The Ergo Support System: An Integrated Set of Tools for Prototyping Integrated Environments

Peter Lee, Frank Pfenning, Gene Rollins, and William Scherlis

Department of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3890

Abstract

The Ergo Support System (ESS) is an engineering framework for experimentation and prototyping to support the application of formal methods to program development, ranging from program analysis and derivation to proof-theoretic approaches. The ESS is a growing suite of tools that are linked together by means of a set of abstract interfaces. The principal engineering challenge is the design of abstract interfaces that are semantically rich and yet flexible enough to permit experimentation with a wide variety of formally-based program and proof development paradigms and associated languages. As part of the design of ESS, several abstract interface designs have been developed that provide for more effective component integration while preserving flexibility and the potential for scaling. A benefit of the open architecture approach of ESS is the ability to mix formal and informal approaches in the same environment architecture. The ESS has already been applied in a number of formal methods experiments.

1 Introduction

We are building an integrated set of tools, the Ergo Support System (ESS), for rapidly prototyping integrated environments. Our immediate goal is to support experimentation with a variety of approaches to program development, with particular emphasis on those approaches based on *formal* methods. This effort is a central activity of the Ergo Project at Carnegie Mellon University, which has the longer-term goal of developing useful, machine-supported methods for formal program development, analysis, and adaptation. It is our intent to develop technology that will ultimately scale up to larger systems and that will mix formal and informal approaches to support development and maintenance of large systems with trusted components. Scaling considerations have influenced much of the ESS design. A summary of the project's activities may be found in [LPR*88]. Initial motivation for the project is given in [SS83] and [Sch84].

We have experimented with many methods for formal program development, ranging from classical program transformation such as the fold/unfold system of Burstall and Darlington [BD77], finite differencing techniques of Paige and Koening [PK80], and the specialization system of Scherlis (see [Sch86] for references), to proof-based approaches such as the Nuprl system [CKB84,C*86] and proof transformation techniques [Pfe88b]. Our experiments with these and other approaches fall into three categories: (1) those done on paper only, (2) those using imported, existing systems, and (3) those carried out in environments which were constructed with the ESS. Our approach in this third category is essentially one of building an environment based on the formal method in question, thereby providing a basis for experimentation.

Our experience has made us aware that we are investigating two separate, but closely related research questions: how can we develop, analyze, and adapt *programs* by formal methods combined with informal methods, and how can we rapidly construct prototype programming *environments* that support these methods? In this paper we will discuss our approach to the latter problem. Rather than building a single system for formal program development, we have concentrated our engineering efforts on building a set of tools that supports the rapid prototyping of program and proof development environments.

In the course of designing such a suite of tools we have arrived at a number of requirements. First, the tools must not force strong commitments to particular language paradigms. The tools must be able to make use of information about languages based upon formal specifications of their syntax and semantics. This requirement derives from the fact that our experimentation must focus not only on *methods*, but also on

This research was supported in part by the Office of Naval Research under contract N00014-84-K-0415 and in part by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 5404, monitored by the Office of Naval Research under the same contract.

To appear at the Third ACM SIGSOFT Symposium on Software Development Environments, Boston, MA, November 28-30, 1988.

the related *language* issues (*e.g.*, programs at various stages of commitment, transformation rules, program analysis data, formal specifications, annotations, interface specifications, prototypes, inference rules, proofs, *etc.*). Secondly, integral support for the *activity* of experimentation with the various methods and languages must be provided even at the earliest stages of experimentation. At such a preliminary phase of research on the use of formal methods in software engineering, it will not suffice to build rigid systems that inhibit the basic "tinkering" necessary for experimentation and discovery.

We have made initial steps towards realizing such a set of tools. The Ergo Support System (ESS) is a collection of language and environment-prototyping tools that are integrated by a small number of carefully chosen, *formally-defined abstract interfaces*. Thus, instead of being an opaque, monolithic system, the ESS is an extensible collection of related parts. Success in building a flexible and powerful system for experimenting with formal methods depends critically on the choice of semantically substantive internal abstract interfaces. In this paper we report on some of the current tools and interfaces, and describe how they interact.

2 Integration via Abstract Interfaces

In developing the ESS, we have concentrated much of our effort on the design and implementation of a small set of formally-defined *abstract interfaces*. We lack a concise definition for the term "abstract interface," but loosely speaking, it consists of a *formalism* together with a supporting *implementation*. The formalism is a conceptual and mathematical basis for the protocol of interaction among system components. The implementation specifies how the protocol is represented by actual programming language data structures.

As a simple example, consider an abstract interface for communicating the syntactic structure of programs among the components of an environment. One obvious possible choice for the underlying formalism might be that of conventional abstract syntax, as specified by a (possibly ambiguous) context-free grammar, or perhaps an abstract data type. The implementation, then, might provide for the actual communication of abstract syntax via tree data structures, along with associated tree-manipulation functions. We will return to this example and others in more detail in the following sections.

Programming language abstract data type specifications alone are often inadequate for specifying abstract interfaces in their full richness. In our examples, we make use of combinations of several kinds of mechanisms to specify abstract interfaces, including abstract data type signatures, axioms, and other notations as appropriate.

