Higher-Order Abstract Syntax[*]

Frank Pfenning[†] Conal Elliott[†]

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

Abstract

We describe motivation, design, use, and implementation of *higher-order abstract syntax* as a central representation for programs, formulas, rules, and other syntactic objects in program manipulation and other formal systems where matching and substitution or unification are central operations. Higher-order abstract syntax incorporates name binding information in a uniform and language generic way. Thus it acts as a powerful link integrating diverse tools in such formal environments. We have implemented higher-order abstract syntax, a supporting matching and

unification algorithm, and some clients in Common Lisp in the framework of the Ergo project at Carnegie Mellon University.

1 Introduction

Higher-order abstract syntax is a generalization of the usual data type of abstract syntax tree that is used to represent syntactic objects in implementations of systems that manipulate programs, formulas, rules, etc.

*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.

†The authors can be reached via electronic mail on the ArpaNet as `fp@cs.cmu.edu` and `conal@cs.cmu.edu`.

Higher-order abstract syntax uses a simply typed λ-calculus enriched with products and polymorphism and thus extends a proposal by Huet and Lang [11].

It differs from LF [1,9] in that we do not allow dependent function types, since no appropriate matching or unification algorithm is known. Moreover, we feel that products and polymorphism, which are absent from LF, are an essential feature for practical applications.

The requirements that led to the design of higher-order abstract syntax arose through one of the Ergo project's goals, namely to build a language-generic environment for formal program development. Components of such an environment that we have completed include a parser and unparser generator, an attribute grammar compiler, a type inference facility, and an interaction facility based on the X window system. Other components now being implemented include a language- and rule-independent transformation system and a logic-independent deduction system.

Such a program design environment should support rapid prototyping of all relevant aspects of programming languages. That is, one should be able to easily specify syntax, semantics, and transformation rules for a language and obtain an environment tailored to manipulating programs in that language. Our goals are even broader, since we believe that logical languages and proof systems are essential for formal program development and, therefore, that we must be able to support these as well.

The requirements for the internal representation of syntactic objects in such a system are quite different from the requirements in a single-language system. On the other hand, many of the same issues come up in compiler generation systems. We believe the basic principles that should guide the design of an internal representation for programs (and other syntactic objects) in our context are:

- Simple, user-friendly syntactic and semantic definition of a language should be supported through the representation. It should also allow definition of transformation and inference rules in a similar style. In particular, it should be possible to avoid complex side-conditions wherever possible.

- Efficient and correct use can be made of the definitions. In particular, the system must be able to efficiently and correctly transform programs or apply inference rules.

- The representation should be general enough to be used by all relevant tools in an environment, thereby providing for a high degree of integration between the tools.

Abstract syntax trees have proved very useful in similar contexts. This has been well established in systems like PSG [2], the Cornell Synthesizer Generator [19], Gandalf [8,6], or Popart [20]. However, abstract syntax trees do not completely satisfy the more general requirements listed above. The crux of the problem is that almost all languages in this context will have name binding constructs. These binding constructs make correct matching and substitution difficult (see Section 2 for some examples). Also, the requirements of a language generic system do not allow incorporation of the information about binding constructs the way it can be done in single-language systems. Instead, the language implementor must explicitly define the binding constructs of the language once and for all, and the system must be able to incorporate this information into the representation.

We found that all static binding constructs we examined can be represented in a simply typed λ-calculus with Cartesian products. We also added polymorphism to the representation to make it powerful to enough to conveniently state general transformation and inference rules. A somewhat less general, but closely related representation was first proposed by Huet and Lang [11]. It should be noted that our representation in no way prohibits dynamic extent of variables, nor does it mean a commitment to a call-by-name over call-by-value semantics. Also, since the types are syntactic sorts, no commitment to the semantic type structure of the object language is made. Higher-order abstract syntax is appropriate and useful for almost all languages, including Prolog, ML, Pascal, various logics and type theories, Hoare logics, etc.

Adopting this typed λ-calculus representation allows us to use the very powerful mechanism of *higher-order unification* (of which higher-order matching and substitution are special cases). We have implemented Huet's algorithm [10] for higher-order unification with extensions to deal with products and polymorphism. It is complete except for some rare cases involving higher-order polymorphism. In certain cases, straightforward matching would produce too many unifiers to be useful, and in this case the client (user or program) can preinstantiate some variables to generate a more specialized version of a pattern. Note that this problem of too many rewrites is inherent in program transformation, and not a defect of our implementation. On the contrary: our implementation provides a simple way of stating a very general, valid rule and then partially instantiating it to a more specialized one which is automatically correct and can be applied efficiently.

