High-Level Information Format for Integrating Front-end and Back-end Compilers (v0.3)

Jenn-Yuan Tsai

September 8, 1997

1 Introduction

In the integration of a source-level optimizing compiler (front end) with an instruction-level optimizing compiler (back end), we need to pass some high-level information from the front end to the back end. For this purpose, we have designed the *High-Level Information (HLI)* file format to transfer the high-level information generated by the front end to the back end. An HLI file for a program includes information about the program that is important for the back-end optimization, but is only available or computable by the front end. The HLI format includes the following information:

- High-level program structure;
- Equivalent memory access information;
- Alias information;
- Loop-carried data dependencies;
- Interprocedural data flow information.

Figure 1 shows the top-level structure of the integrated compiler system. Currently, the front-end compiler is SUIF and the back-end compiler is GCC. The SUIF compiler reads a source program and translates it into the SUIF internal representation. SUIF then performs program transformation and optimization passes on the internal representation. After the program transformations have been completed, the front end outputs the internal representation to a new source file that could be read by the back end. To generate the corresponding HLI file for the new source program, the front end re-reads the new source program into its internal representation to obtain the correct line numbers of the new source program. The front end then performs program analysis passes to generate the high-level information to be passed to the back end and outputs this information to the HLI file. The back end (GCC) parses and translates the transformed source program into its internal RTL representation. An HLI import pass in GCC will read in the corresponding HLI file and map the high-level information to the RTL representation. We create additional data structures inside GCC to hold the high-level information and maintain the mapping between RTL representation and the high-level information. We also provide some utility functions in GCC to retrieve the high-level information related to each RTL instruction and to maintain high-level information across GCC optimization passes.

2 HLI File Format

As shown in Figure 2, the HLI file contains a number of HLI entries. Each HLI entry corresponds to a program unit in the new source file and carries the high-level information regarding the program unit. An HLI entry contains two major tables: a line table and a region table. In this section, We describe the format and contents of the line table and the region table.

2.1 Line Table

The purpose of the line table is to build a connection between the front-end representation and the back-end representation. In the front-end representation, a source line annotation is attached to each expression tree. In the GCC compiler, the parser can insert "source line notes" into the RTL instruction list during RTL generation. If both the front end and the back end read the program from the same source file, we can use the source line numbers to match the expression trees in the front end and the RTL instructions in the back end.

In the high-level information, we are interested only in some special operations such as memory accesses and function calls. We refer to these operations as items in the HLI representation.[1] Since the high-level information available in the front end is represented in terms of expression trees, we need to map the items represented by expression trees in the front end to those represented by RTL instructions in the back end. For this purpose, we use a line table to list interesting items associated with each source line. We also assign each item a unique identification number, which is used by the second part of the HLI entry (region table) to address these items.

Figure 3 shows the format of the line table. The line table consists of a number of line entries where each line entry corresponds to a source line of the program unit in the source file. The first field of a line entry is the line number of the source line. The second field is the line number of the parent line of this line, where the parent line corresponds to the nearest enclosing control statement in the program structure. For program lines immediately inside a loop, the parent line is the loop header. For program lines immediately inside an IF block or an ELSE block, the parent line is the corresponding IF statement or ELSE statement, respectively. Note that the parent line of an ELSE statement line is the corresponding IF statement line.

The third field of a line entry is the line type, which indicates what kind of statement the current line is. The possible values for the line type are listed in Table 1. Using both

[1] An item may also represent an equivalent access class or a whole region, which will be discussed in the next section.

Line Type	Value	Description
PROGRAM_UNIT	0	Entry of a program unit
FOR_LOOP	1	Head of a regular for loop
WHILE_LOOP	2	Head of a while loop
IF_THEN	3	IF statement
ELSE	4	ELSE statement (the parent line is the corresponding IF line)
SWITCH	5	SWITCH statement in C
CASE	6	CASE statement in C (the parent line is the corresponding SWITCH line)
ASSIGNMENT	7	assignment statement
IO	8	I/O statement
CALL	9	Stand-alone function call statement
RETURN	10	Return statement
BREAK	11	BREAK statement in C (the parent line is the corresponding loop header line or SWITCH line)
CONTINUE	12	CONTINUE statement in C (the parent line is the corresponding loop header line)
OTHER	13	Other kinds of statement

Table 1: Possible values for the line type field in the line table.

the parent line number and the line type associated with each program line, the back end can re-construct the high-level program structure and incorporate this information into the source line notes in the RTL representation.

