Automatically Generating Universal Attachments Through Compilation

Karen L. Myers
Department of Computer Science
Stanford University
Stanford, California 94305
myers@neon.stanford.edu

Abstract

Universal attachment is a general-purpose mechanism for integrating diverse representation structures and their associated inference programs into a framework built on logical representations and theorem proving. The integration is achieved by links, referred to as universal attachments, that connect logical expressions to these structures and programs. In this paper, we describe a compilation-based method for automatically generating new programs and new universal attachments to those programs given a base set of existing programs and universal attachments. The generation method provides the means to obtain large collections of attachments and attached programs without the traditional specification overhead. As well, the method simplifies the task of validating that a collection of attachments is *correct*.

1 Introduction

Universal attachment [Myers, 1990] is a mechanism for integrating diverse representation and reasoning techniques into a framework based on logic and theorem proving. The motivations for using universal attachment are increased inferential efficiency and expanded representational capabilities. Following in the tradition of previous attachment methods [Green, 1969] [Weyhrauch, 1980], universal attachment centers on the notion of 'attaching' procedures and data structures to logical expressions. When expressions having attachments are encountered during theorem proving, the attached procedures are executed on the attached data to directly evaluate the logical expressions rather than relying on further deduction. Universal attachment is a more expressive mechanism for merging procedures and data structures into a theorem prover than previously defined attachment techniques, and as such supports a much broader class of attachments.

This paper describes a method for automatically generating both new programs and new universal attachments to those programs. The general problem of automatically producing attachments and attached programs is extremely difficult. We present a method that relies on the existence of a base set of attachments and attached programs. The method exploits the idea that concepts are often hierarchically defined. In particular, if a given logical expression ϕ is defined in terms of subexpressions $\phi_1, \ldots, \phi_n$ all having attachments, then compilation techniques can be used to generate an attached program for ϕ that calls the attached programs for the ϕ_i's. Attachments to these newly generated programs are obtained by appropriately combining the attachments defined for the ϕ_i's.

An automated generation mechanism is important for several reasons. Such a mechanism can be used to create a large collection of attachments and attached programs, thus providing the computational advantages associated with such a collection but without the specification overhead. Another benefit relates to the *correctness* of attachments, where correctness means that the attached programs and the data upon which the programs operate embody the appropriate semantics for the expressions to which they are attached. Provided that the generation process is sound, the task of validating the correctness of a collection of automatically generated attachments reduces to that of validating the smaller initial set of attachments.

We begin in Section 2 by describing universal attachment. Section 3 outlines the generation process and conditions for its applicability, while Section 4 evaluates the approach taken. The automatic generation method has been implemented as part of a universal attachment system built on top of the KADS theorem prover [Stickel, 1988].

2 Universal Attachment

Before presenting the compilation-based generation method, we first summarize universal attachment and describe the condition of *factorability* required for the method to succeed.

2.1 A Definition of Universal Attachment

Definition 1 (Universal Attachment) *A* universal attachment *is a five-tuple represented as*

$$\langle \phi[x_1, \ldots, x_n], \mathsf{F} \rangle \rightarrow \langle \mathsf{p}, \mathsf{A}, \mathsf{D} \rangle \qquad (1)$$

where $\phi[x_1,\ldots,x_n]$ *is the attachment pattern,* F *is the filter set,* p *is the attached program,* A *is the attachment map and* D *is the detachment map.*

The *attachment pattern* $\phi[x_1,\ldots,x_n]$ is a logical expression (a relational, functional, quantified or boolean expression in some logical language $\mathcal{L}$) defined over the *distinguished variables* $x_1,\ldots,x_n$. Distinguished variables are schema variables ranging over the terms in the logical language being employed. As a matter of convenience, we will assume that the distinguished variables for an attachment pattern are simply the variables appearing in the pattern (*i.e.* the distinguished variables for the term *plus(x,y)* are x and y).

