Design and Implementation of whirl2xaif and xaif2whirl

Nathan Tallent Mike Fagan *

November 2003

Abstract

In order to connect the Open64 Fortran front end to the xaifbooster differentiation engine, one must develop bridging tools to translate between Open64's intermediate representation language (WHIRL) and xaifbooster's intermediate representation (XAIF). This report describes the design and implementation of these translation tools: whirl2xaif and xaif2whirl.

1 Introduction

Modular source-to-source automatic differentiation (AD) tools use a three-component design. These components are:

- A front-end to process user programs
- A differentiation engine
- A back-end to convert the differentiation engine output back into a user program

The front-end and the back-end are programming-language-dependent. The three-component framework, however, permits the differentiation engine to be programming-language-independent.

We are currently developing a three-component AD tool using the xaifbooster differentiation engine. In particular, we are developing a Fortran 90 front/back-end and the necessary bridging software to use xaifbooster.

To ensure robustness of the AD tool, we sought industrial strength programming-language-dependent components. Recently, SGI made available the Open64 language tool set. This tool set was originally intended to be a commercial product, meeting our desires.

To connect the Open64 components to the xaifbooster differentiation engine, however, required bridging software. While the Open64 tool set used an intermediate representation language named 'WHIRL', the xaifbooster engine used a different intermediate representation called 'XAIF'. So, building the system required translation tools, insightfully named whirl2xaif and xaif2whirl.

*Thanks to ACTS and LACSI for funding this work

The remainder of this report describes various technical details of the design and implementation of these bridging tools. We have divided it into five sections, excluding appendices. First, we give some relevant background material on both WHIRL and XAIF. Secondly we describe the overall design principles of whirl2xaif and xaif2whirl. Next we devote two sections to describing technical issues involved in the design and implementation of whirl2xaif and xaif2whirl, respectively. Finally, three appendices provide more detailed information on selected topics.

2 Background on Intermediate Languages

2.1 WHIRL

WHIRL is the intermediate language of the Open64 compiler. It is generated by the various front-ends and is used and transformed by the various back-end components. WHIRL was designed to support such imperative languages as C, C++, Java, Fortran 77 and Fortran 90.

Generally speaking, WHIRL is an abstract syntax tree that targets an abstract C machine. In order to support 1) compiler optimizations that may operate best on higher or lower level code, 2) generation of assembly code and 3) generation of high-level source code, WHIRL is actually classified into five levels, from 'Very High' to 'Very Low'. [1] 'Very High' WHIRL retains many of the properties of C or Fortran source code while 'Very Low' WHIRL resembles assembly code. Because we are ultimately interested in source-to-source transformations, we exclusively use and generate 'Very High' WHIRL.

To get a sense of what WHIRL looks like, we include a sample. Here is a very simple Fortran routine:

```
subroutine head(x, y, z)
  double precision, intent(in) :: x,z
  double precision, intent(inout) :: y

  y=x*y*z

end subroutine
```

This is the corresponding 'Very High' WHIRL (Note that in WHIRL, assignments are stores (STID) and variable references are loads (LDID).):

```
 LOC 1 1 subroutine head(x, y, z)
FUNC_ENTRY <1,20,head_>
 IDNAME 0 <2,1,X>
 IDNAME 0 <2,2,Y>
 IDNAME 0 <2,3,Z>
BODY
```

[1] Not surprisingly, the other levels are 'High', 'Mid', and 'Low'.

```
BLOCK
PRAGMA 0 120 <null-st> 0 (0x0) # PREAMBLE_END
LOC 1 4
LOC 1 5   y=x*y*z
F8STID 0 <2,2,Y> T<11,.predef_F8,8>
 F8MPY
  F8F8LDID 0 <2,3,Z> T<11,.predef_F8,8>
  F8MPY
   F8F8LDID 0 <2,1,X> T<11,.predef_F8,8>
   F8F8LDID 0 <2,2,Y> T<11,.predef_F8,8>
LOC 1 6
LOC 1 7 end subroutine
RETURN
END_BLOCK
```

For more information on WHIRL see the documentation located in

```
Open64/documentation
```

Also see section 8 for a primer on WHIRL symbol table types.

2.2 XAIF

XAIF – XML Abstract Interface Form – provides a language-independent representation of constructs common in imperative languages, such as C and Fortran. The main role of the XAIF is to define a layer of abstraction, so that various automatic differentiation algorithms can be expressed in a language-independent manner.

