An Environment for Formal Systems

Timothy G. Griffin†
87-846

June 1987

Department of Computer Science
Cornell University
Ithaca, New York 14853-7501

†This work was supported in part by NSF grant no. MCS-83-03327, NSF/ONR grant no. DCR-85-14862, and the British Science and Engineering Research Council.
An Environment for Formal Systems

Timothy G. Griffin*
Cornell University

June 18, 1987

Abstract

This report describes the Environment for Formal Systems, EFS, that allows a user to interactively define the syntax and inference rules of a formal system and to construct proofs in the defined system. The EFS supports two AUTOMATH-like formalisms for encoding logics: the Edinburgh Logical Framework and the Calculus of Constructions. Facilities are provided for the definition of notational abbreviations and the construction of goal-directed proofs. New goal-directed rules can be interactively defined and checked for validity. The EFS was implemented with the Cornell Synthesizer Generator.

1 Introduction

This report describes the EFS, an Environment for Formal Systems, that supports the definition of formal systems and the construction of proofs in systems so defined. The EFS is an interactive environment that employs a declarative, rather than a procedural, style of definition. In particular, the EFS provides features that allow a user to

- define the syntax and inference rules of a formal system,

*This work was supported in part by NSF grant no. MCS-83-03327, NFS/ONR grant no. DCR-85-14862, and The British Science and Engineering Research Council. The author’s e-mail address is tgg@svax.cs.cornell.edu
• define notational abbreviations,
• construct "bottom-up" proofs,
• construct goal-directed "refinement" proofs,
• define new refinement rules that are checked at for validity declaration
time,
• organize theories in a hierarchical structure.

The EFS supports two AUTOMATH [dB80] related formalisms for en-
coding logical systems: the Edinburgh Logical Framework [HHP87] and
the Calculus of Constructions [Coq85,CH85]. These systems employ simi-
lar typed λ-calculi with dependent types to encode expressions, rules and
proofs, albeit using quite different approaches. Readers are assumed to be
somewhat familiar with these systems. An EFS user can choose to work
in the type system of the Edinburgh Logical Framework or in that of the
Calculus of Constructions.

Outline of report. The report is organized as follows. The remainder
of this section provides some background and motivation for this work.
Section 2 is a user's guide to the EFS. It starts with a brief overview of the
Cornell Synthesizer Generator and then proceeds to illustrate the EFS by
developing a fragment of first-order logic. Section 3 discusses related work
and draws some conclusions. Appendix A contains a complete list of the
built-in refinement rules. Appendix B contains an EFS file developed using
the Calculus of Constructions type system.

Background. Many of the features of the EFS were inspired by those
of the Nuprl system, developed at Cornell under the direction of Bates
and Constable [Bat79,BC83,BC85] and fully documented in [Con86]. In-
deed, the EFS was implemented as a means of exploring issues related to a
reimplementation of Nuprl. Nuprl is an interactive environment that sup-
ports the construction and verification of proofs in a formal system closely
related to Martin-Löf's intuitionistic type theory [Mar82]. Nuprl can be
used, among other things, to develop programs from formal specifications.
Nuprl contains many novel features that we believe are essential in support-
ing the interactive construction and verification of proofs regardless of the
formal system involved. These include a syntactic definition mechanism for
notational abbreviations, a structure-oriented proof editor that supports
goal-directed proof construction, a meta-language facility for programming
proof search routines, and a library facility for storing constructed objects
and maintaining dependency relations.

**Goals.** The design and implementation of systems that support the
interactive construction and verification of proofs is becoming an important
area of research in computer science. Given this, our goal is to provide
high-level, logic-independent tools for the specification and generation of
Nuprl-like environments (as characterized by the features outlined above).
In the conclusion we will discuss to what extent the EFS represents progress
toward achieving this goal.

**Representation.** The EFS relies on AUTOMATH-like representations
of mathematical constructs to provide the necessary level of abstraction
that permits a high-level declarative definition of a formal system. The
pioneering work of the AUTOMATH project [dB70,dB80] made substantial
contributions to the development of machine verification of mathematical
proofs. (It should be noted that when we speak of AUTOMATH we are
actually referring to a large family of related languages PAL, AUT-68, AUT-
PI and others, see [vD80]). AUTOMATH was designed to be a framework
general enough to encode most mathematical constructions while making
as few logical commitments as possible. As de Bruijn colorfully stated it in
[dB80]:

The AUTOMATH system is like a big restaurant that serves all
sorts of food: vegetarian, kosher, or anything else the customer
wants. The languages are not tied to any logical system: hardly
any logic has been built in.

Among the many achievements of the AUTOMATH project was the
complete translation and automated checking of Landau’s *Grundlagen* by
Jutting [Jut77] and the development of a metatheory [vD80] that proved the
internal consistency of many of the languages of the AUTOMATH family.

The Calculus of Constructions (CC) extends the expressive power of
AUTOMATH with notions from Martin-Löf’s 1971 theory of types [Mar71]
and Girard’s second-order types [Gir71]. The emphasis of this approach
can be summarized in the *propositions as types* principle (see also Curry
and Howard [CF58,How80]. Unlike AUTOMATH, the Calculus of Constructions is also meant to be a higher-order programming language where programs can be viewed as proofs of their specifications (as in Nuprl). See Mohring [Moh86] for an example of program development and Coquand and Huet [CH85] for an extensive treatment of mathematics development in the Calculus of Constructions.

The Edinburgh Logical Framework (LF) recently proposed by Harper, Honsell and Plotkin [HHP87] offers a clearly defined methodology for encoding a formal system presented in natural deduction [Pra71] style. The LF framework was inspired by Martin-Löf’s treatment of judgements [Mar85] and Schroeder-Heister’s treatment of higher-order rules. The LF provides a well-defined theory of syntax similar to that of Church [Chu40] as well as an approach to encoding rules and proofs that can be summarized in the judgements are types principle. A judgement is identified with the type of its proofs. Basic judgements are defined as constants while hypothetical and schematic judgements are modeled with the function space type constructor. See Avron and Mason [AM87] for extended examples. The development of a fragment of first-order logic in Section 2 closely follows the presentation given in [HHP87].

A thorough comparison of these systems (CC and LF) is beyond the scope of this report, although it is hoped that the EFS will serve as a tool to facilitate such a comparison. We note here only this — that if one is interested in the algorithmic content of a (constructive) logic, an encoding in the Calculus of Constructions can utilize the evaluation mechanism of the EFS for the execution of proofs.

**Implementation.** The author has implemented the EFS with Cornell Synthesizer Generator [RT85]. The Synthesizer Generator is a tool for building full-screen language-based environments from high-level specifications. The Generator was originally designed for implementing language-based environments for programming languages. We hope that this report will help to demonstrate its usefulness in a much wider range of applications. The EFS is among the sample specifications included with the distribution of the Synthesizer Generator, Release 2.0.
2 The EFS — A User’s Guide

This section is a user’s guide to the EFS. For the sake of completeness, it begins with a brief overview of the Synthesizer Generator and outlines some of the features common to all editing environments implemented with the Generator. The EFS is then illustrated through defining a fragment of first-order logic.

Synthesizer Generator Generated Editors. The Synthesizer Generator takes as input a specification written in the Synthesizer Specification Language (SSL) and produces as output a language-based editing environment. An editor specification consists of a series of declarations that define a context-free grammar, display information, parsing rules, context-sensitive constraints, and editor-specific commands. An attribute grammar formalism is used to define context-sensitive constraints. The editor produced is a full-screen editor with many features resembling those of Emacs [Sta]. All generated editors share a common user interface that is extended with editor-specific commands declared in their specification. The reader is referred to the Synthesizer Generator Manual by Reps and Teitelbaum [RT85] for a complete description of SSL as well as the language-independent commands and how they are invoked.

What distinguishes editors created by the Synthesizer Generator from text editors is that each edited file contains a derivation tree of the context-free grammar defined in the editor’s specification. The editor supports multiple buffers, where each buffer contains a single derivation tree. Interacting with the editor amounts to modifying a derivation tree. Each derivation tree is attributed according to the attribute equations of the specification. Attribute values are incrementally updated to achieve a consistent attribution after each modification [DRT81,Rep85]. Such editors have been referred to as “spread-sheets for trees”.