The *filter set* F is a collection of constraints or *filters* on bindings to the distinguished variables in ϕ, represented as $\mathsf{f}[x_1,\ldots,x_n]$. The language for expressing filters can be defined independently of the universal attachment framework, thus a different filter language can be utilized according to the nature of the applications being considered. We present a sample filter language and its corresponding definition of filter satisfaction later in this section.

The *domain* of the universal attachment, namely the set of logical expressions for which the universal attachment is defined, is determined jointly by the attachment pattern and the filter set. For $\mathsf{F} = \{\mathsf{f}_1[x_1, \ldots, x_n],\ldots,\mathsf{f}_m[x_1,\ldots,x_n]\}$, the domain of the attachment (1) is the set of logical expressions $\{\phi[t_1,\ldots,t_n]\}$ where $\mathsf{f}_1[t_1,\ldots,t_n],\ldots,\mathsf{f}_m[t_1,\ldots,t_n]$ are satisfied.

The *attached program* p is the name of the program to be used in the attachment process. The *attachment map* A takes the list of bindings to the distinguished variables as input and generates the appropriate set of parameters for p (the *attached data*). The *detachment map* translates the result of evaluating p on the attached data to the appropriate expression in the logical language (the *value* of the universal attachment). The attachment and detachment maps provide a very important service in that they translate between logical terms and structures used to represent those terms outside of the theorem prover. For this reason, we refer to them as the *translation maps.*

The interpretation of the attachment (1) is that a logical expression α in its domain will be *evaluated* by first executing p on the parameters obtained by applying the attachment map to the bindings of the distinguished variables, then translating the result according to the detachment map, and finally substituting that value back in for α.

2.2 The Filter Language $\mathcal{F}_{\mathcal{L}}$

Before considering an example of universal attachment, we first describe a sample filter language, $\mathcal{F}_{\mathcal{L}}$, defined for a logical language $\mathcal{L}$. More information on both alternative filter languages and the advantages of using filter languages can be found in [Myers, 1990].

The metalevel filter language $\mathcal{F}_{\mathcal{L}}$ is built on the assumption that filters can be specified for a distinguished variable independently of all other distinguished variables, *i.e.* filters can be written in the form $\mathsf{f}[x_i]$. The language consists of the predicates atomic(x), sort(x,s), and standard-name(x,s), with their satisfaction defined as follows:[1]

Definition 2 (Filter Satisfaction in $\mathcal{F}_{\mathcal{L}}$) *A term t satisfies a filter* f *in $\mathcal{F}_{\mathcal{L}}$ iff:*

1. f *is* atomic(t) *and t is a logical constant symbol in $\mathcal{L}$.*
2. f *is* sort(t,s) *and t has sort* s.
3. f *is* std-name(t,s) *and t is a standard name for sort* s.

The filter atomic requires t to be a constant symbol in $\mathcal{L}$. The filter sort dictates that t be of sort s; the corresponding sort information is included as part of $\mathcal{L}$. Standard name filters are used to make syntactic distinctions within sorts. The user can associate a set of standard names with a sort by making the appropriate specifications in $\mathcal{L}$. For example, we might specify that all numeric representations of integers within our logical language are standard names for the sort int of integers. The filter std-name(t,s) dictates that t must be a member of the set of standard names for sort s. To appreciate the need for standard names filtering, consider the logical constant *the-pope's-favourite-integer.* Although this constant is of sort int, we would not want to attach to expressions in which this constant is bound to a distinguished variable since we don't know the specific number that this constant denotes. We can eliminate such attachments by excluding this constant from the standard names for int.

2.3 A Graph Example

The following example illustrates how universal attachments can be applied.

Example 1 (Path Connectivity)
Consider a directed graph whose edges are described by the axiom

$$edge(a,b) \wedge edge(a,c) \wedge edge(b,c) \wedge edge(c,d) \wedge edge(d,b)$$

Here, the constants a, b, c and d denote particular nodes in the graph.

