Abstract

Many applications have one or more important modules that are written in a language other than conventional procedural or object oriented languages. These languages are often translated into some tabular or byte code representation which is executed by an interpreter. The interpreter is sometimes written in the same language as the rest of the application. Examples of such languages are yacc and Tcl. This paper investigates a technique for linking entities in the interpreted languages to their implementation in the interpreter. The technique is illustrated using the design recovery of a compiler that uses S/SL, a grammar language similar to yacc. We present a model extracted from the compiler that links entities in S/SL to the underlying implementation language and some example maintenance queries that might be posed for the system. Extracting the model requires recognizing idiomatic code in the interpreter. We then discuss how the technique might be extended to similar systems such as those implemented in Tcl.

1. Introduction

Design recovery of conventional procedural and object oriented languages is a well known process. Research includes standard models for procedural languages [22] including C++ [10], extraction of information [1, 9], analysis of the information [23,11,26, 19, 12, 2 16] and presentation of results [18,27]. In most of the research, separate languages are handled at the linking level. That is, separate modules may be written in different languages, and most of the techniques handle relationships between the entities in different languages without difficulty.

Design recovery of sublanguages embedded within the language such as CICS and SQL has also been addressed [30]. However, little attention has been paid to special purposes languages that are part of applications, but are not implemented as sublanguages. Examples are compiler languages such as yacc [17], lex [21] and scripting languages such as Tcl [25]. These languages are often translated to some tabular or byte code form which is executed by an interpreter, often written in the same language as the rest of the application.
2. S/SL

The Syntax/Semantic Language (S/SL) was designed for implementing compilers and has been used by IBM to build one of the later versions of their COBOL compiler[3]. A complete description of S/SL is beyond the scope of this paper, but we present a small example to motivate the language and describe its runtime environment.

Figure 1 shows a simple S/SL program that recognizes the language $(n \ a \ )^n$. The output of the program is the value of $n$ (the number of nested parentheses). The left column contains the definitions of the input symbols, any error signals, and the definition of the single mechanism. The right column contains the grammar specification. The program starts with the first grammar rule which is $EvenBrackets$ which calls a recursive production, $MatchBrackets$ which is responsible for parsing matched brackets.

Semantic mechanisms are hooks to indicate where additional processing is to take place. Without semantic mechanisms, the S/SL is simply a parser that can recognize context free languages. In this case the semantic mechanism $Counter$ adds a simple counter to the program. It contains three operations. The first, $counterSet$, initializes the counter. The $counterIncrement$ operation increments the value of the counter by one, and the last operation, $counterOutput$ writes the value of the counter to the output. The initialization and output routines are called in the first production ($EvenBrackets$) while the increment operation is called just before each recursive call of $MatchBrackets$.

One of the questions that a maintainer of a compiler might want to know is which semantic operations are invoked directly or indirectly by a given rule, and to what semantic mechanisms do those operations belong.

Figure 2 shows the structure of the S/SL system. The S/SL program is turned into a byte code table and a set of definitions for use by the S/SL runtime system.
The bytecodes and definitions are interpreted by a walker which implements the S/SL virtual machine. The walker can be implemented in any conventional programming language such as C or C++. Since S/SL is a simple language, the basics can be provided by thirteen instructions provided by thirteen predefined bytecodes.

The semantic operations are implemented as additional bytecodes. The example S/SL program from figure 1 will contain 16 byte codes: the predefined thirteen byte codes and the three bytecodes used to implement the three operations of the counter semantic mechanism. The compiler writer must extend the walker to execute the code that implements the semantic operations when the new bytecodes are encountered.

Ideally, if software engineering principles have been followed, there is a collection of variables and subroutines that implement each of the mechanisms and some other variables and subroutines that are used to communicate between semantic mechanisms. However, since the implementation language may not directly support modules, the division between the variables may be a matter of convention. There may also be other utility variables and procedures that are commonly used by all mechanisms.

In this development environment, a clear general question to a maintainer of an S/SL based system is to partition the variables and procedures into these three groups plus the variables and procedures that implement the walker itself.

