Translating KL-ONE from Interlisp to FranzLisp

Tim Finin

Franz KL-ONE translation project
University of Pennsylvania

(Presented during the main conference)

This section describes an effort to translate the Interlisp KL-ONE system into Franzlisp to enable it to be run on a VAX. This effort has involved Tim Finin, Richard Duncan and Hassan Ait-Kaci from the University of Pennsylvania, Judy Weiner from Temple University, Jane Barnett from Computer Corporation of America and Jim Schmolze from Bolt Beranek and Newman Inc.

The primary motivation for this project was to make a version of KL-ONE available on a PDP 11/780 VAX. A VAX Interlisp is not yet available, although one is being written and will soon be available. Currently, the only substantial Lisp for a Vax is the Berkeley FranzLisp system. As a secondary motivation, we are interested in making KL-ONE more available in general - on a variety of Lisp dialects and machines.

When we began the effort (summer 1981) we first looked at several existing inter-dialect Lisp translation systems (e.g., Interlisp's TRANSOR, SRI's MacLispify, the MIT MacLisp system developed to transport LUNAR to the Lisp Machine, and several smaller systems). None of the systems quite fit our criteria so we decided to create our own translation system. Our approach was to first build a general purpose inter-dialect Lisp translation system that is driven by transformation rules. We then developed a set of specific Interlisp to FranzLisp translation rules and an appropriate run-time support system for the resulting FranzLisp version of KL-ONE.

The current status of the project is as follows. The basic translation engine (Franzlator) has been implemented and is running smoothly. Our collection of Interlisp to FranzLisp rules, which is tailored for translating KL-ONE, numbers about forty. The run-time support environment (dubbed InterFranz) contains about 250 functions, mostly macros. In addition, a rudimentary DWIM-like facility has been developed to handle certain classes of expressions which tend to slip through the translation process.

Building a general purpose inter-dialect lisp translation system is a fairly large project in its own right and may seem to be an inefficient way to transport KL-ONE from Interlisp to FranzLisp. We have chosen to do this for several reasons. The chief ones are:

- We want a FranzLisp version of KL-ONE which tracks the current Interlisp version. Since the KL-ONE system is still evolving rapidly, this will require periodic re-translations. Thus effort spent to mechanize the translation task will pay off in the long run.

- We anticipate a desire to transport KL-ONE to other Lisp dialects such as LispMachine Lisp or Common Lisp. A properly designed inter-dialect translator will minimize the future cost of this effort.

- We expect to use the translation system to import other Interlisp systems into Franzlisp. Current candidates are RUS and PSI-KLONE. We also expect to write other sets of translation rules to use the translation system to import from other Lisp's and to export native FranzLisp programs. Thus the cost of building a general purpose translation system will be shared by other projects.

A. Translation versus Emulation

In undertaking to transport a large system such as KL-ONE from one dialect of Lisp to another there are two basic approaches: translation and emulation. Translation involves transforming the Lisp code from the initial source-dialect to the desired target-dialect. The result is a program that can be run directly in an unmodified interpreter for the target-dialect. Emulation involves reconstructing the source-dialect's environment in the target-Lisp's interpreter. Properly done, this enables the unaltered source-code to run directly. These two approaches are, of course, poles on a continuum which admit a wide range of hybrid systems.

The emulation approach, or a mixed system which is near to pure emulation, is very attractive from several points of view. An emulator tends to be easier to construct for many of the same reasons that interpreters are typically easier to construct than compilers. The emulator's task is intrinsically easier since all of the work takes place at the last moment (at run time) when all of the information is available. Once we are successful in emulating the environment, other packages of code from the source dialect can be run directly without any additional work. Still another advantage is that the source code which is run in the emulation environment is identical (more or less) with the original code. This is in contrast to a translation system which might transform readable source language code into executable, but unreadable, target language code.

In spite of these apparent advantages, we have taken a translation approach. The major reasons for this are:

- Maintaining a FranzLisp environment.

