From Naïve to Norvig
On Deriving a PROLOG Compiler

Arthur Nunes-Harwitt
Rochester Institute of Technology
102 Lomb Memorial Drive
Rochester, New York 14623
anh@cs.rit.edu

ABSTRACT
An interpreter is a concise definition of the semantics of a programming language and is easily implemented. A compiler is more difficult to construct, but the code that it generates runs faster than interpreted code. This paper introduces rules to transform an interpreter into a compiler, and then provides a concrete demonstration of the rules in the form of a derivation of a PROLOG compiler, much like Norvig’s, implemented in Common LISP. This example also suggests that the approach can be applied to a wide range of interpreter implementations and related algorithms.

Categories and Subject Descriptors
D.3.4 [Programming Languages]: Prolog—Compilers; D.2.m [Software Engineering]: Miscellaneous—Rapid Prototyping

General Terms
Verification, Performance

Keywords
meta-programming, compilers, interpreters, partial evaluation, staging

1. INTRODUCTION
In his now classic text Paradigms of Artificial Intelligence Programming[15], Norvig discusses two PROLOG[6] implementations. The first is a naïve interpreter, and the second is a fairly sophisticated compiler that targets Common LISP[17]. Part of the difficulty of compiler writing is knowing what code to generate. Norvig presents a vision of what the generated Common LISP should look like. But how could someone with less insight make that determination? And further, what is the connection between a naïve interpreter and a compiler? This paper addresses these questions in general and illustrates the ideas by transforming a naïve PROLOG interpreter into a compiler that targets Common LISP.

Rational reconstructions of algorithms or results are worthwhile because they emphasize the key ideas involved. For example, Felleisen[9, 8], inspired by Landin[12], derives an SECD-like abstract machine from a term calculus. And Damvy[5] uses partial evaluation to derive the Boyer-Moore string-matching algorithm.

At one end of the spectrum, partial evaluation[11] is a mostly automatic program specialization technique. It can be used to turn interpreters into compilers. In particular, Futamura’s second projection[10], partially evaluating the partial evaluator with an interpreter, yields a compiler. However, such a fully automatic approach may not be desirable. If partial evaluation fails, it can be difficult to find the problem. For example, while Consel and Khoo’s partial evaluation based PROLOG compiler[4] was an important step and is very interesting, it is unable to compile recursive programs. It appears that the cause of the difficulty eluded the authors despite the fact that they were experts.

At the other end of the spectrum, staged computation[16, 14] is a mostly manual technique for specializing code. It involves writing code that, given a single input, does as much computation as possible and returns a function or function text that characterizes the rest of the computation that depends on additional inputs. For example, a Feeley-style staged interpreter[7, 1] is effectively a compiler that can be modified to generate program text. It is typically assumed that an interpreter is already expressed in a way so that staging is easy. In contrast, we discuss what property makes staging easy and go on to say how an interpreter that does not have this property can be transformed into one that does.

This paper makes the following contributions. It identifies a transformation technique in the form of four essential rules. A case study suggests the utility of these transformation rules and the transformation approach in general for deriving a compiler from an interpreter. The case study shows how the transformation rules are used, and it illustrates how the transformation technique can be applied to a class of interpreters even broader than the class of denotational-style interpreters.

2. TRANSFORMATIONS
When deriving a compiler from an interpreter, each step changes the code from the previous step. In this section, we

1In this paper, ‘denotational-style’ means compositional[18].
identify the changes that are common to the class of derivations we are considering. These changes can be formulated as meaning-preserving transformation rules. The ideas behind two of the rules, currying in section 2.1 and quoting in section 2.4, are well known. However, when two new rules are added, lambda lowering in section 2.2 and expression lifting in section 2.3, the combination becomes a powerful transformation technique.

Using currying, lambda lowering, and expression lifting, one can derive a Feeley-style interpreter from a naive interpreter written in a denotational style. Using the quoting rule, one can transform a Feeley-style interpreter that generates functions into a compiler that generates the text that represents those functions. Applying these rules to a denotational-style interpreter requires no ingenuity. We will see that even interpreters that are not written in a denotational style can be modified, using only a minute amount of ingenuity, so that the application of these rules still leads to a compiler.

In the following subsections, each rule is described informally, summarized more formally, analyzed informally for correctness, and finally demonstrated by a simple example of the application of the rule. Although the focus of this paper is on programming languages, these rules can be applied to any code. To illustrate the rules, the following Common Lisp function, which raises a number b to a natural number power n, is successively transformed.

```
(defun power (n b)
  (if (= n 0)
    1
    (* b (power (- n 1) b))))
```

### 2.1 Currying Dynamic Variables

Currying transforms a function of two arguments into a function of one argument that returns a function. Here we have in mind that one variable will be determined at one stage, or time, and the other will be determined at another stage, or time; we call the variable associated with the earlier stage static and the later stage dynamic. Of course, it is natural to generalize and allow any number of static variables and any number of dynamic variables.

Throughout this paper, λx.e refers to an anonymous function with a sequence of zero or more formal parameters x and with function body e. In rule (1) below, s refers to static variables and d refers to dynamic variables. The rule for currying now follows.

\[ \lambda s, d. e \mapsto \lambda s. \lambda d. e \]  

(1)

Further, procedure calls to the curried functions must be transformed to account for the alternate protocol.

\[ e_c(e_s, e_d) \leftrightarrow e_c(e_s)(e_d) \]  

(2)

if \( e_c \) is an expression that reduces to a curried function.

We omit the discussion of correctness in this case because the equivalence of functions and their curried counterparts is well known.

Currying the original example yields the following code.

```
(defun power (n)
  (lambda (b)
    (if (= n 0)
      1
      (* b (funcall (power (- n 1)) b)))))
```

### 2.2 Lambda Lowering

Lambda lowering involves moving a portion of an expression out of the function that it is inside as long as the portions moved do not depend on the parameters of the function. In particular, if the function body is a conditional, but the conditional does not depend on the function’s parameters, we may regard the conditional as specifying one of two functions. Or if the function body defines an intermediate value that does not depend on the parameters, we may regard the definition as occurring outside the body of the function.

