The HLI Implementor's Guide (v0.1)

(internal document)

Sangyeun Cho and Yonghong Song

Agassiz Project
Dept. of Computer Science
University of Minnesota
Minneapolis, MN 55455, USA
E-mail: {sycho, song}@cs.umn.edu

September 1, 1997

1 Introduction

High-Level Information (or *HLI*) [2] has been proposed to provide a number of important program information that is not available or hard to calculate in back-end compilers. HLI is generated by a front-end compiler (currently SUIF), which extensively analyzes the program at a high or intermediate level, and it is saved in a file to be passed to a back-end compiler (currently GCC).

For the portability purpose, HLI resides in separate data structures from those used in the back-end compiler, and the information is retrieved only by a set of *query functions* specified in [1]. The query functions stand as a common interface to HLI regardless of the back-end compiler used. The implementation of the query functions, however, are in some parts dependent on the internals of the back-end compiler. Another set of functions, called *utility functions*, collect the information needed for some query functions. The relationship between HLI, the information in the back-end compiler, the utility functions, and the query functions is depicted in Figure 1.

This document aims to describe the algorithms, the internal data structures, and utility functions used to implement the query and maintenance functions. Those who want to modify an existing query or maintenance function, or add a new function to the specification will find this document useful.

The rest of the document is organized as follows: in the next section, we begin with the necessary background of HLI and GCC. The utility functions, used by the query functions will then be described, followed by the query functions. The internals and use of maintenance functions are then presented, which will close the document.

2 Background

Query and maintenance functions are intended to be as independent of the underlying back-end compiler as possible, although `HLI_GetLine` is somewhat dependent on the target compiler. The query and maintenance functions are heavily dependent on the generic data structures which are described in a

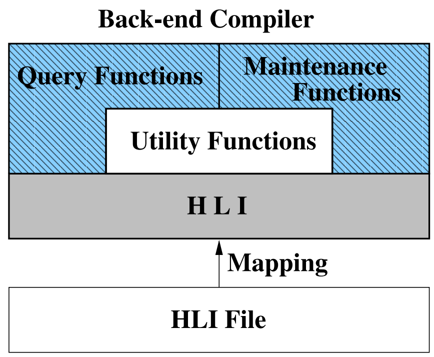

Figure 1: Query functions, utility functions, HLI, and back-end compiler

header file `hli_BE_rep.h`. To form a clear interface between a target compiler and these two kinds of functions three new types are used. One is `MemRefSpec` as the memory reference type, another is `CallRefSpec` as the function call reference type, and the other is `RefSpec`, which is the union of `MemRefSpec` and `CallRefSpec`.

2.1 Data structures for HLI

The generic data structures on which query and maintenance functions rely are in `hli_BE_Rep.h`. Figure 2 shows the relationship between different entities. Please refer to `hli.h` for details.

2.2 Memory access in GCC RTX

The query and maintenance functions are supposed to be blind to the back-end compiler, so the memory access is represented as `MemRefSpec` type and function call access is represented as `CallRefSpec`. Both are equivalent to `rtx` type in gcc 2.7.2. The type `RefSpec` is the combination of these two and acts as the united interface between a back-end compiler and HLI.

In GCC 2.7.2 the memory access rtx has prototype: `(mem:M ADDR)` where `M` specifies how large a unit of memory is accessed and `ADDR` is the address to the main memory. Please refer to `gcc.info.*` for details. To map each memory access in GCC rtx format to one item in the HLI format, a memory access in GCC is represented as a 2-tuple `(insn ID, RefSpec)`; `insn ID` will specify an rtx instruction in the instruction chain, and `RefSpec` will determine one memory access among possibly multiple memory accesses in the instruction. The same identification occurs in case of function call. We can use item type field in `HLI_Item` to separate `MemRefSpec` and `CallRefSpec` in `RefSpec`.

3 Utility Functions

We use a set of utility functions to ease programming of query and maintenance functions. The details about these functions are not given in this document except general algorithm descriptions. Please refer to the file `hli_util.c`.