We also provide a first-order interface to higher-order abstract syntax. This first-order interface is defined through the augments in the grammar and is used by tools that still depend on the conventional notion of abstract syntax (like the parser and unparser generators and the current attribute grammar compiler). As discussed in Section **??** the efficiency loss due to the dual interface seems to be minimal.

In the remainder of this paper, we first point out some of the inadequacies of the representation of programs as abstract syntax trees. We then discuss higher-order abstract syntax and illustrate its use through a diverse set of examples. Finally, some implementation issues concerning higher-order abstract syntax and related work are discussed.

2 Some Motivating Examples

In this section we highlight some of the problems that arise in matching and substitution due to the presence of binding constructs in a language. Almost all languages have these binding constructs, though sometimes they are not immediately apparent. For example, in Prolog the "free" variables in a clause are actually bound in that clause, since they are clearly distinct from variables with the same name in other clauses. A function definition stated as $f(x) = b$ actually binds x and f (see the beginning of Section 3.2).

The rules we present throughout this paper are stated without any *semantic* side conditions such as strictness or termination. Depending on the language semantics, such conditions may still be necessary to ensure full semantic equivalence between the transformed programs. However, it should be noted that in all the examples the *syntactic* side conditions on the rules disappear without compromising the validity of the rule.

2.1 Correct Matching and Substitution

This problem should be very familiar; it is generally called "variable capturing". It appears in two different forms: during matching and during substitution. Consider the rule of **let**-conversion.

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

Here are two incorrect applications of this rule. Note that reading them from right to left shows the problem of doing correct matching against $b[e/x]$.

$$\begin{array}{l}\textbf{let } x = y \textbf{ in let } y = 5 \textbf{ in } x * y \;\not\Longleftrightarrow \\ \quad \textbf{let } y = 5 \textbf{ in } y * y \\ \textbf{let } x = 5 \textbf{ in let } x = x * x \textbf{ in } x \;\not\Longleftrightarrow \\ \quad \textbf{let } x = 5 * 5 \textbf{ in } 5\end{array}$$

What is required for correct substitution is recognition of name conflicts and renaming of bound variables. If this rule is read from right-to-left, it is clear that there are many possible ways of abstracting an expression from a program, and that therefore straightforward matching on *any* representation would be very non-deterministic. In a situation like this the solution is to partially instantiate the pattern before matching.

2.2 Variable Occurrence Restriction

Variable occurrence restrictions again require renaming of bound variables during substitution, or failure of matching. The following example is taken from a formalization of a natural deduction system to show the variety of circumstances in which these problems occur.

$$\frac{\Gamma \vdash P}{\Gamma \vdash \forall x P}\,\forall\mathrm{I},\ \text{where } x \text{ not free in } \Gamma.$$

If this rule is used by matching against the lower line, the restriction on x must be checked separately — it is difficult to formulate the rule simply and concisely. Ideally, x would be renamed to a new variable x^0 if x is already free in Γ. If the rule is used in the other direction, it should simply not match if x appears in Γ. As we will see in Section 3.3, rules incorporating occurrence conditions can be formulated easily and applied correctly using higher-order abstract syntax.

Note that in a system that uses first-order abstract syntax, not only would the rule be conditional, but the language implementor would somehow have to define a predicate `not-free-in` for the language in question.

2.3 Correct Treatment of Contexts

Many program transformation rules can be stated naturally through the use of *contexts*. Correct applications of these rules, however, is tricky. For example, a rule propagating computation into the branches of an **if** expression could be written as

$$C[\textbf{if}\ p\ \textbf{then}\ a\ \textbf{else}\ b] \iff \textbf{if}\ p\ \textbf{then}\ C[a]\ \textbf{else}\ C[b]$$

Consider the following incorrect application.

$$\begin{array}{l}\textbf{let}\ p = \textbf{false in if}\ p\ \textbf{then}\ 1\ \textbf{else}\ 2 \quad \not\Leftrightarrow \\ \quad \textbf{if}\ p\ \textbf{then}\ \ \textbf{let}\ p = \textbf{false in}\ 1 \\ \qquad\quad \textbf{else}\ \ \textbf{let}\ p = \textbf{false in}\ 2\end{array}$$

As noted in [16] syntactic conditions on C are difficult to formulate if one wishes to eliminate the possibility of incorrect rule application as in the example. The use of higher-order abstract syntax solves this problem by allowing the statement of the rule as above, but automatically prohibiting the incorrect use below without any additional conditions.

3 Design and Use of Higher-Order Abstract Syntax

In this section we will sketch higher-order abstract syntax and illustrate how it solves the problems mentioned in the previous section.