XAIF programs, making common compiler analyses explicit, are written in terms of call graphs and control flow graphs. Each node in the call graph is a control flow graph; each node in the control flow graph is a basic block. Basic blocks contain statements; variable references within statements refer to entries in the program's scoped symbol table. It should be noted that XAIF constructs such as call graphs and control flow graphs may not *exactly* correspond to their compiler counterparts. For example, XAIF basic blocks include subroutine calls and are not necessary of maximal length.

It is important to realize that XAIF was designed to represent only the *numerical core* of a program: it is self-consciously not the union of common imperative programming languages. For example, it is not possible to represent a Fortran I/O statement in XAIF. This fact has major ramifications for the design of the bridging tools.

Below is the XAIF of the small Fortran example above, with several elisions. [2]

```
<xaif:CallGraph ...>

  <!-- ...scoped symbol table removed... -->
```

[2] Notice that we did not say "the *small* XAIF of the small Fortran example"!

```
    <xaif:ControlFlowGraph vertex_id="1" symbol_id="head" ...>
      <!-- ...argument list removed... -->

       <xaif:Entry vertex_id="1"/>

       <xaif:BasicBlock vertex_id="2" scope_id="2">

         <xaif:Assignment statement_id="ass1">
           <xaif:AssignmentLHS>
             <xaif:SymbolReference ... symbol_id="y" .../>
           </xaif:AssignmentLHS>
           <xaif:AssignmentRHS>
             <xaif:VariableReference vertex_id="1">
               <xaif:SymbolReference ... symbol_id="x" .../>
             </xaif:VariableReference>
             <xaif:VariableReference vertex_id="2">
               <xaif:SymbolReference ... symbol_id="y" .../>
             </xaif:VariableReference>
             <xaif:VariableReference vertex_id="3">
               <xaif:SymbolReference ... symbol_id="z" .../>
             </xaif:VariableReference>
             <xaif:Intrinsic vertex_id="4" name="mul_scal_scal"/>
             <xaif:Intrinsic vertex_id="5" name="mul_scal_scal"/>
             <xaif:ExpressionEdge ... source="1" target="4" position="1"/>
             <xaif:ExpressionEdge ... source="2" target="4" position="2"/>
             <xaif:ExpressionEdge ... source="3" target="5" position="1"/>
             <xaif:ExpressionEdge ... source="4" target="5" position="2"/>
           </xaif:AssignmentRHS>
         </xaif:Assignment>

       </xaif:BasicBlock>

       <xaif:Exit vertex_id="3"/>

       <xaif:ControlFlowEdge edge_id="1" source="1" target="2"/>
       <xaif:ControlFlowEdge edge_id="2" source="2" target="3"/>

    </xaif:ControlFlowGraph>

  </xaif:CallGraph>
```

For more information on XAIF see http://www-unix.mcs.anl.gov/xaif/.

3 Principles behind whirl2xaif and xaif2whirl

We have divided the AD task into several discrete steps:

1. Open64's Fortran 90 front-end (mfef90) produces a WHIRL file
2. whirl2xaif translates the WHIRL into an XAIF file
3. xaifbooster creates a new XAIF file, introducing code to compute derivatives
4. xaif2whirl generates WHIRL from the new XAIF code
5. Open64's whirl2f back-end generates Fortran source code from the new WHIRL file.
6. A Fortran post-processor expands certain special tags (produced by xaif2whirl) in the new Fortran output.

This report is concerned especially with items two and four.

While several technical issues in the WHIRL $\leftrightarrow$ XAIF translation processes relate exclusively to WHIRL or XAIF, we first discuss two more broadly important considerations. The first is that WHIRL expresses more information than can be represented in XAIF. The second is that we use an abstract analysis engine to create things like call graphs and control flow graphs.

3.1 WHIRL, XAIF and Information Loss

Because XAIF was designed to express only the numerical core of imperative programs, WHIRL contains much information that cannot be expressed within XAIF. Some examples include:

- certain control flow constructs such as GOTO, RETURN, SWITCH/CASE
- PRAGMA statements
- IO statements (representing Fortran I/O)
- expressions involving references to structure members
- source code comment tables
- initialization tables; consider the differences between the following Fortran fragments:

```
! '2' is stored in WHIRL AST (as a constant node)
real :: x
x = 2

! '2' is stored in the WHIRL initialization table
real :: x = 2
```

Because XAIF is 'lossy' with respect to WHIRL – and because this lost information is crucial both from a correctness and practical viewpoint – we needed to find some way to preserve the WHIRL information that could not be translated into XAIF. We chose to cope with this problem by not constructing a completely new WHIRL intermediate representation from xaifbooster's XAIF. Instead, because xaifbooster only added or replaced new statements and expressions within basic blocks, we decided that xaif2whirl would *reuse the original* WHIRL file, generating new WHIRL from the new XAIF, all at the block and statement level.

