CLIPS TEMPLATE SYSTEM FOR PROGRAM UNDERSTANDING

Ronald B. Finkbine, PhD.
Department of Computer Science
Southeastern Oklahoma State University
Durant, OK 74701
finkbine@babbage.sosu.edu

ABSTRACT

Program Understanding is a subfield of software re-engineering and attempts to recognize the run-time behavior of source code. To this point, the success in this area has been limited to very small code segments. An expert system, HLAR (High-Level Algorithm Recognizer), has been written in CLIPS and recognizes three sorting algorithms, Selection Sort, Quicksort and Heapsort. This paper describes the HLAR system in general and, in depth, the CLIPS templates used for program representation and understanding.

INTRODUCTION

Software re-engineering is a field of Computer Science that has developed inconsistently since the beginning of computer programming. Certain aspects of what is now software re-engineering have been known by many names: maintenance, conversion, rehosting, reprogramming, code beautifying, restructuring, and rewriting. Regardless of the name, these efforts have been concerned with porting system functionality and/or increasing system conformity with programming standards in an efficient and timely manner. The practice of software re-engineering has been constrained by the supposition that algorithms written in outdated languages cannot be re-engineered into robust applications with or without automation.

It is believed by this author that existing software can be analyzed, data structures and algorithms recognized, and programs optimized at the source code level with expert systems technology using some form of intermediate representation. This intermediate form will provide a foundation for common tool development allowing intelligent recognition and manipulation. This paper describes a portion of the HLAR (High-Level Algorithm Recognition) system [1].

The Levels of *understanding* in the software re-engineering and Computer Science fields is displayed

in Figure 1 [2]. The lowest of these, *text*, is realized in the simple file operations; opening, reading, writing and closing. The next level is *token* understanding and occurs in compilers within their scanner subsystem. Understanding at the next level, *statement*, and higher generally does not occur in most compilers, which tend to break statements into portions and correctly translate each portion, and, therefore, the entire statement. One exception is the *semantic* level which some compilers perform when searching for syntactically correct but logically incorrect segments such as *while (3 < 4) do S*.

```
[7] Program
[6] Plan
[5] Semantic
[4] Compound Statement
[3] Statement
[2] Token
[1] Text
```

Figure 1: Levels of Understanding

Software re-engineering requires that higher-level understanding take place and the act of understanding should be performed on an intermediate form. The representation method used in this research project is the language ALICE, a very small language with a Lisp-like syntax. It is intended that programs in existing high-level languages be translated into ALICE for recognition and manipulation. This language has only five executable statements; *assign*, *if*, *loop*, *call* and *goto*. All syntactic sugar is removed, replaced with parentheses. Figure 2 displays a simple assignment statement. The *goto* statement is used for translating unstructured programs and the goal is to transform all unstructured programs into structured ones.

```
(assign x 0)
```

Figure 2: Assignment Statement

FACT REPRESENTATION

Prior to initiation of the HLAR system, programs in the ALICE intermediate form are translated into a set of CLIPS facts which activate (be input to and fulfill the conditions of) the low-level rules. Facts come in different types called templates (records) which are equivalent to frames and are ``asserted'' and placed into the CLIPS facts list. Slots (fields) hold values of specified types (integer, float, string, symbol) and have default values. As a rule fires and a fact, or group of facts, is recognized, a new fact containing the knowledge found will have its slots filled and asserted. Continuation of this process will recognize a larger group of facts and, hopefully, one of the common algorithms.

```
(general_node (number 1)
  (sibling 0)
  (address "2.10.5")
  (node_type assign_node))
(assign_node (number 1)
 (lhs_node 1)(rhs_node 1)
(general_node_parent 1))
(identifier_node (number 1)
  (operand 2)(name "x"))
(expression_node (number 1)
  (operand_1 2))
(integer_literal_node
  (number 1)
  (operand 2)(value 1))
```

Figure 3: Facts List

The *assign* statement from the previous Figure is translated into the equivalent list of facts in Figure 2. This is a much more complicated representation, and not a good one for programmers, but much more suitable for an

expert system.

The code of this Figure displays a number of different templates. The *general_node* is used to keep all statements in the proper order within a program and the *node_type* slot describes the type of statement associated with the node. The template *assign_node* with its appropriate *number* slot , its *lhs_slot* slot points to the number one *operand_node*, which is also pointed to by the operand slot in the number one *identifier_node*. The *rhs_node* slot of the assignment points to the number one *expression_node* and its *operand_1* slot points to the number two *operand_node* which is also pointed to by the operand slot in the number one *integer_literal_node*.

This is a more complex representation of a program than its ALICE form, but it is in a form that eases construct recognition. An ALICE program expressed in a series of CLIPS facts is a list and requires no recursion for parsing. Each type of node (i.e. *general, assign, identifier, integer_literal, operand, expression*) are numbered from one and referenced by that number. Once the ALICE program is converted into a facts list, low-level processing can begin.