3.1 Well-Sorted First-Order Abstract Syntax

Ordinarily a parser creates untyped abstract syntax trees. They can be viewed as terms where the operators in the language act as function symbols, and the lexical terminals as constants. Variables as such do not exist in this representation, that is, object language variables are represented as identifiers, which are all considered constants. In some systems in which program schemas are considered, explicit notations for schema-variables are introduced. These schema- or meta-variables become variables in the abstract syntax.

In the context of syntax manipulation tools the need for types in the abstract syntax soon arises. This is because one needs to be able to test whether a substitution of a subterm for another is syntactically legal. The types then are derived from the *syntactic sorts* or *syntactic categories*. The notion of syntactic sort and object language type usually do not coincide, though typically the object language types refine the syntactic sort structure.

As an example throughout this paper we will use a simple functional language with mutually recursive function definitions, because it provides a good framework for illustrating the use of higher-order abstract syntax.

We begin with a grammar that, except for special fonts, can be read directly by the parser and unparser generator in the Ergo Support System. The grammar augments surrounded by "$< >$" indicate the abstract syntax that the parser constructs to represent the concrete syntax on the left. We omit some of the straightforward grammar augments.

$P ::= \quad \mathbf{rec}\ D \qquad <\mathsf{rec}^0(D)>$

$$
\begin{array}{lll}
D ::= & F = E \mid D_0 \ \mathbf{and} \ D_1 & \\
F ::= & V \mid V \ F & \\
E ::= & V & \mathsf{<var}(V)\mathsf{>} \\
\mid & C & \mathsf{<const}(V)\mathsf{>} \\
\mid & \mathbf{if} \ E_0 \ \mathbf{then} \ E_1 \ \mathbf{else} \ E_2 & \mathsf{<ite}(E_0, E_1, E_2)\mathsf{>} \\
\mid & \mathbf{let} \ B \ \mathbf{in} \ E & \mathsf{<let}^0(B, E)\mathsf{>} \\
\mid & E_0 \ E_1 & \mathsf{<apply}(E_0, E_1)\mathsf{>} \\
\mid & \mathbf{lam} \ V \,.\, E & \mathsf{<lam}^0(V, E)\mathsf{>} \\
B ::= & V = E \mid B_0 \ \mathbf{and} \ B_1 &
\end{array}
$$

This completes the outline of a view of abstract syntax as typed terms, where operators correspond to function symbols and syntactic sorts to types. Meta-variables or schema variables would have to be a distinct syntactic category.

In the Ergo Support System, the first-order abstract syntax as defined by means of the grammar above is used by the parser, unparser, formatter, and interaction facility (for highlighting proper subterms). However, if a higher-order view is defined for a language the abstract syntax tree is never built. Instead, we parse directly into higher-order abstract syntax.

The first-order view from the higher-order representation is then created through a function that decomposes a term into its first-order operator and arguments. Currently, this function must be specified directly by the language implementor.

3.2 Higher-Order Abstract Syntax

Motivated by the examples in Section 2 we now generalize to higher-order terms. This representation was suggested by Huet and Lang [11] for use in program transformation. In this first generalization, simply typed lambda terms represent programs. Syntactic sorts become λ-calculus types. The lexical terminals and the operators that do not introduce variable bindings are, respectively, first-order and second-order constants of the λ-calculus.

Another crucial change occurs for the variables. Operators in the object language that are *binding constructs* are now explicitly encoded as third-order constants. As the following examples will show, this requires that bound object language variables actually become variables in the typed lambda calculus.

The generalization to simply typed λ-terms is still insufficient for many languages. The problem is the presence of lists of various forms: lists of arguments, lists of bound variables, lists of mutually recursive function definitions. The usual solution is to curry functions, which enables representation, but not simple, finitary definition of transformation or inference rules. See, for example, the formalization of Hoare logic in LF [1], where the natural encoding remains restricted to a two-register machine. More generally, LF can naturally formalize an n-register machine for any fixed n, but only give a very unsatisfactory encoding of full Hoare logic.

To solve this problem, we have extended higher-order abstract syntax and the unification algorithm to handle products. In our formulation of the λ-calculus with products, we use pattern binders akin to ML, rather than constants fst and snd. The product is binary and associates to the left. Our example language has three constructs that introduce scope: **let**, **lam**, and **rec**. Figure 1 shows how they are represented in higher-order abstract syntax.