Now suppose we define a logical relation $path(x,y)$ in terms of the relation *edge* such that *path(x,y)* holds exactly when there is a sequence of edges connecting x to y. To answer queries about paths in the graph we can create a universal attachment. First, we define the LISP structure gr to be the following edge-based representation of the graph described by *edge*:

[1]Throughout this document, sans-serif font is used for expressions in $\mathcal{F}_{\mathcal{L}}$, *italicized* text indicates expressions in the logical language $\mathcal{L}$, and `typewriter-style` font indicates an attached program or data.

```
(defvar gr
  '((a (b c)) (b (c)) (c (d)) (d (b)))) .
```

We also need a LISP function connp that takes two nodes and an edge-based representation of a graph as parameters; the function call (connp n1 n2 g) returns t if n1 and n2 are connected by a path in g, and nil otherwise. Finally, we require a metalevel function h that maps logical representations of nodes onto the LISP representation of those nodes (*i.e.* $\mathsf{h}(a) = \mathtt{a}$, $\mathsf{h}(b) = \mathtt{b}$, *etc*). Using these components, we can create the universal attachment:

$$\langle path(x,y), \mathsf{F}^{path}\rangle \rightarrow \langle \mathsf{connp}, \mathsf{A}^{path}, \mathsf{D}^{tf}\rangle \qquad (2)$$

$$\mathsf{F}^{path} = \{\text{std-name}(x, \text{node}), \text{std-name}(y, \text{node})\}$$

$$\mathsf{A}^{path}((x,y)) = (\mathsf{h}(x), \mathsf{h}(y), \mathbf{gr}) \ .$$

The domain of this universal attachment is the set of instances of *path(x,y)* where x and y are bound to standard names of sort node. The attachment map A^{path} maps a pair of logical terms denoting nodes into the triple consisting of the LISP representation of those nodes and the LISP representation of our graph, **gr**. The detachment map D^{tf} translates the LISP atoms t to the logical truth value *true* and nil to *false*. The attachment (2) states that instances of *path(x,y)* where x and y are bound to standard names for sort node should be evaluated by applying connp to the arguments $\mathsf{h}(x)$, $\mathsf{h}(y)$ and **gr**.

2.4 Factorable Attachment Maps

The general definition of an attachment map A presented above, whereby A translates the list of bindings to distinguished variables into the list of parameters for the attached program, is very broad. In order to facilitate the automatic generation process, we impose the restriction of *factorability* on attachment maps.

Definition 3 (Factorability) *An attachment map is* factorable *iff it can be represented as a collection of unary and zeroary functions on bindings to distinguished variables.*

The attachment map A^{path} in Example 1 is factorable, being composed of the unary function h applied to bindings to x, h applied to bindings to y, and the constant function **gr** with value **gr**. We will refer to the individual unary and zeroary functions as the *component translations* of the attachment map. Factorability guarantees that attachment maps are definable in terms of bindings to individual distinguished variables, independent of all other bindings. As will be seen in Section 2, this independence is important for the compilation process. The factorability requirement is not burdensome; virtually all standard applications of universal attachments use factorable attachment maps.

For this document, we assume a standardized representation of attachment maps. Each map consists of a list of ordered pairs, where the first entry of the pair is a component translation function and the second entry is the distiniguished variable for which the component translation is defined. If the component translation is zeroary, than the second entry is simply $\emptyset$. Using this notation, we would represent the attachment map in Example 1 as $((\mathsf{h}, x)\ (\mathsf{h}, y)\ (\mathbf{gr}, \emptyset))$.

3 The Generation Method

In this section we describe both the *generation method* and the *compilation criteria* that a logical expression must satisfy for the method to apply. For simplicity, we assume that all attached programs are written in LISP.

The generation method derives from the observation that concepts are often hierarchically defined. Given a logical expression ϕ defined in terms of subexpressions having attachments, the method employs compilation techniques to produce an attached program for ϕ that calls the programs attached to ϕ's subexpressions.[2] A universal attachment from ϕ to the generated program is obtained by merging the subexpressions' attachments in a manner that depends on the logical structure of ϕ.