3 Terms and Occurrences

The simplest abstract interface provided in the ESS is for basic operator-operand terms. This interface is used primarily for representing the syntactic structure of programs and other objects. The interface is based on a collection of abstract data types, two of which we describe below.

The interface distinguishes between terms, occurrences, and term occurrences. Informally, a *term* is a value consisting of an operator and a list of subterms. Two terms are equivalent if their operators and subterms are equivalent. Viewing terms as a trees, an *occurrence* indicates a path through a tree, denoting only the branch choices. Finally, a *term occurrence* is a particular subterm of a term, that is, an occurrence and a containing term.

Formal definitions of these concepts can be given as algebraic specifications of abstract data types (ADTs). Excerpts from specifications are given in Figures 1 and 2.

ADT `Term` **with**
mk-term : `Oper` `List`(`Term`) `Term`
term-op : `Term` `Oper`
term-args : `Term` `List`(`Term`)
end

Auxiliary functions of `Term`
term-argn : `Term` `Nat` `Term`
term-arity : `Term` `Nat`
end

Axioms of `Term`
term-argn (mk-term (op $[t_1 \ \ldots \ t_n]$) i) = t_i
term-arity (mk-term (op $[t_1 \ \ldots \ t_n]$)) = n
end

Figure 1: Excerpts from the specification of the `Term` abstract interface.

The specifications of `Term` and `Occ` refer to other data types, for example, `Oper`, `List`, and `Nat`. `List` and `Nat` refer to the standard data types. `Oper` is the type of term operators. When this abstract interface is used for representing the syntactic structure of programs (as in abstract syntax trees), term operators are used to represent the syntactic operators of the language. They might, for example, represent a conditional statement operator, a variable declaration operator, a plus operator, and so on. This type facilitates integration of the generated parsers and unparsers with the display and semantic analysis components.

The ESS implementation of this interface is used by nearly every component, and has been used to represent expressions in a variety of languages, including Pascal, Prolog, ML, Hoare-logic assertion languages, and others. The interface must provide for the maintenance of many versions of the represented objects. Maintaining such a "history" is made much simpler by providing only nondestructive operations on terms. (See, for example, the specification of replace-subterm in Figure 2.) Structure sharing is used to reduce the amount of copying during transformations or deductions.

ADT `Occ` **with**
nil-occ : `Occ`
null-occ : `Occ` `Bool`
occ-push : `Nat` `Occ` `Occ`
occ-top : `Occ` `Nat`
occ-rest : `Occ` `Occ`
end

Auxiliary functions of `Occ`
subterm : `Term` `Occ` `Term`
replace-subterm : `Term` `Occ` `Term` `Term`
end

Axioms of `Occ`
subterm (t nil-occ) = t
subterm (mk-term (op [t_1 ... t_n]) occ-push(i occ)) = subterm (t_i occ)
replace-subterm (t nil-occ $newt$) = $newt$
replace-subterm (mk-term (op [t_1 ... t_n]) occ-push(i occ) $newt$)
= mk-term (op [t_1 ... t_{i-1} replace-subterm(t_i occ $newt$) t_{i+1} ... t_n])
end

Figure 2: Specification of the abstract data type of occurrences.

4 The Syntax Facility

The Syntax Facility [DPRS88,Die88] is the primary tool dealing with the abstract interface of terms. For input it accepts concise context-free grammars written in an extended BNF, together with associated rules for construction of `Term`s. From this input, it produces a top-down operator precedence parser with two-token lookahead, and an unparser that produces formatted program text. A default formatting algorithm is provided and is usually adequate, but many aspects of formatting may be controlled by the language designer through explicit formatting annotations in the grammar.[1] The grammar for grammars is itself specified using the Syntax Facility.

The abstract interfaces of *terms* and *occurrences* are used to integrate the generated unparsers with the Interaction Facility and the Analysis Facility (see Section 7), for interfacing to windows and mouse, and for semantic analysis, respectively. Attributes (through the abstract interface of *attribute*, see Section 6) are used during formatting and to communicate information about textual position of characters to the Interaction Facility. This information is used by the Interaction Facility to interpret mouse selections in a display window.

Figure 3 shows an excerpt from a sample grammar for a small ML-like language. In the example, single quotes delimit keywords (sometimes called reserved words) in the described language, `{}` parenthesize phrases, and `<>` delimit abstract syntactic expressions that describe the `Term` to build when parsing the concrete syntactic alternative they follow. The comment character is `%`, and `+`, `++`, `*`, and `**` are various grammatical iteration constructs.

In the grammar given, for example, a declaration `d` may be introduced by the keyword **let** followed by a non-empty list of binders `bnd` separated by the keyword **and**. If such a construct is parsed, a `Term` with operator `let` and argument list constructed from the list of binders is created. Such terms, along with this particular instantiation of the `Oper` type, are then used to communicate the structure of programs to other components for, perhaps, display in a window or for semantic analysis.