Each editor provides the same set of language-independent commands for manipulating trees, such as cutting and pasting subtrees and for moving from one subtree to another. One subtree, called the current selection, is highlighted in each buffer and represents the user’s focus of interest. The language-specific commands defined in a specification are called transformations. The current selection determines which transformations are applicable.
A derivation tree may contain “holes” that represent unexpanded non-terminals of the grammar productions. These “holes” are often displayed as \( <X> \) (where \( X \) is the unexpanded grammar symbol) and are referred to as placeholders. Many transformations are applicable only when the current selection is a placeholder and we will refer to the act of invoking such a command as “inserting a template”.

Throughout this report we will use the following conventions regarding type styles. The italic style will indicate the name of a command. Typewriter style will be used to indicate the EFS display. Since attribute values contribute to the display of a derivation tree, and are not themselves selectable, we will use boldface type to distinguish selectable placeholders when a displayed object contains attribute values containing placeholders.

The command alternate-unparsing-toggle allows the user to toggle between two different display forms of a derivation tree. The display choice is persistent between a write and a read of a file. Most often this is used to elide the display of a subtree. We will indicate below the constructs that have multiple display forms.

**EFS Terms.** Those EFS terms not containing definition instances (see below) will be referred to a basic terms. In the following grammar \( e \) ranges over basic terms and \( x \) over identifiers:

\[
\begin{align*}
\text{e} := & x \mid (\text{e}) \mid \text{e}(\text{e}) \mid \text{\textbackslash x}:\text{e.e} \mid \text{PI} \ x:\text{e.e} \mid \text{e}\text{-->}\text{e} \mid \text{TYPE}.
\end{align*}
\]

The language described in [HHP87] is defined by a grammar comprised of three levels corresponding to values, types and kinds. We have collapsed these levels into one and implemented a type inference mechanism that sorts out values, types and kinds. The “\(--\)” constructor has been added for the non-dependent function space (\( A\text{-->}B \) represents \( \text{PI} \ x:A.B \) when \( x \) is not free in \( B \)). The similarity of the languages employed in the LF and CC type systems allows us to use the same grammar for both. It is hoped that the translations to the EFS syntax are obvious.

Placeholders for terms and identifiers are displayed as \(<\text{term}>\) and \(<\text{id}>\) respectively. Commands for inserting templates at a \(<\text{term}>\) placeholder are shown in Figure 1. In addition, there are commands for formatting terms. The newline command moves the current selection to the next line, while the indent command moves to the next line and indents one tab stop. Basic terms and identifiers can also be typed directly and parsed when the
<table>
<thead>
<tr>
<th>Command</th>
<th>Template</th>
</tr>
</thead>
<tbody>
<tr>
<td>apply</td>
<td>&lt;term&gt; (&lt;term&gt;)</td>
</tr>
<tr>
<td>lambda</td>
<td>&lt;id&gt;:&lt;term&gt; . &lt;term&gt;</td>
</tr>
<tr>
<td>π</td>
<td>&lt;id&gt;:&lt;term&gt; , &lt;term&gt;</td>
</tr>
<tr>
<td>-- &gt;</td>
<td>&lt;term&gt; -- &gt; &lt;term&gt;</td>
</tr>
<tr>
<td>type</td>
<td>TYPE</td>
</tr>
<tr>
<td>()</td>
<td>(&lt;term&gt;)</td>
</tr>
</tbody>
</table>

Figure 1: Templates for basic terms

The current selection is a <term> or <id> placeholder. A "?" parses to <term>. Entering "? -- > ?" will result in the expression <term> -- > <term> being inserted.

**Error Messages.** The EFS displays error messages when various correctness constraints are violated. These messages are (hopefully) self-explanatory and they will not be explicitly discussed below.

**Files and Chapters.** Figure 2 illustrates an EFS file and an EFS chapter template. Each file consists of a list of named chapters. Comment lines can be entered directly at the <optional comment> placeholder. This placeholder can be removed with the no-comment command. The user can choose the CC type system by selecting the LF and invoking the CC command, or the inverse with the LF command. Although the examples developed below use LF type system, all the features of the EFS described apply within the CC system as well. Error messages are reported in French when using the Calculus of Constructions type system.

Each chapter contains a list of declarations. A chapter template is shown in Figure 2. The <optional import list> placeholder can be removed with the no-imports command or expanded to an identifier list template with the imports command. This provides a simple mechanism for structuring files as hierarchically dependent chapters.

Chapters can be "opened" or "closed" by selecting a chapter’s name and invoking the alternate-unparsing-toggle command. The open display
EFS Version 1
Using Type System : LF
<optional comment lines>

---------------------------------

\textit{chapter}_1
\textit{chapter}_2
\ldots
\textit{chapter}_n

BOTTOM OF FILE

\textbf{CHAPTER <id>}
<optional comment>
<optional import list>
\textbf{BEGIN}
<declaration>
\textbf{END}

---

Figure 2: An EFS file and a chapter template.
mode is the default. In the closed display mode the entire declaration list of a chapter is displayed as “...” to indicate the elision.

**Declarations.** Each chapter contains a list of declarations. Each declaration occurs in the context of a set of environments. There are four such environments:

- The *constant environment* associates constants with their types and is extended by a constant declaration.
- The *value environment* associates a name with a value of some type. It is extended by a value declaration or a refinement.
- The *definition environment* associates a name with a syntactic template.
- The *rule environment* associates a name with a user-defined refinement rule.

The initial value of any environment at the beginning of a chapter is the union of all environments of the same type exported by the chapters named in the import list. The active environments at any declaration in a chapter are the initial environments augmented by the declarations that precede it in the chapter. The environments exported from a chapter are those in effect after the last declaration.

The remainder of this section is devoted to describing the declaration templates that are obtained from the `<declaration>` placeholder by invoking the appropriate command.

**Comments.** The `<declaration>` placeholder can be transformed to a template that provides a way to enter text into a chapter. Comments can extend for any number of lines. The string “***” is appended to the front of each comment line.

**Constants.** A constant declaration associates an identifier with a type. The *constant* command transforms a `<declaration>` placeholder to a template and is displayed as:

```
CONSTANT <id> : <term>
```
CONSTANT o : TYPE

CONSTANT i : TYPE

CONSTANT Absurd : o

CONSTANT Equality : i --> (i --> o)

CONSTANT And : o --> (o --> o)

CONSTANT Implies : o --> (o --> o)

CONSTANT ForAll : (i --> o) --> o

CONSTANT Exists : (i --> o) --> o

CONSTANT True : o --> TYPE

---

Figure 3: Constant declarations.

A constant declaration is well-formed if both the identifier and term are complete, the identifier is unique in the current environment, and the term is a well-formed type expression. A binding is not added to the constant environment unless the constant declaration is well-formed.

We begin our implementation of first-order logic with a declaration of a few syntactic constants and the basic judgement form as shown in Figure 3. (The definition of first-order terms and rules used in this section follows closely the one given in [HHP87].)

**Definitions.** Given the constant declarations above, the mathematical expression "∀x.∃y.x = y" would be represented as

ForAll(\x:i.Exists(\y:i.Equality(x(y)))

Representing mathematics in this notation would quickly become unmanageable. Even if our primitive constructs were nicely displayed, we would still require some mechanism for defining notational abbreviations.
DEFINITION eq
\(<a> = <b> == Equality(a)(b)\)

DEFINITION and
\(<a> & <b> == And(a)(b)\)

DEFINITION implies
\(<a> \Rightarrow <b> == Implies(a)(b)\)

DEFINITION not
\(\neg<a> == (a \Rightarrow Absurd)\)

DEFINITION all
\(\text{all } <x>.<a> == All(\forall x:i.a)\)

DEFINITION some
\(\text{some } <x>.<a> == Exists(\forall x:i.a)\)

DEFINITION true
\(<a> \text{ true } == True(a)\)

---

Figure 4: Definitions for basic constants.
Definition declarations allow for the interactive creation of new templates that function as notational abbreviation for terms. The declaration style has been taken from the definition mechanism that has proved to be indispensable in the Nuprl system (although the implementations differ). The <declaration> placeholder can be transformed to a definition declaration by the definition command:

```
DEFINITION <id>
  <lhs> == <term>
```

The left-hand-side defines the new template as abbreviating the term on the right-hand-side. A completed definition declaration has the form:

```
DEFINITION defname
  s_1 <v_1> s_2 <v_2> ... s_n <v_n> s_{n+1} == T
```

The s_i are strings (possibly null) and the <v_i> designate parameters. (This is the only place where the angle brackets do not signify a placeholder, but are used here so that we are able to distinguish parameters from surrounding text.) Figure 4 contains defined templates for the constants of Figure 3.