We begin by stating a concise definition of the compilation criteria. This definition will serve as a reference for the remainder of the section. Following the definition, we provide a thorough explanation of its parts (in particular the concepts of *weak satisfaction*, *nesting constraints*, *preserving truth values* and *enumerability* used in the criteria are defined) as well as a full description of the generation process.

Definition 4 (Compilation Criteria) *A logical expression ϕ satisfies the* compilation criteria *iff one of conditions* **C1**, **C2**, *or* **C3** *is met:*

C1 *There is an attachment* $\langle\alpha[x_1,\ldots,x_k], \mathsf{F}^\alpha\rangle \rightarrow \langle \mathsf{p}^\alpha, \mathsf{A}^\alpha, \mathsf{D}^\alpha\rangle$ *such that:*

- **C1.1** $\alpha[x_1,\ldots,x_k]$ *unifies with ϕ, producing bindings* $x_1 : t_1,\ \ldots,\ x_k : t_k$.
- **C1.2** *For each filter* $\mathsf{f}[x_i] \in \mathsf{F}^\alpha$, $\mathsf{f}[t_i]$ *is* weakly satisfied.
- **C1.3** *Each non-atomic t_i satisfies the compilation criteria.*
- **C1.4** *The detachment maps in the attachments defined by* **C1.3** *satisfy the* nesting constraints.

C2 ϕ *is a boolean expression, either* $\neg\beta_i$, $\bigwedge_{i=1}^{n}\beta_i[t_1^i,\ldots,t_{k_i}^i]$, *or* $\bigvee_{i=1}^{n}\alpha_i[t_1^i,\ldots,t_{k_i}^i]$, *where:*

- **C2.1** *Each β_i satisfies the compilation criteria.*

[2]This form of compilation is stronger than that associated with logic programming [Warren, 1977]. First of all, logic programming compilation is limited to Horn clauses. Furthermore, the compilation of a logic program produces an encoding in terms of pre-defined primitives (such as unifications and stack manipulations) of the search process followed by the interpreter, while the compilation we propose builds code from arbitrary attached programs.

C2.2 *The attachments defined by* **C2.1** *for each* β_i *have detachment maps that* preserve truth values.

C3 ϕ *is a quantified expression of the form* $\exists z.\ \alpha$ *or* $\forall z.\ \alpha$ *such that:*

C3.1 α *satisfies the compilation criteria.*

C3.2 z *is bound to a distinguished variable in the attachment pattern of the attachment defined by* **C3.1**.

C3.3 *The terms satisfying the filters specified for* z *in the attachment defined by* **C3.1** *are* enumerable.

C3.4 *The attachment defined by* **C3.1** *has a detachment map that* preserves truth values.

C1: Base Case

C1 constitutes the base case of the definition. For a logical expression ϕ to be compilable in the base case, condition **C1.1** requires the existence of an attachment

$$\langle \alpha[x_1,\ldots,x_k], \mathsf{F}^\alpha\rangle \rightarrow \langle \mathsf{p}^\alpha, \mathsf{A}^\alpha, \mathsf{D}^\alpha\rangle \qquad (3)$$

where ϕ unifies with $\alpha[x_1,\ldots,x_k]$. Condition **C1.2** further requires that if $\mathsf{f}[x_i]$ is in F^α and x_i is bound to t_i in the unification for **C1.1**, then $\mathsf{f}[t_i]$ must be *weakly satisfied.* In contrast to the notion of satisfaction from Definition 2, weak satisfaction only requires that the nonvariable components of a term satisfy a given filter. This relaxation of the satisfaction criteria accounts for variables in the expression being compiled. For the filter language $\mathcal{F}_\mathcal{L}$, a term t weakly satisfies $\mathsf{atomic}(t)$ iff t is a variable or logical constant name, and t weakly satisfies $\mathsf{std\text{-}name}(t,\mathsf{s})$ iff t unifies with a standard name for sort s. Weak satisfaction is identical to standard satisfaction for the filter $\mathsf{sort}(t,\mathsf{s})$.