5 Higher-order abstract syntax

`Term`s, as outlined in the previous subsection, provide the simplest abstract interface between the view of programs (or formulas, deductions, derivations, *etc*.) as strings (seen by the user) and the representation which is manipulated by the tools in the environment. *Higher-order abstract syntax* is a more advanced and sophisticated abstract interface that is at the heart of the ESS. In order to understand the motivations behind this interface, we will have to digress somewhat and discuss *meta-programming*.

We use the term *meta-program* to describe any program that manipulates other programs or, more generally, expressions in complex languages. A meta-program is written in a *meta-language* and manipulates expressions in an *object-language*.

Meta-programs are just as diverse as programs and may be written in many different meta-languages. One example of a meta-program would be a program implementing the Unfold

[1] This is a relatively new feature, not shown in the example.

```
Comment Character delimited '%' '%'
Lexical Terminals id, number, string
Bracketing Information e '(' ')'
Precedence Information
   d 'and' medial right
   e ','   medial right
     '+'   medial left, '-' medial left
     '-'   initial
     jux   medial left
program ::= {{{d ';;'} <d>}+ '(' e ')'} <prog(decls(plus),e)> ;
d ::= {'let' {bnd ++ 'and'}}            <let(doubleplus)>
    | {'letrec' {rbnd ++ 'and'}}        <letrec(doubleplus)> ;
bnd ::= {p '=' e}  <bind(p,e)> ;
p ::= id                            <variable(id)>
    | { '(' p^0 ',' p^1  ')' }      <pairpat(p^0,p^1)> ;
e ::= number                        <numb(number)>
    | id                            <var(id)>
    | { e^0 jux '(' e^1 ')' }       <fapply(e^0,e^1)>       % function application
    | { '-' e }                     <neg(e)>                % unary minus
    | { e^0 '+' e^1 }               <add(e^0,e^1)>          % addition
    | { e^0 '-' e^1 }               <minus(e^0,e^1)>        % subtraction
    | { d 'in' e }                  <dec(d,e)>              % local declaration
    | { '\\'  p+ '.' e }            <lambda(plist(plus),e)> % lambda abstraction
    | { '[' {e ** ';'} ']' }        <lst(doublestar)>       % list
    | {'if' e^0 'then' e^1 'else' e^2 }                     % conditional
                                    <if-t-e(e^0,e^1,e^2)> ;
```

Figure 3: Excerpt from the EML grammar.

rule in a program transformation system. Another example would be an interactive theorem prover that generates a set of subgoals, given a current goal and inference rule. Meta-programs may also be very large, such as compilers and compiler generators.

We are increasingly convinced that no single meta-language can be sufficient to conveniently express the wide variety of meta-programs that must be written in program development environments. Instead, a meta-language should just be viewed as *just one tool* in such an environment. In the ESS, we are already using a variety of meta-languages, such as Common Lisp (which is also the base language of the ESS), an attribute specification language (see Section 6 and [NP88]), ADT-OBJ (an object-oriented language used for programming user interfaces, see [Fre88b]), and Prolog (a logic programming language used for program transformation and theorem proving, see [NM88,HM88b,HM88a]). Work on LEAP (a strongly typed functional language with explicit polymorphism, see [PL88]) and others is in progress.

When faced with such a diversity of meta-languages, the integration problem is formidable, and the abstract interface between them is crucial for the utility and practicality of the whole environment. `Terms` (one may think of them as abstract syntax trees) have a crucial weakness, namely that they do not carry information about the name-binding rules of the object-language. This makes it impossible to write, for example, *one* function for substitution that is syntactically correct for *all* object-languages.

As a simple example, consider the problem of variable capture. In the subset of ML introduced earlier, we may want to express and implement the rule of **let**-conversion.

$$\textbf{let } x = e \textbf{ in } b \qquad\qquad b[e\ \ x]$$

$b[e\ \ x]$ is our notation for the result of substituting e for x in b. Here are two incorrect applications of this rule. Note that reading them from right to left shows the problem of doing correct matching against $b[e\ \ x]$.

$$\textbf{let } x = y \textbf{ in let } y = 5 \textbf{ in } x \quad y \qquad\qquad \textbf{let } y = 5 \textbf{ in } y \quad y$$
$$\textbf{let } x = 5 \textbf{ in let } x = x \quad x \textbf{ in } x \qquad\qquad \textbf{let } x = 5 \quad 5 \textbf{ in } 5$$

Correct substitution requires recognition of name conflicts and renaming of bound variables. The correct results are of course

$$\textbf{let } x = y \textbf{ in let } y = 5 \textbf{ in } x \quad y \qquad\qquad \textbf{let } y \ = 5 \textbf{ in } y \quad y$$
$$\textbf{let } x = 5 \textbf{ in let } x = x \quad x \textbf{ in } x \qquad\qquad \textbf{let } x = 5 \quad 5 \textbf{ in } x$$

and they require knowledge of the binding constructs of the object-language (ML, in this case). For more on these and other examples (like variable-occurrence conditions in inference rules and context propagation transformations) see [PE88].

From this and other examples it is clear that one would like to include information about the binding properties of object-language constructs in the representation of programs. Higher-order abstract syntax achieves this in a uniform way. The basic idea was proposed first by Huet and Lang in [HL78], namely to

use the simply typed -calculus as a representation language. We found it essential to enrich this by allowing products and some amount of polymorphism.