In order to express transformation rules through patterns, we now augment the grammar for the object language to a grammar for an associated pattern language. The pattern language is not universal, but rather is be constructed in harmony with a given object language. Because of the higher-order nature of

$$\begin{array}{rcl}
\textbf{let } V_1 = E_1 \textbf{ and } \ldots \textbf{ and } V_n = E_n \textbf{ in } E & \text{as} & <\mathsf{let}(\lambda\langle V_1 \ldots V_n\rangle \,.\, E)\langle E_1, \ldots, E_n\rangle> \\
\textbf{lam } V \,.\, E & \text{as} & <\mathsf{lam}(\lambda V \,.\, E)> \\
\textbf{rec } V_1 V_{11} \ldots V_{1n_1} = E_1 \textbf{ and } \ldots \textbf{ and } V_m V_{m1} \ldots V_{mn_m} = E_m & \text{as} &
\end{array}$$
$$<\mathsf{rec}(\lambda\langle V_1, \ldots, V_m\rangle \,.\, \langle \mathsf{lam}(\lambda V_{11} \,.\, \mathsf{lam}(\lambda V_{12} \ldots E_1 \ldots)), \ldots, \mathsf{lam}(\lambda V_{m1} \,.\, \mathsf{lam}(\lambda V_{m2} \ldots E_m \ldots))\rangle)>$$

Figure 1: Representation of **let**, **lam**, and **rec**.

our abstract syntax, this requires more than just the introduction of schema variables. Typically, an object language and its associated pattern language will differ by three or four productions. These additional productions provide for specifying variable applications, in some cases also for variables, abstractions, and products.

Here we include higher-order application and abstraction. Note how the augments of these productions differ from the cases building apply and lam.

$$
\begin{array}{llll}
E ::= \ldots & | & E_0[E_1] & \texttt{<}E_0(E_1)\texttt{>} \\
 & | & \lambda V \,.\, E & \texttt{<}\lambda V \,.\, E\texttt{>}
\end{array}
$$

Let us reconsider the **let**-conversion rule from Section 2.1. We can now state a single polymorphic rule for **let**-conversion. The first line shows how it would actually be stated, the second line shows the hidden type variable α. The given type is the most general, with let being a polymorphic constant of type $(\alpha \to E) \to \alpha \to E$. Remember that the types in the representation correspond roughly to the syntactic categories of the language definition.

$$
\begin{array}{l}
\mathbf{let}\ x = e\ \mathbf{in}\ b[x] \iff b[e] \\
\mathbf{let}\ x_\alpha = e_\alpha\ \mathbf{in}\ b_{\alpha \to E}[x_\alpha] \iff b[e]
\end{array}
$$

We can match this rule, for example, against the following term

$$\textbf{let } x = 5 \textbf{ and } y = 6 \textbf{ in } x * y$$

In this example, the substitution will be

$$\begin{aligned} \alpha &\longleftarrow E \times E \\ e &\longleftarrow \langle 5, 6 \rangle \\ b &\longleftarrow \lambda \langle x, y \rangle \ . \ x * y \end{aligned}$$

Then the right-hand side becomes

$$b[e] = (\lambda \langle x, y \rangle \ . \ x * y)[\langle 5, 6 \rangle] \underset{\beta\eta}{=} 5 * 6$$

The Fold and Unfold rules [3] provide two more examples of how rules can be concisely and easily formulated in this framework. Here we show the simpler Unfold rule and give a definition that states one single rule for any number of mutually recursive function definitions.

$$\textbf{rec}\ f = b[f][f] \iff \textbf{rec}\ f = b\,[b[f][f]]\,[f]$$

We now consider an example where this rule is applied to a recursive function definition. The objective of the transformation is to replace the first recursive call to gcd by the properly instantiated function body. This is achieved in two steps: the first is to replace a recursive function call by its definition, the second is to use **lam**-reduction (analogous to β-reduction, but on the level of the object language) to obtain the instantiated body.

```
rec gcd $x$ $y$ = if $x = 1$ then 1
                  else if $x = y$ then $x$
                  else if $x < y$ then gcd $y$ $x$
                  else gcd $(x - y)$ $y$
```

This program matches the unfold rule in four different ways, each either unfolding or not unfolding each of the two recursive calls. In the substitution below, g abstracts the occurrences of gcd that are to be unfolded and f the ones that remain unchanged in the transformation.

```
$b \longleftarrow \lambda g\ .\ \lambda h\ .$ if $x = 1$ then $1$
                        else if $x = y$ then $x$
                        else if $x < y$ then $g\ y\ x$
                        else $h\ (x - y)\ y$
```