The left-hand-side of a definition declaration can be entered in its entirety and parsed. If an s_i is to contain any character from the set {","}> it must escaped with the \ character on entry. The tokens {"n","t","b} have special meaning on the left-hand side of a definition declaration. The \t moves to the current left margin on the next line, while \t and \b move the current left margin left and right one indentation stop respectively. For example, the left-hand-side of the definition declaration

```
DEFINITION let
  let <x> = <a> : <b>
in
  <c>
  ni == (\x:b.c)(a)
```

was produced by typing

```
let <x> = <a>;<b>\nin\nlet <x> = <a>;<b>\nin
```
when the <lhs> placeholder was selected.

Invoking a definition is accomplished in one of two ways:

1. Typing "@defname" when positioned at a <term> placeholder. This will result in the an instance of this definition with all of the <v_i> replaced by <term> or <id> placeholders.

2. Typing "@defname t_1, t_2, ... , t_k" when positioned at a <term> placeholder. Here the t_i represent terms or definition invocations. This will result in an instance of the definition with the first k slots filled in with the t_i from left to right.

For example, typing "@and" in the context of the definitions of Figure 4 will cause the following template

<term> & <term>

to be inserted. Whereas typing "@and A (@implies B))" will result in the term

A & (B => <term>)

being inserted. Currently, there are no facilities for updating the parser to recognize these defined terms, leaving this as the only method of entering them.

Invoking a definition that contains a binding variable will result in a template with all binding variable slots replaced by <id>. For example, invoking the all definition of Figure 4 will produce the template all <id>.<term>. All definition instances are treated as "first-class citizens" in the land of terms. The definition mechanism insures that the left-hand side inherits the binding structure of the right-hand side. The procedures for computing the free variables of a term, substitution, and α-conversion can work directly on a definition instance without expanding the definition. For example, the term

\[ \lambda i. (\lambda x. i.all w. x = w)(w) \]

would β-reduce to the term

\[ \lambda i. all w'.w = w' \]
DEFINITION hyp1  
<j1> |- <j2> == j1 --> j2

DEFINITION hyp2  
<j1>,<j2> |- <j3> == j1 --> j2 |- j3

DEFINITION hyp3  
<j1>,<j2>,<j3> |- <j4> == j1 --> j2,j3 |- j4

DEFINITION rule-scheme1  

DEFINITION rule-scheme2  

DEFINITION rule-scheme3  

Figure 5: Definitions for judgement forms.
CONSTANT AndIntro :
   (A,B). A true, B true |- A & B true

CONSTANT AndElim :
   (A,B,C).
   A & B true, (A true, B true |- C true) |- C true

CONSTANT ImpliesIntro :
   (A,B). (A true |- B true) |- A => B true

CONSTANT ImpliesElim :
   (A,B). A => B true, A true |- B true

CONSTANT OrIntroLeft :
   (A,B). A true |- A | B true

CONSTANT OrIntroRight :
   (A,B). B true |- A | B true

CONSTANT DoubleNeg : (A).¬¬A true |- A true

Figure 6: Inference rules.
where the instance of the definition all has been $\alpha$-converted to avoid capture.

It may be necessary to ascertain which definition a term is an instance of. Selecting a definition instance $d$ and invoking the alternate-unparsing-toggle command will change its display to $[\text{name}:d]$ where name is the definition's name. This is particularly useful for differentiating between instances of two definitions having the same display.

We are now ready to continue our implementation of first-order logic. Figure 5 contains templates declarations for a few judgement forms and Figure 6 contains a some inference rules.

Values. A value declaration allows the user to name a well-formed term. A <declaration> placeholder can be transformed by the value command to the template:

\begin{verbatim}
VALUE <id> == <term>
    : <term>
\end{verbatim}

The displayed attribute <term> is the inferred type of a term inserted at the <term> placeholder and is recomputed whenever the term is modified. For example, a partially completed value declaration might look like this:

\begin{verbatim}
VALUE f == \x:o.\y:o.<term>
    : o --> o --> <term>
\end{verbatim}

After completing the term its type is updated:

\begin{verbatim}
VALUE f == \x:o.\y:o.And(x)(y)
    : o --> o --> o
\end{verbatim}

The binding is not added to the value environment until the identifier and term are complete and the term is found to be well formed. All value declarations are required to have unique names. Figure 7 exhibits a few examples.

Evaluation of terms. The EFS provides a facility for the evaluation of terms. Terms can be evaluated to normal form or to display normal form. A term is reduced to normal form by expanding all value names and definitions and contacting all $\beta$ and $\eta$ redexes. A term is reduced to display
VALUE \( h =: A: o. \overset{- \text{A}}{} 
\)
\[ o \rightarrow o \]

VALUE \( g == \backslash A: o. A \& h(A) 
\)
\[ o \rightarrow o \]

VALUE \( \text{Iff == } \backslash A: o. \backslash B: o. (A \Rightarrow B) \& (B \Rightarrow A) 
\)
\[ o \rightarrow o \rightarrow o \]

Figure 7: Value declarations.

normal form by expanding all value names and non-normal definitions and and contacting all \( \beta \) and \( \eta \) redexes. Definitions are normal by default, but a user may want certain definitions, such as the \text{let} construct defined above, to be expanded in \textit{display normal form}. In such cases the definition declaration's name should be selected and the the \text{alternate-unparsing-toggle} command invoked. This adds \text{NORMAL Yes} to the display. Selecting the \text{Yes} and transforming it to \text{No} will have the desired effect.

The \text{<declaration>} placeholder can be transformed with the \text{normal} command to the template

\text{NORMALIZE} \text{<term>}
\[ \Rightarrow \text{<term>} \]
\[ : \text{<term>} \]

or with the \text{eval} command to the template

\text{EVALUATE} \text{<term>}
\[ \Rightarrow \text{<term>} \]
\[ : \text{<term>} \]

In each case the result of evaluation is displayed along with its type. For example, given the declarations of Figure 7 we have:

\text{NORMALIZE} \text{\backslash \( W: o. g(W) \)}
\[ \Rightarrow \text{\backslash \( W: o. \text{And}(W)(\text{Implies}(\text{Implies}(W)(\text{Absurd}))(\text{Absurd})) \)} \]
\[ : o \rightarrow o \]
whereas with the *eval* command we obtain

\[
\text{EVALUATE } \lambda W: o. g(W) \\
\Rightarrow \lambda W: o. W \land \neg W \\
: o \rightarrow o
\]

which does not expand the definitions *and* and *not*.

**Refinement.** Figure 8 contains two "bottom-up" proofs using some of the declarations of Figures 3-6. Clearly this is not an ideal method for construction proofs.

The EFS supports a *goal-directed* method of constructing terms that adopts the notion of a *refinement* from Nuprl. Using the Nuprl notation, we write a goal as \( H \gg G \), where \( H \) is a well-formed context binding identifiers to type expressions and \( G \) is a type expression. An *achievement* of a goal \( H \gg G \) will be any term \( a \) such that \( a \) has the type \( G \) in the context \( H \). Implicit here is the type system (CC or LF) and the constant and value environments active at the point of the refinement. In a refinement, the user states a goal to be achieved and applies *refinement rules* to decompose it into (usually simpler) subgoals in such a way that achievement of the subgoals entails achievement of the original goal. Figure 9 displays the general form of a refinement rule \( R \). The \( e_j \) represent arguments to the rule \( R \). The \( i^{\text{th}} \) subgoal is represented as \( H_i \gg G_i \) with an achievement \( a_i \). Note that in the case \( G \) represents a judgement, \( H \) can be taken to represent an assumption list and \( a \) to represent a proof that \( G \) holds.

Appendix A contains a complete description of the basic refinement rules provided by the EFS. It should be emphasized that the subgoals of each rule are automatically computed when a refinement rule is invoked.