Without going into too much detail, we illustrate here the use of higher-order abstract syntax. Our notation for higher-order abstract syntax uses for abstraction, juxtaposition for application. Juxtaposition associates to the left. Constants such as let or mult are typeset in a sans-serif font, variables are in italics. First an example program in concrete syntax and its representation in the second line:

let $x = y$ **in let** $y = 5$ **in** $x \quad y$
$\mathsf{let}(\ x \quad \mathsf{let}(\ y \quad \mathsf{mult}\, x\, y)\, 5)\, y$

Note that **let** is represented as a third-order constant let, and is represented as a second-order constant mult. This can now be matched against a pattern with the use of higher-order matching (only second-order matching is needed in this example). We use b as a pattern variable that represents a *function* which, given an expression, will return an expression. It is part of the second-order matching process to synthesize this function. The concrete syntax for patterns is an extension of the ML syntax and uses $b[e]$ as a notation for the application of a pattern variable b to an argument e. In the representation this simply becomes application (juxtaposition). Below is the pattern for the **let**-conversion rule and its representation.

let $x = e$ **in** $b[x]$ $\qquad$ $b[e]$
$\mathsf{let}(\ x \quad b\, x)\, e$ $\qquad$ $b\, e$

The substitution matching this against the example program:

$$\begin{array}{ll} e & y \\ b & x \quad \mathsf{let}(\ y \quad \mathsf{mult}\, x\, y)\, 5 \end{array}$$

After substitution, the right-hand side becomes

$$\begin{array}{rcl} b\, e & = & (\ x \quad \mathsf{let}(\ y \quad \mathsf{mult}\, x\, y)\, 5)\, y \\ & = & \mathsf{let}(\ y \quad \mathsf{mult}\, y\, y\ ) \end{array}$$

which is the representation of the correct result given above.

One can see that the -conversion involves renaming of bound variables when necessary. This is possible in a way independent of the object-language, since is the only binder in higher-order abstract syntax. Of course, it is the language implementor's responsibility that all binding constructs of his object language are properly represented through 's in higher-order abstract syntax. Note that higher-order abstract syntax only captures lexical binding properties — it does not make a commitment to the dynamic semantics (*e.g.*, call-by-value vs. call-by name, or dynamic vs. static binding).

Higher-order abstract syntax can be applied to a wide range of languages, though the amount of effort required to achieve the representation for different languages varies greatly. We have used ML, PASCAL, Prolog and others as examples. The crucial problem in each of these cases is of course the translation between the `Term` abstract interface and higher-order abstract syntax in both directions. The language implementor is free to choose any number of meta-languages to program these translation functions. One appealing option is to use attribute grammars (which are discussed in Section 7). We now discuss the abstract interface of attributes which integrates attribute grammars into the rest of the environment, but which also has uses beyond the attribute grammar compiler.

6 Attributes

The abstract interface of *attributes* is described in elsewhere in this volume [NP88], so we will give only a very brief description here.

This abstract interface is formulated as a higher-order abstract data type. It ensures attribute consistency, while providing for inter-program structure sharing, demand-driven attribute evaluation, incremental attribute update, and attribute caching. It is completely independent of the attribute grammar compiler, and is in fact also used by other tools, such as the Syntax Facility and the Interaction Facility (see Section 8).

The abstract interface of attributes was developed in response to requirements in the context of a collection of independent tools, such as the ESS. We found that conventional solutions to the incremental attribute update problem require too much care on the part of the client programs, and also prohibit several, closely related versions of the program to coexist with their attributes. In other words, only one, the "current" version of the program is available with its attributes. This can lead to gross inefficiencies when, for example, the program derivation space is explored in a non-linear fashion, since old attribute values have to be reconstructed.

Our abstract data type solves this problem by postulating a functional character for attributes, that is, an attribute value should be completely determined by the term and the context it appears in, where the definition of context is made together with the definition of an attribute. Roughly, the implementation of the data type caches the mapping from contexts to attribute values in terms in order to avoid recomputation as much as possible. This scheme incurs a certain overhead in attribute access, but in our applications the attribute caching and demand-driven nature of attribution more than makes up for this overhead.

Typical clients of the attribute interface are the Syntax Facility which uses it to communicate information to the Interaction Facility, and the attribute grammar compiler itself, which is written almost entirely as an attribute grammar and is discussed in the next Section.

7 The Analysis Facility

The Analysis Facility (described in more detail in [NP88]) provides for general-purpose program analysis by means of attribute grammars. The Analysis Facility accepts attribute grammars specified in a simple meta-language closely related to the meta-language used for syntax specification. The class of attribute grammars accepted is conventional and is in many respects similar to the MUG2 system [Wil81].

From an attribute grammar the Analysis Facility generates attribute evaluators based on the abstract interface of attributes described in the previous section.