Following the line type is a list of item entries. Each item entry consists of an ID field and a type field. The ID field stores a unique number (within the scope of the program unit) that is to reference the item. The type field stores the type of the item, as listed in Table 2.

With the help of the source line annotations, we can map groups of items from the front end to the back end by matching their source line numbers. However, this map information is not precise enough to map items inside a group (i.e. a source line) from the front end to the back end. To precisely map items from the front end to the back end, the front end needs to know the RTL generation rules of the back end and the order of items associated with each source line. Specifically, the order of item listed in the item table must match the order

Item Type	Value	Description
CALL	0	Function call
CONSTANT_R	1	Load a constant from a constant table (read only)
SCALAR_R	2	Load data from a scalar variable
SCALAR_W	3	Store data to a scalar variable
ARRAY_R	4	Load data from an array element
ARRAY_W	5	Store data to an array element
POINTER_R	6	Load data from an address pointed by a pointer
POINTER_W	7	Store data to a address pointed by a pointer
EQUIVALENT_REF	8	All memory accesses of an equivalent access class
REGION_REF	9	All memory accesses inside a region

Table 2: Possible values for item type.

of the item appearing in the RTL instruction list. We can then precisely map items from the front end to the back end while using the item IDs to communicate high-level information between the front end and the back end.

2.2 Region Table

To simplify the representation of high-level information and in the meantime maintain precise data dependence information with respect to each loop, we represent the high-level information of a program unit with scopes of regions. A region can be a program unit or a loop and can include sub-regions. The basic idea of using region scopes in HLI is to partition all of the memory access items inside a region into equivalent access classes and then describe data dependences and alias relationships among those equivalent access classes with respect to the region.

High-level information for a region includes alias information between memory access items, loop-carried data dependences between memory access items with respect to the region (if it is a loop), and side effects of function calls inside the region. The region table of a HLI entry stores the high-level information for every region in the program unit corresponding to the HLI entry. Figure 4 shows the layout of the region table. Each region entry in the region table has a region header describing the type and scope of the region. In addition to the region header, each region entry holds four sub-tables: (1) an equivalent access table, (2) an alias table, (3) a loop-carried data dependence table, and (4) a function call REF/MOD table. In the following subsections, we describe the representations of the high-level information associated with each region.

2.2.1 Region Header

A region header includes five fields. The first field is the identification number (ID) of the region. Similar to the items defined in the line table, each region has a unique ID number inside a program unit. In the HLI representation, a region ID can also be used as an item ID to refer to all of the memory accesses inside the region. The second field is the ID of the parent region, which immediately encloses the current region. The third and fourth fields of the region header specify the first line number and the last line number of the region, respectively. The last field of the region header stores the region type, which can be either a loop or a program unit.

2.2.2 Equivalent Access Table

A region can have a large number of memory access items inside. To record all of the data dependences and alias relationships between every pair of the memory access items will result in a large amount of data. In fact, a lot of memory access items in a region may access the same memory location because of the multiple references of the same variable in a region. We can thus group memory access items that will access the same memory location into an equivalent access class and use the equivalent class to represent all of the memory access items in its class. By using equivalent access classes to describe the data dependences among memory access items we can reduce the amount of the high-level information.

The equivalent access table of a region partitions all memory access items inside the region (including memory access items enclosed by its sub-region) into equivalent access classes. As shown in Figure 5, the region table consists of a number of equivalent access classes. Each equivalent access class has a unique item ID, which, like the region item ID, can be used as item ID to represent all of the memory access items belonging to the class. The members of an equivalent access class can be either memory access items immediately enclosed by the region (not enclosed by any sub-regions) or equivalent access classes of its immediate sub-regions. Equivalent access classes of immediate sub-regions are used to represent memory access items that are enclosed by sub-regions. In other words, memory access items which are not immediately enclosed by the region will not directly appear in the equivalent access table of the region. Instead, they are represented by the the corresponding equivalent access classes at the sub-regions that enclosed them.