But more specific questions arise when a maintainer of an S/SL based system is tasked with modifying the system. For example, which variables in the walker are modified by a given S/SL rule? That is which variables are modified by semantic operations invoked by the given rule or invoked by any other rules that are called by the given rule?

To answer any of these questions we need three things:

1. A design model of the walker program. Since the walker is implemented in a conventional language such as Pascal, C or C++, this is a well known problem in the reverse engineering and program comprehension communities.

2. A design model of the S/SL program. While there are some analogies with models of conventional languages, there are some interesting issues.

3. Links between the two models. The entities that represent semantic operations must be linked to the implementation of those semantic operations.

These issues are discussed further in Section 4 and 5.

3. PT Pascal

PT Pascal is a subset of Pascal used for teaching compiler classes originally implemented at the University of Toronto by J.A. Rosselet[24]. PT Pascal is used to teach the 4th year compiler class at Queen’s University. The students are given a project that involves modifying the compiler to accept a slightly different language.

The compiler is implemented in four passes: scanner, parser, semantic analyzer and code generator. All of the passes use S/SL as the basis of the program. The walker and semantic mechanisms of all passes are implemented in PT, illustrating the bootstrapping concept.

The S/SL processor produces PT Pascal constant definitions for input, output and error symbols and for the semantic operations. The definitions allow the walker to refer to the operations and data used in the byte code symbolically. Within the walker, the bytecode table is represented as an array of integers.

Since PT does not have record structures, multiple variables must be used. For multi-valued structures such as the symbol table and type stack, multiple arrays must be used. This particular implementation will influence the questions that the maintainer needs to ask. One of these questions is: which variables represent the state of a particular semantic mechanism?

4. Models

For this investigation, we used a relatively simple model for our design database, but got interesting results. Figure 3 shows an ER Diagram of the model for the information that we extracted from the source code. Rather than extracting a more general model such as DMM [22], our model is based, in part, on the needs of a specific maintenance task. The task was provided by the 2002 compiler class at Queen’s. Students, including Ms. Chen were required to modify the PT compiler to accept a C++ like language.

We did not attempt to complete the extraction and analysis in time to use for the compiler class project. Instead we kept track of the type of questions asked during the class project and used them as a guide when designing the model and the queries. The simplicity of the model also highlights the relations that cross the language boundary.

The left hand side of the figure is the model for S/SL language. The entities are rules (SSLRule), semantic mechanisms (Mechanism), and semantic operations (Operation). Rules can call other rules (RuleCall) and can invoke semantic operations (Invokes). Each operation is part of a particular semantic mechanism (partOf).
The right hand side of the figure represents the model for PT Pascal. The only entities modeled are procedures and variables. The main program of the Pascal program is considered a procedure. The modeled Pascal relations are straightforward. The modify relation indicates that the variable is used on the left hand side of an assignment in the procedure, and is a subset of the reference relation which indicates the variable is used in the procedure.

Variables are subclassed into several classes representing global and local variables, global and local constants, and procedure parameters. Local variables and constants are variables and constants declared within a procedure.

Three relations link the entities of the S/SL model to the entities of the Pascal model. As we mentioned in Section 2, semantic mechanisms are implemented in the S/SL walker by the addition of code and data structures. Section 5 discusses how the the code for the semantic operations is recognized. This code may modify and reference variables and it may call procedures. These actions are modeled by the opModify, opReference and opCalls relations.

5. Model Extraction

The extraction of the model is slightly different for the S/SL code and for the PT. In general, both are similar to the extraction methods presented elsewhere [20, 8, 7].

The source program is first run through a unique naming phase [20] which gives each entity in the system a unique name. These names are the names that will be used to refer to each entity in the model. Several extractors are run on the uniquely named source and generate the model. In our particular case-study, we used TA [14] to encode the model. TXL [5, 6] was used for all phases of the extraction process.

In the case of the S/SL model extraction, unique naming was not needed. The language has no lexical scoping and all symbols in the language are global. As such, the name of a rule, mechanism, operation, type or type value are already unique.