In order for this to work – i.e., in order to make use of the *original* WHIRL file when generating *new* WHIRL from xaifbooster's output – we needed a mechanism for correlating the original WHIRL representation with the XAIF. To illustrate, consider an example where xaif2whirl reads its inputs – a WHIRL file (from whirl2xaif) and the xaifbooster XAIF file – and begins translating a node in the XAIF call graph. [3] xaif2whirl needs to somehow locate the WHIRL function that corresponds to this call graph node (or control flow graph). In other words, as xaif2whirl descends into basic blocks of the control flow graph, it is ready to create new WHIRL statements for the whole block. But in order to know where to place these statements, it needs to know where the statements are located in the original WHIRL function. If this information is known, xaif2whirl can simply delete the statements corresponding to the original block and replace them with new statements. The modified WHIRL file may than be saved to disk and translated to new Fortran source code.

There was still another issue, however. xaifbooster operated at the basic block – not the statement – level, which meant that while the correctness of the original computation would be preserved, it might generate completely new statements for the *entire* basic block. We therefore needed some mechanism for knowing that certain statements that could not be represented in XAIF would still be preserved. Consider the case of a basic block that contains a WHIRL RETURN statement. The RETURN statement cannot be directly represented in XAIF, but removing it from the (original) WHIRL tree is almost certainly incorrect. The same is true for other items in the above list, such as WHIRL IO statements. Thus, we needed some way of knowing that a statement should be preserved even though XAIF could not represent it. Moreover, we needed to represent this in XAIF in order to instruct xaifbooster to preserve the relative ordering between the numerical code and these unknown statements. In other words, it would be incorrect for xaifbooster to reorder a collection of statements by placing something representing a WHIRL RETURN statement before the derivative accumulation code for the basic block.

To resolve both of the problems above we introduced two new XAIF constructs. First, to enable general correlation between WHIRL and XAIF, we added an 'annotation' attribute to XAIF nodes. whirl2xaif could fill these attributes with persistent IDs to various WHIRL constructs such as symbol tables, symbols, functions and nodes in the function's WHIRL tree. Second, in order to represent WHIRL statements such as RETURN and IO, we introduced a new XAIF element, 'xaif:Marker'. This element could represent anything whirl2xaif defined it to mean. Moreover, we changed

[3] We intentionally omit for now discussion of a most necessary step: inserting the new symbols xaifbooster creates in the WHIRL symbol table.

xaifbooster to view xaif:Marker statements as 'partitions' that essentially created mini-basic blocks: xaifbooster would never change the order of statements with respect to any surrounding xaif:Marker statements.

To summarize, creating annotation attributes and xaif:Marker elements gave whirl2xaif the ability to generate persistent IDs for important components of the WHIRL data. xaif2whirl then was able to use these IDs to locate the original WHIRL information, and remove, modify or extend it. (See section 6 for more details on the persistent IDs.) Because xaifbooster only modified code at the basic block level, leaving the call graph and control flow graphs intact, much of the original WHIRL information could be successfully reused.

In this connection, we should make explicit some implications that follow from this design. First, xaif2whirl cannot arbitrarily change or create new control flow. (At present, xaif2whirl cannot change or create *any* control flow, but it should be able to handle a few certain well-defined and simple transformations that may be needed for reverse-mode AD.) Secondly, because XAIF requires tools to define the precise meanings of the annotation attribute and xaif:Marker element, XAIF data files become *tool-dependent*. While xaifbooster can work with any XAIF file, xaif2whirl will *only* work with XAIF data originally generated by whirl2xaif. We can note, though, that this is not simply a limitation of xaif2whirl: because the extra information that cannot be encoded in XAIF must be preserved in some fashion, all front-ends to xaifbooster will need to generate some sort of tool-specific data.