Applying the substitution to **rec** $f = b[b[f][f]][f]$, the right-hand side of the rule, yields:

```
rec gcd $x\ y =$ if $x = 1$ then $1$
                  else if $x = y$ then $x$
                  else if $x < y$ then
                       (lam $x$ . lam $y$ .
                        if $x = 1$ then $1$
                         else if $x = y$ then $x$
                         else if $x < y$ then gcd $y\ x$
                         else gcd $(x - y)\ y)\ y\ x$
                  else gcd $(x - y)\ y$
```

Now we can apply **lam**-reduction twice, yielding the correctly unfolded program:

```
rec gcd $x$ $y$ = if $x = 1$ then 1
                  else if $x = y$ then $x$
                  else if $x < y$ then
                      if $y = 1$ then 1
                         else if $y = x$ then $y$
                         else if $y < x$ then gcd $x$ $y$
                         else gcd $(y - x)$ $x$
                  else gcd $(x - y)$ $y$
```

Then, using two simple rules for simplifying **if-then-else** expressions we can eliminate some of conditions to get

```
rec gcd $x$ $y$ = if $x = 1$ then 1
                  else if $x = y$ then $x$
                  else if $x < y$ then
                      if $y = 1$ then 1
                         else gcd $(y - x)$ $x$
                  else gcd $(x - y)$ $y$
```

Higher-order rule descriptions, like the description of Unfold, can be embedded in a programming language so that compositions of transformations like the one above can be described as short pieces of programs. A good example of such a system is λProlog [15]. Higher-order abstract syntax provides a way for systems like λProlog to use readable concrete syntax for programs without sacrificing power, simplicity of expression, or efficiency.

3.3 Propagating Contexts

Here is the promised solution to the challenge in Section 2.3. The context variable C simply becomes a second-order variable and will be instantiated only to proper contexts. A typical rule would be the following:

$$C[\textbf{if}\ p\ \textbf{then}\ a\ \textbf{else}\ b] \iff \textbf{if}\ p\ \textbf{then}\ C[a]\ \textbf{else}\ C[b]$$

See Figure 2 for an example of a context propagation rule and its use with higher-order abstract syntax. The last counterexample demonstrates how higher-order matching ensures that bound variables will not leave their scope. This restriction is not enforced through a test after a potential match is produced, but is an integral part of the higher-order unification algorithm.

3.4 Raising Rules

So far all examples have been second-order. However, there are many applications, in which one would like to use third-order matching. In the context of raised rules (as discussed here), our algorithm will always terminate (a result due to Miller [13]). Raised rules are a significant generalization of Huet & Lang's template matching [11].

Here is an example of an equivalence we would to obtain as a consequence of a general rule for context propagation.

```
let $x = y * y$ in
  let $z = x * y$ in
     if $y > 0$ then $z$ else $x$
$\Longleftrightarrow$
if $y > 0$
```

```
then let $x = y * y$ in
          let $z = x * y$ in $z$
else let $x = y * y$ in
          let $z = x * y$ in $x$
```

This does not match the context propagation rule from above, since a substitution like $a \longleftarrow z$ would be captured by the binding on z. A general solution in a case like this is to *raise* the order of the rule through explicit abstraction. This solution is inspired by Paulson's $\forall$-lifting [18] that was discovered independently by Miller and called raising [13]. Both raise the order of certain equations in the presence of parameters. Raising requires that we be able to explicitly mention the "λ" of the λ-calculus representation in the pattern. The following is a raised version of context propagation.

$$C[\lambda x \;.\; \textbf{if } p \textbf{ then } a[x] \textbf{ else } b[x]] \iff \textbf{if } p \textbf{ then } C[\lambda x \;.\; a[x]] \textbf{ else } C[\lambda x \;.\; b[x]]$$

$C[\textbf{if } p \textbf{ then } a \textbf{ else } b] \iff \textbf{if } p \textbf{ then } C[a] \textbf{ else } C[b]$

$x * (\textbf{if } x > 0 \textbf{ then } 1 \textbf{ else } -1) \iff \textbf{if } x > 0 \textbf{ then } x * 1 \textbf{ else } x * (-1)$
where $C \longleftarrow \lambda z \;.\; x * z,\;\; a \longleftarrow 1,\;\; b \longleftarrow -1,\;\; p \longleftarrow x > 0.$

$$(\textbf{if } x > 0 \textbf{ then } 1 \textbf{ else } -1) * x \iff \textbf{if } x > 0 \textbf{ then } 1 * x \textbf{ else } (-1) * x$$
where $C \longleftarrow \lambda z \,.\, z * x,\ a \longleftarrow 1,\ b \longleftarrow -1,\ p \longleftarrow x > 0.$

$$\textbf{let } q = \textbf{false in if } q \textbf{ then } 1 \textbf{ else } -1 \not\Longleftrightarrow \textbf{if } q \textbf{ then let } q = \textbf{false in } 1 \textbf{ else let } q = \textbf{false in } -1$$
since $C \longleftarrow (\lambda z.\textbf{let } q = \textbf{false in } z),\ a \longleftarrow 1,\ b \longleftarrow -1$ and $p \longleftarrow q$ leads to a clash.