The *refinement* command transforms a `<declaration>` placeholder to insert the template:

```
REFINEMENT <id>:
  Display achievement : Yes
  \gg <term>
  <refinement>
  [achievement = <term>]
```

The `<term>` is a placeholder for the initial goal (having no assumptions). Refinement rules can be inserted at any `<refinement>` placeholder with the
VALUE First ==
\A:o.\B:o.
  ImpliesIntro(A & B)(A)
  (\p:(A & B true).
    AndElim(A)(B)(A)(p)(\a:A true.\b:B true.a))
  : PI A:o.PI B:o.A & B => A true

VALUE Excluded-Middle ==
\A:o.
  DoubleNeg(A | ~A)
  (ImpliesIntro(~(A | ~A))(Absurd)
   (\p1:(A | ~A) true.
    ImpliesElim(A | ~A)(Absurd)
     (p1)
     (OrIntroRight(A)(~A)
      (ImpliesIntro(A)(Absurd)
       (\p2:A true .
        ImpliesElim(A | ~A)(Absurd)
         (p1)
         (OrIntroLeft(A)(~A)(p2))))))
  : PI A:o.A | ~A true

Figure 8: Two bottom-up proofs.
appropriate command. Transforming the Yes to a No with the no command results in deleting the achievement term from the display.

Figure 10 contains four successive steps in a refinement named simple. In the first step only the goal has been completed. The rule intro* (see appendix A) is applied in the second step resulting in one subgoal, A true in the context A:o;h1:A true. The third step (removal of A true from the goal) demonstrates that the top-level goal can be modified at any time and that these changes will propagate through the refinement. The forth step completes the refinement with the hypothesis rule. Notice that the achievement is built incrementally as the refinement is constructed. The completed refinement has the effect of binding simple to the achievement \A:o.\h1:A true.h1 in the value environment.

Selecting the name of a refinement and invoking the alternate-unparsing-toggle command will elide all but the name and goal of the the refinement. Selecting any rule instance and invoking the alternate-unparsing-toggle command will elide the display of that subtree in the refinement rooted at the rule. In both cases the display is replaced with "..." to indicate the elision.

**User-defined Refinement Rules.** The refinement rules described in appendix A are sufficient for building a term of any legal type. However, they not quite adequate since a user will surely want refinement rules tailored to the specific logic being developing.

The EFS provides a facility for the declarative definition of user-defined
REFINEMENT simple:
Display achievement: No
>> PI A:o.A true --> A true
  <refinement>

REFINEMENT simple:
Display achievement: Yes
>> PI A:o.A true --> A true
  By intro*
    A:o
    h1:A true
    >> A true
    <refinement>
    [achievement = \A:o.\h1:A true.<term>]

REFINEMENT simple:
Display achievement: Yes
>> PI A:o.<term> --> A true
  By intro*
    A:o
    h1:<term>
    >> A true
    <refinement>
    [achievement = \A:o.\h1:<term>.<term>]

REFINEMENT simple:
Display achievement: Yes
>> PI A:o.A true --> A true
  By intro*
    A:o
    h1:A true
    >> A true
      By hypothesis
    [achievement = \A:o.\h1:A true.h1]

Figure 10: Successive stages of a simple refinement proof.
REFINEMENT RULE R

\[ H \gg G \]

By \( R \ s_1 <v_1:P_1> \ s_2<v_2:P_2> \ldots s_n <v_n:P_n> s_{n+1} \)

\[ H_1 \gg G_1 \ \ [a_1] \]

\[ H_2 \gg G_2 \ \ [a_2] \]

\[ \ldots \]

\[ H_n \gg G_n \ \ [a_n] \]

\[ \text{[achievement} = a] \]

Figure 11: General form of a user-defined refinement rule.

refinement rule schemes. In these rule declarations the user must specify the
goal, subgoals, and arguments to a rule as well as a term that achieves the
initial goal given achievements of the subgoals. The validity of each rule is
automatically checked at declaration time with respect to the environments
active at the point of declaration. That is, the type-checking and type-
inference mechanisms of the EFS are used to verify that the type of the
achievement matches the goal.

Figure 11 displays the general form of a user-defined rule declaration.
Here \( H \gg G \) and the \( H_k \gg G_k \) represent the goal and subgoals respectively,
where \( G \) and the \( G_k \) are terms and \( H \) and the \( H_k \) are contexts. The \( v_i \)
are variables that represent argument parameters while the \( P_i \) are terms
that represent the types of these parameters. The \( s_j \) are strings as in the
definition declarations described above. The \( a_i \) are variables that represent
the achievement of the subgoal \( H_i \gg G_i \) while \( a \) is a term that represents
the achievement of the rule \( R \).

The actual subgoals and achievement of a given rule instance within a
refinement are computed by means of pattern matching. The actual goal
and argument types are matched against the patterns in a rule’s declaration.
The bindings of the context \( H \) of a rule declaration indicate which variables
must be instantiated by pattern matching when the rule is invoked. The
term \( G \) and the terms \( P_i \) and \( G_k \) must be patterns. In the following grammar
$p$ ranges over basic patterns and $x$ over variables:

$$p := x | (p) | p(p) | p \rightarrow p | \text{TYPE}.$$ 

A pattern is either a basic pattern or a definition instance that contains no binding variables, or a definition instance that expands to a pattern. The expansion of a definition instance involves a textual replacement of arguments for parameters as well as reducing $\beta$ and $\eta$-redexes of the of the definition’s right-hand side. For example, if $A$ and $P$ are variables then $A & \text{all } x.P(x)$ is a pattern since $A$ is a pattern and the definition and contains no binding variables and the term $\text{all } x.P(x)$ macro expands to $\text{All}(\lambda x:i.P(x))$ which in turn $\eta$-reduces to the pattern $\text{All}(P)$.

Figure 12 contains three user-defined refinement rules derived from constants given in Figure 6. When a user-defined rule is invoked in a refinement the current goal and the types of the actual arguments are matched against the patterns of the rule to obtain a substitution. This substitution is then applied to the rule’s subgoals and achievement to arrive at the actual subgoals and achievement of the refinement step. For example, in the rule and-elim the context declares $A$, $B$ and $C$ to be variables that must be instantiated when a rule is invoked. If and-elim is invoked in a refinement with goal $W \text{ true}$ and an argument $e1$ of type $X \& Y \text{ true}$, it will produce a subgoal $W \text{ true}$ in a context extended with the bindings $h1:X \text{ true}$ and $h2:Y \text{ true}$, where $h1$ and $h2$ are new variables. If $e2$ is the achievement of this subgoal then the rule states that the term

$$\text{AndElim}(X)(Y)(W)(e1)(\lambda h1:X \text{ true}.\lambda h2:Y \text{ true}.e2)$$

will be the achievement of the initial goal.

Figure 13 illustrates the use of the rules of Figure 12 in a refinement named proof1. It should be emphasized again that the subgoals and achievement terms are automatically instantiated when a rule is invoked.
REFINEMENT RULE and-intro
A:o;
B:o
>> A & B true
   By and-intro
   >> A true [a]
   >> B true [b]
[achievement = AndIntro(A)(B)(a)(b)]

REFINEMENT RULE and-elim
A:o;
B:o;
C:o
>> C true
   By and-elim with <w:(A & B true)>
     a:A true
     b:B true
     >> C true [c]
[achievement = AndElim(A)(B)(C)(w)(\a:A true.\b:B true.c)]

REFINEMENT RULE implies-intro
A:o;
B:o
>> A => B true
   By implies-intro
     a:A true
     >> B true [b]
[achievement = ImpliesIntro(A)(B)(\a:A true.b)]

Figure 12: User defined refinement rules.
REFINEMENT proof 1:
Display achievement: No

\[ \text{PI } X : o . \text{PI } Y : o . X \land Y \Rightarrow Y \land X \text{ true} \]

By intro*
\[ X : o ; \]
\[ Y : o \]

\[ \Rightarrow X \land Y \Rightarrow Y \land X \text{ true} \]
By implies-intro
\[ h 1 : X \land Y \text{ true} \]
\[ \Rightarrow Y \land X \text{ true} \]
By and-elim with \( h 1 \)
\[ h 2 : X \text{ true} \]
\[ h 3 : Y \text{ true} \]
\[ \Rightarrow Y \land X \text{ true} \]
By and-intro

\[ \Rightarrow Y \text{ true} \]
By hypothesis
\[ \Rightarrow X \text{ true} \]
By hypothesis

Figure 13: Using user-defined rules.
The refinement proof1 is completed after using only six rules and has the effect of binding the name proof1 to the refinement's achievement term

\[ X:o. \ Y:o. \]
\[ \text{ImpliesIntro}(X \& Y)(Y \& X) \]
\[ (\text{\h1:X \& Y \ true.} ) \]
\[ \text{AndElim}(X)(Y)(Y \& X)(h1) \]
\[ (\text{\h2:X \ true.} \text{\h3:Y \ true.}) \]
\[ \text{AndIntro}(Y)(X)(h3)(h2)) \]

in the value environment.