The Analysis Facility is almost completely bootstrapped within the ESS: its parser is generated by the Syntax Facility, its evaluator generator is written as an attribute grammar and thus generated by the Analysis Facility itself. This entails a great deal of flexibility that has been exploited many times when changing the syntax of attribute grammars, or incorporating new features. Integration is also improved by the encapsulation of the demand-driven evaluation strategy in the attribute abstract interface. The Analysis Facility provides a convenient metalanguage for specification of attribute grammars, but is otherwise simply one of several clients of the attribute abstract interface.

The example attribute grammar given in Figure 4 specifies a translator from first-order terms in their default implementation to higher-order abstract syntax for our sample ML-like language. The scoping rules for ML are not particularly difficult, but they are not trivial, since ML allows pattern binding. For example,

$$\backslash\ ((x\ \ y)\ \ (u\ \ v))\quad (x+u\ \ y+v)$$

defines a function that takes two pairs as arguments and adds corresponding components (\ is our ML notation for abstraction). In order to realize that $x\ \ y\ \ u\ \ v$ are all bound in the body of the abstraction, the pattern $((x\ \ y)\ \ (u\ \ v))$ must be analyzed.

A second complication arises because a program consists of a sequence of **let**'s, each one opening a scope for the defined function or variable name that extends to the end of the program. Simultaneous recursive function definition binds the name of all the functions in every definition body.

In the attribute grammar in Figure 4, we use the prefix `ho` to mean "higher-order," so, for example, `hoe` is the higher-order representation of an expression.

Here is a brief explanation of this attribute grammar. We use the auxiliary functions `m` (which creates a higher-order term by applying its first argument the remaining arguments) and `lam` (which creates an abstraction from a list of variables and their scope). The only other auxiliary functions are `vsnil()` (which returns an empty list of variables), `vscons(v,l)` (which adds the variable `v` to the list `l`), and `lamcurry` (which basically creates a nested list of abstractions from a list of variable lists and a scope).

A declaration `d` receives a the higher-order representation of its scope as an inherited attribute and synthesizes a higher-order representation of the declaration *including* its scope. Note that in a **let**-declaration, the expressions translated from the binders do not appear in the scope the created abstraction, while they do in the case of a **letrec**. The list of variables in the pattern p and the translations of e and p are synthesized attributes of a binder `bnd` of the form $p = e$. The calculation of most of the attributes is straightforward.

The Program Flow Analyzer [Nor87] is a component of the ESS which generates attribute grammars from higher-level descriptions of the program and data flow properties of a language. It is bootstrapped within the ESS using the Syntax and Analysis Facilities and is therefore, like many of our tools, also an application of the ESS.

8 The Interaction Facility

Much of the complexity of formally-based program development can be made easier to manage by providing sophisticated means for user interaction, for example by supporting multiple graphical views of programs and specifications. There are several ESS tools and abstract interfaces for constructing advanced user interfaces. We refer to these collectively as the Interaction Facility. It contains the following:

ADT-OBJ: Development and refinement of user interfaces is an application for which the object-oriented programming paradigm is particularly well-suited. The standard object system for Common Lisp, CLOS [B*88], provides mechanisms for inheritance and method update which are crucial for developing user interfaces. During experimentation in a larger system, we have found that some care must be taken when using these mechanisms. ADT-OBJ [Fre88b] is a restrictive variant of CLOS which prohibits the violation of system component integrity. As its name implies, ADT-OBJ combines the features of object-oriented programming and abstract data types, thereby providing secure encapsulation of object methods, yet still providing many of the advantages of inheritance and method update.

DISPLAY: One of the most difficult requirements imposed by our environments is the support for interaction with multiple graphical representations of programs, specifications, program derivation structures—virtually any syntactic object. The DISPLAY system [Fre88a] provides such support for objects using the `Term` and `Occ` abstract interfaces, and the X window system [GR87]. Using DISPLAY, term occurrences can be displayed in windows and menus, and manipulated via mouse interactions. Standard graphical representations, for example as pretty-printed text or as trees, can be customized to suit particular applications. DISPLAY is based on the ADT-OBJ object system.

CPS-LISP: The object-oriented paradigm provides useful support for the specification and implementation of the *structure* of user interfaces, but is of little help in the specification of their *control*. We have found the mechanism of *continuations* to be ideal for specifying such user interface control. Hence, we have developed a mini-language called CPS-LISP, which is a subset of Common Lisp augmented with constructs for manipulation of the current continuation, as in the `call/cc` construct of the Scheme language [Cli85]. With CPS-LISP (and DISPLAY), user interface control programs can be easily written and integrated into an environment.