The extraction of the PT model from the PT source code is also straightforward. The PT Pascal code is first uniquely named, and other TXL programs are used to extract a conventional model. The main difference from most previous research in extraction is that the extractor is also designed to recognize the implementation of the walker as well to a direct model of program entities. The approach is, however, similar to other extractions based on idiomatic code such as recognition of business rules [29] or the recognition of state machines [28].

A prototype S/SL walker is included with the standard S/SL distribution. Figure 5 shows a portion of the prototype walker for PT Pascal. The main processing is a repeat–until loop that contains instructions to update the current byte code followed by a case statement to decode and execute the byte code. The first thirteen entries (starting with oCall and ending with oSetResult) are the predefined bytecodes that implement the base S/SL language and are already implemented in the prototype walker. In the figure, we have omitted the middle 11 predefined bytecodes to save space (the phrase “impl. of other ssl language codes” shows where they are in the prototype walker).

The byte code labels used for the case statements representing language byte codes (oCall, oResult, etc.) are predefined. The labels for the byte codes representing semantic operations are generated by the S/SL language processor as Pascal constants and incorporated into the constant section of the walker program. The developer extends the case statement to include alternatives using the labels generated by the S/SL language processor. In the figure the place where the implementation of the new
The implementation of semantic byte codes is given by the phrase, “impl. of semantic byte codes”.

Extracting the model involves recognizing this particular idiom in the expanded walker. In particular, the idiom that must be recognized is the case statement that decodes the virtual machine instructions. The extraction finds the case statement which contains as labels the 13 predefined opcodes and associates the code for each case alternative with the semantic operation. That section of code is checked for variables that are referenced, variables that are modified and procedures that are called. It then generates the relations \( op_{\text{Modify}}, op_{\text{Reference}} \) and \( op_{\text{Calls}} \).