Throughout this paper, \( \text{FV}(e) \) refers to the set of free variables of an expression e. The formal rules for lambda lowering follow.

\[ \lambda \overline{x}. \text{if } e \text{ then } e_1 \text{ else } e_2 \mapsto \text{if } e \text{ then } (\lambda \overline{x}. e_1) \text{ else } (\lambda \overline{x}. e_2) \]  

if \( x_i \notin \text{FV}(e) \)  

(3)

\[ \lambda \overline{x}. \text{let } z = e \text{ in } e_0 \mapsto \text{let } z = e \text{ in } \lambda \overline{x}. e_0 \]  

if \( x_i \notin \text{FV}(e) \) and \( z \neq x_i \)  

(4)

For rule (3), if e reduces to a value v, then the body of the function depends on v. When false, the body is \( e_2 \); otherwise the body is \( e_1 \). And that is what the right-hand-side says.

For rule (4), if e reduces to a value v, then the let on the left-hand side substitutes v for \( z \) in \( e_0 \). The let on the right-hand side substitutes v for \( z \) in the abstraction; however, since \( z \) is distinct from the formal parameters, it passes right through and becomes a substitution in \( e_0 \).

Note that both formulations assume that e always reduces to a value. If that is not the case, looping outside of a function is always observed; but looping inside a function is observed only if the function is called. In practice, as in the example above, it is clear whether or not the expression in question loops or not; typically it does not loop.

Lambda lowering the previous curried code now yields the following.

```
(defun power (n)
  (lambda (b)
    (if (= n 0)
      (lambda (b) 1)
      (lambda (b)
        (* b (funcall (power (- n 1)) b))))))
```

### 2.3 Expression Lifting

Expression lifting is similar to lambda lowering insofar as both involve moving an expression out of a function. With expression lifting the entire expression is moved completely out of the function if it does not depend on the parameters of the function. Typically, the expression being lifted is an application.

In the rules below in this subsection and the next, \( e[x := e'] \) means replace the variable \( x \) with the expression \( e' \) in the expression e avoiding variable capture. The formal rule for expression lifting follows.

\[ \lambda x.e[u := e'] \mapsto \text{let } z = e' \text{ in } \lambda x.e[u := z] \]  

if \( z \) is fresh and \( x_i \notin \text{FV}(e') \)  

(5)

If \( e' \) reduces to a value v, then the body of the function on the left-hand side will replace u with v. The let on the right-hand side also ultimately replaces u with v since the substitution for \( z \) passes right through the abstraction.

Here too, it is necessary to assume that e reduces to a value. If that is not the case, as before, looping outside
of a function is always observed, but looping inside a function is observed only if the function is called. However, in contrast to lambda lowering, one may encounter expressions that do loop when lifted. For interpreters not written in a denotational style, it may be that the recursion variable can be curried and then the remaining application can be lifted out; in such circumstances, recursion cannot be terminated! An example of this phenomenon involves the initial PROLOG implementation below in sections 3.1 and 3.2.

Expression lifting the previous lambda lowered code now yields the following.

\[
\begin{align*}
(\text{defun } \text{power} (n) & \quad \text{(if } (= n 0) \quad (\text{lambda (b) 1}) \quad \text{let ((f (power (- n 1)))}) \quad (\text{lambda (b)}) \quad (* b (funcall f b))))))
\end{align*}
\]

2.4 Code Via Quoting

Finally, the code via quoting rule introduces text generation. It transforms an expression that returns a function into an expression that returns the text that represents that function. The formal rule for quoting follows.

\[
\begin{align*}
\text{let } z_1 = e_1 \quad \text{in} \quad \text{let } z_n = e_n \text{ in } \lambda d. e \leftrightarrow \quad (6) \quad \text{let } z_1 = e_1 \quad \text{in} \quad [\lambda d. e[z_1 := z_1] \cdots [z_n := z_n]]
\end{align*}
\]

when \( \text{FV}(\lambda d. e) = \{z_1, \ldots, z_n\} \), each \( z_i \) is fresh, and \( \text{FV}(d_i) \cap \{z_1, \ldots, z_n\} = \emptyset \) for each \( i \).

Brackets \([\cdot]\) are used to indicate quoting. They are part of the implementation language rather than the meta-language; the substitution brackets are still part of the meta-language. The substitutions indicate that the let-bound variables \( (\cdot_1) \) are replaced with the unquoting form of the variables \( (\cdot_1) \).

With this rule, the transformed expression reduces to a different value from the original, and so here the notion of correctness is different. Rather, we take for granted an eval operator that treats text as a program to evaluate. Correctness means that applying the eval operator to the text that results from reducing the transformed expression leads to the same value as the original expression. We simply argue that since the text looks like the expression we would have evaluated right away, and name capture is avoided, then it must be that the same value is computed.

Quoting the code from the previous section now yields the following.

\[
\begin{align*}
(\text{defun } \text{power} (n) & \quad \text{(if } (= n 0) \quad (\text{lambda (b) 1}) \quad \text{let ((f (power (- n 1)))}) \quad (\text{lambda (b)}) \quad (* b (funcall f b))))))
\end{align*}
\]

2.5 Beyond Denotational Interpreters

These transformations will work on a denotational-style interpreter, but what about other kinds of interpreters? Indeed, for non-denotational interpreters these transformations are often insufficient. For example, consider an operational-style interpreter. In that case, it is quite reasonable to say that, just as a number evaluates to a number, an abstraction evaluates to an abstraction (or a closure): \( \mathcal{E}(\text{fun}(x)E, \text{env}) = \text{closure}(x, E, \text{env}) \), where \( \mathcal{E} \) is the evaluator, \( x \) is a variable, \( E \) is an expression, and env is the environment. The transformations do not introduce calls, and so there cannot be a compiler call on the sub-expression \( E \) since there is not one in the interpreter. Nevertheless, such an interpreter can be modified to be denotational. The call to the evaluator can be moved\(^2\) inside the closure hidden inside an abstraction:

\[
\begin{align*}
\mathcal{E}(\text{fun}(x)E, \text{env}) &= \text{closure}(x, \lambda (\text{env}'). \mathcal{E}(E, \text{env}'), \text{env})
\end{align*}
\]