The equivalent classes defined at a region must be exclusive, so that every memory access item inside the region, including those enclosed by its sub-regions, is represented by exactly one equivalent access class at the region. However, a memory access item can be represented by different equivalent access classes defined at regions of different levels.

Figure 6 demonstrates the region structures of a procedure and corresponding equivalent access tables. The outermost region of the procedure is Region 1, which represents the whole procedure. Region 1 has two immediate sub-regions (Region 2 and Region 3), which represent the two i loops in the procedure. The second i loop (Region 3) has an inner j loop, which is represented by Region 4. In the equivalent access table of Region 1, all memory access items in the procedure are partioned into three equivalent access classes: *sum*, *a[0..9]*, and *b[0..9]*. From the point of view of Region 1, every memory access item inside the procedure is represented by exact one of those equivalent access classes. For example, at Region 1 Item 11 (*a[i]*) inside the j loop is represented by the equivalent access class of *a[0..9]*. As mentioned before, equivalent access classes use the IDs of sub-regions' equivalent access classes to refer to memory access items residing in its sub-regions. For example, the equivalent access class of *sum* at Region 1 uses the equivalent access class of *sum* defined at Region 3 to refer to memory access items 13 and 14 enclosed by Region 3.

If a region is a loop, its equivalent access table only describes the equivalent access relationships among memory access items or sub-region equivalent access classes WITHIN a loop iteration. Also, when referred to by the alias table and by the loop-carried data dependence table of the region, an equivalent class only represents the memory locations accessed in one loop iteration. However, when referred to by the outer region, the equivalent class represents all of the memory locations that will be accessed by the whole loop.

In general, the memory access items of the same equivalent access class are considered to be definitely equivalent and always access the same memory location. However, the front-end compiler might want to group memory access items from different equivalent access classes at sub-regions that may access the same memory location into an equivalent access class to reduce the amount of high-level information. In this case, the memory access items of an equivalent access class may not always access the same memory locations. To distinguish this case, every equivalent access class has an equivalent access type field, whose value can be *definitely equivalent* or *maybe equivalent*. Note that the property of maybe equivalent type will propagate along the corresponding equivalent access classes at enclosing regions. That is, if the type of an equivalent access class is *maybe equivalent*, the equivalent access class that refers to it at the enclosing region will also be *maybe equivalent*.

In Figure 6, equivalent access classes *b[j]* and *b[j-1]* at Region 4 represent different memory locations in each loop iteration. However, from the point of view of Region 3 the memory access items in these equivalent access classes may access the same memory locations, so they are grouped into one equivalent access class at Region 3. Since the memory access items in the equivalent access class do not cover exactly the same memory locations, the type of the equivalent access is *maybe equivalent*.

2.2.3 Alias Table

The alias table describes the possible alias relationships among the equivalent access classes of the region. Two or more equivalent access classes are said to be alias if they may access the same memory location at runtime. If two equivalent access classes are alias, all of the memory access items represented by the two equivalent access classes are also alias. The format of alias table is shown in Figure 7. Each alias entry consists of a set of equivalent access classes that are alias with respect to the current region. The equivalent access classes referred to in the alias table must be equivalent access classes defined at the current region. As mentioned before, if the region is a loop, the alias table only describes the alias relationships among the equivalent access classes within a loop iteration. Data dependences caused by equivalent access classes in different loop iteration will be described in the loop-carried data dependence table.

Unlike an equivalent access class, an alias entry does not have item ID because we are not going to use the alias entry to represent equivalent access classes or memory access items in other tables. While the equivalent access table of a region requires every memory access item inside the region to belong to exactly one equivalent access class at the region, the alias table allows a equivalent access class to appear in multiple alias entries at the region. Note that the alias relationship is not transitive. For example, the alias of A and B and the alias of B and C do not imply that A and C are alias.