The condition **C1.3** accounts for the binding of distinguished variables to non-atomic functional expressions in ϕ. In general, an expression ϕ that satisfies **C1.1** will have the form

$$\alpha[\beta_1[t_1^1,\ldots,t_{n_1}^1],\ldots\beta_k[t_1^k,\ldots,t_{n_k}^k]] \qquad (4)$$

where each $\beta_i[t_1^i,\ldots,t_{n_i}^i]$ is either a variable, a constant, or a non-atomic functional expression. A program that evaluates ϕ must necessarily include code to evaluate every non-atomic $\beta_k[t_1^k,\ldots,t_{n_k}^k]$. Thus the nesting of functional expressions in ϕ invalidates the compilability of ϕ unless an attached program exists to evaluate these expressions, or an appropriate program can be created through compilation. In either case, the nested expressions must satisfy the compilability criteria, prompting condition **C1.3**.[3]

If we let the set $\mathcal{I}$ index the non-atomic $\beta_i[t_1^i,\ldots,t_{n_i}^i]$, condition **C1.3** guarantees the existence of the attachments:[4]

$$\langle \beta_i[x_1,\ldots,x_{m_i}], \mathsf{F}^{\beta_i}\rangle \rightarrow \langle \mathsf{p}^{\beta_i}, \mathsf{A}^{\beta_i}, \mathsf{D}^{\beta_i}\rangle\,, \qquad i \in \mathcal{I}. \ (5)$$

C1.4 requires that these attachments satisfy certain *nesting constraints.* The nesting constraints ensure the soundness of consolidating programs attached to nested functional terms with p^α from (3) into a single attached program for the entire expression ϕ. We can express the nesting constraints as follows:

Definition 5 (Nesting Constraints) *For every* $i \in \mathcal{I}$*:*

NC1 *The range of* D^{β_i} *must satisfy the filters in* F^α *defined for* x_i.

NC2 *If* $(\mathsf{m}, x_i) \in \mathsf{A}^\alpha$ *and* $i \in \mathcal{I}$, *then* $\mathsf{m}(\mathsf{D}^{\beta_i}(\mathsf{c})) = \mathsf{c}$ *for every* c *in the range of the function* p_{β_i}.

Condition **NC1** states that if $\beta_i[t_1^i,\ldots,t_{n_i}^i]$ is a non-atomic expression bound to the distinguished variable x_i then the range of the detachment map D^{β_i} must satisfy the filters defined for x_i. This constraint ensures that substituting the values produced by the attachments in (5) for the corresponding nested expressions in $\alpha[\beta_1[t_1^1,\ldots,t_{n_1}^1],\ldots\beta_k[t_1^k,\ldots,t_{n_k}^k]]$ yields an expression in the domain of the attachment (3). Condition **NC2** requires that the composition of the component attachment map m defined for a distinguished variable x_i and the detachment map D^{β_i} for the nested expression $\beta_i[t_1^i,\ldots,t_{n_1}^i]$ is the identity map. This constraint ensures that the attachments in (5) and (3) have the same interpretation for attached symbols that they both utilize.

The program generated for ϕ is simply an application of p^α with nested function applications of the p^{β_i} for $i \in \mathcal{I}$ used to evaluate the non-atomic $\beta_i[t_1^i,\ldots,t_{n_i}^i]$. Defining the parameter list for the new program is somewhat complex. In particular, a parameter must be defined for each distinct component translation map applied to a distinguished variable binding. The factorability of the attachment maps makes it possible both to identify these translations and to build the required new attachment map that accounts for all such translations; details are in [Myers, 1990]. The detachment map is simply D^α from (3) while the new filter set is obtained by 'merging' the filter sets for the nested expressions and F^α. The merging process is an enhanced union operation that eliminates filters subsumed by other filters in the union. For example, $\mathsf{sort}(t,\mathsf{s1})$ subsumes $\mathsf{sort}(t,\mathsf{s2})$ if sort s2 contains sort s1.