Invoking a user-defined rule named R is accomplished in one of two ways:

1. Typing “R” when positioned at a <refinement> placeholder.

2. Typing “R \ t_1,t_2, \ldots,t_k” when positioned at a <refinement> placeholder. Here the \ t_i represent terms. This will result in an instance of the refinement rule R with the first k argument slots filled in with the \ t_i from left to right.

The declaration of a user-defined rule is constructed in the following manner. The <declaration> placeholder is transformed by the rule command to insert this template:

```
REFINEMENT RULE <id>
<optional context>
>> <term>
   By <id><optional arguments>
   <optional subgoals>
   [achievement = <term>]`
```

The <optional context> can be removed with the no-context command or replaced with a context template with the context command. The <optional arguments> placeholder can be removed with the no-args command or replaced with a <rule args> placeholder with the args command. The rule argument can be typed directly and parsed while selecting the <rule args> placeholder. The same conventions apply as in the left-hand side of a definition with regard to the special tokens \{",",","",","",","",","",","",","",","",","",","",","",","",","",","",","",","",","",","",","",","",","",","",","",","",","",","",","",","",","",","",","",","",","",","",","","\}.

26
<optional subgoals> placeholder can be removed with the no-subgoals command or replaced with subgoal list template with the subgoals command. Each subgoal template has the form

<optional assumptions>
>> <term> [<id>]

where the <optional assumptions> placeholder can be removed with the no-assumptions command or replaced with a context template with the assumptions command.

Since this is a somewhat tedious process, the EFS provides a method of automatically generating such rules. Selecting a <declaration> placeholder and typing "make-rule name" invokes a mechanism that attempts to automatically construct a rule from the type of name (an identifier). Failure to generate a rule will result in the insertion of an incomplete rule declaration template.

3 Related Work and Conclusions.

Reps and Alpern [RA84] introduced a scheme for representing goal-directed proofs as an attribute grammar having one grammar production for each refinement rule and having the relationship between goals and subgoals defined by attribute equations. Proof editors implemented using this scheme allow for a flexible style of interactive proof construction. Goal-directed proof trees can pass through inconsistent states and contain incomplete terms. For example, a proof can proceed even when the initial goal is incomplete. The Reps-Alpern approach has been used with good results by the author in previous implementations and by Brian Ritchie in Burstall's IPE [Bur86].

We feel that our main contribution is in extending the Reps-Alpern approach while maintaining the flexible proof editing model that makes it so attractive. First, we have devised a general scheme for implementing editors with the Synthesizer Generator in such a way that the collection of templates available to the user can be interactively extended. Second, this scheme was applied to the specification of the EFS to provide the definition and the user-defined refinement rule facilities. Thus, the set of syntactic
templates and refinement rules are not fixed at editor specification time. Rather than write a new SSL specification for every formal system that we wish to implement, we can instead define such systems interactively and at a higher level of abstraction using the EFS (assuming, of course, that the formal system we wish to implement can be encoded with the EFS tools using one of the two type systems that it supports). The EFS syntactic definition mechanism is an improvement over Nuprl’s in that substitution, α-conversion and free variable checks can be performed directly on definition instances without expansion.

A great deal is gained by utilizing an AUTOMATH-style representation for formal systems. Routines such as substitution, α-conversion and free variable checks, are taken care of once and for all by the implementation of a λ-calculus in the EFS. It is not necessary to write hand coded routines that check whether expressions or proofs are well formed. Instead, this is handled by the type checking and type inference mechanisms of the typed λ-calculus that have been implemented as a part of the EFS. Rather than writing low-level procedural descriptions of syntax, rules and proofs, the EFS user is able to specify these in a declarative fashion using a high-level notation.

The Synthesizer Generator greatly simplified the task of implementing the EFS. Implementing it from scratch would have been a large undertaking involving the coding of routines related to buffer management, object display, object manipulation, evaluation and propagation of context sensitive constraints, and many other facilities required to support such a system. The Generator “factors out” many such language-independent concerns, leaving the specification writer to concentrate on the language-dependent aspects.

An analogy between the Synthesiser Generator and AUTOMATH suggests itself in that both abstract away from the details of their domains (language-based editors and mathematical formalisms respectively) to provide elegant descriptive frameworks. The extent to which we have made progress toward the goal outlined in the introduction could surely be measured by the mileage we get out of these abstractions.

The EFS has been used to define several systems including sorted and unsorted first-order logics, Church’s higher-order logic [Chu40,HHP87], a fragment of type theory [Mar82], a Hoare-style logic and the modal logic
In addition, the EFS has been used to define several small programming languages utilizing the ideas of Landin [Lan66,Lan65] and Reynolds [Rey81] for representing programming language constructs using a $\lambda$-calculus. Thus, at least to some extent the EFS can be used to interactively define editing environments for programming languages. The time required for the interactive definition of the basic constructs in each of the examples mentioned above was on the order of hours rather than weeks or months.

Perhaps the biggest drawback of the EFS is its lack of a programmable meta-language. Systems such as Nuprl [CKB85], Edinburgh LCF [GMW79], or the Calculus of Constructions implementation described in [CH85], use the programming language ML as a meta-language to support facilities for writing very general proof search routines. It should be noted, however, that the user-defined refinement rules represent a very restricted class of tactics. The EFS provides a high-level notation for declaring such tactics, as well as a mechanism for verifying their validity at declaration time. Future work will address the incorporation of a meta-language into an EFS-like system.

Another possible drawback of the EFS concerns the question of efficiency. Even if we ignore the issue of tactics, it is not yet clear that the EFS would be able to support the kind of very large theory building that has been pursued in the Nuprl system [How86,How87]. Two areas of future work will address this problem. First, the current EFS is a prototype and represents work in progress — considerable speedup can be gained from the redesign of its SSL implementation. Second, the Synthesizer Generator project is currently directing substantial effort toward increasing the efficiency of generated editors (see for example [Hoo87]).

Finally, it should be noted that our insistence upon supporting both the LF and CC systems, and not tailoring the implementation of the EFS to the particulars of any one system, may well have prevented us from more fully exploiting the potential of either approach.

Acknowledgments. This work was made possible by support from the Nuprl and Synthesizer Generator projects at Cornell, and the LF project at the Laboratory for the Foundations of Computer Science, Edinburgh University. I am grateful to the LFCS group for their encouragement and support during my stay in Edinburg — special thanks to Arnon Avron,
Robert Harper, Furio Honsell, and Ian Mason. I would like to thank Robert Constable and Tim Teitelbaum for many helpful suggestions concerning the presentation of this report. Many members of the Synthesizer Generator project provided invaluable guidance and advice without which the EFS could not have been implemented — special thanks to Roger Hoover in this regard. Stuart Allen, Micah Beck, Jim Hook, Doug Howe, Nils Klarlund, Todd Knoblock, and Carla Marceau provided numerous valuable comments on preliminary drafts of this report.

4 Appendices

A Refinement Rules

This appendix lists the built-in refinement rules of the EFS. These rules are divided into two groups. The names of the basic rules appear on the transformation menu when positioned at a <refinement> placeholder. The extended rules require the user to type a special symbol ( ! ) and an argument at the <refinement> placeholder.

We use a Nuprl-style notation to represent a refinement rule $R$ as in Figure 9. In an actual EFS refinement only the achievement of the initial goal can be displayed. With each rule the condition(s) of its valid application is(are) stated. The only conditions not stated explicitly are those concerning the form of a goal or an arguments type — these conditions are implicit in the presentation of the rules. By the condition $A = B$ we mean that $A$ and $B$ are $\alpha \beta \eta$-equal. The condition "$x : A$ in $H$" means that the binding $x : A$ occurs in the list $H$. The condition $H \vdash a : A$ means that the term $a$ has type $A$ in the context $H$. Implicit in these conditions are the type system (CC or LF) and the constant and value environments active at the point of the refinement. The metavariables $h$ and $h_i$ represent fresh variables.