TEMPLATE REPRESENTATION

Templates in CLIPS are similar to records or frames in other languages. In the HLAR system, templates are used in a number of different areas including general token, *plan* and knowledge representation. A properly formed template has the general form of the keyword *deftemplate* followed by the name of the template, an optional comment enclosed in double quotes and a number of *field* locations identifying attributes of the template. Each attribute must be of the simple types in the CLIPS system: *integer, string, real, boolean* or *symbol* (equivalent to a Pascal enumerated type) and default values of all attributes can be specified.

Currently there are three types of template recognition; *general* templates are used to represent the statement sequence that constitute the original program., *object* templates are used to represent the clauses, statements and algorithms that are recognized, and *control* templates are used to contain the recognition process.

GENERAL TEMPLATES

The names of all the general templates are listed in Figure 4. These are the templates that are required to represent a program in a list of facts having been translated from its original language which is similar to a compiler derivation tree.

```
general_node
argument_node
assign_node
call_node
define_routine_node
define_variable_node
define_structure_node
evaluation_node
expression_node
if_node
loop_node
parameter_node
program_node
identifier_node
integer_literal_node
struc_ref_node
struc_len_node
```

Figure 4: General Templates

The *general_node* referenced in this Figure is used to organize the order of the statements in the program. Prior to the HLAR system being initialized, a utility

program parses the ALICE program depth-first and generates the facts list needed for analysis. The *general_node* template is utilized to represent the order of the statements and contains no pointers to statement nodes. The statement-type nodes identified in the next section contain all pointers necessary to maintain program structure. Each token-type has its own templates for representation within an ALICE program. Included are node types for routine definitions, the various types of compound and simple statements, and the attributes of each of these statements. The various types of simple statement nodes contain pointers back to the *general_node* to keep track of statement order. These statement node templates contain the attributes necessary to syntactically and semantically reproduce the statements as specified in the original language program, prior to translation into ALICE.

In an effort to reduce recognition complexity, which is the intent of this research, specialty templates for each item of interest within a program have been created. An earlier version of this system had different templates for each form, instead of one template with a symbol-type field for specification. This refinement has reduced the number of templates required, thus reducing the amount of HLAR code and the programmer conceptual-difficulty level. The specialty templates are listed in Figure 5.

```
spec_call
spec_parameter
spec_exp
spec_assign
loop_algorithm
spec_eval
minimum_algorithm
swap_algorithm
if_algorithm
sort_algorithm
```

Figure 5: Special Templates

OBJECT TEMPLATES

Expressions are the lowest-common denominator of *program understanding*. They can occur in nearly all statements within an ALICE program. To reduce the complexity of *program understanding*, each expression for which we search is designated as a special expression with a specified symbol-type identifier. Expressions and structure references present a particular problem since they are mutually recursive as expressions can contain structure references and structure references can contain expressions. This problem came to light as the HLAR system became too large to run on the chosen architecture (out of memory) with the number of templates, rules and facts present at one time. Separation of the rules into several passes required that expressions and structure references be detected within the same rule group. Examples *a[i]* and *a[i+1] > a[i]* represent these concepts.

Figure 6 contains the specialty template for expressions. This template contains a number of fields for specific values, but most of interest is the field *type_exp*. Identifiers that represent specific expressions are listed as the allowed symbols. Complexity is reduced in this representation since multiple versions of the same statement can be represented by the same symbol. An example is two versions of a variable increment statement; $x = x + 1$ and $x = 1 + x$.

```
(deftemplate spec_exp
 (field type_exp (type SYMBOL)
  (default exp_none)
   (allowed-symbols exp_none
      exp_0 exp_1 exp_id exp_first
      exp_div_id_2 exp_plus_id_1
      exp_minus_id_1 exp_plus_id_2
      exp_minus_id_2 exp_mult_id_2
      exp_plus_div_id_2_1
      exp_minus_div_id_2_1
      exp_div_plus_id_id_2
      exp_ge_id_id exp_gt_const_id
      exp_gt_id_id exp_gt_id_const
      exp_gt_id_minus_id_1
      exp_lt_id_minus_id_1
      exp_gt_strucid_strucid
      exp_lt_strucid_strucid
      exp_lt_strucid_struc_plus_id_1
      exp_gt_strucid_id
      exp_ne_id_true
      exp_or_gt_id_min_id_1_ne_id_t
      exp_or_gt_id_id_ne_id_t
      exp_strucfirst))
(field id_1 (type INTEGER)
 (default 0))
(field id_2 (type INTEGER)
 (default 0))
(field id_3 (type INTEGER)
 (default 0))
(field exp_nr (type INTEGER)
 (default 0)))
```

Figure 6: Expression Template

Recognition of various forms of the *assign* statement occurs within the variable rules of HLAR. Common statements such as increments and decrements are recognized, as well as very specific statements such as $x = y / 2 + 1$.

Variable analysis is performed from one or more *assign* statements. The intent is to classify variables according to their usage, thus determining knowledge about the program derived from programmer intent and not directly indicated within the encoding of the program. An example would be saving the contents of one specific location of an array into a second variable such as *small = a[0]*.