In Figure 6, equivalent access classes *b[0]* and *b[0..9]* at Region 3 may access the same memory location, so the alias table of Region 3 will include an entry indicating that these two equivalent access classes are alias.

2.2.4 Loop-carried Data Dependence Table

If the region is identified as a loop, the loop-carried data dependence table will list all of the loop carried data dependencies caused by the loop. As shown in Figure 8, loop-carried data dependencies are represented by pairs of equivalent access classes defined at the region. Each pair specifies a data dependence arc caused by the loop. The data dependence type can be definitely dependent or maybe dependent. In addition, each dependence pair includes a distance field. To simplify the representation dependence distance, the direction of a dependence is always normalized to be '>' (from an earlier iteration to a later iteration). If the distance of the dependence is known, the distance value is stored in the distance field. Otherwise, a value of -1 is stored in the field.

For the example shown in Figure 6, the only loop-carried data dependence is between equivalent access classes *b[j]* and *b[j-1]* at Region 4. The distance of the loop-carried data dependence is one.

2.2.5 Function Call REF/MOD Table

The function call REF/MOD table of a region describes the side effects of enclosed function calls made on the equivalent access classes of the region. The format of the sub region REF/MOD table is shown in Figure 9. If a function call is immediately enclosed by the region, the function call REF/MOD table will use the function call item ID defined in the line table to refer the function call and list the equivalent access classes that may be referenced/modified by the called function. For function calls inside a sub-region, the function call REF/MOD table will use the sub-region ID to represent all of the function calls and list the equivalent access classes that may be referenced/modified by the function calls inside the the sub-region. With the function call REF/MOD table, the front end can pass inter-procedure data-flow information to the back end to enable the back end to move instructions around a function call.