Basic Rules.

• named-hyp

  Conditions: $x : G'$ in $H$ and $G = G'$

  $H >> G$                $[x]$
By hypothesis <x>

• hyp
  Conditions: x:G' in H and G = G'
  Note: This rule selects the last such x in G.

  H >> G
  [x]
  By hypothesis

• intro

  H >> PI x:A.B
  [\x:A.e]
  By intro
  H,x:A >> B
  [e]

  H >> A --> B
  [\h:A.e]
  By intro
  H,h:A >> B
  [e]

• intro*
  Invoking this rule is the same as repeatedly invoking intro until the
  goal type is not of the form PI x:A.B or A --> B.

• -->analysis
  Condition: H |- e:A --> B

  H >> G
  [e2[e(e1)/h]]
  BY analysis of <e>
  H >> A
  [e1]
  H,h:B >> G
  [e2]

• pi-analysis
  Conditions: H |- e : PI x:A.B , H |- e1 : A' , A = A'
H >> G \quad [e2[e(e1)/h]]

By pi-analysis on \langle e \rangle with \langle e1 \rangle
H,h:B[e1/x] >> G \quad [e2]

• cut
  Condition : H |- A:TYPE

H >> G \quad [((\h:A.e2)(e1))]
by cut with \langle A \rangle
  H >> A \quad [e1]
  H,h:A >> G \quad [e2]

• fact
  Condition : H |- e1:A

H >> G \quad [((\h:A.e2)(e1))]
BY fact \langle e1 \rangle
  H,h:A >> G \quad [e2]

• def
  Condition: G' is G with outermost instance of definition “i” expanded.

H >> G \quad [e]
By expanding definition \langle i \rangle
  H >> G' \quad [e]

• hyp-def
  Condition: j:A in H and A' is A with outermost instance of definition “i” expanded.

H >> G \quad [e[j/h]]
By expanding definition \langle i \rangle in \langle j \rangle
  H,h:A' >> G \quad [e]

• eval
  Note : G' is the display-normal-form of G.
H \gg G \quad [e]
By evaluation
H \gg G' \quad [e]

- hyp-eval
  Condition: \( j : A \) in H and \( A' \) is the display-normal-form of A

H \gg G \quad [e[j/h]]
By evaluation \( \langle j \rangle \)
H, h : A' \gg G \quad [e]

- equiv
  Condition: \( G = G' \)

H \gg G \quad [e]
By equivalence with \( \langle G' \rangle \)
H \gg G' \quad [e]

- explicit
  Conditions: \( H \vdash e : G', G = G' \)

H \gg G \quad [e]
By explicit use of \( \langle e \rangle \)

- gen
  Condition: \( H \vdash e_1 : A \) and \( e_2(e_1) = G \) and \( h : A \) in H

H \gg G \quad [e[e_1/h]]
By gen with \( \langle h \rangle \), \( \langle e_1 \rangle \), \( \langle e_2 \rangle \)
H \gg e_2(h) \quad [e]

- auto

**Extended Commands.** These rule must be entered by typing \( \langle j \rangle \) when the cursor is positioned at the \( \langle \)refinement\( \rangle \). The \( j \) should be variable of H.
• \( \text{H} \vdash \ \text{hj:} \text{A}_1 \rightarrow \text{A}_2 \rightarrow \ldots \rightarrow \text{A}_n. \)

\[
\begin{align*}
\text{H} & \gg \text{C} & \text{[en[j(e1)(e2)\ldots(e(n-1))/h]]} \\
\text{BY analysis of } \langle j \rangle & \\
\text{H} & \gg \text{A}_1 & \text{[e1]} \\
\text{H} & \gg \text{A}_2 & \text{[e2]} \\
\vdots & \\
\vdots & \\
\text{H} & \gg \text{A}(n-1) & \text{[e(n-1)]} \\
\text{H}, \text{h:An} & \gg \text{C} & \text{[en]} \\
\end{align*}
\]

• \( \text{H} \vdash \ \text{Pi}\ x_1: \text{A}_1. \ \ldots \ \text{Pi}\ x_n: \text{A}_n. \text{B} \ \text{in H and H} \vdash \text{e}_i: \text{A}_i \ \text{for } \ i = 1, \ldots, n. \)

\[
\begin{align*}
\text{H} & \gg \text{G} & \text{[e[j(e1)(e2)\ldots(en)/h]]} \\
\text{By pi-analysis* on } \langle j \rangle \ \text{with } \langle e1>, \langle e2>, \ldots, \langle en > \\
\text{H}, \text{h.B[e1,e2,\ldots,en/x1,x2,\ldots,xn]} & \gg \text{G} & \text{[e]} \\
\end{align*}
\]
B  A Sample File

This appendix contains a short EFS file developed using the Calculus of Constructions type system. It develops a few examples that are taken from Coquand and Huet [CH85]. The reader is invited to compare the refinement style, in the EFS, to the "bottom-up" style of proof construction in the presentation of Coquand and Huet.

EFS - Version 1
Using Type System : CC
** This file contains a demonstration of the EFS using the
** Calculus of Constructions type system.

CHAPTER Basic-Definitions
BEGIN

** Higher Order Judgements

DEFINITION hyp1
<j1> |- <j2> == j1 --> j2

DEFINITION hyp2
<j1>, <j2> |- <j3> == j1 --> j2 |- j3

DEFINITION hyp3
<j1>, <j2>, <j3> |- <j4> == j1 --> j2, j3 |- j4

DEFINITION hyp4
<j1>, <j2>, <j3>, <j4> |- <j5> == j1 --> j2, j3, j4 |- j5

** Schematic types

DEFINITION schemel
(<<t>>).j == PI t:TYPE.j

DEFINITION scheme2
(<<t1>, <<t2>>).j == PI t1:TYPE.(t2).j

DEFINITION scheme3
(<<t1>, <<t2>, <<t3>>).j == PI t1:TYPE.(t2, t3).j

DEFINITION scheme4
(<<t1>, <<t2>, <<t3>, <<t4>>).j == PI t1:TYPE.(t1, t2, t3).t4

DEFINITION tschemel
(<<x:t>>).j == PI x:t.j

DEFINITION tscheme2
(<<x1:<<t1>, <<x2:<<t2>>>>).j == PI x1:t1.(x2:t2).j

DEFINITION tscheme3
(<<x1:<<t1>, <<x2:<<t2>, <<x3:<<t3>>>>).j == PI x1:t1.(x2:t2, x3:t3).j

DEFINITION tscheme4
(<<x1:<<t1>, <<x2:<<t2>, <<x3:<<t3>, <<x4:<<t4>>>>).j == PI x1:t1.(x2:t2, x3:t3, x4:t4).j

END CHAPTER Basic-Definitions
VALUE Equality == \A:TYPE.\x:A.\y:A.\p:A.\p:A \p(A --> TYPE).\p(x) --> \p(y)  
                  : PI A:TYPE.A --> A --> TYPE

DEFINITION eq
<e1> = <e2> in <T> == Equality(T)(e1)(e2)

REFINEMENT eq-is-reflexive :
Display Achievement : No
>> (A).(x:A).x = x in A
By intro
  A : TYPE;
  x : A;
  P : (A --> TYPE);
  h1 : P(x)
>> P(x)
  By hypothesis

REFINEMENT RULE eq-is-reflexive
T : TYPE;
  e : T
>> e = e in T
  By eq-is-reflexive
  [achievement : eq-is-reflexive(T)(e)]

REFINEMENT eq-is-symmetric :
Display Achievement : No
>> (A).(x:A, y:A).x = y in A |- y = x in A
By intro
  A : TYPE
>> (x:A, y:A).x = y in A |- y = x in A
By intro
  x : A
>> (y:A).x = y in A |- y = x in A
By intro
  y : A
>> x = y in A |- y = x in A
By intro
  h4 : x = y in A
>> y = x in A
By expanding definition eq in h4
  h5 : PI P:(A --> TYPE).\p(x) --> \p(y)
>> y = x in A
By pi-elim on h5 with \z:A.z = x in A
  h6 : x = x in A --> y = x in A
>> y = x in A
By -->elim on h6
>> x = x in A
  By eq-is-reflexive
  h7 : y = x in A
>> y = x in A
  By hypothesis
REFINEMENT eq-is-transitive:
Display Achievement: No
>> (A). (x:A, y:A, z:A). x = y in A, y = z in A \implies x = z in A
By intro:
A : TYPE;
x : A;
y : A;
z : A;
h1 : x = y in A;
h2 : y = z in A;
P : (A \rightarrow TYPE);
h3 : P(x)
>> P(z)
By expanding definition eq in h1
h9 : PI P : (A \rightarrow TYPE). P(x) \rightarrow P(y)
>> P(z)
By pi-elim on h9 with P
h10 : P(x) \rightarrow P(y)
>> P(z)
By \rightarrow elim on h10
>> P(x)
By hypothesis
h11 : P(y)
>> P(z)
By expanding definition eq in h2
h12 : PI P : (A \rightarrow TYPE). P(y) \rightarrow P(z)
>> P(z)
By pi-elim on h12 with P
h13 : P(y) \rightarrow P(z)
>> P(z)
By \rightarrow elim on h13
>> P(y)
By hypothesis
h14 : P(z)
>> P(z)
By hypothesis

REFINEMENT eq-of-application:
>> (A). (x:A, y:A, f:(A \rightarrow A)). x = y in A \implies f(x) = f(y) in A
...

