Novel approach to a rule-based general purpose program translator using paramodulation

Yoshiyasu Takefuji and Michael Dowell

In this paper a rule-based Lisp dialect translator using paramodulation is presented as an example of a general purpose program translator application where the knowledge about the translation is embedded in rules. The advantage of using a rule-based system is to allow the user to supply his own rules for translation, thus the translator can be considered as a general purpose converter. Also, the rule-based LDT has the ability to test individual rules for correctness to aid in rule development. The translation being used for development is Franz to Common Lisp.

Keywords: Lisp, rule-based program translator, paramodulation, converter

Since the advent of Lisp languages we have been suffering from a great number of dialects. Rapid progress in artificial intelligence research has directed researchers to focus on the dialect problem in Lisp languages. Recently, the Department of Defense decided to use only Common Lisp as the official Lisp language. However, a great number of useful Lisp programs are not coded in Common Lisp: eventually those useful programs will have to be converted. In order to simplify the conversion of programs, rule-based programming may be one of the more promising solutions. Rule-based systems allow users to create new conversion rules, and in this sense, the rule-based translator will become a general purpose dialect converter.

In this paper a rule-based Lisp Dialect Translator (LDT)[1] is presented where paramodulation[2] is a key feature of the LDT. The LDT takes advantage of the fact that the structure, and much of the semantics of different dialects of Lisp, are the same, and therefore do not need to be changed. The LDT makes one-to-one, one-to-many and many-to-many translations. In addition, for the Franz to Common Lisp translation, a special one-to-many translation is needed to allow for the use of superparentheses. The rule form, the different types of translation and the relationship rules between Franz[3,4] and Common Lisp[5-7] are discussed below.

Center for Machine Intelligence, Department of Electrical and Computer Engineering, University of South Carolina, Columbia, SC 29208, USA

RULE FORM

Paramodulation is an inference rule based on the substitution properties of the equality relation. Inference rules are processes for producing new clauses from existing clauses[2]. The rules are in a paramodulation form, e.g.:

```
P(a)
EQUAL (a,b)
------------
P(b)
```

In this example, the result P(b) is called the paramodulant. The clause P(b) is said to be obtained by paramodulating into the clause P(a) from the equality EQUAL (a,b). The expressions 'into' and 'from' can also refer to the terms being matched. In the above example, one would say that paramodulation occurred from the term a into the term b[2].

As another example, suppose we have the following facts: John's son is handsome and Bob is John's son. From the given facts we can conclude that Bob is handsome:

```
is_handsome(son(John))
EQUAL(son(John),Bob)
------------
is_handsome(Bob)
```

In the above example, one would say that paramodulation occurred from the term son(John) into the term Bob.

The LDT restricts the form of the equality literal such that the 'from' term is always to be the left-hand side

0950–7051/88/020090–04 $03.00 © 1988 Butterworth & Co (Publishers) Ltd

of the equality literal, and the 'into' term is always to be the right-hand side of the equality literal. When a substitution is made for a variable it must be made for all occurrences of the variable in the clause.

The rules represent the relationship between the dialects, and will be of the form:

EQUAL (old_expression : new_expression)

with the colon acting as a deliminator. In our case the 'old_expression' will be Franz Lisp and the 'new_expression' will be Common Lisp.

Within the expressions, variables will allow for one-to-many and many-to-many translations. The variables in different equality literals are completely independent, even if they have the same name. The first part of the rule 'old_expression' will allow for single- or multiple-atom instantiation. Single-atom instantiation will be specified by using the '?' symbol. For example:

match '((? *who*) goes to school) '(Nancy goes to school)

will cause variable *who* to be instantiated to 'Nancy'.

Multiple-atom instantiation will be specified by using the '+' symbol. For example:

match '(Nancy goes (+*where*)) '(Nancy goes to school)

will cause variable *where* to be instantiated to 'to school'.

The second part of the rule 'new_expression' will have the variables replaced by the instantiated value, regardless of either single- or multiple-atom instantiation. For example:

(Nancy goes to the store)
EQUAL ((? *who*) goes (+ *where*) : who will go where after lunch)
- - - - - - - - - - - -
(Nancy will go to the store after lunch)

In the above example, the variable *who* is instantiated to 'Nancy', and the variable *where* to 'to the store'. Looking at the second part alone, it is not clear which are variables, but *who* and *where* will be replaced due to the first part.

TRANSFORMATIONS

One-to-one transformations

The rule form for one-to-one transformations is the simplest to derive with the 'new_expression' directly replacing the 'old_expression'. This can be thought of as direct substitution, e.g.:

The relationship:	EQUAL (add : +)
The Franz function:	(defun example_1 () (add 2 3))
The Common function:	(defun example_1 () (+ 2 3))

The paramodulation occurred from the 'add' function into the '+' function.