Example 2 Let int and rat be functions that map logical representations of integers and rationals onto appropriate LISP representations of those numbers. Using

[3]Note that **C1.3** makes the base case of the definition itself recursive. The well-definedness of the criteria is ensured since the base case recursion parallels the logical structure of ϕ and hence must be finite.

[4]These attachments may be in the set of predefined attachments or may be generated recursively by the compilation process. The guaranteed attachments for subexpressions in the recursive cases **C2** and **C3** also have this characteristic.

int and rat we can specify attachments for the expressions *add(x,y)* and *div(x,y)*, which have the intended interpretations of integer addition and rational division. Let add and div be LISP programs of two inputs that compute integer addition and rational division, and int and rat be the sorts of integer and rational numbers. By defining $\mathsf{A}^{\mathsf{int}} = ((\mathsf{int}, x), (\mathsf{int}, y))$ and $\mathsf{D}^{\mathsf{int}} = \mathsf{int}^{-1}$, along with $\mathsf{A}^{\mathsf{rat}} = ((\mathsf{rat}, x), (\mathsf{rat}, y))$ and $\mathsf{D}^{\mathsf{rat}} = \mathsf{rat}^{-1}$, we can create the attachments:

$$\langle add(x,y), \{\text{std-name}(x, \mathsf{int}), \text{std-name}(y, \mathsf{int})\}\rangle \longrightarrow \langle \mathtt{add}, \mathsf{A}^{\mathsf{int}}, \mathsf{D}^{\mathsf{int}}\rangle$$

$$\langle div(x,y), \{\text{std-name}(x, \mathsf{rat}), \text{std-name}(y, \mathsf{rat})\}\rangle \longrightarrow \langle \mathtt{div}, \mathsf{A}^{\mathsf{rat}}, \mathsf{D}^{\mathsf{rat}}\rangle \ .$$

Now consider the logical definition

$$\forall xy.\ avg(x,y) = div(add(x,y), 2) \ .$$

The expression *div(add(x,y))* satisfies the base case of the compilation criteria. Using the generation method, we obtain the following program and attachment:

```
(defun avg-lisp (n1 n2)
  (div (add n1 n2) 2)))
```

$$\langle avg(x,y), \{\text{std-name}(x, \mathsf{int}), \text{std-name}(y, \mathsf{int})\}\rangle \longrightarrow \langle \mathtt{avg\text{-}lisp}, \mathsf{A}^{\mathsf{int}}, \mathsf{D}^{\mathsf{rat}}\rangle \ .$$

C2: Boolean Expressions

If ϕ is a boolean expression that doesn't satisfy **C1**, ϕ is still compilable provided that condition **C2** is satisfied. All boolean expressions share the same compilation criteria and generation method; here we restrict attention to conjunctive expressions, where ϕ has the form $\wedge_{i=1}^{n} \alpha_i[t_1^i, \ldots, t_{k_i}^i]$.

Compilation condition **C2.1** guarantees the existence of attachments

$$\langle \alpha_i[x_1, \ldots, x_{k_i}], \mathsf{F}^{\alpha_i}\rangle \longrightarrow \langle \mathsf{p}^{\alpha_i}, \mathsf{A}^{\alpha_i}, \mathsf{D}^{\alpha_i}\rangle, \quad i = 1, \ldots, n.$$

These attachments provide the programs used to construct the new attached program for ϕ.