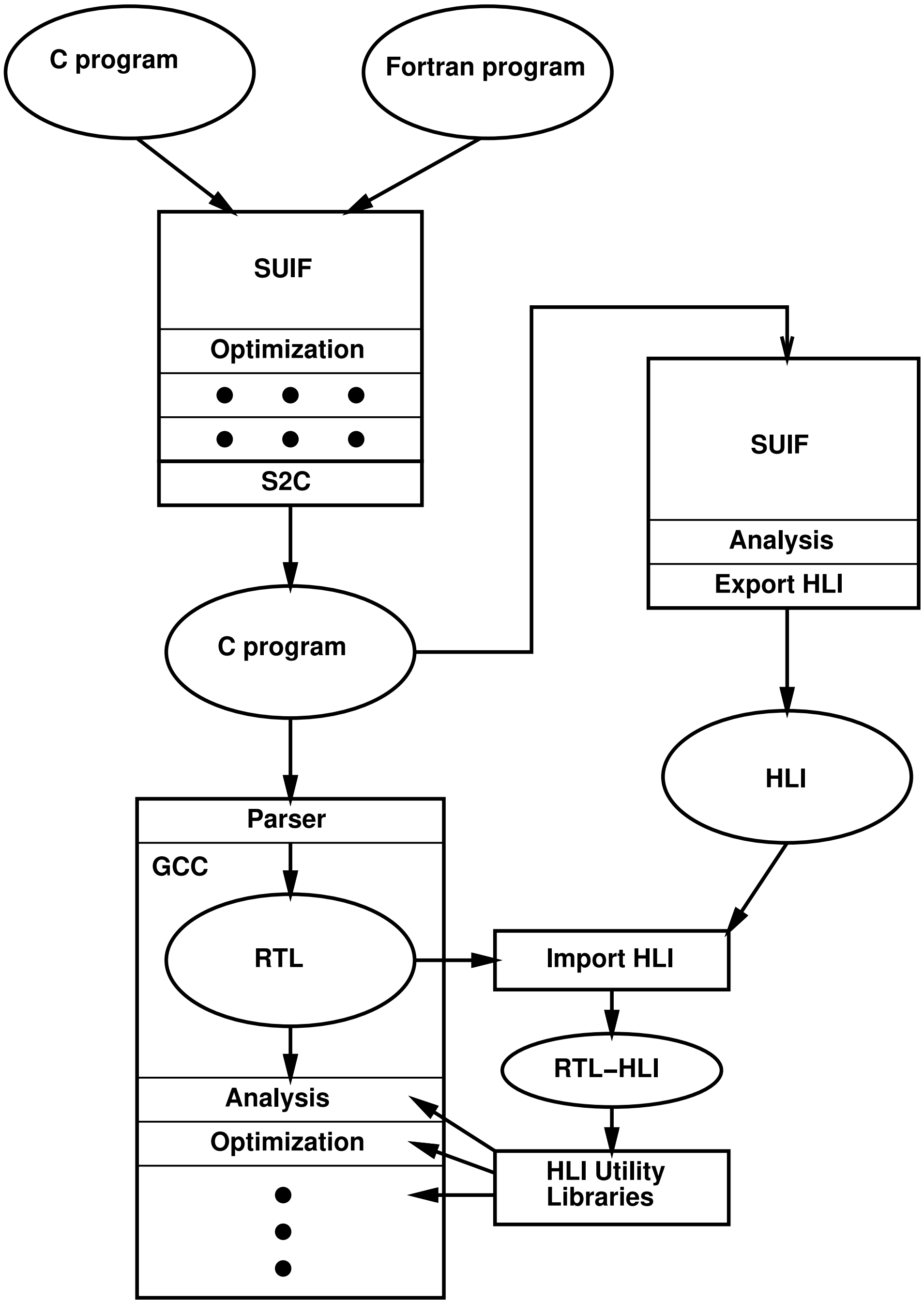

Figure 1: Top-level compiler structure for passing high-level information from the front end (SUIF) to the back end (gcc).