REFINEMENT RULE eq-is-symmetric
T : TYPE;
el : T;
e2 : T
>> el = e2 in T
By eq-is-symmetric
>> e2 = el in T [p]
[achievement : eq-is-symmetric(T)(e2)(el)(p)]

REFINEMENT RULE eq-is-transitive
T : TYPE;
el : T;
e2 : T
>> el = e2 in T
By eq-is-transitive with \langle e3 : T \rangle
>> el = e3 in T [p1]
>> e3 = e2 in T [p2]
[achievement : eq-is-transitive(T)(el)(e3)(e2)(p1)(p2)]
REFINEMENT RULE substitution
A : TYPE;
  e1 : A
  >> Q(e1)
    By substitution with <e2 : A> over <Q : A --> TYPE>
    >> e2 = e1 in A [p1]
    >> Q(e2) [p2]
    [achievement : p1(Q)(p2)]

REFINEMENT RULE eq-of-application
T : TYPE;
g : T --> T;
e1 : T;
e2 : T
>> g(e1) = g(e2) in T
    By eq-of-application
    >> e1 = e2 in T [p1]
    [achievement : eq-of-application(T)(e1)(e2)(g)(p1)]

END CHAPTER Equality

CHAPTER Product
** Definition of conjunction and related rules.
Import Chapter(s) : Basic-Definitions
BEGIN

VALUE And == \A:TYPE.\B:TYPE.\C:TYPE.\(A --> B --> C) --> C
          : TYPE --> TYPE --> TYPE

DEFINITION and
<a> & <b> == And(a)(b)

REFINEMENT and-intro :
Display Achievement : No
>> (A, B).A, B |- A & B
By intro\x
A : TYPE;
B : TYPE;
h1 : A;
h2 : B;
C : TYPE;
h3 : (A --> B --> C)
>> C
By -->elim\x on h3
  >> A
    By hypothesis
  >> B
    By hypothesis
  h7 : C
  >> C
    By hypothesis

REFINEMENT and-elim :
>> (A, B, C).A & B, (A, B |- C) |- C
...

38
REFINEMENT RULE and-intro
A : TYPE;
B : TYPE
>> A & B
  By and-intro
  >> A [a]
  >> B [b]
[achievement : and-intro(A)(B)(a)(b)]

REFINEMENT RULE and-elim
A : TYPE;
B : TYPE;
C : TYPE
>> C
  By and-elim <\omega : A & B>
  a : A;
  b : B
  >> C [c]
[achievement : and-elim(A)(B)(C)(\omega)(\{a:A.\ b:B.c\})]

END CHAPTER Product

CHAPTER Implication
** Implication is defined in terms of -->,
** we do not need to define intro and elim rules
** for --> since we get these "for free" with the
** into and -->elim rules of the EFS.
Import Chapter(s) : Product, Equality
BEGIN

DEFINITION implies
<a> --> <b> == a --> b

DEFINITION iff
<a> <-> <b> == (a --> b) & (b --> a)

REFINEMENT RULE iff-intro
A : TYPE;
B : TYPE
>> A <-> B
  By iff-intro
  i : A
  >> B [b]
  j : B
  >> A [a]
[achievement : and-intro(A --> B)(B --> A)(\{i:A.b\}(\{j:B.a\}])
The refinement iff-test demonstrates the iff-intro and eq-is-symmetric refinement rules defined above. Note that the achievement term is displayed.

REFINEMENT iff-test :
Display Achievement : Yes
>> (A).(el:A, e2:A).(el = e2 in A) <=> (e2 = el in A)
By intro
A : TYPE
>> (el:A, e2:A).(el = e2 in A) <=> (e2 = el in A)
By intro
el : A
>> (el = e2 in A) <=> (e2 = el in A)
By intro
el : A
>> (el = e2 in A) <=> (e2 = el in A)
By iff-intro
h4 : (el = e2 in A)
>> (e2 = el in A)
By eq-is-symmetric
>> el = e2 in A
By hypothesis
h4 : (e2 = el in A)
>> (el = e2 in A)
By eq-is-symmetric
>> e2 = el in A
By hypothesis

ACHIEVED WITH iff-test =
\(\forall A:TYPE.\bin A:A.\bin e2:A.\)
\(\text{and-intro}((el = e2 in A) \Rightarrow (e2 = el in A))\)
\(\text{eq-is-symmetric}(A)(el)(e2)(h4)\)
\(\text{eq-is-symmetric}(A)(e2)(el)(h4)\)

END CHAPTER Implication

CHAPTER Quantifiers
**
Import Chapter(s) : Basic-Definitions, Equality
BEGIN

as with implication, we get the intro and elim rules of "all" for free.

DEFINITION all
all <x>:<a>,<b> == PI x:a.b

VALUE Sigma == \(\forall A:TYPE.\bin B:A --\rightarrow TYPE.(C).((x:A).B(x) \mid - C) \mid - C\)
: PI A:TYPE.(A --\rightarrow TYPE) --\rightarrow TYPE

DEFINITION exists
Exists <x>:<a>.<b> == Sigma(a)(\(\forall x:a.b\))

40
REFINEMENT exists-intro :
Display Achievement : No
>> (A).(B:A --> TYPE, a:A).B(a) | - Exists x:A.B(x)
By intro*
A : TYPE;
B : A --> TYPE;
a : A;
h1 : B(a);
C : TYPE;
h2 : ((x:A).B(x) |- C)
>> C
By pi-elim on h2 with a
h7 : B(a) |- C
>> C
By -->elim on h7
>> B(a)
By hypothesis
h8 : C
>> C
By hypothesis

REFINEMENT exists-elim :
...

REFINEMENT RULE exists-intro
T : TYPE;
P : T --> TYPE
>> Exists x:T.P(x)
   By exists-intro with <w : T>
>> P(w) [p]
[achievement : exists-intro(T)(P)(w)(p)]

REFINEMENT RULE exists-elim
A : TYPE;
B : TYPE;
P : A --> TYPE
>> B
   By exists-elim with <w : Exists x:A.P(x)>
   h : A;
p : P(h)
>> B [pr]

REFINEMENT exists-example :
Display Achievement : No
>> (A).Exists f:A --> A.all a:A.f(a) = a in A
By intro
A : TYPE
>> Exists f:A --> A.all a:A.f(a) = a in A
By exists-intro with \z:A.z
>> all a:A.a = a in A
By intro
a : A
>> a = a in A
By eq-is-reflexive

END CHAPTER Quantifiers
CHAPTER Fix-Point-Theory

Import Chapter(s) : Equality, Product, Implication, Quantifiers

BEGIN

VALUE FixPt == \A:TYPE. \f:(A --> A). \x:A. f(x) = x in A
   : PI A:TYPE. (A --> A) --> A --> TYPE

VALUE Commute == \A:TYPE. \f:(A --> A). \g:(A --> A). all a:A. g(f(a)) = f(g(a)) in A
   : PI A:TYPE. (A --> A) --> (A --> A) --> TYPE

VALUE Unique == \A:TYPE. \P:(A --> TYPE). \x:A. P(x) & all y:A. P(y) => x = y in A
   : PI A:TYPE. (A --> TYPE) --> A --> TYPE