To motivate **C2.2**, we consider the nature of the program to be generated for ϕ. The form of this program will mirror the logical structure of ϕ: the program consists of an application of the LISP function and to the results obtained by evaluating the attached programs p^{α_i}. This strategy is sound only if the values produced by the p^{α_i} 'match' the notion of truth values utilized in LISP. In particular, when an attached program p^{α_i} returns t (or nil), then the truth (or falsity) of the corresponding conjunct should be established. The need for this semantic correspondence motivates the following definition.

Definition 6 (Truth Value Preservation) *A function* g *from a representation language* $\mathcal{L}_1$ *to a representation language* $\mathcal{L}_2$ preserves truth values *iff* g *maps truth and falsity in* $\mathcal{L}_1$ *onto truth and falsity, respectively, in* $\mathcal{L}_2$.

Example 3 The detachment map D^{tf} defined in Example 1 preserves truth values from LISP to our logical language: it maps t to *true* and nil to *false*.

Although there may be many functions that preserve truth values from one language to another, we will designate D^{tf} as the *canonical* truth preserving function from LISP to our logical language. Verifying that a detachment map D preserves truth values thus reduces to checking that $\mathsf{D} = \mathsf{D}^{tf}$.

As noted above, the new attached program for $\wedge_{i=1}^{n} \alpha_i[t_1^i, \ldots, t_{k_i}^i]$ consists of an application of the LISP function and to the values obtained by evaluating the p^{α_i}. In analogy with programs generated for the base case **C1**, a parameter is required for each unique translation applied to a distinguished variable binding and the new attachment map is the collection of these translations. The new filter set is the union of the filter sets for the individual conjuncts where subsumed filters are once again removed. The detachment map is the canonical truth value preserving function, D^{tf}.

C3: Quantified Expressions

Condition **C3** constitutes the compilation criteria for a quantified expression $\forall z.\ \alpha$ or $\exists z.\ \alpha$ that fails to satisfy **C1**.

C3.1 requires the existence of an attachment

$$\langle \alpha[x_1, \ldots, x_k], \mathsf{F}^{\alpha}\rangle \longrightarrow \langle \mathsf{p}^{\alpha}, \mathsf{A}^{\alpha}, \mathsf{D}^{\alpha}\rangle \qquad (6)$$

for the matrix of the quantified expression. The underlying idea for compiling quantified expressions is to apply the program p^{α} from the attachment (6) to the attached LISP representations of all possible bindings to the quantified variable z.

To make this approach feasible, the quantified variable must be bound to a distinguished variable x_i in $\alpha[x_1, \ldots, x_k]$ (**C3.2**), and the set of terms satisfying the filters in F^{α} defined for x_i must be *enumerable* (**C3.3**). With standard names filters, enumerability is obtained by using the list of standard names defined for the sort of the quantified variable. More complex mechanisms for achieving enumerability are described in [Myers, 1990]; these mechanisms use attached structures directly to determine the domain of quantification. For example, with quantified expressions defined relative to a graph, the domain of iteration can be limited to the set of standard names for nodes in that graph rather than the set of standard names for all nodes appearing in any graph.

The new attached program consists of an iteration whose domain is the set of LISP objects obtained by applying the component translation map m defined for x_i (*i.e.* $(\mathsf{m}, x_i) \in \mathsf{A}^{\alpha}$) to the enumeration of terms satisfying the filters for x_i in F^{α}. At each step, p^{α} is called. For existential quantifiers, the program returns t only if p^{α} evaluates to t for some value in the domain. For universal quantifiers, the program returns nil if a value is found for which p^{α} evaluates to `nil` and returns `t` otherwise. As with the compilation of boolean expressions, we require that the detachment map D^{α} preserve truth values (**C3.4**) to ensure that the semantics of the iteration process match the semantics of the logical quantifier.

The parameters for the new program are identical to those of p^{α} except that the parameter corresponding to the variable of quantification is replaced by a parameter representing the domain of the iteration. Correspondingly, the new attachment map is obtained from A^{α} by replacing the pair (m, x_i) by a function that returns the domain of the iteration. The new filter set is obtained from F^{α} by removing filters defined on x_i. The new detachment map is D^{tf}.