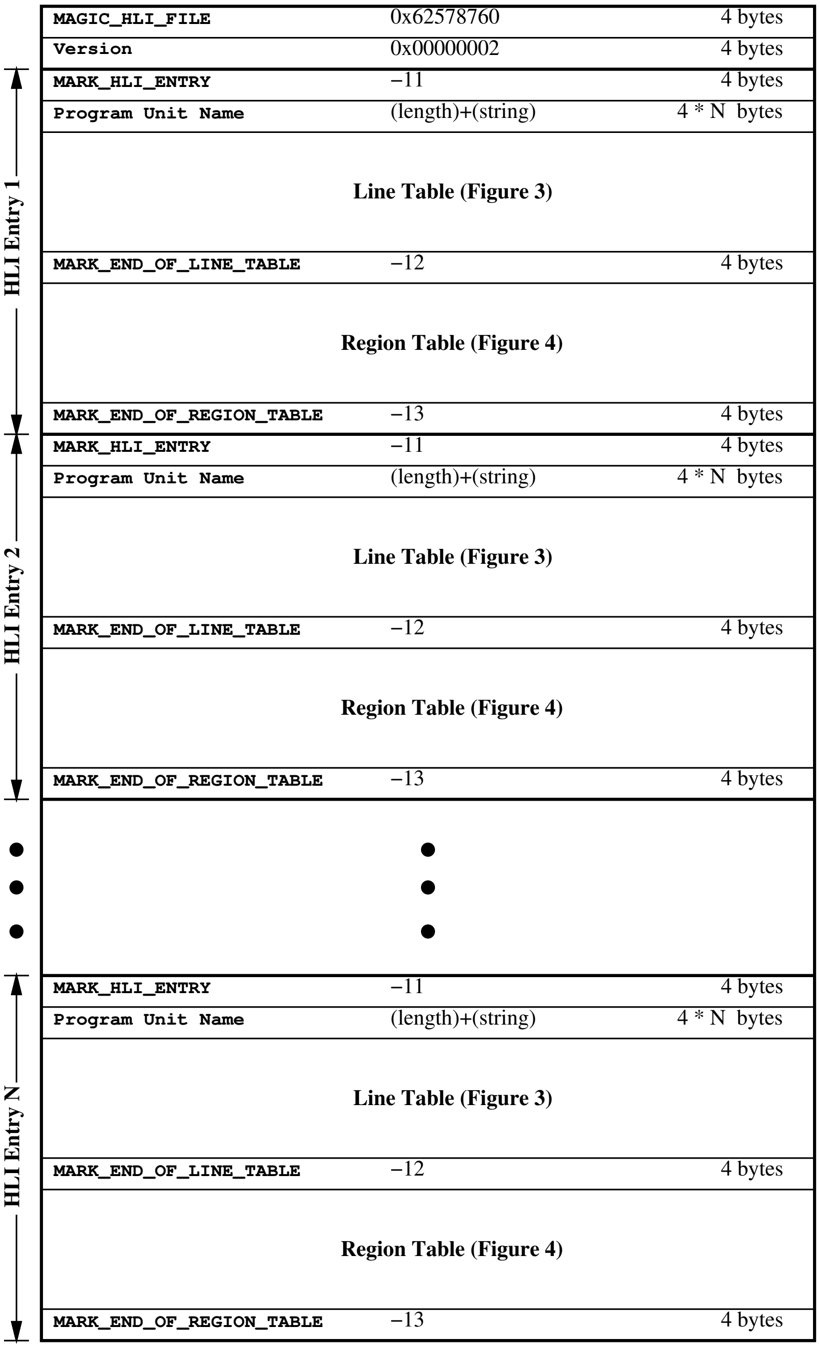

Figure 2: Top level layout of an HLI file.

Line #1	`Line #1`	(integer)	4 bytes
	`Parent Line #`	(integer)	4 bytes
	`Line Type`	(integer)	4 bytes
	`Item ID`	(integer)	4 bytes
	`Item Type`	(integer)	4 bytes
	`Item ID`	(integer)	4 bytes
	`Item Type`	(integer)	4 bytes
		⋮	
	`MARK_END_OF_LINE`	–14	4 bytes
Line #2	`Line #2`	(integer)	4 bytes
	`Parent Line #`	(integer)	4 bytes
	`Line Type`	(integer)	4 bytes
	`Item ID`	(integer)	4 bytes
	`Item Type`	(integer)	4 bytes
	`Item ID`	(integer)	4 bytes
	`Item Type`	(integer)	4 bytes
		⋮	
	`MARK_END_OF_LINE`	–14	4 bytes
⋮		⋮	
Line #N	`Line #N`	(integer)	4 bytes
	`Parent Line #`	(integer)	4 bytes
	`Line Type`	(integer)	4 bytes
	`Item ID`	(integer)	4 bytes
	`Item Type`	(integer)	4 bytes
	`Item ID`	(integer)	4 bytes
	`Item Type`	(integer)	4 bytes
		⋮	
	`MARK_END_OF_LINE`	–14	4 bytes
	`MARK_END_OF_LINE_TABLE`	–12	4 bytes

Figure 3: The format of the line table.

Region 1	`Region ID`	(integer)	4 bytes
	`Parent Region ID`	(integer)	4 bytes
	`Begin Line #`	(integer)	4 bytes
	`End Line #`	(integer)	4 bytes
	`Region Type`	(integer)	4 bytes
	Equivalent Access Table (Figure 5)		
	`MARK_END_OF_EQU_ACC_TABLE`	−15	4 bytes
	Alias Table (Figure 7)		
	`MARK_END_OF_ALIAS_TABLE`	−16	4 bytes
	Loop Carried Data Dependence Table (Figure 8)		
	`MARK_END_OF_LCDD_TABLE`	−17	4 bytes
	Fuction Call REF/MOD Table (Figure 9)		
	`MARK_END_OF_CALL_RM_TABLE`	−18	4 bytes
● ● ●	● ● ●		
Region N	`Region ID`	(integer)	4 bytes
	`Parent Region ID`	(integer)	4 bytes
	`Begin Line #`	(integer)	4 bytes
	`End Line #`	(integer)	4 bytes
	`Region Type`	(integer)	4 bytes
	Equivalent Access Table (Figure 5)		
	`MARK_END_OF_EQU_ACC_TABKE`	−15	4 bytes
	Alias Table (Figure 7)		
	`MARK_END_OF_ALIAS_TABLE`	−16	4 bytes
	Loop Carried Data Dependence Table (Figure 8)		
	`MARK_END_OF_LCDD_TABLE`	−17	4 bytes
	Fuction Call REF/MOD Table (Figure 9)		
	`MARK_END_OF_CALL_RM_TABLE`	−18	4 bytes
	`MARK_END_OF_REGION_TABLE`	−13	4 bytes

