Pascal to Smalltalk

Barry Haynes
April 11, 1985

Current Status

A new class, PascalCodeParser has been created. It is a subclass of Parser that parses Pascal source code to produce a Smalltalk parse tree. From this parse tree, Smalltalk source is created and each method is added to the Smalltalk class that should include that method. Currently, the user must create the Smalltalk class manually. All methods are automatically placed in a category called **pascal source**. PascalCodeParser can now handle parsing procedures and functions and converting Pascal formal parameters into Smalltalk keyword parameters. It converts simple local variables from Pascal to Smalltalk. Pascal expressions and assignment statements are correctly converted into equivalent Smalltalk statements. Handling function return results is also implemented.

At this point, we know the basic conversion mechanism works. All that is required to do complete conversion from Pascal to Smalltalk is Smalltalk methods and a scheme for each of the parts of the Pascal railroad diagrams that have not yet been implemented. The rest of this memo deals with these issues and problems that must be resolved to finish the job.

Issues for the Rest of the Conversion

1. In general it is assumed the Object Pascal that is being translated has been compiled, has no syntax errors and is correct, complete Object Pascal in every way.

2. It will be assumed that real numbers are not used in the Pascal that is being translated.

3. Globals could be handled in the following way. For MacApp itself, global variables and constants can be added as class variables to class TObject. For the case of constants, code will be added to TObject initialize to set the constants to the correct values. For a particular application, globals and constants declared in the interface and implementation of that application will be stored in a shared pool shared by all classes of that application. The shared pool will be initialized by the initialize method of whatever class the parser comes comes across first. It is generally assumed this will be a sub-class of TApplication.

4. There will be a symbol table that will remember needed information about particular types and variables, like if it is an array or a PascalRecord. Variables that require a PascalRecord will have code generated in either the appropriate class initialize method or the local method, depending on where the Variable is located, to create the PascalRecord.

5. Both the interface and implementation of MacApp or a specific application will have to be parsed. This will allow the parser to know about the types of all variables. The parser will need to know about global procedures as well as methods that haven't yet been parsed. Only by parsing the interfaces first can this information be obtained. Procedures that are global to MacApp will be defined as class methods of TObject. Procedures that are global to a particular application will be defined as class methods of the first class that is parsed in that application. This is assumed to be a subclass of TApplication. Identifiers will have to be looked up in the symbol table to see if they are a function call. When they are, the name of the class will have to be placed in front of them. Calls to the Toolbox will also have to be recognized by symbol table lookup so 'Mac' can be placed in front of them. When a procedure has parameters, the Smalltalk keywords will have to be looked up and added as the Smalltalk call is made.

6. Pascal character literals need to be converted to Smalltalk equivalents.

7. Pascal declarations for enumerated types will be handled by making a constant for each type starting with the value 0. Comments will be added to aid in reverse translation.

8. Sub-Range types will be handled by treating them like integers and assuming the Pascal range checking has delt with any problems. Upon translating back, Pascal range checking should flag problems added while in the Smalltalk world.

9. "Packed" will be ignored for now.

10. Code will be generated at the beginning of the appropriate method to allocate the correct ammount of space for any array variable that is declared.

11. Varient Records would have to be a new sub-class of the original record for each varient. For now, we will assume we don't have any varient records. Can we assume this?

12. Object type statements will cause the definition of a new Smalltalk class with instance variables equivalent to the Pascal ones. The Smalltalk methods will be added while parsing the interface part and they will be put in the categories as specified by the Pascal comments.

13. Things like "override", "forward", "external", "inline" will be saved in comments for translation back to Pascal at a later date.

14. There will be a general escape mechanism that will skip things we have not yet decided how to translate. Skipped source will be left in it's Pascal form within a Smalltalk comment with the flag "**Skipped**" at the start of the comment.

15. Within Smalltalk, there will be no difference between variables that are declared to be some type of Pascal Record verses those that are declared to be a pointer to a Pascal Record. Both of these must use a subclass of PascalRecord to access the Toolbox. So statements like **foo^.bar** and **foo.bar** will both be translated into the Smalltalk **foo bar**. The statements **foo^.bar :=** and **foo.bar :=** will both be translated into **foo bar: <**. We will have to leave in appropriate comments to aid in the reverse translation.

    There is one subtle thing having to do with VAR parameters within Toolbox calls. If the Pascal declaration is "VAR foo:PascalRecord" then the type encoding should be "D" for direct pointer since the contents of the PascalRecord will be changed by the procedure. If the Pascal declaration is "VAR foo:^PascalRecord", for a pointer to a PascalRecord, then the type encoding should be "VD" since the pointer value itself will change. Within the Toolbox calls that are accessed through the table lookup, all of them are encoded as "VD". We will have to look through these for the problem cases and encode them by hand.

16. Pascal Records that contain 4 bytes of data or less have to be passed as LONGINTs instead of a subclass of PascalRecord.