We want to maintain an environment in which any native FranzLisp code will run. Constructing a fairly complete Interlisp emulator would entail fundamental changes in the environment.

- Avoiding naming conflicts.

  Many of the differences between Interlisp and Franz can be handled by adding definitions for those built-in functions which Interlisp provides but Franz does not (e.g. TCONC). In many cases, however, Franz and Interlisp use the same name for different functions. One common difference arises when the same symbol refers to two unrelated functions. The function *, for example, is a comment introducing function in Interlisp and multiplication in FranzLisp). A second class of differences arises when there is variation in the "syntax" of the function. The function MAPCAR, for example, takes it arguments in a different order in Interlisp and Franzlisp. A third class of differences involves the "semantics" of the function. An example here is LISTP function which in Interlisp returns T only for non-empty lists and in FranzLisp returns T for any list, including NIL.

- To have a stable textual Franz Lisp version of KL-ONE.

  The output of the translator is a set of files which comprise a stable text-level representation of Franz Lisp KL-ONE. We believe that this makes it easier to debug, maintain and modify a large system like KL-ONE.

- Generality.

  We believe that a translation approach will be easier to extend so that we can eventually produce versions of KL-ONE for other Lisp dialects. An emulation approach is more likely to depend on features of the target language will may not be present in a new candidate target language. Macros, for example, would typically be used in an emulation approach whenever possible. Some Lisp systems, such as MTS Lisp, do not support Macro functions.

ALthough we characterize Franzlator as a translation system, it is in fact a mixed system which has a significant run-time component. Most of that runtime component consists of definitions of functions which Interlisp provides but FranzLisp does not. We have chosen to define these functions as macros whenever appropriate (e.g., for simple functions like GEQ, which compares two numbers for a "greater than or equal to" relation). This has the effect of enabling us to vary the size of the run-time component by simply including a translation rule which expands calls to macros at translation time instead of run time or compile time.

II. The Translation Process

Although we have written a translator in FranzLisp, the entire translation process involves a total of four machines. The process begins on a JERICHO where the Interlisp DWIMIFY function is used to translate all of the CLISP code into standard Interlisp. The resulting dwimified files are then transferred to the BBNG machine from which they are FTPed to WHARTON-10 and finally transferred by a local networking facility to a VAX in Penn's CIS department. There the files are passed through the FRANZLATOR system to produce two sets of files. One set represents the FranzLisp version of KL-ONE and the other a collection of notes about the translation process (e.g. unrecognized functions, expressions which may require hand translation, etc.).

In translating KL-ONE, the FRANZLATOR system uses three major databases:

- A database of Interlisp to FranzLisp translation rules.
- A database containing information about the Interlisp system functions (e.g., function type, number of arguments, special forms).
- A database describing functions in the InterLisp runtime environment (InterFranz).

III. Organization of the Translator

The Translator is organized as a two-pass system which is applied to a set of source-dialect files and produces a corresponding set of target-dialect files. During the first pass all of the source files are scanned to build up a database of information about the functions defined in the file. In the second pass the expressions in the source files are translated and the results written to the target files. The translation of an s-expression is driven by transformation rules applied according to an "eval-order" schedule (i.e., the arguments to a function call are translated before the call to the function itself). In addition to the transformation rules, the translator is guided by the data base of information about the functions, both the built in Interlisp functions and the user-defined ones.

An additional pass, to be done initially, may be required to perform certain character-level transformations. In translating KL-ONE from Interlisp to FranzLisp, however, we found that all of the necessary character level transformations could be done through the use of multiple readtables. The readtable used when reading the original Interlisp files ,for example, treats the character ";" as a normal alpha-numeric and "%" as an escape character.