Figure 4 shows the Standard S/SL walker. Figure 5 shows an example extract of code from semantic.pt, the semantic analysis phase of the PT compiler. In particular, it shows the code in the walker for the two operations \( o_{\text{SymbolStkPushFormalParameter}} \) and \( o_{\text{SymbolStkSetKind}} \). In the first case, the code that implements the semantic operation is abstracted into a procedure (and is in fact shared between several semantic operations). The second case, the entire semantic operation is implemented in a single PT Pascal line. While both cases are short (short samples were chosen for the figure), more extensive case alternatives exists in the compiler.

```pascal
procedure SSLWalker;
begin

  { Initialize Table Walker State }
  processing := true;
  sslStackTop := 0;
  sslPointer := 0;
  noErrors := 0;
  sslabort := false;
  AcceptInputToken;

repeat { until processing = false }

  operation := sslTable[sslPointer];
  sslPointer := sslPointer + 1;
  if tracing then
    SslTrace;

  case operation of
    oCall:
      if sslStackTop < sslStackSize then
        begin
          Assert((symbolStkKind[symbolStkTop] = syProcedure), assert24);
          SymbolStkPushIdentifier(symbolStkSymbolTblRef[symbolStkTop] +
              countStack[countStackTop]);
        end;
    oSymbolStkPushFormalParameter:
      { The top symbol is a procedure. Push the
        procedure's i'th formal parameter where i
        is the value of the top count stack entry. }
      begin
        Assert((symbolStkKind[symbolStkTop] = syProcedure), assert24);
        SymbolStkPushIdentifier(symbolStkSymbolTblRef[symbolStkTop] +
            countStack[countStackTop]);
      end;
    oSymbolStkSetKind:
      symbolStkKind[symbolStkTop] := parameterValue;
      if not sslabort then
        Assert (eof (tCode), assert25);
      end;
  end;
  until not processing;
Figure 4. Standard S/SL walker

Figure 5. Some semantic operations from semantic.pt
As illustrated by this example, the three relations linking the S/SL model to the PT Pascal model cannot represent all of the code that implements the semantic operation, since the code in the case alternative may consist of a single procedure call (e.g. \textit{oSymbolStkPushFor- malParameter} in figure 5). Instead the relations represent the particular pieces of code that we can guarantee is dedicated to the given semantic operations. However, the rest of the code that implements a given semantic operation may be inferred from the model based on the calls relation between procedures on the PT Pascal side of the model.

Figure 6 shows the TA relations that were extracted from the snippet of code given in figure 5. The first 9 relations represent the extraction of the \textit{oSymbolStkPushFormalParameter} semantic operation, while the last three lines represent the extraction of the \textit{oSymbolStkSetKind} operation.

6. Queries

We used grok \cite{15} to process several queries against our extracted model. Some of the queries are entirely about entities in the S/SL model, some queries concern the PT Pascal model, and some queries cross the boundaries. PT Pascal model queries are similar to conventional queries made on similar models for other procedural languages. This section shows one of the queries that one might pose for a grammar language. We also discuss two of the queries that cross the boundary between S/SL and PT Pascal.

6.1 Mechanism and Operation Invocation

The first question that we consider is an impact question. The semantic analyzer and code generator are coded as S/SL programs that parse the output of the previous phase (parser and semantic analyzer respectively) and perform their function using semantic mechanisms. A single rule may call several other rules to process a particular piece of the input.

So when modifying the semantic analyzer or the code generator, one question that arises: what semantic operations are called directly or indirectly by a given S/SL rule? This is equivalent to a query in a conventional procedural language as what procedures are called directly or indirectly from a given procedure. The S/SL version of the query can be expressed in grok as:

\begin{verbatim}
{ “givenRule”}. ((ruleCall*) o invokes)
\end{verbatim}

Figure 7 shows the result of this query for the rule “TypeBody” from the semantic phase. This is the rule that recognizes a type within a declaration of variable or another type. The operations listed are part of the SymbolStack, TypeStack, ValueStack and CountStack mechanisms.

6.2 Semantic Mechanism State Queries

The next set of queries is designed to answer the question which variables in the PT implementation represent the state of a particular semantic mechanism. To handle this question We start with two queries:

\begin{verbatim}
(inv partOf) o opReference. By composing the inverse of the partOf relation with the opReference relation this query gives all of the variables that are referenced in the case alternatives associated with all semantic operations for each semantic mechanism.
\end{verbatim}
This query is similar to the last, but it reports the variables referenced by procedures called by the case alternatives. The calls relation is reflexively and transitively closed to include the entire call tree reachable from each case alternative.

These two queries result in two relations that give all of the variables that are referenced by semantic mechanisms. This includes all of the variables that make up the mechanism, and all of the utility variables such as loop counters. It also includes local variables in functions that are used to hold temporary results. A further query restricts the results to included only global variables, but filtering out other global utility variables is a bit more difficult. In some cases, one mechanism may access the variables that represent the state of another mechanism. For example the symbol stack mechanism, used for evaluating correctness of expressions in the semantic analyzer may be accessed by the type stack mechanism.

Further filtering however is not needed in this case as the list for each mechanism is small enough that it may be evaluated by the maintainer directly. Figure 8 shows the union of both queries filtered by global variables the for the TypeStack mechanism. The result of this query shows the variables used to implement the semantic mechanism (those starting with ’typeStk’) and the variables used to communicate with other semantic mechanisms.

Figure 7. Semantic operations invoked by the rule TypeBody

Figure 8. Global variables referenced by the TypeStack semantic mechanism

6.3 Side Effects of Rules

While semantic mechanisms are supposed to be abstract mechanisms and treated as black boxes, sometimes the compiler developer may want to ask questions about the effects of a a particular rule on the runtime environment. Thus the last question is similar to both of the previous two questions: Which PT Pascal variables are modified directly or indirectly by a given S/SL rule. This query is simply the first query unioned with the second query (both modified to use opModify and Modifies instead of opReference and Reference). The result may or may not be filtered to restrict the result to global variables.

Figure 9 shows the result of the modified query restricted to global variables. This query gives a better clue to which variables are used to implement the semantic mechanism. The results of the reference query (fig 8) contained variables from other semantic mechanisms which were referenced by the code implementing the TypeStack semantic mechanism.

7. Future Work

Some links between interpreted languages are straightforward such as the link between a Java method and a native method which may be modeled as if it was a call between languages. However, some links are more subtle such as the links in S/SL and Tcl. While this paper has investigated the form of combined models between these
types of languages, much more remains to be done in identifying the idiomatic code that links features in extensible languages to features in the implementation of those languages.

For example, the Tcl interpreter is provided as a C library. An application built around the Tcl interpreter creates an instance of the interpreter using the library routine `Tcl_CreateInterp`. It then registers new commands using the library routine `Tcl_CreateCommand`. A simple example similar to the introductory example in chapter 29 of [25] is shown in Figure 10. In the example two commands are added to the interpreter, `readgraph` and `drawgraph`. The implementations of these commands are in the procedures `RdGrph` and `DrwGrph` (not shown here) which are passed as parameters to the `Tcl_CreateCommand` library routine.

Thus connecting a model of the Tcl language to the code that implements each command requires recognizing the appropriate calls to the `Tcl_CreateCommand` library routine. This may be done with a source transformation in TXL or ASF+SDF[30], or in this case, it could be recovered by analyzing a model for C/C++ that includes procedure calls such that produced by CPPX [9].

It could be argued that the queries examined are influenced to some extent by the academic nature of the sample project. We are investigating making the design database and query engine available to students taking future versions of the class to see if the queries are actually useful when modifying the compiler. The next logical step is to try it on a larger use of S/SL. S/SL has been used in some commercial contexts (i.e. by IBM[3]) and it would be interesting to pursue some analysis of commercial compilers using this technique.

The model can also be extended to be more complete. The implementation of the second semantic operation shown in figure 5 also shows the implementation of passing parameters to semantic operations. Recall that the values of the parameters were enumerated types given in the S/SL program. The values are passed by the walker from the predefined instruction `oSetParameter` in a predetermined variable `parameterValue`. Semantic operations may also return values which may be used to influence the grammar (choice operations). In our case-study we did not model the relationship between operation parameters and results in the S/SL language and the implementation in PT.

8. Conclusions

This paper has shown that by recognizing idiomatic code in the interpreter, one can relate entities in an interpreted language such as S/SL to the code in the runtime environment that implements them. This technique has applications beyond the compiler domain, with possible applications to scripting languages such as Tcl and fourth generation languages common in many business environments.

We have shown a model for the S/SL language that includes links between the S/SL language and the underlying implementation language. Several queries based on experience of graduate students modifying an existing compiler were also presented.