3.2 OpenAnalysis

OpenAnalysis (OA) is an open source analysis engine for imperative/procedural languages such as a compiler's intermediate language. OA defines a generic, abstract interface for representing constructs and information in the language. When an OA user couples the abstract OA interface to a particular language, OA can perform certain inter-procedural, control flow and data-flow analyses. Until recently, OpenAnalysis has largely been a nice concept, based on assumptions that significantly limited its practical use. See the OpenAnalysis README for more information.

Because Open64's analysis code was (and is) not cleanly compartmentalized, it was not possible, for example, to use Open64 to create a control flow graph without also dealing with most of the rest of the analysis code. For this reason, and because we also wanted an abstract analysis engine that could be used by another group working on a C front-end to xaifbooster, we chose to begin devoting the resources necessary to transform OA into something that could be used by full-scale compiler intermediate languages.

We extended some work begun by Jason Eckhardt in order to create an OA abstract interface for WHIRL. With this, we could use OA's control flow graph builder, the most robust of its analysis engines. We later extended OA to create simple call graphs (assuming no function parameters or pointers). Work is presently underway to significantly improve OA's data flow analysis.

Because OA's analysis engines performed compiler-targeted analysis, they did not always exactly conform to what we needed. For example, XAIF control flow graphs had some peculiar characteristics. This required us to have a 'post-processing' pass to

perform the minor transformations necessary for converting the graph into XAIF. See section 7 for more details on this.

4 Design and Implementation of whirl2xaif

whirl2xaif was derived from Open64's whirl2f, though by now it has been significantly modified.

Here is a big-picture sketch of the translation process:

```
Create global persistent ID maps (SymTab, PU)
Create an OA call graph
Create and emit XAIF ScopeHeirarcy for XAIF CallGraph
For each OA call graph vertex that represents a function definition
  Translate the WHIRL function into XAIF
    Create local persistent ID maps (WHIRL nodes)
    Create an OA control flow graph
    For every basic block in the OA control flow graph
      For every statement in the OA basic block
        Emit XAIF for the statement
```

Instead of creating a complete XAIF internal representation (such as a DOM Tree), we decided that whirl2xaif would construct only a minimal set of ancillary data structures. In order to translate WHIRL's PU-forest into XAIF, we created a call graph and control flow graphs using OpenAnalysis. However, these two graphs merely guided the order of translation. Because translation was strictly sequential, it was easy to use stream output. All important translation context information was stored in an 'XlationContext' object which maintained an internal stack of translation contexts and was passed by reference to nearly all routines involved in the translation. The 'XlationContext' both stored data structure pointers (such as persistent id maps) and provided useful contextual information (indicating, e.g., that the current expression was part of an xaif:VariableReference). Generally speaking, the 'XlationContext' was designed to represent the translated XAIF content instead of the source WHIRL content. We imposed the convention that a function would assume that its caller provided all the context it needed within the passed 'XlationContext'. Moreover, a function would assume that the passed output stream was ready to accept whatever output it was assigned to emit. This, of course implied that callers would prepare both the 'XlationContext' and output stream for any callees.

In XAIF, expressions are represented as DAG's. However, instead of creating explicit graph structures to guide the translation – as we did with the control flow graph – we exploited the DAG property and created the graphs implicitly while recursively traversing the WHIRL tree. In order for this to work, we imposed the convention that all functions involved in the translation would emit the XAIF expression graphs as if they were nodes in a top-down expression tree. When this convention was followed, the translation could safely emit the XAIF graph edges for nodes generated earlier in the recursion stack.

5 Design and Implementation of xaif2whirl

Translation from XAIF to WHIRL is harder than the reverse and is conceptually similar to a compiler's front end.

Here is a big-picture sketch of the translation process:

```
Create global persistent ID maps (SymTab, PU)
Enter new symbols from XAIF ScopeHeirarcy into WHIRL symbol tables
For each XAIF call graph vertex
  Translate the XAIF control flow graph into WHIRL
    Create local persistent ID maps (WHIRL nodes)
    For every basic block in the XAIF control flow graph
      Remove corresponding statements in WHIRL tree
      Create new WHIRL statements from XAIF and insert into WHIRL tree
```

As was previously indicated, we decided that xaif2whirl would not create a completely new WHIRL file. Rather, it would reuse the original WHIRL information, replacing only blocks of statements. As was also mentioned above, we decided that it would use persistent the ID information embedded in annotation attributes and xaif:Marker elements to correctly modify the WHIRL symbol table and to remove and replace WHIRL statement. Also, as in whirl2xaif, we developed a version of 'XlationContext' to conveniently communicate contextual information and facilitate the translation.