A. The First Pass

During the first pass all of the source files are scanned to build up a database of information about the functions defined in the file. In particular, for each user-defined function we need to know how many arguments it expects and whether or not it evaluates them. The translator must know how many arguments each function expects in order to supply default values for missing arguments or to remove any extra arguments. This is important since Interlisp functions can take any number of arguments. Missing arguments are supplied as NIL arguments and extra arguments are not passed to the function. It is common practice for many Interlisp programmers to rely on this convention, especially with regards to missing arguments. An example is to write (CONS X) rather than (CONS X NIL).

The translator needs to know how each user-defined function evaluates its arguments in order to correctly translate the arguments in a call to that function. If a function parameter is not evaluated (as is the case in a Fexpr or Nlambda type function) then the translator should not translate the corresponding argument in any calls to the function. If the argument is evaluated, either by the interpreter or explicitly by a call to EVAL from within the function, the the translator must translate the argument. The problem, of course, is how to determine whether or not a function explicitly evaluates an initially un-evaluated argument.

The handling of function arguments which may or may not be evaluated is problematic in systems such as this. The proper thing to do is to examine the code to the function and try to determine whether there is an explicit call to EVAL. Franslator takes the more practical approach of assuming that the function either will or will not explicitly evaluate all of its un-evaluated arguments. The decision is controlled by the value of a global variable. A facility is supplied which allows one to directly inform the translator about a function's type, number of arguments and exactly which arguments are evaluated.

B. The Second Pass

During the second pass, each source-dialect file in the set is processed independently, resulting in a corresponding file in the target-dialect. The processing, for the most part, is simply a matter of translating each s-expression in the source file and writing the result in the target file. In addition, for each of the source files, a file containing notes about the translation is produced. Entries are made, for example, when the translator discovers a function which is not in its data base and upon encountering a function call with an improper number of arguments. In addition, any rule can add notes to this file as one of its side effects.

IV. Transformation Rules

The actual translation is done by a set of transformation rules. Each rule specifies the translation of one s-expression into one or more resultant s-expressions. In addition to the usual "pattern" and "result" parts, rules can be easily augmented with arbitrary conditions and actions.

A. The Structure of a Rule

A rule has two obligatory parts: a pattern which determines the expressions the rule applies to and a result which specifies the result of the transformation. In addition to these, a rule can have up to five optional attributes such as a test and priority. The syntax of a rule is:

```
          <rule> -> (<pattern> <result> . <attributes>)
    <attributes> -> ()  |  (<attribute> . <attributes>)
     <attribute> -> (<attribute name> <attribute value>)
<attribute name> -> test  |  side-effect  |  priority ...
<attribute value> -> {an s-expression}
```

Variables in the pattern are specified using a variation of the MacLisp "backquote" convention. Any symbol in the rule's pattern which is preceded by a "," is taken as a variable which can match any one s-expression. A symbol preceded by ",@" can match any number of sister s-expressions. In the result part of a transformation rule the comma and @ have a slightly different interpretation. There, s-expressions which are preceded by a "," are replaced by their values and those preceded by ",@" have their values "spliced in" in their place.

Some examples of transformation rules are shown below.

```
[r1]  (NIL nil)
[r2]  ((NLISTP ,x)  (not (dtpr ,x)))
[r3]  ((PROG1 ,@args)  (prog2 nil ,@args))
[r4]  ((MAPCAR ,list (FUNCTION ,f))
       (mapcar ',(makeMonadic f) ,list))
[r5]  ((DECLARE: ,@args)  ,(translateDeclare: ,args))
```

Rule [r1] is the simplest, mapping the symbol "NIL" into the symbol "nil". Rule [r2] introduces the use of a simple variable. The third example rule shows an application requiring a "@" variable. The rule [r4] shows an computation embedded in the result part of the rule. The second element of the result will be a list whose CAR is QUOTE and whose CADR is the result of calling the function makeMonadic with argument f. The last rule, [r5] is one in which the entire result is computed.