A related but more subtle form of that problem occurs in a first-order language in which functions are not expressed values. If the interpreter is not denotational, the function environment contains merely program text. Instead of hitting a brick wall of program text, the transformations may appear to work, but the assumption for rule (5) is not satisfied, and recursive programs lead to infinite loops. For example, consider evaluating a procedure call:

\[
\begin{align*}
\mathcal{E}(y(E), \text{Fenv}, \text{env}) &= \mathcal{E}(E', \text{Fenv}, \text{env}_0[x \mapsto \mathcal{E}(E, \text{Fenv}, \text{env})])
\end{align*}
\]

where \( (x, E') = \text{Fenv}(y) \), and \( \mathcal{E} \) is the first-order evaluator, \( y \) is a function-variable, \( E \) and \( E' \) are expressions, \( \text{Fenv} \) is the function environment, \( \text{env}_0 \) is the initial environment, and \( \text{env} \) is the environment.

If the expression and the function environment are considered static, and if \( E' \), the body of the function \( y \), contains a recursive call \( (\text{i.e., a call to } y) \), then the transformed evaluator will forever attempt to determine that meaning.

It could be that this was the phenomenon that stumped Consel and Khoo. The solution is to consider the denotation of the function environment, and as before move the call to the evaluator to the program text. Thus the denotation of the function environment \( \{ (y, (x, E')) \}, \ldots \) is the following. \( \{ (y, (x, \lambda (\text{Denv}' . \text{env}'). \mathcal{E}(E', \text{Denv}', \text{env}'))), \ldots \} \)

Evaluating a procedure call now looks as follows.

\[
\begin{align*}
\mathcal{E}(y(E), \text{Denv}, \text{env}) &= f (\text{Denv}, \text{env}_0[x \mapsto \mathcal{E}(E, \text{Denv}, \text{env})])
\end{align*}
\]

where \( (x, f) = \text{Denv}(y) \).

Now the expression is the only static parameter, and every call is on a smaller sub-expression.

3. COMPILER DERIVATION

We start with a naïve PROLOG interpreter written in a natural style similar to Norvig’s. This original interpreter is transformed into a denotational-style interpreter with efficient unification. Then the rules in section 2 are applied so as to derive a compiler.

3.1 A Naïve Interpreter

Our naïve PROLOG interpreter is not identical to Norvig’s. While he represents a success continuation as a single list of terms to prove, we represent both success and failure continuations explicitly as functions. Although the naïve interpreter is written in a functional style, there is no notion of a denotation of the program/database (\( \Delta \)). The notation for the pseudo-code borrows from ML and Haskell.

First we introduce the types used in the interpreter pseudo-code below. The class of predicates \( \text{Pred} \) and the class of terms \( \text{Term} \) are at the core of a description of \( \text{PROLOG} \) syntax. A predicate looks like a Boolean-valued function call, and a term is either a number, a variable, or a predicate.

\(^2\) Of course, the meaning of the application must be correspondingly modified as well. Our solution is not unique, but we feel that it is the simplest.
Backus-Naur form is used to formally define these classes:

\[
\begin{align*}
  t & \in \text{Term} \quad ::= \quad n \\
  & \quad ::= \quad X \\
  & \quad ::= \quad P \\
  P & \in \text{Pred} \quad ::= \quad i(t_1, \ldots, t_n)
\end{align*}
\]

where \( n \) is a number, \( i \) is an identifier, \( V \) is the set of all variables, \( X \in V \) is a variable, and \( t_k \) is a Term.

A substitution is used to map variables to terms. Sub denotes the class of substitutions, which is shorthand for \( \text{Var} \rightarrow \text{Term} \). A clause is a predicate that is the consequence of some conjunction of predicates. Clause denotes the class of clauses, which is shorthand for \( \text{Pred} \times [\text{Pred}] \). Now a Prolog program/database is understood as a sequence, or list, of clauses.

Pseudo-code for a naïve Prolog interpreter consists of four functions charged with proving results and support functions for unification and for copying terms. The last four arguments for the proving functions are the unification substitution \( \theta \), the database \( \Delta \), the success continuation \( \kappa_s \), and the failure continuation \( \kappa_f \). A success continuation takes a failure continuation so that additional solutions can be found. The types of these variables now follow\(^3\).

\[
\begin{align*}
  \theta : & \quad \text{Sub} \\
  \Delta : & \quad \text{Clause} \\
  \kappa_s : & \quad \text{Sub} \times (()) \rightarrow \text{Answer} \rightarrow \text{Answer} \\
  \kappa_f : & \quad () \rightarrow \text{Answer}
\end{align*}
\]

The interactive entry point to the proving functions is the prove-query function \( \mathcal{P}_Q \) which attempts to prove all the predicates of a query using the initial substitution and the initial continuations. It takes a list of predicates and a database.

\[
\mathcal{P}_Q(\Pi, \Delta) = \mathcal{P}_A(\Pi, \theta_0, \Delta, \kappa_{s0}, \kappa_{f0})
\]

Prove-query delegates to a function prove-all \( \mathcal{P}_A \) that attempts to prove all the predicates in the list \( \Pi \): if the list is empty it succeeds, otherwise prove \( P \) is called on the first term with the success continuation extended to prove the rest.

\[
\begin{align*}
  \mathcal{P}_A([], \theta, \Delta, \kappa_s, \kappa_f) & = \kappa_s(\theta, \kappa_f) \\
  \mathcal{P}_A(P : \Pi, \theta, \Delta, \kappa_s, \kappa_f) & = \\
  & = \mathcal{P}(P, \theta, \Delta, (\lambda(\theta', \kappa_f'). \mathcal{P}_A(\Pi, \theta', \Delta, \kappa_s, \kappa_f')), \kappa_f) \\
\end{align*}
\]

The function prove \( \mathcal{P} \) attempts to prove a predicate by delegating to prove-goal \( \mathcal{P}_G \).

\[
\mathcal{P}(P, \theta, \Delta, \kappa_s, \kappa_f) = \mathcal{P}_G(\Delta, P, \theta, \Delta, \kappa_s, \kappa_f)
\]