Figure 2: Examples of context propagation.

This operation raised the order of a and b to be second-order variables, C is now a third-order variable of type $(E \to E) \to E$. A match of this pattern against its motivating example is given through the substitution

$$\begin{array}{lcl} C & \longleftarrow & \lambda f \,.\, \textbf{let } x = y * y \textbf{ in let } z = x * y \textbf{ in } f[\langle x, z\rangle] \\ p & \longleftarrow & y > 0 \\ a & \longleftarrow & \lambda\langle z_0, z_1\rangle \,.\, z_1 \\ b & \longleftarrow & \lambda\langle z_0, z_1\rangle \,.\, z_0 \end{array}$$

Again note the use of polymorphism in the rule description and the instantiation of type variables to product types to capture the fact that this should apply to any number of bound variables that may appear in the branches of the **if**, but not in the test p.

This also shows that variable occurrence conditions become unnecessary. The fact that p could not depend on any variable bound in the context is implicit in the formulation of the rule.

Our current incomplete support for polymorphism cannot handle this example.

4 Other Applications

Higher-order abstract syntax captures the binding information present in a language. We have shown in Section 3 how this can be used for correct unification, matching, and substitution and direct formulation of rewrite rules in the context of a language-generic environment for manipulating programs, formulas, and other syntactic objects.

In the following subsections we indicate how other components of such a generic environment can exploit the scoping information embedded in the representation.

4.1 Type Inference

In the Ergo Support System, we have prototyped a component for general type inference, given the semantic type signature of a language. Conceptually, the type signature is translated into an attribute grammar that will do type inference on a given term, though the actual implementation is slightly more efficient.

An early problem with our type signature specification language was that implicit binding constructs could not be handled. For example, in Prolog the binding of the free variables over a clause is implicit, and it need not have the same type as a variable with the same name in another clause. Other languages, where binding constructs are complicated could not be handled either. An example is an imperative language where an arbitrary number of local variable declarations may follow a **begin**.

With the use of higher-order abstract syntax these problems disappear, since the necessary analysis is done once and for all at parse time. Without going into detail, here is a possible type signature for our functional example language that is based on the higher-order abstract syntax representation. Note that this completely eliminates the need for analyzing the syntactic categories D, F, and B, which are not present in the higher-order abstract syntax. Constants of the object language may be declared separately — the last line is an example.

Briefly, the operator in the language (constant in the higher-order abstract syntax) is listed with their semantic types. Type constructors of the object language are $\mathsf{funtype}$, $\mathsf{prodtype}$, $\mathsf{boolean}$, and number, type variables are α and β.

rec	:	$(\alpha \to \alpha) \to \alpha$
ite	:	$\mathsf{boolean} \to \alpha \to \alpha \to \alpha$
let	:	$(\alpha \to \beta) \to \alpha \to \beta$
apply	:	$(\mathsf{funtype}\ \alpha\ \beta) \to \alpha \to \beta$
lam	:	$(\alpha \to \beta) \to (\mathsf{funtype}\ \alpha\ \beta)$
$*$	:	$(\mathsf{funtype}\ (\mathsf{prodtype}\ \mathsf{number}\ \mathsf{number})\ \mathsf{number})$

The higher-order nature of the abstract syntax makes this straightforward specification possible.

Note that this specification does not treat let in the way ML does, that is, it does not copy the type variables in instances of α.

4.2 Attribute Grammar Evaluator

Most attribute grammars on a language with binding constructs will have to pass around an environment as context in which the attribute value will be computed. Typically that environment is enlarged when descending through a binding construct of language. As mentioned above, determining where variables are bound often requires a significant amount of work, like collecting free variables from a term.

Using higher-order abstract syntax the repeated work can be avoided, thereby improving the efficiency of attribute analysis. Moreover, expression of attribute equations is more concise, since the environment of bound variables can effectively be hidden from the user in most cases.

The Analysis Facility in the Ergo Support System can currently process only attribute grammars that traverse first-order abstract syntax. The implementation of an extension of attribute grammars to take advantage of the higher-order abstract syntax is outlined here and planned for the near future.