Figure 4: The format of the region table.

Equivalent Access Class 1	`Equivalent Access Class ID`	(integer)	4 bytes
	`Equivalent Access Type`	(integer)	4 bytes
	`Item ID*`	(integer)	4 bytes
	`Item ID`	(integer)	4 bytes
		⋮	
	`Item ID`	(integer)	4 bytes
	`MARK_END_OF_EQUIVALENT_ENTRY`	−19	4 bytes
Equivalent Access Class 2	`Equivalent Access Class ID`	(integer)	4 bytes
	`Equivalent Access Type`	(integer)	4 bytes
	`Item ID`	(integer)	4 bytes
	`Item ID`	(integer)	4 bytes
		⋮	
	`Item ID`	(integer)	4 bytes
	`MARK_END_OF_EQUIVALENT_ENTRY`	−19	4 bytes
⋮		⋮	
Equivalent Access Class N	`Equivalent Access Class ID`	(integer)	4 bytes
	`Equivalent Access Type`	(integer)	4 bytes
	`Item ID`	(integer)	4 bytes
	`Item ID`	(integer)	4 bytes
		⋮	
	`Item ID`	(integer)	4 bytes
	`MARK_END_OF_EQUIVALENT_ENTRY`	−19	4 bytes
	`MARK_END_OF_EQ_ACC_TABLE`	−15	4 bytes

* Items must be memory access items immediately enclosed by the current region or equivalent access classes defined at immediate sub-regions.

Figure 5: The format of the equivalent access table.

```
int a[10];
int b[10];
int sum;

foo ()  /* region 1*/
{
    int i, j;

    for (i = 0; i < 10; i++) /* region 2 */
    {
        a[i] = 0;
        {1} /* item 1 */
        b[i] = i;
        {2}
    }

    sum = 0;
     {3}
    for (i = 0; i <10; i++) /* region 3 */
    {
        a[i] = a[i] + b[0];
         {4}    {5}     {6}
        for (j = 1; j  < 10; j++) /* region 4 */
        {
            b[j] = b[j] + b[j−1];
            {7}     {8}      {9}
            a[i] = a[i] + b[j];
            {10}   {11}   {12}
        }
        sum = sum + a[i];
         {13}    {14}    {15}
    }
}
```