Finally, the function prove-goal \( \mathcal{P}_G \) attempts to prove a predicate by proving one clause \( \eta \) from its database of clauses. If the list of clauses is empty it fails. Otherwise it attempts to unify the predicate with the predicate in the first clause. If that succeeds it then attempts to prove all the predicates in the clause predicate list by invoking prove-all \( \mathcal{P}_A \); if that fails it tries another clause.

\[\text{The specific type of Answer is left undefined. It is possible for the initial continuations to convey the results using a variety of types.}\]

\[
\begin{align*}
  \mathcal{P}_G([], P, \theta, \Delta, \kappa_s, \kappa_f) & = \kappa_f() \\
  \mathcal{P}_G(\eta : \Delta', P, \theta, \Delta, \kappa_s, \kappa_f) & = \\
  & \quad \text{let } (P', \Pi) = \text{copy}(\eta) \\
  & \quad \kappa_f' = \lambda(), \mathcal{P}_G(\Delta', P, \theta, \Delta, \kappa_s, \kappa_f) \\
  & \quad \quad \text{in unify}(P', P, \theta, (\lambda(\theta'). \mathcal{P}_A(\Pi, \theta', \Delta, \kappa_s, \kappa_f')), \kappa_f')
\end{align*}
\]

The function prove-goal \( \mathcal{P}_G \) makes use of two functions: \text{copy} and \text{unify}. The function \text{copy} copies terms and renames any variables in the copied clause. It is this mechanism that ensures that local variables do not affect other local variables of the same name. Generating fresh variable names requires state, which could be maintained using an additional parameter; the details are not shown here.

The function unify determines if two terms unify. If so, the unify success continuation, which takes only a substitution parameter, is called. If not, the unify failure continuation is called. The details of the unification implementation are also not shown.

Missing features include primitive predicates, cut, and special cases for tail-recursion. These pose no particular challenge and are omitted for the sake of brevity. A hook for primitive predicates could be added to prove-all \( \mathcal{P}_A \). Supporting cut requires a third continuation parameter initialized by prove. Supporting tail-recursion involves adding cases for singleton lists to prove-all \( \mathcal{P}_A \) and prove-goal \( \mathcal{P}_G \).

### 3.2 Efficiency and Denotation

The function prove-goal \( \mathcal{P}_G \) looks through the entire data-base \( \Delta \) to find a match. Thus unify is invoked on the head of every clause. Unification is somewhat heavyweight. It is possible to filter out clauses that cannot unify using lighter-weight comparisons based on the clause head’s identifier and the clause head’s arity. We use the notation \( \Delta(i/n) \) to indicate filtering by identifier and arity because this filtering resembles an environment look-up. The pseudo-code is modified as follows.

Instead of passing the predicate to prove, prove-all \( \mathcal{P}_A \) passes the components of the predicate: the identifier, the arity, and the list of terms.

\[
\begin{align*}
  \mathcal{P}_A([], \theta, \Delta, \kappa_s, \kappa_f) & = \kappa_s(\theta, \kappa_f) \\
  \mathcal{P}_A([i(t_1, \ldots, t_n)]) : & \quad \Pi, \theta, \Delta, \kappa_s, \kappa_f) = \\
  & \quad \mathcal{P}(i, n, [t_1, \ldots, t_n], \theta, \Delta, (\lambda(\theta', \kappa_f'). \mathcal{P}_A(\Pi, \theta', \Delta, \kappa_s, \kappa_f')), \kappa_f)
\end{align*}
\]

Prove \( \mathcal{P} \) then uses the identifier and the arity to find only the relevant clauses in the database. Since all the clauses passed to prove-goal \( \mathcal{P}_G \) have the same identifier, only the list of terms \( \tau \) needs to be passed.

\[
\mathcal{P}(i, n, \tau, \theta, \Delta, \kappa_s, \kappa_f) = \mathcal{P}_G(\Delta(i/n), \tau, \theta, \Delta, \kappa_s, \kappa_f)
\]

Since prove-goal \( \mathcal{P}_G \) now takes a term-list parameter rather than a predicate, the unification function must be correspondingly adjusted.

\[
\begin{align*}
  \mathcal{P}_G([], \tau, \theta, \Delta, \kappa_s, \kappa_f) & = \kappa_f() \\
  \mathcal{P}_G(\eta : \Delta', \tau, \theta, \Delta, \kappa_s, \kappa_f) & = \\
  & \quad \text{let } (i(t_1, \ldots, t_n), \Pi) = \text{copy}(\eta) \\
  & \quad \kappa_f' = \lambda(), \mathcal{P}_G(\Delta', \tau, \theta, \Delta, \kappa_s, \kappa_f) \\
  & \quad \quad \text{in unify}(i(t_1, \ldots, t_n), \tau, \theta, (\lambda(\theta'). \mathcal{P}_A(\Pi, \theta', \Delta, \kappa_s, \kappa_f')), \kappa_f')
\end{align*}
\]

From this point of view it becomes apparent that \( \Delta(i/n) \) maps to a list of clauses (i.e., program text) and not to any
denoted value. As discussed in section 2.5, the transformation rules may fail on an interpreter that is not denotational. In fact, if the transformation technique were applied to the interpreter above, it would yield a compiler that looped on recursive PROLOG programs. We make use of the solution discussed in that section, and we introduce a new function \( D \) that maps a database to its denotation. The call to prove-goal (\( P_G \)) is moved from prove (\( P \)) to the denotation of a collection of clauses. A database denotation can be understood either as a table or a function that maps an identifier and arity to a single function that succeeds or fails based on whether or not the terms provided satisfy the indicated predicate. We use \( \delta = D(\Delta) \) as notation for a database denotation.

\[
D(\Delta) = \{(i/n, (\lambda(\tau, \theta, \delta, \kappa_s, \kappa_f). P_G(\Delta(i/n), \tau, \theta, \delta, \kappa_s, \kappa_f))) | i \text{ is an identifier, } n \in \mathbb{N}\}
\]

Prove-query, prove-all, prove, and prove-goal must all be modified to take the database denotation.