One of the most important parts of translation was correctly defining and correlating statement intervals between XAIF and WHIRL. Because xaifbooster could potentially replace all statements between xaif:Markers and because xaif:Markers were used to represent only certain statements, we stored a list of persistent statement IDs in the enclosing xaif:BasicBlock element. This list of persistent IDs precisely defined what original WHIRL statements were part of the basic block. When combined with the information within xaif:Markers – which represented statement that should be preserved – we could precisely define the statement interval sets that were to be replaced.

In order to mitigate some of the difficulties of creating WHIRL from a language like XAIF, we allowed xaif2whirl to employ two moderating assumptions. First, the critical standard by which xaif2whirl would be judged was the unparsed source code that the whirl2f back-end generated (excluding back-end bugs!). The significance of this was that while xaif2whirl was required to create correct WHIRL, it did not necessarily need to create efficient or fully complete WHIRL. For example, when creating assignment statements, xaif2whirl could always use the more general (and possibly less efficient) ISTORE node, instead selecting between ISTORE and STID. When creating subroutine or function calls, it could conservatively tag all arguments as pass-by-reference (even if the called function might allow for pass-by-value). Finally, when creating array references, there was no need to specify extent information for each dimension. (If known at compile time, this information would already be in the symbol table, enabling generation of correct declarations by whirl2f.)

The second moderating assumption we employed was that, if convenient, we would allow a Fortran post-processor to expand special tags inserted by xaif2whirl and preserved by whirl2f. In order to compute derivatives, xaifbooster would turn all appro-

priate variables into structures that contained on one hand the original variable and on the other hand the derivative of that variable. This meant that wherever this structure was used, the variable's Fortran type declaration needed to change. Instead of forcing xaif2whirl to change the WHIRL symbol table and also to turn scalar references into structure member references, we decided to simply generate special tags around the references and require that the post-processor both change the types and the references. Because a scheme like this was used in the original Adifor, we knew that it would work. More importantly, we already had a post-processor that needed only minimal changes. In the future, however, we may try to shift the burden of the post-processor's work onto xaif2whirl and make all necessary changes in WHIRL.

6 Appendix A: WHIRL Persistent IDs

xaif2whirl relies on the following WHIRL persistent IDs.

Symbol tables ('ST_TAB') Symbol table IDs are guaranteed to be unique among all program units (the PU-forest). They can be simply and safely generated because the order of PUs remains the same between whirl2xaif and xaif2whirl.

Symbols Symbol IDs simply correspond to the ST_TAB indices embedded within a ST_IDX. These are guaranteed to be unique within the ST_TAB.

Program Units ('PU') Program unit IDs are guaranteed to be unique among all program units (the PU-forest). They can be simply and safely generated because the order of PUs remains the same between whirl2xaif and xaif2whirl.

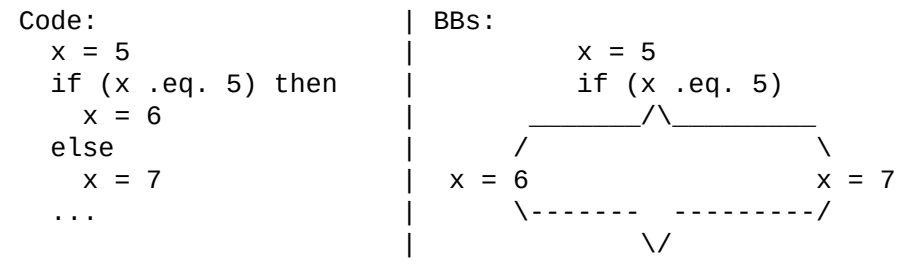
WHIRL nodes ('WN') Created by a depth first search numbering, guaranteed to be unique within the WHIRL PU.

7 Appendix B: OpenAnalysis Post-Processor

At present, the OpenAnalysis post-processor only manipulates the OA's control flow graph.

The OA control flow graph post-processor contains two passes.

