivability_preserving $: \mathbb{P}\ KERNEL[ST,\ IP,\ OP]$

$$\begin{array}{lll} \forall\ ker{:}KERNEL[ST,\ IP,\ OP]\bullet & & \\ & ker \in derivability\_preserving & \\ \Leftrightarrow & (\forall st1,\ st2 : ST;\ ip{:}\ IP;\ op{:}OP \mid (op,\ st2) = ker.tr\_f(ip,\ st1)\bullet & \\ & \quad (ker.int\ \ st1).thms \subseteq\ \ consequences((ker.int\ \ st1).thy) & \\ & \Rightarrow \quad (ker.int\ \ st2).thms \subseteq\ \ consequences((ker.int\ \ st2).thy)) & \end{array}$$

Critical Property 2 The second critical property demands that the tool makes a proper distinction between conservative and axiomatic extensions. It asks that every transition of the tool which changes the definitions component of the abstract state does so via *new_specification*. For the want of a better word, we use the term *standard* for this property.

z

$$
\begin{array}{l}
[ST, IP, OP] \\
\quad \mathbf{standard} : \mathbb{P}\ KERNEL[ST,\ IP,\ OP] \\
\hline
\forall\ ker{:}KERNEL[ST,\ IP,\ OP]\bullet \\
\quad ker \in standard \\
\Leftrightarrow \quad (\forall st1,\ st2\ :\ ST;\ ip{:}\ IP;\ op{:}OP \mid (op,\ st2) = ker.tr\_f(ip,\ st1)\bullet \\
\qquad (ker.int\ st2).defs \neq (ker.int\ st1).defs \\
\quad \Rightarrow \quad (\exists p{:}form\bullet \\
\qquad\qquad (ker.int\ st2).thy = new\_specification\ ((ker.int\ st1).thy)\ p \\
\qquad \wedge \qquad (ker.int\ st2).defs \\
\qquad\qquad = \quad (ker.int\ st1).defs \cup ((ker.int\ st2).thy \setminus (ker.int\ st1).thy)))
\end{array}
$$

4 Design of the Kernel

In this section we give the design of the critical kernel for a very simple proof tool, which, we believe, satisfies the specification we have given. The design deliberately underspecifies certain aspects, and in section 5.2 below we consider how a simple program could be implemented which realises it and also discuss some shortcomings arising from our simplification of the actual work done for HOL.

States In the design we use finite functions over names to represent the sets which appear in the abstract state. We also decide to hold the 'conservative axioms' and the 'axiomatic axioms' separately.

Thus, the state in the design will range over the following set:

z

$$
\begin{array}{l}
\mathbf{C\_STATE} \\
\quad axs,\ defs,\ thms \quad : name \twoheadrightarrow form \\
\hline
\quad ran\ thms \subseteq wff(ran\ axs \cup ran\ defs)
\end{array}
$$

(Here $C\_$ and, later, $c\_$ stand for 'concrete'.)

Interpretation Function This is very straightforward:

z

$$
\begin{array}{l}
\quad \mathbf{int\_c\_state} \quad : C\_STATE \rightarrow STATE \\
\hline
\quad \forall c\_st{:}C\_STATE\bullet \\
\qquad (int\_c\_state\ c\_st).thy = ran\ (c\_st.axs) \cup ran\ (c\_st.defs) \\
\quad \wedge \quad (int\_c\_state\ c\_st).defs = ran\ (c\_st.defs) \\
\quad \wedge \quad (int\_c\_state\ c\_st).thms = ran\ (c\_st.thms)
\end{array}
$$

Inputs and Outputs Since our specification is defined purely in terms of the state of the system, we do not wish to specify the inputs or outputs in detail. We represent them both as given sets:

z

$[\mathbf{C\_IP}, \mathbf{C\_OP}]$

Inference rules The function *infer* maps inputs to state transitions which correspond to various forms of proof activity. None of the transitions computed by *infer* change the axioms and definitions components of the state. To allow *infer* to reject input which is invalid, it may also fail to change the theorems component. Otherwise it adds a new entry in the theorems component associating a formula with some name. The formula is either a logical axiom, or an axiom or definition of the theory, or is obtained by applying *modus ponens* or generalisation to some arguments in which any formulae occur in the theorems component.