The Interaction Facility has been used to generate user interfaces for many applications, including several program transformation environments, a Hoare-logic proof system for a simple parallel language, and the Deduction Facility.

```
Grammar eml
Pass eml-hoas

program(^ho) = prog(decls(d(^env0/hod, !etag\hoe)*), e(^hoe)) where ho = m('hoprog,hod); final ho;
d(!scope, ^hod) =
      let(bnd(!novars/etag, ^etag\patvars, ^hop, ^hoe)*)
        where hod = m('let,lam(patvars,m('guard0(m('plist,hop*),scope))),m('elist,hoe*))
          and novars = vsnil()
    | letrec(bnd(!novars/etag, ^etag\patvars, ^hop, ^hoe)*)
        where hod = m('letrec,lam(patvars,m('guard1(m('plist,hop*),scope,m('elist,hoe*)))))
          and novars = vsnil() ;;
bnd(!vars, ^newvars, ^hop, ^hoe) = bind(p(!vars, ^newvars, ^hop), e(^hoe)) ;;
p(!vars, ^newvars, ^hop) =
      variable(id(^spelling))      where newvars = vscons(spelling, vars) and hop = m('spelling)
    | pairpat(p(!vars,^newvars0,^hop0), p(!newvars0, ^newvars, ^hop1))
                                   where hop = m('pairpat,hop0,hop1) ;;
e(^hoe) = numb (number(^litvalue)) where hoe = m('numb,number(litvalue))
    | var (id(^spelling))          where hoe = m('spelling)
    | fapply (e(^hoe0), e(^hoe1))  where hoe = m('fapply,hoe0,hoe1)
    | neg    (e(^hoe1))            where hoe = m('neg,hoe1)
    | add    (e(^hoe0), e(^hoe1))  where hoe = m('add,hoe0, hoe1)
    | minus  (e(^hoe0), e(^hoe1))  where hoe = m('minus,hoe0, hoe1)
    | dec(d(!hoe1, ^hoe), e(^hoe1))
    | lambda(plist(p(!novars, ^patvars, ^hop)*), e(^hoe0))
                                   where hoe = lamcurry(patvars*,hop*,hoe) and novars = vsnil()
    | lst(e(^hoe0)*)               where hoe = m('lst,hoe0*)
    | if-t-e(e(^hoe0), e(^hoe1), e(^hoe2)) where hoe = m('if-t-e,hoe0, hoe1, hoe2) ;;
```

Figure 4: Excerpt from an EML attribute grammar.

9 Boxes

The Box Facility provides configuration management for Common Lisp in which program components may themselves be generated by other program components. *Boxes* provide a framework for specifying the functional relationship between the various functions and tools used to generate an environment. The Box Facility uses box descriptions to determine what files to regenerate when a source change is made. Conceptually, a box is a collection of functions mapping between abstract interfaces. For example, the Syntax Facility consists of two higher-order functions: (1) the parser generator, maps a grammar to a function from strings to abstract syntax trees, and (2) the unparser generator, maps a grammar to a function from abstract syntax trees to strings.

At present, the ESS still uses a file-oriented (rather than object-based) development environment. Hence, boxes are annotated with information about the files comprising a box. The functionality of the Box Facility is thus an analogue to the Unix *make* facility for Common Lisp. The use of a file-oriented environment means that we use endings of file names to indicate types of data. For example, any file whose name ends in `-gr.txt` is assumed to contain a grammar, and any file ending in `-agr.txt` is assumed to contain an attribute grammar. Similarly, a file ending `-parser.lisp` is assumed to define a parser and can thus be regenerated from the corresponding `-gr.txt`. This is expressed in the following box definition:

```
(defbox sb
  "The syntax-box parser/unparser generator."
  (:maintainers "srd" "tsf")
  (:needs oper occur terms sorts lang attr)
  (:path "/usr/ergo/ess/lang/sb-term/rel/")
  (:generates "sb:sb-make" &key
     (gr-file "*-gr.txt" :input)
     (lexer-out "*-lexer.lisp" :output)
     (parser-out "*-parser.lisp" :output)
     (unparser-out "*-unparser.lisp" :output))
  (:files ...))
```

The `needs` clause in this box definition specifies the boxes that the Syntax Facility (`sb`) depends upon. The directory containing the `sb` source files is listed in the `path` clause. The `sb` exports a function `sb:sb-make` which generates output files `*-unparser.lisp`, `*-parser.lisp`, and `*-lexer.lisp` when given an input file `*-gr.txt`. The Box Facility stores this dependency and uses it when the definition of another box contains a grammar file.

Here is part of the box definition for our ML subset:

```
(defbox eml
  "ESS Document Version of ML"
  (:needs sb-support ab-support)
  (:files "eml-gr.txt" "eml-unparser"
          "eml-parser" "eml-lexer" ...))
```

Since the files listed for EML match the filename patterns in the `sb` box definition, the Box Facility knows how to regenerate the Lisp files for the EML parser, unparser, and lexer.

10 The Deduction Facility

The Deduction Facility is a very high-level component of the ESS. It is independent of any particular logic and can be used for a variety of tasks such as natural deduction, type inference, Hoare-style reasoning about programs, or for reasoning in semantic domains. Its design is strongly influenced by Prolog [MN86,MN87] and LF [HHP87]. Its central abstract interface is a generalization of higher-order abstract syntax by dependent types which allows straightforward expression of inference rules in most logics.

The most novel aspect of the Deduction Facility is its close integration of interactive and automatic proof search through its meta-language Elf [Pfe88a]. The Deduction Facility is still under development, with current emphasis on meta-language design and implementation. We anticipate integrating it later with imported general purpose theorem provers in first-order classical logic and equational reasoning systems that are available in Common Lisp.