\[
P_Q(\Pi, \delta) = P_A(\Pi, \theta_0, \delta, \kappa_s0, \kappa_f0)
\]

\[
P_A([], \theta, \delta, \kappa_s, \kappa_f) = \kappa_s(\theta, \kappa_f)
\]

\[
P_A((i(t_1, \ldots, t_n)), \theta, \delta, \kappa_s, \kappa_f) =
\begin{align*}
& P(i, n, [t_1, \ldots, t_n], \theta, \delta, (\lambda(\theta', \kappa_f'). P_A(\Pi, \theta', \delta, \kappa_s, \kappa_f')), \kappa_f) \\
& \text{Prove } (P) \text{ now looks very different. Instead of supplying prove-goal } (P_G) \text{ with some text from the database, it calls the function associated with the identifier and arity.}
\end{align*}
\]

\[
P(i, n, \tau, \theta, \delta, \kappa_s, \kappa_f) = \delta(i/n)(\tau, \theta, \delta, \kappa_s, \kappa_f)
\]

\[
P_G(\eta, \tau, \theta, \delta, \kappa_s, \kappa_f) = \kappa_f(\eta)
\]

\[
P_G(\eta \triangleright \Delta, \tau, \theta, \delta, \kappa_s, \kappa_f) =
\begin{align*}
& \text{let } (i(t_1, \ldots, t_n), \Pi) = \text{copy}(\eta) \\
& \kappa_f' = \lambda(). P_G(\Delta, \tau, \theta, \delta, \kappa_s, \kappa_f) \\
& \text{in unify}(t_1, \ldots, t_n), \tau, \theta, (\lambda(\theta'). P_A(\Pi, \theta', \delta, \kappa_s, \kappa_f')), \kappa_f')
\end{align*}
\]

At this point, the interpreter in the form of the four proving functions is expressed in denotational-style, and the transformation technique applies. Before applying the technique we improve the efficiency of unification.

### 3.3 Improving Unification

Norvig comments that a possible next step is improving unification. He forgoes adding a union-find based unification algorithm to the interpreter in order to jump directly to a compiler. In this paper, we add a more sophisticated unification algorithm to the interpreter so as to derive a compiler. The substitution (\( \theta \)) is removed. We also take the opportunity here to avoid completely copying a term, and instead determine a mapping from variables to logic variables. This mapping is written in factored form (\( \pi \circ \phi \)) to allow more static computation to occur. The first factor is referred to as the pre-frame and the second as the frame.

A pre-frame is the class of mappings from variables to natural numbers. \( \text{PreFrame} \) denotes the class of pre-frames, which is shorthand for \( \mathbb{N} \rightarrow \text{LogicVar} \). A frame is the class of mappings from natural numbers to logic variables. \( \text{Frame} \) denotes the class of frames, which is shorthand for \( \mathbb{N} \rightarrow \text{LogicVar} \).

Additional types are necessary when the unification algorithm changes. The class of terms is still used to describe what a PROLOG program looks like, but since the union-find approach to unification turns variables into data-structures, a new class of values is needed that describes the run-time data structures. A functor looks like a constructor function call, and a value is either a number, a logic variable, or a functor. \( \text{Funct} \) denotes the class of functors and \( \text{Value} \) denotes the class of values. Backus-Naur form is used to formally define these classes.

\[
\begin{align*}
v & \in \text{Value} \quad ::= \quad n \quad ::= \quad \chi \\
& \quad ::= \quad F \\
F & \in \text{Funct} \quad ::= \quad i(v_1, \ldots, v_n)
\end{align*}
\]

where \( n \) is a number, \( i \) is an identifier, \( \chi \in \text{LogicVar} \) is a logic variable, and \( v_k \) is a Value.

It is possible to say more about the structure of logic variables. When displaying logic variables it is useful to have the original variable from the program and a number to distinguish it from others, but the essence is simply a container. Thus \( \text{LogicVar} \) is short-hand for \( \mathbb{N} \times \text{Var} \times \text{Location} \).

Now the arguments to the proving functions are more varied. The last two arguments for all of them are again the success continuation and the failure continuation. Because a substitution is no longer used, the type of the success continuation has changed.

\[
\begin{align*}
\kappa_s & : (\delta) \rightarrow \text{Answer} \rightarrow \text{Answer} \\
\kappa_f & : () \rightarrow \text{Answer}
\end{align*}
\]

Prove-all (\( P_A \)) takes a pre-frame and a frame.

\[
\begin{align*}
\pi & : \text{PreFrame} \\
\phi & : \text{Frame}
\end{align*}
\]

Prove (\( P \)) and prove-goal (\( P_G \)) take a list of values.

\[
\begin{align*}
\nu & : [\text{Value}]
\end{align*}
\]

The previous pseudo-code is transformed to make use of the more sophisticated unification algorithm. Destructive unification and logic variables require state. Here too the state parameter and the implementation details are not shown. This improvement to unification does improve the overall performance of the interpreter.

The functions in the database denotation are now constructed so that the arguments correspond to the arguments that prove-goal (\( P_G \)) needs.

\[
D(\Delta) = \{(i/n, (\lambda(\nu, \delta, \kappa_s, \kappa_f). P_G(\Delta(i/n), \nu, \delta, \kappa_s, \kappa_f))) | i \text{ is an identifier, } n \in \mathbb{N}\}
\]

Instead of merely supplying an initial substitution, prove-query (\( P_Q \)) must construct an initial pre-frame and frame based on the variables in the predicate list \( \Pi \) to supply to prove-all (\( P_A \)). The function \text{varsFromPred} returns a list of all the unique variables in a predicate list. The function \text{newPreFrame} turns a list of variables into a map from variables to numbers, and the function \text{newFrame} turns that same list of variables into a map from numbers to logic variables.

\[
P_Q(\Pi, \delta) =
\begin{align*}
& \text{let } V = \text{varsFromPred}(\Pi) \\
& \pi_0 = \text{newPreFrame}(V) \\
& \phi_0 = \text{newFrame}(V) \\
& \text{in } P_A(\Pi, \pi_0, \phi_0, \delta, \kappa_s0, \kappa_f0)
\end{align*}
\]
Prove-all \( (P_A) \) now uses the function \( \text{toValues} \) to convert its term list to a list of values so that prove \( (P) \) will not need to keep track of a frame.