z

$$\mathbf{infer} : C\_IP \to C\_STATE \to C\_STATE$$

$$\begin{array}{lll}
\multicolumn{3}{l}{\forall ip{:}C_IP;\ st1,\ st2 : C_STATE \mid st2 = infer\ ip\ st1 \bullet} \\
& & st2.axs = st1.axs \\
\land & & st2.defs = st1.defs \\
\land & (& st2.thms = st1.thms \\
& \lor & (\exists p{:}logical_axioms;\ pn{:}name \bullet \\
& & \quad pn \notin dom(st1.thms) \land st2.thms = st1.thms \cup \{pn \mapsto p\}) \\
& \lor & (\exists p{:}ran(st1.axs) \cup ran(st1.defs);\ pn{:}name \bullet \\
& & \quad pn \notin dom(st1.thms) \land st2.thms = st1.thms \cup \{pn \mapsto p\}) \\
& \lor & (\exists p,\ q{:}form;\ rn{:}name \mid \{p,\ q\} \subseteq ran(st1.thms) \bullet \\
& & \quad rn \notin dom(st1.thms) \land st2.thms = st1.thms \cup \{rn \mapsto mp(p,\ q)\}) \\
& \lor & (\exists p{:}form;\ n,\ qn{:}name \mid p \in ran(st1.thms) \bullet \\
& & \quad qn \notin dom(st1.thms) \land st2.thms = st1.thms \cup \{qn \mapsto gen(n,\ p)\}))
\end{array}$$

Note that there is a close correspondence between the definition of *infer* and the definition of *direct_consequence* in section 2.3 above.

Extending Theories The function *extend* maps inputs to state transitions which introduce new axioms or definitions. Such transitions do not change the theorems component of the state. One or other of *new_axiom* or *new_specification* is used to compute an updated axioms or definitions component. Note that the defining predicate for *C_STATE* ensures that the formula passed to *new_specification* is indeed a consequence of the axioms and definitions of the theory.

z

$$\mathbf{extend} : C\_IP \rightarrow C\_STATE \rightarrow C\_STATE$$

$$\begin{array}{l}
\forall ip{:}C_IP;\ st1,\ st2 : C_STATE \mid st2 = extend\ ip\ st1 \bullet \\
\qquad st2.thms = st1.thms \\
\quad \land \quad (\quad (\exists p{:}form \bullet \\
\qquad\qquad ran(st2.axs) = new_axiom\ (ran(st1.axs) \cup ran(st1.defs))\ p \\
\qquad\qquad\qquad \setminus ran(st1.defs) \\
\qquad\quad \land \quad st2.defs = st1.defs) \\
\qquad \lor \quad (\exists p{:}ran(st1.thms) \bullet \\
\qquad\qquad ran(st2.defs) = new_specification(ran(st1.axs) \cup ran(st1.defs))\ p \\
\qquad\qquad\qquad \setminus ran(st1.axs) \\
\qquad\quad \land \quad st2.axs = st1.axs))
\end{array}$$

Transition Function This is very straightforward:

z

$$\mathbf{c\_trans\_fun} : C\_IP \times C\_STATE \rightarrow C\_OP \times C\_STATE$$

$$\begin{array}{l}
\forall ip{:}C_IP;\ op{:}C_OP;\ st1,\ st2 : C_STATE \mid (op,\ st2) = c_trans_fun\ (ip,\ st1) \bullet \\
\qquad st2 = infer\ ip\ st1 \lor st2 = extend\ ip\ st1
\end{array}$$

Kernel Construction The design of our proof tool is completed by combining the transition and interpretation functions to give an instance of the type *KERNEL*:

z

$$\mathbf{c\_ker} : KERNEL[C\_STATE,\ C\_IP,\ C\_OP]$$

$$c\_ker.tr\_f = c\_trans\_fun \land c\_ker.int = int\_c\_state$$

5 Discussion

5.1 Verification Issues

Now we have given a design as an instance of the type *KERNEL*, we can state the critical requirements for it formally. Thus, the overall correctness proposition for the design is the conjecture:

1. $\quad ?\vdash c\_ker \in derivability\_preserving \cap standard$

We would like to reduce this to conjectures about the inference and extension subsystems. The proof of this reduction of the problem would precede along the following lines:

We would expand conjecture 1 using the definitions of *derivability_preserving*, *standard*, *c_ker*, *int_c_state* and *trans_fun* and then observe that inference and *new_axiom* transitions must be standard (because they do not change the definitions) and that extension transitions preserve derivability (because they do not change the theorems component and do not remove any axioms or definitions). Thus the conjecture reduces to the following conjectures, which, in effect, assert that inference transitions preserve derivability and that *new_specification* transitions are standard.

$$
\begin{array}{lll}
2. & ?\vdash \forall ip{:}C\_IP;\ st1,\ st2 : C\_STATE \mid st2 = infer\ ip\ st1 \bullet & \\
 & \quad ran(st1.thms) \subseteq consequences(ran(st1.axs) \cup ran(st1.defs)) & \\
 & \Rightarrow \quad ran(st2.thms) \subseteq consequences(ran(st2.axs) \cup ran(st2.defs)) &
\end{array}
$$

$$
\begin{array}{llll}
3. & ?\vdash \forall ip{:}C\_IP;\ st1,\ st2 : C\_STATE \mid st2 = extend\ ip\ st1 \bullet & & \\
 & \quad ran(st2.defs) \neq ran(st2.defs) & & \\
 & \Rightarrow \quad (\exists p{:}form \bullet & ran(st2.axs \cup st2.defs) & \\
 & \qquad = & new\_specification\ (ran(st1.axs) \cup ran(st1.defs))\ p & \\
 & \quad \wedge & ran(st2.defs) & \\
 & \qquad = & ran(st1.defs) \cup (ran(st2.axs \cup st2.defs) & \\
 & & \setminus (ran(st1.axs) \cup ran(st1.defs)))) &
\end{array}
$$

Thus, we have reduced the correctness proposition for the complete system to properties of its subsystems taken separately. Continuing this decomposition process would lead fairly quickly to an informal proof of correctness for the system in an example of this kind (probably after one or more rounds of corrections to the design, and perhaps the specification, where the proof attempt revealed flaws). Although the sketch given in the previous paragraph deliberately omits many details (e.g. the justification of the definitions of *infer* and *extend* against the defining property of C_STATE), a fully formal proof using a real-life proof development system such as HOL would be quite feasible.

5.2 Implementation Issues

The actual implementation of the system would follow the LCF paradigm. The kernel would be implemented as an abstract data type in an interactive functional programming language such as Standard ML (see [4]). Implementing the types and functions defined in sections 2.1 and 2.2 is very straightforward in such a language. Assuming this to have been done, and also assuming a type *'a dict* implementing string-indexed lookup tables for items of type *'a* with operations *empty*, *enter* etc., the abstract data type might start as follows:

Standard ML Example

```
abstype theorem = mk_theorem of form
with    local   val c_state:
                        {axs:form dict ref, defs:form dict ref, thms:form dict ref}
                = {axs = ref empty, defs = ref empty, thms = ref empty};
        in      fun new_axiom (n:string, p:form) : theorem = (
                        #axs c_state := enter (n, p)(!(#axs c_state));
                        mk_theorem p
                );
        (* .... *)
        end;
end;
```

The user interacts with the system using ML as a command language (usually referred to as the metalanguage in this context). Thus from the point of view of our design the input to the system would be the metalanguage commands, and the outputs might be taken as the values printed by the ML system. Note that there is no explicit implementation of the functions *infer* and *extend*, which were introduced in the design just to identify a decomposition of the kernel into subsystems. In a fully formal treatment of the implementation, one would have to display an interpretation function describing how the implementation realised the design

One very naive aspect of our design is that it stores the result of every inference in the theory. In practice, the design should allow the inference rules to be coded as functions returning values of type *theorem* and theorems would only be saved when the user made an explicit request for the system to do so. For this reason, it is arguable that the design does not really capture the "essence of LCF". To do this properly the design needs to model the metalanguage store in which the results of inferences reside. Our actual work on HOL addresses this issue as well as others which arise in supporting the features we wish for in a real system (e.g. management of a hierarchy of named theories allowing deletion of theories, and of definitions and axioms within them).

5.3 Supporting Specification

In section 1.3, we mentioned that the conservative extension mechanism was intended to help in treating specifications without prejudicing logical consistency. In this section we sketch how this works in practice and how we arrange to defer the proof obligations until specification work is completed.