In order to show how the Deduction Facility is organized and used, let us suppose that we are trying to implement a Hoare-logic reasoning system for a simple imperative language. How would we proceed? First, we would write a grammar for the imperative language. We would then embed this grammar in the grammar for Hoare triples, which also uses an assertion language as another sublanguage. (The Syntax Facility provides support for such language embeddings in the form of *external grammars*.)

The next step is to define the higher-order abstract syntax for the language of Hoare triples, including the assertion language and programming language. Depending on the programmer's inclination, this could defined in Common Lisp or as an attribute grammar (using the Analysis Facility). Finally the language of Hoare triples would be embedded in a language for *deductions* in the Hoare logic, which includes a definition of the inference rules as typed constants in the style of LF.

As a result of the process, we would obtain an environment for Hoare logic deduction. This consists of an interactive component (using the Interaction Facility, see Section 8) with proof tree display and browsing, inference rule menus, a definition and library facility, and a meta-programming component for programming theorem proving and program synthesis strategies in the style of [FM88]. One important feature that distinguishes this from an environment such as provided by PRL [C*86] is that the current proof goal may contain free variables ("logical" variables, in Prolog terminology), and thus allows the "theorem" to be created during deduction. This means that the Hoare logic implemented as outlined above can be used for program synthesis without any further work, since one may start with pre- and post-condition, but with a free variable instead of a program. Deduction will then synthesize the program. The Deduction Facility thus also provides a convenient and powerful framework for experiments with this sort of combination of program verification and synthesis.

11 A Prototype Environment

As an illustration of how abstract interfaces can be used to integrate system components we present a simple EML system. Arrows represent components, and boxes represent data. Only a limited number of components and interfaces are shown here. Note that some components generate other components; the parser generator produces an EML parser. The loop on the right side of the diagram indicates that the EML transformer can be applied repeatedly to higher-order abstract syntax. The path along the bottom of the diagram shows that the HOAS produced at each stage can be translated back to EML program text for display to the user.

Figure 5: Overview of the EML example.

12 Conclusions

The design of ESS addresses two distinct sets of engineering concerns. First we are seeking flexible tools to support the rapid development and evolution of prototype implementations of formal methods approaches. The tools help by enabling users to rapidly reach the point of directly addressing research issues surrounding the mechanization of formal methods. Much of the contribution of the tools derives from the support they provide for the rapid assembly of syntax-crunching and user-interaction machinery. These functions may not be particularly interesting from a scientific point of view, but they can often overwhelmingly dominate the engineering of formal methods prototypes.

The second set of concerns arise from our interest in the meaningful application of formal methods to practical large scale software development. Increasingly stringent requirements in large scale software systems for trust, adaptability,

security, and effective use of concurrency create demand for software engineering approaches that permit effective incorporation of formal methods technology on a selective and incremental basis. A particular challenge is the meshing of formal and informal approaches in the development of larger systems in which, for example, certain security and safety properties must be verified formally while less critical performance properties need only be established through empirical or other informal means.

In both cases, careful and early attention must be devoted to the environment engineering issues of component decomposition and abstract interface design. Component decomposition must be sufficiently aggressive that environments can scale gracefully as applications scale, formalisms improve, requirements evolve, and component sources diversify. Decomposition into components requires conventionalization of abstract interfaces, so it is not without cost, but at this stage of development the benefits far outweigh these costs. The abstract interfaces must be semantically substantive while remaining sufficiently flexible and abstract to enable support for scaling and for the diversity of methods and tools that will likely be needed to maintain an effectively scalable overall approach.

The placements of abstract interfaces in a complex system are often the most hard-won of design decisions. In the case of ESS, several iterations of design over a period of years have yielded the current set of interfaces and tools. We feel, however, that we have reached only a crude initial approximation for a design that will effectively scale. ESS is at the point where interesting experimental applications are being developed, and there is already need to include tools in the ESS for persistent object management and search, and for more advanced term matching and replacement. These will no doubt yield more mature interface designs that will more effectively sustain growth in scale and capability.

Acknowledgements

The authors would like to acknowledge all those who have participated, past or present, in the development of the Ergo Support System: Bill Chiles (Syntax Facility), Ken Cline (Type Inference Facility, Deduction Facility), Scott Dietzen (Syntax Facility, Transformation Facility), Conal Elliott (Higher-Order Unification Facility, Prolog), Tim Freeman (Interaction Facility, Transformation Facility), Nevin Heintze (Prolog), Bill Maddox (Analysis Facility), Rod Nord (Analysis Facility, Program Flow Analyzer Facility), and Anne Rogers (Syntax Facility).

13 Bibliography

[B*88] Daniel G. Bobrow et al. Common lisp object system specification. Working Draft. February 1988.

[BD77] R. M. Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the Association for Computing Machinery*, 24(1):44–67, January 1977.

[C*86] Robert L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[CKB84] Robert L. Constable, Todd Knoblock, and Joseph L. Bates. Writing programs that construct proofs. *Journal of Automated Reasoning*, 1(3):285–326, 1984.

[Cli85] William Clinger. *The Revised Revised Report on Scheme*. AI Memo 848, MIT, Cambridge, August 1985.