VALUE Iterate == \A:TYPE. \f:(A --> A). \g:(A --> A).
   PI P:(A --> A) --> TYPE.
   P(f) --> (PI h:(A --> A). P(h) --> (P(\x:A. f(h(x)))) --> P(g)
   : PI A:TYPE. (A --> A) --> (A --> A) --> TYPE

DEFINITION fix
<f> has fixed point <a>:<A> == FixPt(A)(f)(a)

VALUE Fix2 == \A:TYPE. \f:(A --> A). \a:A.f has fixed point a:A
   : PI A:TYPE. (A --> A) --> A --> TYPE

DEFINITION unique-fix
<f> has unique fixed point <a>:<A> == Unique(A)(Fix2(A)(f))(a)

DEFINITION comm
<f> commutes with <g> over <A> == Commute(A)(f)(g)

DEFINITION iter
<g> is an iterate of <f> over <A> == Iterate(A)(f)(g)

VALUE Compose == \A:TYPE. \f:(A --> A). \g:(A --> A). \a:A. f(g(a))
   : PI A:TYPE. (A --> A) --> (A --> A) --> A --> A

REFINEMENT RULE iter-elim
A : TYPE;
\ f : A --> A;
\ g : A --> A
>> P(g)
   By iter-elim with \w : (g is an iterate of f over A)> and \p : (A --> A) --> TYPE
   >> P(f)
   [p1]
   h : A --> A;
   p2 : P(h)
   >> P(Compose(A)(f)(h))
   [p3]

REFINEMENT Lemma1 :
Display Achievement : No
   \ g commutes with f over A & Exists x:A.g has unique fixed point x:A =>
   Exists y:A.f has fixed point y:A
By intro
A : TYPE
   \ g commutes with f over A & Exists x:A.g has unique fixed point x:A =>
   Exists y:A.f has fixed point y:A

42
By intro
f : A --> A
>> (g:A --> A).
g commutes with f over A & Exists x:A.g has unique fixed point x:A =>
   Exists y:A.f has fixed point y:A
By intro
g : A --> A
>>
g commutes with f over A & Exists x:A.g has unique fixed point x:A =>
   Exists y:A.f has fixed point y:A
By intro
h4 : g commutes with f over A & Exists x:A.g has unique fixed point x:A
>> Exists y:A.f has fixed point y:A
   By and-elim h6
   h5 : g has unique fixed point h7:A
>> Exists y:A.f has fixed point y:A
   By exists-intro with h6
   h7 : A;
   h8 : g has unique fixed point h7:A
>> Exists y:A.f has fixed point y:A
   By exists-intro with h7
>> f has fixed point h7:A
   By expanding definition fix
   >> f(h7) = h7 in A
   By expanding definition unique-fix in h8
   h9 : g has fixed point h7:A &
       all y:A.g has fixed point y:A => h7 = y in A
>> f(h7) = h7 in A
   By and-elim h9
   h10 : g has fixed point h7:A;
   h11 : all y:A.g has fixed point y:A => h7 = y in A
>> f(h7) = h7 in A
   By cut with g(f(h7)) = f(h7) in A
>> g(f(h7)) = f(h7) in A
   By eq-is-transitive with f(g(h7))
   >> g(f(h7)) = f(g(h7)) in A
   By expanding definition comm in h5
   h12 : all a:A.f(g(a)) = g(f(a)) in A
>> g(f(h7)) = f(g(h7)) in A
   By pi-elim on h12 with h7
   h13 : f(g(h7)) = g(f(h7)) in A
>> g(f(h7)) = f(g(h7)) in A
   By eq-is-symmetric
   >> f(g(h7)) = g(f(h7)) in A
   By hypothesis
>> f(g(h7)) = f(h7) in A
   By eq-of-application
   >> g(h7) = h7 in A
   By expanding definition fix in h10
   h12 : g(h7) = h7 in A
>> g(h7) = h7 in A
   By hypothesis
   h12 : g(f(h7)) = f(h7) in A
>> f(h7) = h7 in A
   By pi-elim on h11 with f(h7)
   h13 : g has fixed point f(h7):A => h7 = f(h7) in A
>> f(h7) = h7 in A
   By -->elim on h13
   >> g has fixed point f(h7):A
   By expanding definition fix
   >> g(f(h7)) = f(h7) in A
By hypothesis
h14 : h7 = f(h7) in A
>> f(h7) = h7 in A
By eq-is-symmetric
>> h7 = f(h7) in A
By hypothesis

REFINEMENT comm-is-reflexive :
Display Achievement : No
>> (f:A --> A), f commutes with f over A
By intro
f : A --> A
>> f commutes with f over A
By expanding definition comm
>> all a:A, f(f(a)) = f(f(a)) in A
By intro
a : A
>> f(f(a)) = f(f(a)) in A
By eq-is-reflexive

REFINEMENT Lemma2 :
Display Achievement : No
>> (f:A --> A, g:A --> A),
   (g is an iterate of f over A) => (f commutes with g over A)
By intro
f : A --> A
>> (g:A --> A),
   (g is an iterate of f over A) => (f commutes with g over A)
By intro
g : A --> A
>> (g is an iterate of f over A) => (f commutes with g over A)
By intro
h4 : (g is an iterate of f over A)
>> (f commutes with g over A)
By expanding definition comm
>> (all a:A, g(f(a)) = f(g(a)) in A)
By intro
a : A
>> g(f(a)) = f(g(a)) in A
By iter-elim with h4 and \(z:(A --> A), z(f(a)) = f(z(a))\) in A
>> f(f(a)) = f(f(a)) in A
By eq-is-reflexive
h6 : A --> A;
h7 : h6(f(a)) = f(h6(a)) in A
>> f(h6(f(a))) = f(f(h6(a))) in A
By eq-of-application
>> h6(f(a)) = f(h6(a)) in A
By hypothesis

REFINEMENT Andrews_Lemma :
Display Achievement : No
>> (f:A --> A, g:A --> A),
g is an iterate of f over A & Exists x:A, g has unique fixed point x:A =>
   Exists y:A, f has fixed point y:A
By intro
A : TYPE
  g is an iterate of f over A & Exists x:A.g has unique fixed point x:A =
  Exists y:A.f has fixed point y:A
By intro
f : A --> A
>> (g:A --> A).
  g is an iterate of f over A & Exists x:A.g has unique fixed point x:A =
  Exists y:A.f has fixed point y:A
By intro
g : A --> A
>> g is an iterate of f over A & Exists x:A.g has unique fixed point x:A =
  Exists y:A.f has fixed point y:A
By intro
h4 : g is an iterate of f over A & Exists x:A.g has unique fixed point x:A =
  Exists y:A.f has fixed point y:A
By and-elim h4
h5 : g is an iterate of f over A;
  h6 : Exists x:A.g has unique fixed point x:A
  >> Exists y:A.f has fixed point y:A
    By cut with f commutes with g over A
    >> f commutes with g over A
    By fact Lemma2
    h7 : (A),(f:A --> A, g:A --> A).
      (g is an iterate of f over A) => (f commutes with g over A)
      >> f commutes with g over A
      By pi-elim on h7 with A,f,g
    h8 : (g is an iterate of f over A) => (f commutes with g over A)
      >> f commutes with g over A
      By --elim on h8
      >> (g is an iterate of f over A)
      By hypothesis
    h9 : (f commutes with g over A)
      >> f commutes with g over A
      By hypothesis
    h7 : f commutes with g over A
  >> Exists y:A.f has fixed point y:A
By fact Lemma1
h8 : (A),(f:A --> A, g:A --> A).
  g commutes with f over A &
  Exists x:A.g has unique fixed point x:A =>
  Exists y:A.f has fixed point y:A
  >> Exists y:A.f has fixed point y:A
By pi-elim on h8 with A,f,g
h9 : g commutes with f over A &
    Exists x:A.g has unique fixed point x:A =>
    Exists y:A.f has fixed point y:A
  >> Exists y:A.f has fixed point y:A
By --elim on h9
  >> g commutes with f over A & Exists x:A.g has unique fixed point x
    By and-intro
    >> g commutes with f over A
    By hypothesis
    >> Exists x:A.g has unique fixed point x:A
    By hypothesis
  h10 : Exists y:A.f has fixed point y:A
  >> Exists y:A.f has fixed point y:A
  By hypothesis

END CHAPTER Fix-Point-Theory

END OF FILE
References