Again, lacking space, the example is not fully explained but should indicate the form of such generalized attribute grammars. We show a fragment of an attribute grammar that evaluates expressions from our simple language. A higher-order subterm is treated as if it returns a function from the attribute value for the bound variable to the attribute value for the whole expression. That function can then be applied in the definition of other attributes. In the implementation, the function objects are never built, but they serve as a convenient, conceptual device. In effect, the attribute grammar will be translated into another computing the same value, but using environments in which the variable name is bound to the value to which the function would eventually be applied.

$$\begin{array}{rcl} E(\uparrow\mathsf{value}) & = & C(\uparrow\mathsf{value}) \mid \ldots \\ & \mid & \mathsf{let}([E \to E](\uparrow\mathsf{valuefn}), E(\uparrow\mathsf{value}^0)) \end{array}$$

$$
\begin{array}{ll}
 & \textbf{where } \mathsf{value} = \mathsf{valuefn}(\mathsf{value}^0) \\
| & \mathsf{apply}(E(\uparrow\mathsf{value}^0), E(\uparrow\mathsf{value}^1)) \\
 & \textbf{where } \mathsf{value} = \mathsf{value}^0(\mathsf{value}^1) \\
| & \mathsf{lam}([E \to E](\uparrow\mathsf{valuefn})) \\
 & \textbf{where } \mathsf{value} = \mathsf{valuefn}
\end{array}
$$

In this example, the evaluation of a **let** would be significantly more efficient that evaluation of and application of an abstraction to an argument. The reason is that the term corresponding to the **lam** expression is actually built first, then later reduced by application, while the reference to valuefn in the clause for **let** will be compiled into a form using environments.

4.3 Denotational Specification of Language Semantics

We have found that using higher-order abstract syntax significantly simplifies denotational semantics specifications. The key idea here, proposed in [11], is to specify the semantics of a language simply by giving an interpretation of the constants, and adopting the standard interpretations for variables, abstractions and applications.

Semantic specifications are *syntax-directed*, that is, the meaning of a compound expression should be a "function" of the meanings of its components in the abstract syntax. The constant interpretations are exactly these functions, and putting together terms with application has the semantic effect of applying the meaning of the constant to the meanings of the components. This frees the language specifier from this step and makes the semantic specifications very direct.

The standard technique for handling binding constructs is to make the meaning of an expression be a function from environments to values. For example, the semantics of a **let** expression in a call-by-name language is commonly given as

$$\mathcal{E}[\![\textbf{let}\ v = e_0\ \textbf{in}\ e_1]\!]\rho = \mathcal{E}[\![e_1]\!](\rho + [v \to \mathcal{E}[\![e_0]\!]])$$

Using the formulation of **let** given above, this simplifies greatly to

$$\mu[\![\mathsf{let}]\!] = \lambda f\ .\ \lambda a\ .\ f\ a$$

The formulation of the call-by-value case is only slightly more complicated.

Formal syntactic systems involving semantic properties (such as total or partial equivalence of meanings) that are based on higher-order abstract syntax will naturally be much simpler than their traditional counterparts. This is because there is less distance between the (abstract) syntax being manipulated and the meanings that motivate them.

5 Some Implementation Issues and Further Work

5.1 Implementation of Higher-Order Abstract Syntax

It seems that the most costly operations on abstract syntax are higher-order matching and unification. We therefore decided to implement the typed λ-terms, and provide the first-order interface through explicit conversion functions that are called when the first-order components of a term are accessed. Our positive experience with the current implementation confirms that this was the right decision. The first-order constituents of a λ-term are cached as they are computed, thus making term-traversal almost as efficient as for first-order abstract syntax. We have not noticed a performance loss during formatted unparsing, which is the most time-consuming operation carried out using the first-order view of abstract syntax. However, the representation of λ-terms themselves leaves room for improvement. Currently, we use the normal form described by Huet in [10], but lazy normalization and the use of deBruijn's notation [5] should lead to some efficiency improvements in the implementation of the unification algorithm.

5.2 Specification of Name Binding Constructs

Currently the language implementor has to supply some highly stylized Lisp functions that achieve the translations between higher- and first-order view of abstract syntax. We have not yet had enough experience to feel confident in a design of a user-oriented way of specifying the binding properties of a language.

5.3 Nondeterminism

In general during program transformation, many of the rules will be too non-deterministic to be simply matched against a given program. In such a case we need a more specific instance of a general rule. This more specific instance can be obtained through substitution for some of the free variables in the pattern. The correctness of the specialized rule follows directly from the correctness of the general rule.

If one uses higher-order matching as part of the execution of a higher-order logic program [15] then it is the programmer's responsibility to formulate and order the clauses in such a way that the non-determinism of the higher-order unification is controlled. This requires some experience with λProlog, but is in general not too difficult.