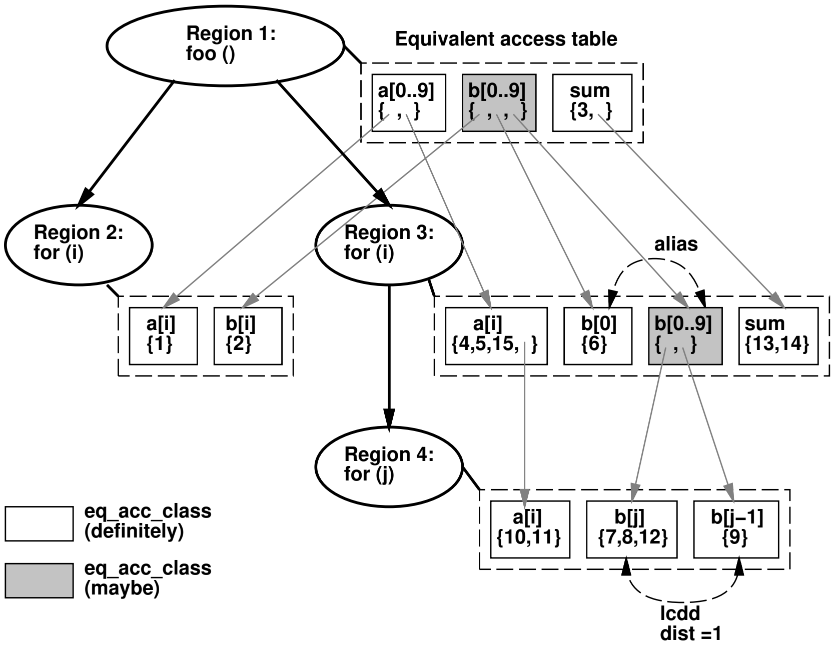

Figure 6: An example program and the structures of its regions and equivalent access classes.

Alias Entry 1	Equivalent Access Class ID*	(integer)	4 bytes
	Equivalent Access Class ID	(integer)	4 bytes
	Equivalent Access Class ID	(integer)	4 bytes
		• • •	
	Equivalent Access Class ID	(integer)	4 bytes
	MARK_END_OF_ALIAS_ENTRY	−20	4 bytes
Alias Entry 2	Equivalent Access Class ID	(integer)	4 bytes
	Equivalent Access Class ID	(integer)	4 bytes
	Equivalent Access Class ID	(integer)	4 bytes
		• • •	
	Equivalent Access Class ID	(integer)	4 bytes
	MARK_END_OF_ALIAS_ENTRY	−20	4 bytes
• • •		• • •	
Alias Entry N	Equivalent Access Class ID	(integer)	4 bytes
	Equivalent Access Class ID	(integer)	4 bytes
	Equivalent Access Class ID	(integer)	4 bytes
		• • •	
	Equivalent Access Class ID	(integer)	4 bytes
	MARK_END_OF_ALIAS_ENTRY	−20	4 bytes
	MARK_END_OF_ALIAS_TABLE	−16	4 bytes

* Items must be equivalent access classes defined at the current region.

Figure 7: The format of alias table.

DD 1	**Equivalent Access Class ID***	(integer)	4 bytes
	Equivalent Access Class ID	(integer)	4 bytes
	Data Dependence Type	(integer)	4 bytes
	Distance	(integer)	4 bytes
DD 2	**Equivalent Access Class ID**	(integer)	4 bytes
	Equivalent Access Class ID	(integer)	4 bytes
	Data Dependence Type	(integer)	4 bytes
	Distance	(integer)	4 bytes
		• • •	
DD N	**Equivalent Access Class ID**	(integer)	4 bytes
	Equivalent Access Class ID	(integer)	4 bytes
	Data Dependence Type	(integer)	4 bytes
	Distance	(integer)	4 bytes
	MARK_END_OF_LCDDS_TABLE	−17	4 bytes

*** Items must be equivalent access classes defined at the current region.**

Figure 8: The format of the loop-carried data dependence table.

Func Call DEF/USE Entry 1	Function Call/Sub-Region ID	(integer)	4 bytes
	REF Equivalent Class ID*	(integer)	4 bytes
	REF Equivalent Class ID	(integer)	4 bytes
		• • •	
	MARK_END_OF_REF	–21	4 bytes
	MOD Equivalent Class ID	(integer)	4 bytes
	MOD Equivalent Class ID	(integer)	4 bytes
		• • •	
	MARK_END_OF_MOD	–22	4 bytes
		• • •	
Func Call DEF/USE Entry N	Function Call/Sub-Region ID	(integer)	4 bytes
	REF Equivalent Class ID	(integer)	4 bytes
	REF Equivalent Class ID	(integer)	4 bytes
		• • •	
	MARK_END_OF_REF	–21	4 bytes
	MOD Equivalent Class ID	(integer)	4 bytes
	MOD Equivalent Class ID	(integer)	4 bytes
		• • •	
	MARK_END_OF_MOD	–22	4 bytes
	MARK_END_OF_CALL_RM_TABLE	–18	4 bytes

* Items must be equivalent access classes defined at the current region.

Figure 9: The format of the function call REF/MOD table.