```
#include <stdio.h>   vm = Tcl_CreateInterp();
#include <tcl.h>   Tcl_CreateCommand(vm, "readgraph", RdGrph,
main(int argc, char * argv[]){   (ClientData)NULL, (TclCmdDeletProc *)NULL);
   Tcl_Interp * vm;
   int retcode;
   Tcl_CreateCommand(vm, "drawgraph", DrwGrph,
   if (argc != 2) {
      (ClientData)NULL, (TclCmdDeletProc *)NULL);
      fprintf(stderr, "usage: %s filename\n",
      retcode = Tcl_EvalFile(vm, argv[1]);
      argv[0]);
      if (*vm -> result != NULL){
         exit(1);
      } else if (*argc == 2) {
         fprintf(stderr, "usage: %s filename\n",
      retcode = Tcl_EvalFile(vm, argv[1]);
         argv[0]);
         exit(1);
      } else {
         fprintf(stderr, "usage: %s filename\n",
         argv[0]);
         exit(1);
      }
   }
```
The particular idiom recognized in this research is limited to S/SL based compilers. While the case-study was made on a teaching compiler, real world compilers have been built using S/SL[3]. An environment providing such queries would be useful to developers that are responsible for these compilers as they evolve over time.

One lesson learned both from this and other research recognizing idioms in the source is an important technique in recovering higher level design concepts. Another lesson is that recovered design models do not have to be comprehensive to be useful. Although the validation of our model is limited to personal experience with the task, it was more than sufficient to answer the important questions posed by the task.

If applications built using languages such as S/SL and TCL are to be understood, the analysis cannot be restricted to the underlying implementation language. Design recovery must work at the same level that the programmer deals with the code.

Acknowledgements

This research was funded by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References