One-to-many transformations

The rule form for one-to-many transformations involves matching, which is done by using variables within the expressions. The first part of the rule 'old_expression' will allow for single- or multiple-atom instantiation. The second part of the rule 'new_expression' will have the variables replaced by the instantiated value, regardless of either single- or multiple-instantiation, e.g.:

The relationship:	EQUAL (nequal (? *x*) (? *y*) : not (equal *x y*))
The Franz function:	(defun example_2 () (cond ((nequal 'a 'b) (princ 'a))))
The Common function:	(defun example_2 () (cond ((not (equal 'a 'b)) (princ 'a))))

The paramodulation occurred from the 'nequal' function into 'not (equal', the *x* variable was instantiated to 'a, the *y* variable was instantiated to 'b. Notice the variables in the right-hand side were not in the same form as the left-hand side. This is what allows for substitution regardless of single- or multiple-atom instantiation.

Many-to-many transformations

This type of translation can be seen as a combination of one-to-one and one-to-many translations, with operations being performed on the variables and a direct substitution between the two instructions, e.g.:

The relationship:	EQUAL (append1 (? *x*) (? *y*) : append *x* (list *y*))
The Franz function:	(defun example_3 () (append1 '(a b c) 'a))
The Common function:	(defun example_3 () (append '(a b c) (list 'a)))

The paramodulation occurred from the 'append1' function into the 'append', the *x* variable was instantiated to '(a b c), the *y* variable was instantiated to 'a. Notice that the form of the *y* variable was changed on the left-hand side to make the replaced value a list of *y* instead of an atom *y*, resulting in 'a being replaced by '(list a).

Special one-to-many transformation

To translate superparentheses from Franz to Common Lisp a special type of translation is needed. In Franz Lisp a right superparenthesis is represented by ']', and can close off as many open left parentheses as needed until the end of the function is reached or until an open left superparenthesis is encountered. A left superparenthesis is represented by '[', and closes one right superparenthesis. Superparentheses are not allowed in Common Lisp, so a transformation is needed to change left superparenthesis to a single left parenthesis, and a right superparenthesis to the correct number of right parentheses. The algorithm used is as follows:

character	- - - - - - - - -
(	add 1 to the current count
[	start a new count (= 1), save the old count and replace with (
)	subtract 1 from the current count
]	replace with the correct number of (determined by the current count, resume the old count (unless this is level 0 then start over)

The Franz function:

```
     level 0: 1      21   2                               0
     level 1:                12          321   2          0
  (defun example_4 () (cond [(null ()) (print 'hello'] ]
```

The Common function:

```
     level 0                                            | |
     level 1                 |                          | |
  (defun example_4 () (cond ((null ()) (print 'hello')) ))
```

The left superparenthesis was replaced by a single left parenthesis and a new count was started. When the first right superparenthesis was encountered, it was replaced by two right parentheses and the old count was resumed. Then the second right superparenthesis was replaced by two right parentheses, which returned the original count to zero, and thus signified that the function was finished.

LDT GENERAL STRUCTURE

At present the source code of the LDT is written in Franz Lisp, and when completed will be translated into Common Lisp using itself. The final structure of the LDT has not been established, but a successful prototype is in operation. The architecture of the LDT is shown in Figure 1.

The LDT makes two passes, with the intermediate results being stored in a file. The first pass replaces the superparentheses by processing every character. The second pass replaces all one-to-one, one-to-many and many-to-many translations by processing functions, e.g.:

FIRST PASS	SECOND PASS
special one-to-many	one-to-one
replace ['s &]'s	one-to-many
	many-to-many

EXTENSIONS

One immediate extension will be to upgrade the matching function to allow matching of variables to characters within a word. This will allow for translation of the combination of cars and cdrs which exceed a length of four characters in Franz Lisp to the correct sequence of cars and cdrs of length less than four characters in Common Lisp. In Common Lisp no more than four combinations of car and cdr may be combined into one function name, e.g.:

```
(caddaddr list)
EQUAL (c*r (? x) : c*4r (c*4r c))
------------
(caddar (cddr list))
```

The paramodulation occurred from the 'caddaddr' function into a sequence of c____r's where only four were allowed in each function.

AIDING RULE DEVELOPMENT

The rule-based LDT allows for testing of one rule at a time, which will help to derive correct rules. This can be done by writing a test program which contains only the function which is to be translated. Using this program and the single rule under development as the input for the translator, the result of the translation can be studied to determine how the rule must be changed. Once the correct form has been reached, the program can be run in the new dialect to determine if the semantics are correct.

RELATIONSHIP BETWEEN FRANZ AND COMMON

Franz and Common Lisp overlap, with some functions being present in both dialects (see Figure 2). Therefore, not all Franz Lisp functions have to be translated: only those functions which are different or are not present in Common Lisp (see Figure 3).