\[
P_A([], \pi, \phi, \delta, \kappa_s, \kappa_f) = \kappa_s(\kappa_f)
\]

\[
P_A((i(t_1, \ldots, t_n)) :: \Pi, \pi, \phi, \delta, \kappa_s, \kappa_f) = \mathcal{P}(i, n, \text{toValues}([t_1, \ldots, t_n], \pi, \phi), \delta, (\lambda(\kappa_f'). P_A(\Pi, \pi, \phi, \delta, \kappa_s, \kappa_f')), \kappa_f)
\]

\[
P(i, n, \nu, \delta, \kappa_s, \kappa_f) = \delta(i/n)(\nu, \delta, \kappa_s, \kappa_f)
\]

Prove-goal \( (P_G) \) does not take a substitution any more, nor does it take a frame. However, it creates a frame based on the variables in the first clause from its list of clauses using the function \( \text{varsFromClause} \). This frame is supplied to the unification algorithm where the values associated with the terms can be unified with the list of values \( (\nu) \) that were supplied as an argument. The unification function is adjusted again so that destructive unification is used and logic variables are set and unset. The unsetting is hidden in the unification failure continuations.

\[
P_G([], \nu, \delta, \kappa_s, \kappa_f) = \kappa_f()
\]

\[
P_G(\eta :: \Delta, \nu, \delta, \kappa_s, \kappa_f) = \eta
\]

\[
\text{let } (i(t_1, \ldots, t_n), \Pi) = \eta
\]

\[
V = \text{varsFromClause}(\eta)
\]

\[
\pi = \text{newPreFrame}(V)
\]

\[
\phi = \text{newFrame}(V)
\]

\[
in \text{unify}([t_1, \ldots, t_n], \pi, \phi, \nu, (\lambda(\kappa_f'). P_A(\Pi, \pi, \phi, \delta, \kappa_s, \kappa_f')), (\lambda(). P_G(\Delta, \nu, \delta, \kappa_s, \kappa_f)))
\]

At this point, not only is the interpreter expressed in a denotational-style, but it also uses an efficient unification algorithm. The transformation technique can now effectively be applied to this interpreter. The next subsections describe their application.

### 3.4 Currying and Lambda Lowering

Currying involves little more than moving the dynamic parameters from the left to the right side of the equal sign. Since conditionals are implicit and involve separate equations, this movement also achieves lambda lowering. The lambda lowering that remains is to lower the lambda through the let in prove-goal \( (P_G) \). Of course, all the calls to the curried functions must be modified.

The pseudo-code from the previous subsection is transformed using rules \( (1), (2), \) and \( (3) \) from section 2. When prove-goal \( (P_G) \) is curried, it produces a function. Thus the abstraction in the database denotation can be eta-reduced.

\[
D(\Delta) = \{ i/n, P_G(\Delta(\lambda i/n)) \mid i \text{ is an identifier, } n \in \mathbb{N} \}
\]

For prove-query \( (P_Q) \), the query itself is static, but the database denotation remains dynamic and is curried. We also see that prove-all \( (P_A) \) has been curried and that the call has been changed.

\[
P_Q(\Pi) = \lambda(\delta).
\]

\[
\text{let } V = \text{varsFromPred}(\Pi)
\]

\[
\pi_0 = \text{newPreFrame}(V)
\]

\[
\phi_0 = \text{newFrame}(V)
\]

\[
in P_A(\Pi, \pi_0)(\phi_0, \delta, \kappa_{\phi_0}, \kappa_{\phi_0})
\]

The static parameters for prove-all \( (P_A) \) are the list of predicates and the pre-frame. The frame itself, the database denotation, and the continuations remain dynamic and are curried. We see that the recursive call to \( P_A \) and the call to \( \text{toValues} \) have been adjusted.

\[
P_A([], \pi) = \lambda(\phi, \delta, \kappa_s, \kappa_f). \kappa_s(\kappa_f)
\]

\[
P_A((i(t_1, \ldots, t_n)) :: \Pi, \pi) = \lambda(\phi, \delta, \kappa_s, \kappa_f). \mathcal{P}(i, n, \text{toValues}([t_1, \ldots, t_n], \pi)(\phi), \delta, (\lambda(\kappa_f'). P_A(\Pi, \pi)(\phi, \delta, \kappa_s, \kappa_f')), \kappa_f)
\]

The function \( \text{prove} \) is, in effect, a call operator. All of the parameters are dynamic, so no currying occurs here.

\[
P(i, n, \nu, \delta, \kappa_s, \kappa_f) = \delta(i/n)(\nu, \delta, \kappa_s, \kappa_f)
\]

For prove-goal \( (P_G) \), only the list of clauses is static. All the other parameters are curried. Not only are the calls to prove-goal and prove-all adjusted, but also the call to \( \text{unify} \) is adjusted as well since it is also curried.

\[
P_G(\eta :: \Delta) = \lambda(\nu, \delta, \kappa_s, \kappa_f).
\]

\[
\text{let } (i(t_1, \ldots, t_n), \Pi) = \eta
\]

\[
V = \text{varsFromClause}(\eta)
\]

\[
\pi = \text{newPreFrame}(V)
\]

\[
\phi = \text{newFrame}(V)
\]

\[
in \text{unify}([t_1, \ldots, t_n], \pi)
\]

\[
(\phi, \nu, (\lambda(\kappa_f'). P_A(\Pi, \pi)(\phi, \delta, \kappa_s, \kappa_f')), (\lambda(). P_G(\Delta)(\nu, \delta, \kappa_s, \kappa_f)))
\]

### 3.5 More Lambda Lowering

In addition to lowering the lambdas into the conditional expressions, we lower the lambdas into the let\(^*\) as well. This move allows the variables to be determined statically. Then only the frame is allocated dynamically.

The previous pseudo-code is now transformed using rule \( (4) \) from section 2.