Another solution we have used successfully is to interactively point at subterms displayed in a window, for example, to identify subterms over which to abstract.

5.4 Language Representation

When one is faced with the task of implementing an object language using higher-order abstract syntax, one is faced with a variety of choices. The most straightforward representation is described in this paper: syntactic categories (usually a subset of the non-terminals) become types and there is a distinct, possibly higher-order constant for each operator in the language.

However, sometimes one would like to eliminate some of the syntactic sugar from the concrete syntax and thus map different concrete syntax expressions to the same representation. This is currently possible in our implementation, but not very well supported, since the unparser would only unparse into concrete syntax corresponding to the core language. The advantage of using a smaller core language is that one needs to define semantics, transformation rules, etc., only for the core language.

Another possibility for a more concise representation of a typed language is to use the types of the λ-calculus to represent the types of the object language. The class of languages for which this is possible is limited by the expressiveness of the type system of the higher-order abstract syntax.

6 Related Work

LF, the Edinburgh Logical Framework [1,9,7] makes explicit use of a different version of the λ-calculus to represent logical formulas and programs *and* the deduction system for reasoning about them. The way quantifiers are encoded is similar to the way it is done in higher-order abstract syntax and goes back to Church [4]. Because of the power of their λ-calculus with dependent function types, it is not known whether there is a natural unification algorithm that could be used for proof search in their system in a general way. However, their system is at the same time weaker in a different respect, since it does not allow products or polymorphism. This means that many natural formulations of transformation or inference rules cannot be represented in LF.

Isabelle [18] uses a representation similar to ours for the statement of rules, and uses higher-order unification for deduction. Isabelle's λ-calculus representation does not have the expressive power of higher-order abstract syntax, but explicitly encodes quantifier dependencies.

λProlog [14,12,17] provides a natural *metalanguage* for writing programs acting on higher-order abstract syntax. This was the basic point made in [15]. We plan a generalization of λProlog that would deal with products and thus allow us to use it as a very expressive metalanguage for metaprogramming in the context of language-generic program derivation and theorem proving.

7 References

[1] Arnon Avron, Furio A. Honsell, and Ian A. Mason. *Using typed Lambda Calculus to Implement Formal Systems on a Machine*. Technical Report ECS-LFCS-87-31, Laboratory for Foundations of Computer Science, University of Edinburgh, Edinburgh, Scotland, June 1987.

[2] Rolf Bahlke and Gregor Snelting. The PSG system: from formal language definitions to interactive programming environments. *ACM Transactions on Programming Languages and Systems*, 8(4):547–576, October 1986.

[3] 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.

[4] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[5] N. G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indag. Math.*, 34(5):381–392, 1972.

[6] David Garlan. *Views for Tools in Integrated Environments*. PhD thesis, Carnegie Mellon University, 1987. Available as Technical Report CMU-CS-87-147.

[7] Timothy G. Griffin. *An Environment for Formal Systems*. Technical Report 87-846, Department of Computer Science, Cornell University, Ithaca, New York, June 1987.

[8] Gandalf Group. Special issue on the Gandalf project. In *The Journal of Systems and Software*, Volume 5, Number 2, May 1985.

[9] 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.

[10] Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

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

[12] Dale Miller, Gopalan Nadathur, and Andre Scedrov. Hereditary Harrop formulas and uniform proof systems. In *Second Annual Symposium on Logic in Computer Science*, pages 98–105, IEEE, June 1987.

[13] Dale A. Miller. Unification under mixed prefixes. 1987. Unpublished manuscript.

[14] Dale A. Miller and Gopalan Nadathur. Higher-order logic programming. In *Proceedings of the Third International Conference on Logic Pro-*

gramming, Springer Verlag, July 1986.

[15] 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.

[16] B. Möller. *A survey of the project CIP: Computer-aided, intuition-guided programming.* Technical Report TUM–18406, Institut für Informatik der TU München, Munich, West Germany, 1984.

[17] Gopalan Nadathur. *A Higher-Order Logic as the Basis for Logic Programming.* PhD thesis, University of Pennsylvania, 1987.

[18] Lawrence Paulson. Natural deduction as higher-order resolution. *Journal of Logic Programming*, 3:237–258, 1986.

[19] Thomas Reps and Tim Teitelbaum. The synthesizer generator. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 42–48, ACM, New York, April 1984.

[20] David S. Wile. *POPART: Producer of Parsers and Related Tools, System Builder's Manual.* Technical Report, University of Southern California, Information Sciences Institute, 1987.