In Common Lisp[7], the function '+' can add any type of numbers, while in Franz Lisp the function '+' will only add integers within a certain range. The equivalent Franz Lisp function for the Common Lisp function '+' is represented by one of the following one-to-one equality literals:

```
EQUAL ( add : + )
EQUAL ( plus : + )
EQUAL ( sum : + )
```

In Common Lisp the member function uses EQL test,

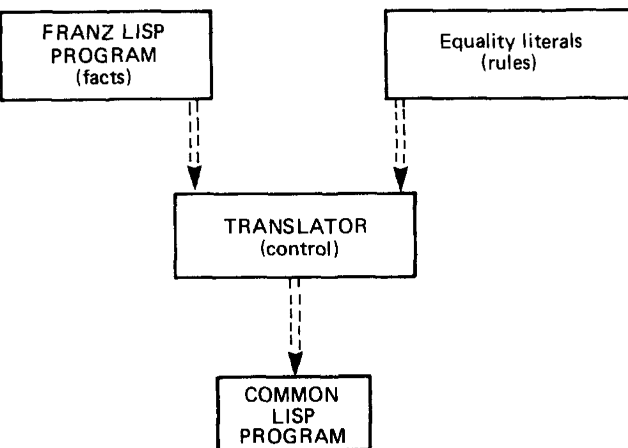

Figure 1. LDT prototype architecture

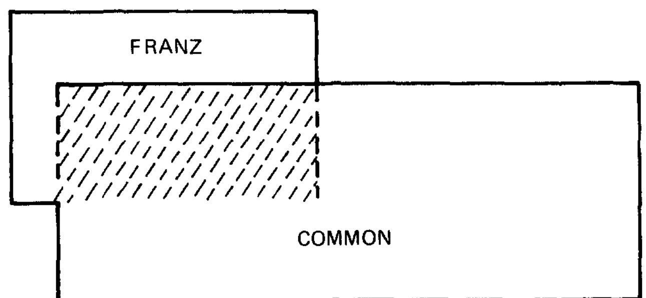

Figure 2. Relationship between Franz and Common Lisp

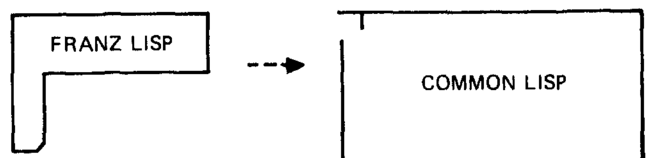

Figure 3. View of translation

while in Franz Lisp the member function uses the EQUAL test. The equivalent member function in Common Lisp compared to the member function in Franz Lisp is represented by the following one-to-many equality literal:

EQUAL (member (? *x*) (? *y*) : MEMBER *x y* :TEST 'EQUAL)

Finally one of the many-to-many equality literals:

EQUAL (append1 (? *x*) (? *y*) : append *x* (list *y*))

A worked example using the Franz-to-Common Lisp translator is shown in Figure 4.

CONCLUSION

In this paper a Franz-to-Common Lisp dialect translator was presented as an applied example of a rule-based general purpose program translator.

It has been shown that rule-based programming and the paramodulation technique mentioned in this paper can bring simplification to the process of program translation, and improve software productivity and re-usability, in that all the user has to do is provide rules in order to obtain the target translation.

The first function is a Franz Lisp program, which will simply take two numbers as input, see if they are within a certain range, then print a message depending upon the range.

```
(defun test1
  (num1 num2)
  (let [(x (add1 num1)) (y (sub1 num1))]
    [setq 1
      (do [(range (list y num1 x) (append1 range (add1 x)))
            (x x (add1 x))]
            [(member num2 range)
            (length range)])])
  (cond [(greaterp 1 12) (print 'too much difference')]
      [(plusp (difference 1 3)) (print 'ok')]
    [t (print 'too little difference')]]
        Franz Lisp function
(defun test1
  (num1 num2)
  (let ((x (| 1 + | num1)) (y (| 1 - | num1)))
    (setq 1
      (do ((range (list y num1 x) (APPEND range (LIST (
            (x x (| 1 + | x)))
            ((MEMBER num2 range :TEST 'EQUAL)
            (length range)))))
  (cond ((> 1 12) (PRIN1 'too much difference'))
    ((plusp (- 1 3)) (PRIN1 'ok'))
    (t (PRIN1 'too little difference'))))
                    Common Lisp function
```

The translated Franz Lisp functions were add1, sub1, append1, member, greaterp, print and difference, along with translating all of the superparentheses used in the Franz Lisp program to demarcate the significant structures.

Figure 4. Worked example of the Franz-to-Common Lisp dialect translator

REFERENCES

1. **Dowell, M and Takefuji, Y** 'A rule-based Lisp dialect translator using paramodulation', *Proc. Future Directions in Computer Hardware and Computer Software* (1986)
2. **Lusk, E L and Overbeek, R A** *The Automated Reasoning System ITP* Argonne National Laboratory
3. **Wilensky, R** *LISPcraft* W W Norton and Co. (1984)
4. **Foderaro, J K and Sklower, K L** *The FRANZ LISP Manual* University of California (1982)
5. **Winston, P H and Horn, B K P** *LISP* (2nd Ed) Addison-Wesley Publ. Co. (1984)
6. **Steele Jr., G L, Fahlman, S E, Gabriel, R P, Moon, D A and Weinreb, D L** *Common Lisp* Digital Equipment Corp. (1984)
7. **Brooks, R A** *Programming in Common Lisp*, MIT, John Wiley and Sons, Inc. (1985)