17. The filebuffer symbol within Pascal, f^, will be ignored for now assuming it is not used.

18. NIL is passed back and fourth between Smalltalk and the Toolbox in the following way. The statement "PascalRecord new" creates a PascalRecord with all three of it's fields equal to NIL. When this is passed to the conversion routines, it is converted to 0, the Pascal version of NIL. When 0 is passed back from the Toolbox as a pointer, handle, function result or VAR parameter, the conversion routines pass back a PascalRecord with all three fields equal to NIL.

19. If statements will be done by moving the expression to the front then using ifTrue: for the "THEN" part and ifFalse: for the "ELSE" part.

20. Case statements will be done by assigning the expression to a tempory variable then doing a compare of the tempory to each Pascal case using nested ifTrue: ifFalse: statements. If there is an otherwise clause, it will be the code for the final ifFalse.

21. While statements translate into: (expression)
    whileTrue: [code]

22. Repeat statements translate into: [code, expression]
    whileFalse:[ ]

23. For statements translate into: x to: y by: delta do: [ ]

24. The statement "WITH foo DO [block]" can be implemented by looking at each expression within the block foo and seeing if it is a local variable. If it is not a local variable, then combine it with foo and see if it creates a valid method of whatever PascalRecord foo is a subclass of. If it does, then use that method instead of the original expression. To translate back, all the foos can be removed from an area deliminated by "with foo" comments of some sort.

25. For procedure parameters,say to call foo, one can pass a block that looks something like [generateWhateverKindOfReceiverFooNeeds foo]. If the procedure parameter has parameters, then you send a block with parameters like [:param1 :param2 | generateWhateverKindOfReceiverFooNeeds foo: param1 with: param2].

26. To implement Pascal VAR parameters with Smalltalk, if there is just one Var parameter and the call is currently not a function, convert the call to a function and return the Var parameter return value. If there are more than one Var parameters or the call is already a function, you pass in a block that returns as its value the input value of the Var parameter. This block also has a parameter that you pass to it. It assigns the value of this parameter to a local variable in the method that called the method with the Var parameter. Example:

```
| returnValue initialValue |
initialValue < 5.
   "call foo passing in 5 as the initial value for the VAR parameter"
foo: [ :newReturnValue | returnValue < newReturnValue.    ^initialValue]


"within foo"
foo: intValue  <VAR INTEGER>

| inputValue |
   "send the value: message to intValue to get the initial value of the VAR
    parameter. The parameter you pass it, 10, is the return value for the VAR
    parameter. You can evaluate the block more than once if the return value
    is based on the input value."
inputValue < intValue value: 10
```

Will this always work? Is there an easier way!

27. Pascal comments using {} will be translated into Smalltalk comments using "". Additional information that is added to the Smalltalk source to translate back into Pascal will use comments within {}. The Smalltalk parser will be changed to recognize the {} comments.

28. Problems discovered during hand translation of the early Text Edit version of MacApp. These need to be dealt with during the automatic translation process.

a. Objects in Smalltalk can get deallocated if nobody is referencing them. The trick of allocating a window object then stuffing it with setWRefcon to later retrieve it with getWRefcon has the problem that the object may be de-allocated while it is only referenced by the Toolbox refcon field. I saved the window in a Smalltalk collection within class TWindow and used the index I saved with setWRefcon to later retrieve the correct window.
b. Toolbox constants and globals need to be available for the MacApp application.
c. Some Object Pascal methods get really too large within the Smalltalk environment.

d. Calls to the Toolbox that return a VAR parameter value can sometimes have subtle problems when translated. Example:

the Pascal statement GlobalToLocal(theEvent.where) translates to:

Mac globalToLocal: theEvent where

In this case, **where** never gets the local value returned to it since the message **theEvent where** returns a tempory object and that tempory object is the one that gets the new value returned to it. The solution I used was:

tempPt < theEvent where.
Mac globalToLocal: tempPt.
theEvent where: tempPt.

e. Toolbox globals that might get changed by the ToolBox, for example thePort, require a method to access them. This method gets the current value from the real global. Maybe all Toolbox global vars should be accessed via a Mac method call instead of having equivalent Smalltalk class variables.

f. In Smalltalk you can't write new values into procedure parameters as you can in Pascal. You need to generate a local tempory variable along with the code to copy the param's value into that tamp. Now, within the method one can assign a new value into that temp.

g. Pascal create methods do **SELF :=**. The translation is to create the new object in a tempory variable then return the object you created.

h. Pascal uses **SELF.VarName** to access instance variables, Smalltalk just uses the variable name alone. Pascal doesn't use **SELF.ProcName** to access methods within the same class, Smalltalk does use **SELF** in this case.

i. As mentioned earlier, the encoding of some ToolBox calls is wrong. Example:

FUNCTION GetNextEvent(eventMask: INTEGER; VAR theEvent: EventRecord);

is encoded in the table as 'B-IVD'. It should be encoded as 'B-ID' since theEvent is declared as a Record, not a pointer to a Record. We don't want the pointer value to change, as would happen in the original encoding. We want the contents of what is pointed to to change, as per the correct encoding. We need to look through all interfaces and fix the encodings on VAR paramethers where this is a problem.

j. Resources need to be created using the Workshop. There are other parts of the development cycle that would still have to be done using the Workshop or the finder. This is a frustration but we can certainly live with it for now.