The optional rule attributes include TEST, SIDE-EFFECTS, PRIORITY, TYPE and REGIME. The value of a TEST attribute is an Lisp expression which must evaluate to non-NIL before the rule can be applied. The test is run after the pattern has matched so that the pattern's variables will be bound to values. An example of a rule using the TEST attribute is:

```
((PLUS ,@args1 ,x ,@args2 ,y ,@args3)          ; pattern

 (PLUS ,@args1 ,@args2 ,@args3 ,(+ x y))       ;result

 (test (and (numberp x) (numberp y))))         ;test
```

This rule causes any numeric arguments to PLUS to be collected and summed at translation time.

The SIDE-EFFECT attribute introduces a Lisp expression which will be evaluated whenever the rule is applied and the result has been computed. Side effect attributes are typically used to write messages about the translation into the file of translation notes or to the terminal.

The PRIORITY attribute is used to rank the rules. Whenever two rules both apply to an expression being translated, the one with higher PRIORITY is applied first. Our current Interlisp to Franzlisp translation rules do not use the priority feature.

The TYPE attribute should have as its value either splicing or replacing (which is the default rule type). A splicing rule is one in which the result is a list of expression which are to be "spliced" into the list containing the expression being translated. A splicing rule is used, for example, to transform a call to DEFINEQ into a sequence of calls to defun at the top level of the file. In a replacing rule, the result is simply replaces the original expression.

The REGIME attribute must be either cyclic or acyclic (the default case). A cyclic rule can apply more than once to the same expression whereas a

acyclic rule can only be applied once. The default REGIME is acyclic. An example where a cyclic rule is appropriate is:

```
((and ,@x (and ,@y) ,@z)      ; pattern
 (and ,@x ,@y ,@z)            ; result
 (regime cyclic))
```

This rule eliminates a call to AND if it is embedded in another AND by raising its arguments.

B. Rule Representation

The translation rules are presented to the system in the form described above and are immediately "compiled" (by macro-expansion) into Lisp code. Each rule becomes a monadic function whose argument is an s-expression to be translated. If that expression matches the rule's pattern then the function will compute and return the translated form. If the expression and pattern do not match, then a special symbol indicating failure is returned. The Lisp code generated for a rule is optimized for efficiency. The pattern matching operation, for example, is "open coded" into a conjunction of primitive tests and action (e.g., EQ, EQUAL, LENGTH, SETQ).

Each rule is indexed by first assigning it to one of four classes depending on the nature of its pattern. The four classes are rules whose patterns are: (1) atomic; (2) lists with literal atoms as their first element; (3) lists with variables as their first elements; and (4) lists with lists as their first element. Rules in class two are indexed on the property list of the symbol in their CARs. The other classes are not further indexed.

C. Controlling the Translation

The translation system was designed to provide a high degree of transformational power in a simple format. A person writing a set of transformation rules may want to have greater control of the translation engine. In order to provide for such situations, the translation system makes available a number of control functions and certain relevant global variables. For example there exist functions for aborting the application of a translation rule and for prematurely ending the translation of expression

without considering the application of any other rules. The rule writer has access to such values as the stack of forms undergoing translation (to allow for context sensitive rules), and the name of the current Lisp function being translated.

In addition, there are various support and debugging functions which facilitate the development of new sets of translation rules.

V. Summary, Current Status and Future Directions

This section has reported on the development of a general inter-dialect Lisp translation system and its application to the task of translating the Interlisp implementation of KL-ONE into FranzLisp. The translation system is running smoothly and fairly efficiently. The current set of translation rules and run-time support functions appear to cover all of the basic facilities needed by KL-ONE. We are currently in a cycle in which a translation of KL-ONE is made and then run to discover bugs in the translation rules or run-time support system.

Some future work will be directed towards experimenting with extensions to translation system to allow for more flexibility, power and/or efficiency. Other work will involve broadening the set of interlisp to Franzlisp translation rules to handle constructions not required in translating KL-ONE and to handle CLISP code. A third direction is the writing of rules for translating between other pairs of Lisp dialects. A set of Interlisp to CommonLisp rules might be very useful, for example.