The initial pre-frame \( (\pi_0) \) depends only on the static list of predicates \( (\Pi) \), so the lambda can be lowered into the first let. However, the initial frame \( (\phi_0) \) implicitly depends on the dynamic store since allocating a frame involves storage allocation. Therefore, the lambda cannot be lowered into that let.

\[
P_Q(\Pi) = \lambda(\delta).
\]

\[
\text{let } V = \text{varsFromPred}(\Pi)
\]

\[
\pi_0 = \text{newPreFrame}(V)
\]

\[
in \lambda(\delta). \text{let } \phi_0 = \text{newFrame}(V)
\]

\[
in P_A(\Pi, \pi_0)(\phi_0, \delta, \kappa_{\phi_0}, \kappa_{\phi_0})
\]

\(^*\)The let pseudo-code construct should be understood as a sequence of individual lets; i.e., a Common LISP \texttt{let*}. 
Similarly, in prove-goal ($P_G$) the lambda can be lowered past the pre-frame but no lower.

\[ P_G(\emptyset) = \lambda(\nu, \delta, \kappa_s, \kappa_f) \]
\[ P_G(\eta :: \Delta) = \]
\[
\begin{align*}
\text{let } & (i(t_1, \ldots, t_n), \Pi) = \eta \\
V & = \text{varsFromClause}(\eta) \\
\pi & = \text{newPreFrame}(V) \\
in & \lambda(\nu, \delta, \kappa_s, \kappa_f). \\
\text{let } & \phi = \text{newFrame}(V) \\
& \text{in } \text{unify}([t_1, \ldots, t_n], \pi) \\
&& (\phi, \nu, \\
&& (\lambda(\kappa'_f). P_A(\Pi, \pi)(\phi, \delta, \kappa_s, \kappa'_f)), \\
&& (\lambda()). P_G(\Delta)(\nu, \delta, \kappa_s, \kappa_f)) 
\end{align*}
\]

### 3.6 Expression Lifting

Since the calls to the prove functions do not depend on the dynamic variables, it is possible to lift them out of the lowered lambdas. We do this now so that the functions can be generated statically rather than dynamically.

Now the pseudo-code from the previous section is transformed using rule (5) from section 2.

In prove-query ($P_Q$), the call to prove-all ($P_A$) only depends on the list of predicates and the pre-frame, so it can be lifted.

\[ P_Q(\emptyset) = \]
\[
\begin{align*}
\text{let } & V = \text{varsFromPred}(\Pi) \\
\pi_0 & = \text{newPreFrame}(V) \\
f & = P_A(\Pi, \pi_0) \\
in & \lambda(\delta). \text{let } \phi_0 = \text{newFrame}(V) \\
& \text{in } f(\phi_0, \delta, \kappa_s, \kappa_f) 
\end{align*}
\]

Prove-all ($P_A$) has a recursive call. This call is lifted as well as a call to toValues which depends only on the predicate and the pre-frame.

\[ P_A(\emptyset, \pi) = \lambda(\phi, \delta, \kappa_s, \kappa_f). P_r(\phi, \delta, \kappa_s, \kappa_f) \]
\[ P_A(i(t_1, \ldots, t_n), \Pi, \pi) = \]
\[
\begin{align*}
\text{let } & f_1 = \text{toValues}([t_1, \ldots, t_n], \pi) \\
f_2 & = P_A(\Pi, \pi) \\
in & \lambda(\phi, \delta, \kappa_s, \kappa_f). P_r(\phi, \delta, \kappa_s, \kappa_f), \\
& (f_1(\phi), \nu, \\
& (\lambda(\kappa'_f). f_2(\phi, \delta, \kappa_s, \kappa'_f)), \\
& (\lambda()). f_3(\nu, \delta, \kappa_s, \kappa_f)) 
\end{align*}
\]

In prove-goal ($P_G$), it is also possible to lift the call to prove-all ($P_A$). In addition, the recursive call is lifted as is the static portion of unification.

\[ P_G(\emptyset) = \lambda(\nu, \delta, \kappa_s, \kappa_f) \]
\[ P_G(\eta :: \Delta) = \]
\[
\begin{align*}
\text{let } & (i(t_1, \ldots, t_n), \Pi) = \eta \\
V & = \text{varsFromClause}(\eta) \\
\pi & = \text{newPreFrame}(V) \\
f_1 & = \text{unify}([t_1, \ldots, t_n], \pi) \\
f_2 & = P_A(\Pi, \pi) \\
f_3 & = P_G(\Delta) \\
in & \lambda(\nu, \delta, \kappa_s, \kappa_f). \\
& \text{let } \phi = \text{newFrame}(V) \\
& \text{in } f_1(\phi, \nu, \\
& (\lambda(\kappa'_f). f_2(\phi, \delta, \kappa_s, \kappa'_f)), \\
& (\lambda()). f_3(\nu, \delta, \kappa_s, \kappa_f)) 
\end{align*}
\]

### 3.7 Code Generation

Since we are interested in generating text, we quote the relevant portions: the functions with dynamic variables. Rule (6) is applied to the previous pseudo-code.

In prove-query ($P_Q$), the abstraction inside the let is quoted using the bracket notation, and the variables $V$ and $f$ are unquoted using the comma notation. Technically, for rule (6) the variables $V$ and $f$ should be defined such that one immediately comes after the other. While this technical issue could be resolved by introducing an additional let binding, we omit this binding for readability.

\[ P_Q(\Pi) = \]
\[
\begin{align*}
\text{let } & V = \text{varsFromPred}(\Pi) \\
\pi_0 & = \text{newPreFrame}(V) \\
f & = P_A(\Pi, \pi_0) \\
in & [\lambda(\delta). \text{let } \phi_0 = \text{newFrame}(V) \\
& \text{in } (f(\phi_0, \delta, \kappa_s, \kappa_f)) ] 
\end{align*}
\]

The prove-query function now generates text that interfaces with the text of the database denotation.