Example 4 Consider the logical expression

$$\forall z.\ incycle(z) \equiv \exists y.\ path(z,y) \land path(y,z)\ .$$

Given the attachment (2) defined in Example 1, the expression $\exists y.\ path(z,y) \land path(y,z)$ satisfies the compilation conditions in **C3**. The compilation process produces the program `incycle-lisp` defined by

```
(defun incycle-lisp (u v domain)
  (some #'(lambda (item)
            (and (connp u item v)
                 (connp item u v)))
        domain)).
```

The formal parameters `u` and `v` to `incycle-lisp` correspond to a node and a graph, respectively. The parameter `domain` is the list of values over which the iteration is defined. During each step of the iteration, the current iterated value `item` is tested to see if the code fragment `(and (connp u item v) (connp item u v)))` evaluates to `t`. This code fragment is produced by recursively compiling the logical expression $path(z,y) \land path(y,z)$.

The following attachment for *incycle(z)* is also created, according to the principles described above:

$$\langle incycle(z), \{\text{std-name}(x, \text{node})\}\rangle \\ \longrightarrow \langle \texttt{incycle-lisp}, A^{incycle}, D^{tf}\rangle$$

$$A^{incycle}((z)) = (\text{h}(z), \texttt{gr}, \texttt{'(a b c d)})\ .$$

4 Closing Remarks

This paper has presented a compilation-based method for generating new programs and universal attachments to those programs from a base set of existing programs and universal attachments. We have tested an implementation of the method in the domain of graph theory. The class of new attached programs and attachments produced by the implementation method has proven to be quite rich. In addition, the generated attachments appear to provide significant gains in the run-time efficiency of the theorem prover.

For non-quantified expressions, the compilation process consolidates all relevant attachments into a single program. This consolidation eliminates repeated transitions between the theorem prover and the attached computational mechanism, thus reducing the total overhead for translations. The programs constructed for quantified expressions can provide efficiency gains for a different reason: by limiting the domain of iteration for these programs to the relevant terms, the attached programs can be much faster than theorem proving.

The only previous attempt at automating the generation of attachments was Aiello's work on producing new semantic attachments through compilation [Aiello, 1980a] [Aiello, 1980b]. Because universal attachments subsume semantic attachments, Aiello's work is necessarily less ambitious than that reported here. In particular, the compilation of quantified expressions is not addressed. Her work also fails to delimit sufficient restrictions on the class of expressions that are compilable, leading to situations where her method generates programs that are incorrect for the expressions they are designed to evaluate.

Acknowledgements

I would like to thank Jens Christensen, Nils Nilsson, Eunok Paek and Carolyn Talcott for many useful discussions related to this work. The author is supported by an IBM Graduate Fellowship.

References

[Aiello, 1980a] Luigia Aiello. Automatic generation of semantic attachments in FOL. In *Proceedings of the First National Conference on Artificial Intelligence*, 1980.

[Aiello, 1980b] Luigia Aiello. Evaluating functions defined in first order logic. In *Proceedings of the Logic Programming Workshop, Debrecen, Hungary*, 1980.

[Green, 1969] Cordell Green. Application of theorem proving to problem solving. In *Proceedings of the First International Joint Conference on Artificial Intelligence*, pages 219–239, 1969.

[Myers, 1990] Karen L. Myers. Universal attachment. Forthcoming Ph.D. dissertation, Stanford University.

[Stickel, 1988] Mark E. Stickel. The KLAUS automated deduction system. In *Proceedings of the Ninth International Conference on Automated Deduction*, pages 750–751, 1988.

[Warren, 1977] David Warren. Implementing PROLOG — Compiling predicate logic programs. Technical Report 39, University of Edinburgh, 1977.

[Weyhrauch, 1980] Richard W. Weyhrauch. Prolegomena to a theory of mechanized formal reasoning. *Artificial Intelligence*, 13:133–170, 1980.