[Die88] Scott R. Dietzen. *The Ergo Approach to Syntax*. Ergo Report In preparation, Carnegie Mellon University, Pittsburgh, 1988.

[DPRS88] Scott R. Dietzen, Mary Ann Pike, Anne M. Rogers, and William L. Scherlis. *User's Guide to the Ergo Syntax Facility*. Ergo Report In preparation, Carnegie Mellon University, Pittsburgh, 1988.

[FM88] Amy Felty and Dale A. Miller. Specifying theorem provers in a higher-order logic programming language. In Ewing Lusk and Ross Overbeek, editors, *9th International Conference on Automated Deduction, Argonne, Illinois*, pages 61–80, Springer-Verlag LNCS 310, Berlin, May 1988.

[Fre88a] Tim Freeman. *DISPLAY — A Window System for Customizing the User*. Ergo Report 88–066, Carnegie Mellon University, Pittsburgh, September 1988.

[Fre88b] Tim Freeman. *Overriding Methods Considered Harmful; or ADT-OBJ: Rationale and User's Guide*. Ergo Report 88–064, Carnegie Mellon University, Pittsburgh, July 1988.

[GR87] J. Gosling and D. Rosenthal. A window-manager for bitmapped displays and UNIX. In F. R. A. Hopgood et al., editors, *Methodology of Window Managers*, North-Holland, 1987.

[HHP87] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Symposium on Logic in Computer Science*, pages 194–204, IEEE, June 1987.

[HL78] Gérard Huet and Bernard Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.

[HM88a] John Hannan and Dale Miller. Enriching a metalanguage with higher-order features. In John Lloyd, editor, *Proceedings of the Workshop on Meta-Programming in Logic Programming*, University of Bristol, Bristol, England, June 1988.

[HM88b] John Hannan and Dale Miller. Uses of higher-order unification for implementing program transformers. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming: Proceedings of the Fifth International Conference and Symposium, Volume 2*, pages 942–959, MIT Press, Cambridge, Massachusetts, August 1988.

[LPR*88] Peter Lee, Frank Pfenning, John Reynolds, Gene Rollins, and Dana Scott. *Research on Semantically Based Program-Design Environments: The Ergo Project in 1988*. Technical Report CMU-CS-88-118, Carnegie Mellon University, Pittsburgh, March 1988.

[MN86] Dale A. Miller and Gopalan Nadathur. Higher-order logic programming. In *Proceedings of the Third International Conference on Logic Programming*, Springer Verlag, July 1986.

[MN87] Dale A. Miller and Gopalan Nadathur. A logic programming approach to manipulating formulas and programs. In *Symposium on Logic Programming, San Francisco*, IEEE, September 1987.

[NM88] Gopalan Nadathur and Dale Miller. An overview of Prolog. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming: Proceedings of the Fifth International Conference and Symposium, Volume 1*, pages 810–827, MIT Press, Cambridge, Massachusetts, August 1988.

[Nor87] Robert L. Nord. *A Framework for Program Flow Analysis*. Ergo Report 87–038, Carnegie Mellon University, Pittsburgh, November 1987.

[NP88] Robert L. Nord and Frank Pfenning. The Ergo attribute system. In *Proceedings of the SIGPLAN'88 Symposium on Software Development Environments, Boston, November 28–30*, ACM Press, 1988. To appear.

[PE88] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the SIGPLAN '88 Symposium on Language Design and Implementation*, pages 199–208, ACM Press, June 1988. Available as Ergo Report 88–036.

[Pfe88a] Frank Pfenning. *Elf: A Language for Logic Definition and Verified Meta-Programming*. Ergo Report, Carnegie Mellon University, Pittsburgh, Pennsylvania, September 1988. In preparation.

[Pfe88b] Frank Pfenning. Program development through proof transformation. In Wilfried Sieg, editor, *Logic and Computation*, AMS, Providence, Rhode Island, 1988. To appear.

[PK80] Robert Paige and Shaye Koening. *Finite Differencing of Computable Expressions*. Technical Report LCSR-TR-8, Laboratory for Computer Science Research, Rutgers University, August 1980.

[PL88] Frank Pfenning and Peter Lee. *LEAP: A Language with Eval and Polymorphism*. Ergo Report 88–065, Carnegie Mellon University, Pittsburgh, Pennsylvania, July 1988.

[Sch84] William L. Scherlis. Software development and inferential programming. In *NATO ASI Series, Vol. F8 Program Transformation and Programming Environments*, Springer-Verlag, 1984.

[Sch86] William L. Scherlis. Abstract data types, specialization and program reuse. In *International Workshop on Advanced Programming Environments*, Springer-Verlag LNCS 244, 1986.

[SS83] William L. Scherlis and Dana S. Scott. First steps towards inferential programming. In R.E.A. Mason, editor, *Information Processing*, pages 199–212, Elsevier Science Publishers, 1983.

[Wil81] Reinhard Wilhelm. *Global Flow Analysis and Optimization in the MUG2 Compiler Generating System*, pages 132–159. Prentice Hall, 1981.