The idea is to have (non-critical) code which does a certain amount of automatic proof while the specification is loaded into the system to build the corresponding theory. In our example, one might consider processing Z-like implicit, possibly loose, definitions of the form:

Example

$$\begin{array}{|l} c \\ \hline P \end{array}$$

where c is the name of a new function to be introduced and P is its desired defining property (in which $x_1, \ldots, x_k$ occur free, say). This would be handled by a procedure which automatically proved the following trivial theorem:

Example

$$\vdash \forall x1 \bullet ... \forall xk \bullet \exists c \bullet ((\forall x1 \bullet ... \forall xk \bullet \exists c' \bullet P[c'/c]) \Rightarrow P)$$

Passing this to *new_specification* gives the defining axiom:

Example

$$\vdash \forall x1 \bullet ... \forall xk \bullet \exists c \bullet ((\forall x1 \bullet ... \forall xk \bullet \exists c' \bullet P[c'/c]) \Rightarrow P[c(x1,\ ...\ xk)/c])$$

where the second last c is the new function symbol rather than a variable. This defining axiom is clearly equivalent to $P[c(x1, ...xk)/c]$ if we can prove the *consistency* proposition for the definition, namely the conjecture:

Example

$$?\vdash \forall x1 \bullet ... \forall xk \bullet \exists c' \bullet P[c'/c]$$

Thus a specification may be used to construct a theory by means of conservative extensions each of which introduces a defining axiom which is logically equivalent to the desired axiom provided the corresponding consistency proposition is provable.

In fact, the ideas we have outlined would not be of much use in first order logic, in general. However in Z or in a higher-order system like HOL, it becomes quite powerful. In many useful cases it is possible to automate the proof of the consistency proposition. For example, definitions by recursion over a free type or definitions of a set like the definition of *wff* in section 2.1 above may usefully be handled in this way.

6 Conclusions

Work in progress within ICL to construct a high integrity proof tool is beginning to get a good grip on the problem using methods which we have illustrated in this paper in a much-simplified example. A key aspect of the approach is explicit specification of a formal object modelling the kernel of the system of which one can postulate formally the desired critical properties. A formal verification of the design for the kernel against these critical properties seems feasible and work on this is in progress. In the future, it may be possible to bridge the gap between the design and the implementation by means of a formalisation of the implementation language semantics.

A major part of the problem of how specifications are handled in the proof tool has been solved using the conservative extension mechanism which we have discussed. While this technique caters well for implicit definitions of values, we do not yet have a usable, elegant and effective analogue for implicitly defining types in typed systems such as Z or HOL. Moreover, Z has a facility for a user to introduce an arbitrary predicate as a constraint. It is not yet clear how in practice one can handle all of Z in a conservative way. Work on these topics is in progress.

Acknowledgments

The FST project (IED project 1563) is jointly funded by International Computers Limited and by the Information Engineering Directorate of the UK Department of Trade and Industry.

[1] Michael J.C. Gordon. HOL:A Proof Generating System for Higher-Order Logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*. Kluwer, 1987.

[2] Michael J.C. Gordon, Arthur J. Milner, and Christopher P. Wadsworth. *Edinburgh LCF. Lecture Notes in Computer Science. Vol. 78.* Springer-Verlag, 1979.

[3] Elliot Mendelson. *Introduction to Mathematical Logic.* Wadworth and Brook/Cole, third edition, 1987.

[4] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[5] J.M. Spivey. *The Z Notation: A Reference Manual.* Prentice-Hall, 1989.

[6] DS/FMU/INFRA/001. *Infra Project Overview Document.* K. Blackburn, ICL Secure Systems, WIN01.

[7] INTERIM Defence Standard 00-55 Issue 1. *The Procurement of Safety Critical Software in Defence Equipment.* Ministry of Defence, 5th April 1991.

[8] *The HOL System: Description.* SRI International, 4 December 1989.

Index of Z Names

ERRATUM

I am grateful to Ceri Rees for pointing out an error in the definition of matching for sequences of terms. The condition *disjoint matches* should not be imposed on the last line of the definition: the condition that *match* is a function and that it is the union of the range of *matches* is sufficient. The definition as it stands will, incorrectly, disallow matching of any free variable appearing more than once in a term.