Next come the multiple or compound statements such as the *if* or *loop* statements. These statements are the first of the two-phase rules to fire a *potential* rule and an *actual* rule. The *potential* rule checks for algorithm form as is defined against by the standard algorithm. The *actual* rule verifies that the proper statement containment and ordering is present. This allows for smaller rules, detection of somewhat *buggy* source code and elimination of *false positive* algorithm recognition.

The *loop* and *if* statements are the first compound statements that are handled within HLAR. Both have an *eval* clause tested for exiting the *loop* or for choosing the boolean path of the *if*. Both of these statements require a field to maintain the limits of the control of the statement. Generally, a *loop* or *if* statement will contain statements of importance within them and the concept of statement

containment will be required to be properly accounted for.

Higher-level algorithms, such as a *swap*, *minimization*, or *sort*, require that *subplans* be recognized first. This restriction is due to the size of the Clips rules. The more conditional elements in a rule, the more difficult and unwieldy for the programmer to develop and maintain.

Control templates are listed in Figure 7. These are used to control the recognition cycle within Clips. Control templates are necessary after preliminary recognition of a *plan* by a *potential* rule to allow for detection of any intervening and interfering statements prior to the firing of the corresponding *actual* rule.

```
not_fire
assert_nots_scalar
asserts_not_struc
```

Figure 7: Control Templates

Figure 8 is a hierarchical display of the facts from the previous Figure. It expresses the relationships among the nodes and shows utilization of the integer pointers used in this representation.

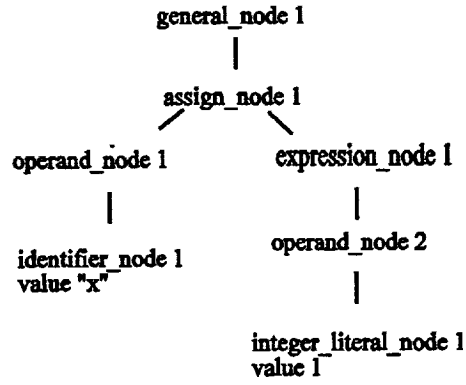

Figure 8: Derivation Tree

STATEMENT RECOGNITION

This section will describe the recognition process of a simple variable increment as displayed in Figure 9 as an example of the recognition process. The first rule firing will recognize a *special expression*, an identifier plus the integer constant one. The second rule firing will recognize a variable being assigned a *special expression* and a third rule will recognize that the identifier on the right hand side of the statement and the variable on the left hand side are the same, thus signifying an incrementing assignment statement.

```
(assign x (plus x 1))
```

Figure 9: Variable Increment

These three rule firings process one statement and further rule conditional elements will attempt to group this statement with related statements to recognize multi-statement constructs. An example would have this *assign* statement within a *loop* statement, and both preceded by an $x = 0$

initialization statement. This would indicate that the variable *x* is an index of the *loop* statement.

SUMMARY AND FUTURE RESEARCH

This research has lead to the development of a template hierarchy for *program understanding* and a general procedure for developing rules to recognize program *plans*. This method has been performed by the researcher, but will be automated in future versions of this system.

There are currently three algorithm *plans* recognized including the selection sort (SSA), quicksort (QSA) and heapsort (HSA). The SSA requires 50 rule firings, the QSA requires 90 firings, and the HSA, the longest of the algorithms at approximately 50 lines of code and taking over 60 seconds on a Intel 486-class machine requires 150 firings. The complete HLAR recognition system contains 31 templates, 4 functions, and 135 rules.

The research team intends to concentrate on algorithms common to the numerical community. The first version of the HLAR system has been successful at recognizing small algorithms (less than 50 lines of code). Expansion of HLAR into a robust tool requires: rehosting the system into a networking environment for distributing the recognition task among multiple CPUs, automating the generation of recognition rules for improved utility, attaching a database for consistent information and system-wide (multiple program) information, and a graphical user-interface.

AUTHOR INFORMATION

The author has: a B.S. in Computer Science, Wright State University, 1985; an M. S. in Computer Science, Wright State University, 1990; and a Ph.D. in Computer Science, New Mexico Institute of Mining and Technology, 1994. He is currently an Assistant Professor of Computer Science at Southeastern Oklahoma State University.

REFERENCES

1. 'High-Level Algorithm Recognition,' R. B. Finkbine, Ph.D. Dissertation, New Mexico Institute of Mining and Technology, 1994.

2. 'Pat: A Knowledge-based Program Analysis Tool,' M. T. Harandi and J. Q. Ning. In 1988 IEEE Conference of Software Maintenance, IEEE CSP, 1988.

AN IMPLEMENTATION OF FUZZY CLIPS AND ITS APPLICATIONS UNCERTAINTY REASONING IN MICROPROCESSOR SYSTEMS USING FUZZY CLIPS

Yuen & Lam

Abstract unavailable at time of publication.