The abstraction in the let is quoted in prove-all ($P_A$). The variables $f_1$ and $f_2$ are unquoted as are $i$ and $n$. Again, technically, for rule (6) the variables $i$ and $n$ should be part of the let bindings, but these are omitted for readability.

\[ P_A(\emptyset, \pi) = [\lambda(\phi, \delta, \kappa_s, \kappa_f). \kappa_s(\kappa_f)] \\
P_A(i(t_1, \ldots, t_n), \Pi, \pi) = \]
\[
\begin{align*}
\text{let } & f_1 = \text{toValues}([t_1, \ldots, t_n], \pi) \\
f_2 & = P_A(\Pi, \pi) \\
in & [\lambda(\phi, \delta, \kappa_s, \kappa_f). P_r((i, i), \\
& (n), \nu, \delta, \\
& (\lambda(\kappa'_f). (f_2(\phi, \delta, \kappa_s, \kappa'_f)), \kappa_f)) ] 
\end{align*}
\]

For prove-goal ($P_G$), the abstraction in the let is again quoted. The variables $f_1, f_2, f_3, \text{and } V$ are unquoted.

\[ P_G(\emptyset) = [\lambda(\nu, \delta, \kappa_s, \kappa_f). \kappa_s(\kappa_f)] \\
P_G(\eta :: \Delta) = \]
\[
\begin{align*}
\text{let } & (i(t_1, \ldots, t_n), \Pi) = \eta \\
V & = \text{varsFromClause}(\eta) \\
\pi & = \text{newPreFrame}(V) \\
f_1 & = \text{unify}([t_1, \ldots, t_n], \pi) \\
f_2 & = P_A(\Pi, \pi) \\
f_3 & = P_G(\Delta) \\
in & [\lambda(\nu, \delta, \kappa_s, \kappa_f). \\
& \text{let } \phi = \text{newFrame}(V) \\
& \text{in } (f_1(\phi, \nu, \\
& (\lambda(\kappa'_f). (f_2(\phi, \delta, \kappa_s, \kappa'_f)), \\
& (\lambda()). f_3(\nu, \delta, \kappa_s, \kappa_f)))] 
\end{align*}
\]

The database denotation $D(\Delta)$ is now text that constructs a table associating identifiers augmented with the arity with program text generated by prove-goal ($P_G$), and $\delta$ is initialized to the table generated by that text when executed.
Hence $D$ is now a compiler. To use this compiler interactively, the database denotation must be loaded, and any query text generated must be subsequently executed.

4. PERFORMANCE

To gauge the performance of the compiler derived in section 3 we make use of a simple benchmark: the naïve Fibonacci function using Peano arithmetic. We found Norvig’s implementation to be quite respectable with timing results that appeared to be the same as SWI PROLOG (version 6.6.1). We directly compare our implementation only to Norvig’s. We used SBCL Common Lisp version 1.1.6.0-3c5581a using the default configuration on a MacBook Pro running Mac OS 10.7.5 on a 2GHz Intel Core i7 with 6GB of memory.

<table>
<thead>
<tr>
<th>fib(15)</th>
<th>Norvig</th>
<th>compiler</th>
<th>interpreter</th>
</tr>
</thead>
<tbody>
<tr>
<td>user time (ms)</td>
<td>2.7</td>
<td>4.3</td>
<td>20.0</td>
</tr>
<tr>
<td>GC time (ms)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MB consed</td>
<td>0.5</td>
<td>2.8</td>
<td>6.5</td>
</tr>
</tbody>
</table>

First we compare the denotational interpreter with efficient unification to the derived compiler and observe the performance boost of the transformation technique. The compiler is about 4.7 times faster than the interpreter. The interpreter allocates about 2.3 times as much memory.

We see that Norvig’s implementation is only about 1.6 times faster than ours. Our implementation allocates about 5.5 times as much memory. However, when $n$ is only 15, the benchmark does not stress the memory. We consider this next and raise $n$ to 20.

<table>
<thead>
<tr>
<th>fib(20)</th>
<th>Norvig</th>
<th>compiler</th>
<th>interpreter</th>
</tr>
</thead>
<tbody>
<tr>
<td>user time (ms)</td>
<td>17.2</td>
<td>52.0</td>
<td>238.6</td>
</tr>
<tr>
<td>GC time (ms)</td>
<td>0</td>
<td>160.0</td>
<td>145.3</td>
</tr>
<tr>
<td>MB consed</td>
<td>3.6</td>
<td>37.7</td>
<td>88.3</td>
</tr>
</tbody>
</table>

The differences are more pronounced here. Nevertheless, without including the garbage collection time, Norvig’s implementation is still only about three times faster than ours. In this case, our implementation allocates about 6.7 times as much memory.

We have also looked at other benchmarks, including the $n$-queens problem. We find that the performance results are very similar. The derived compiler is several times faster than the interpreter and it is in the ballpark of Norvig’s implementation. Garbage collection continues to have a significant negative impact on performance, but the variation in the factors has not been completely characterized.

One reason that Norvig’s implementation has higher performance is that it has an explicit optimization phase akin to copy-propagation. Further, his implementation effectively allocates frames on the Common Lisp run-time stack, whereas our implementation allocates frames in the heap. Even though our implementation currently lacks these enhancements, we consider its performance quite respectable.

5. CONCLUSION

This paper has presented a new transformation technique for deriving a compiler from an interpreter. Further, it has provided a case study that illustrates the effectiveness of the ideas and shows how to derive a serious Prolog compiler from a naïve interpreter.

In particular, this compiler goes beyond Consel and Khoo’s implementation and can compile fully recursive predicates. The case study shows that although the transformation technique was developed with denotational-style interpreters in mind, non-denotational interpreters can be transformed into denotational ones, so the technique can be made to apply to them as well. It also shows that the transformation technique can be applied to related algorithms such as the Prolog unification algorithm. The generated code is reminiscent of the instructions generated for the WAM[19, 13, 2] abstract machine implementation.

Currently, Common Lisp is viewed as the target language, but it could be viewed as an intermediate language. Additional transformations, such as closure-conversion[3] of the continuations, might be used to generate lower-level code.

Manual transformation is a double-edged sword. It is more flexible than automatic transformation, yet it allows for the introduction of human error. It may be worthwhile to create software tools to help perform some of the suggested transformations. We leave such tools as future work.

6. ACKNOWLEDGEMENTS

I would like to thank Axel Schreiner, Matthew Fluet, and Melissa Nunes-Harwitt for carefully reading previous drafts of this paper. I would also like to thank the anonymous referees for their valuable comments.

7. REFERENCES