First, OpenAnalysis creates basic blocks with labels at the beginning and branches at the end. For example, for 'TWOWAY_CONDITIONAL' statements (an OA classification), OA may generate basic blocks (BBs) such as:

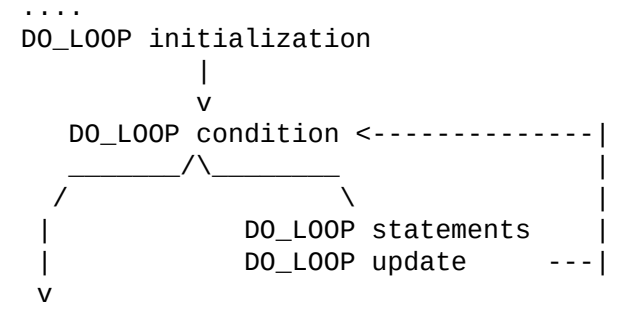


While OA creates correct BBs, in order to create valid XAIF, the first BB must be split so that the if condition can be placed within xaif:If. We create a new BB here so that the translation into XAIF is easy:

```
<xaif:BasicBlock>
  <xaif:Assignment...
  </xaif:Asignment>
</xaif:BasicBlock>

<xaif:If>
  <xaif:Condition...
  </xaif:Condition>
</xaif:If>
```

Second, OA places the initialization and update nodes of WHIRL DO_LOOPs in the appropriate basic block: they are virtually (but not really) spliced out of the DO_LOOPs. For example an DO_LOOP node with initialization, condition, update, and a block of statements may become:



Because XAIF can preserve and exploit high level control structures such as DO_LOOP, we want to *remove* the initialization and update statement so they can be placed in the special 'xaif:ForLoop' construct. More importantly, xaif2whirl also depends on this transformation.

8 Appendix C: WHIRL Symbol Table types primer

WHIRL's symbol table can be difficult to work with. What follows are some important things we have learned about it. As always, see the WHIRL symbol table documentation for more information.

First, it is implemented in C++ (templates and classes) with a C function wrappers for an interface! Most likely (or one hopes!) this has something to do with code reuse, where the symbol table implementation was rewritten but its interface was preserved.

Secondly, WHIRL's scoped symbol table works as advertised for intra-procedural operations. However, once one desires examine symbol tables in an inter-procedural fashion, the full moon rises and the werewolf starts to howl. Or at least it seems that way.

Most significantly in this connection, it is important to realize that a WHIRL PU – program unit, representing a procedure or function – is *not* a self-contained encapsulation. Instead, it is a wrapper for a WHIRL tree and some PU specific symbol tables. However, nodes in the WHIRL tree contain symbol table references that do *not* point directly into these tables, but refer to tables within the global Scope_tab[] (Scope Table), the table of the *current* visible lexical scopes. Consequently, while multiple WHIRL trees and symbol table can reside in memory, the only way to access the symbols for a PU is when it is within the *current* lexical scope. Hence all the symbol table references in other WHIRL trees effectively point to junk. The global 'current' pointers must be updated *each time* one moves to a different PU. Because Open64 did not provide a good way of switching between PUs during inter-procedural algorithms, we developed a tolerable way of doing so.

Thirdly, types in the WHIRL symbol table are sometimes difficult to keep straight. The following is a symbol table types primer (that might nicely supplement the WHIRL symbol table documentation).

SYMTAB Not actually a type, but refers to all of the tables at a particular level/scope. Besides a global scope, there is a local scope for each nested PU. Each scope contains a number of different tables, some of which are common to all levels (e.g. ST_TAB) and some of which are specific to global (e.g. PU_TAB) or local levels (e.g. LABEL_TAB).

SYMTAB_IDX The type of an index into the scope table Scope_tab[]. The global scope is always at the index GLOBAL_SYMTAB; the scope for the current lexical PU is at index CURRENT_SYMTAB. (This is set by the WHIRL reader function Read_Local_Info().)

ST_TAB The type of the symbol table proper, a table that appears at all lexical levels.

ST_IDX A two-part index into any ST_TAB within the Scope_tab[]. The two-part bit field contains an index into the ST_TAB at a certain lexical level.

ST The type of a ST_TAB entry.

Scope_tab[ ] An array of SCOPEs, indexed by SYMTAB_IDX, the lexical level. A SCOPE contains pointers to all the tables for a lexical level, including a ST_TAB.

St_Table[ ] Essentially a class wrapper for Scope_tab[] with member functions for indexing both the Scope_tab[] and the appropriate ST_TAB with a ST_IDX. (TABLE_INDEXED_BY_LEVEL8_AND_INDEX24)