Note that whenever utility, query and maintenance functions return `HLI_UNKNOWN`, it is user's responsibility to check global error indicator `HLI_Error` to figure out the problem. So these functions set the error flag accordingly when it has to return `HLI_UNKNOWN`.

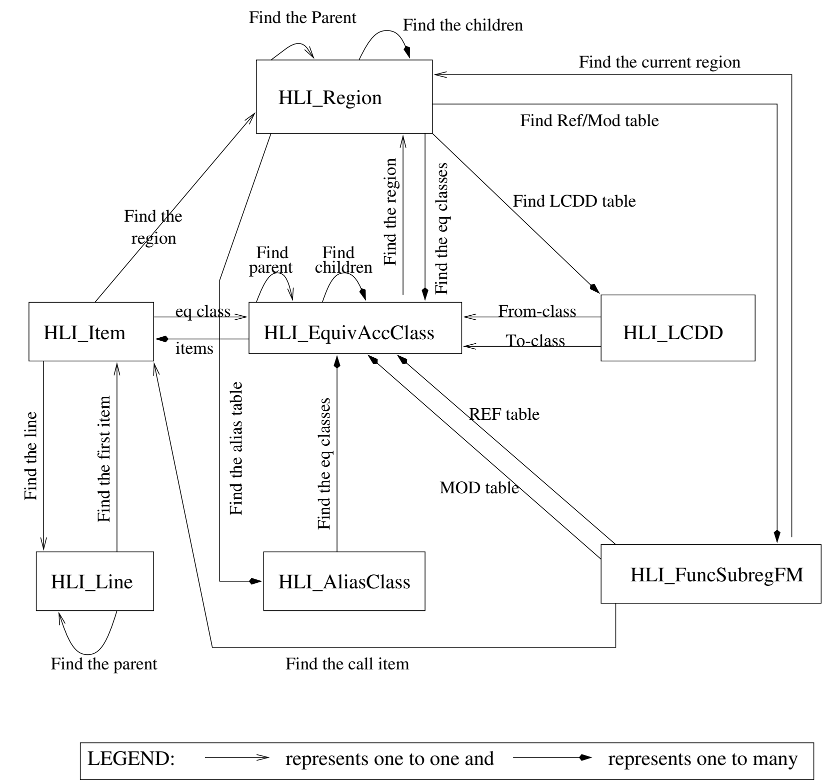

Figure 2: Data structures for HLI

3.1 `HLI_FindClassType`

- Prototype: `HLI_EquivAccType HLI_FindClassType(HLI_EquivAccClass *class1);`
- This function tries to find the equivalent access class type of the given class, say **C**.
- The possible return value:
  - `HLI_EQ_DEFINITE` if **C** is `HLI_EQ_DEFINITE`
  - `HLI_EQ_MAYBE` if **C** is `HLI_EQ_MAYBE`
  - `HLI_UNKNOWN` otherwise.

3.2 `HLI_FindAliasSet`

- Prototype: `HLI_AliasClass *HLI_FindAliasSet(HLI_EquivAccClass *eq,HLI_AliasClass **al,unsigned len);`
- This function tries to find the alias set which contains the given equivalent class in the given alias table. The algorithm is straightforward: search the alias table to find the alias class which contains the given equivalent class. If successful, the corresponding `HLI_AliasClass` pointer is returned, otherwise NULL.

3.3 HLI_FindAliasInfo

- Prototype: `HLI_EquivAccType HLI_FindAliasInfo(HLI_EquivAccClass *c1,HLI_EquivAccClass *c2,HLI_Region *r);`
- This function tries to find the relationship between two equivalent classes in a region. For example, assume that two classes are **C1** and **C2**.

  1. If **C1** is equal to **C2**, `HLI_EQ_DEFINITE` is returned.
  2. If **C1** and **C2** are in the same alias set, `HLI_EQ_MAYBE` is returned.
  3. Otherwise, `HLI_EQ_NONE` is returned.
  4. If any necessary information is lost in previous steps, `HLI_EQ_UNKNOWN` is returned.

3.4 HLI_FindUpperEq

- Prototype: `HLI_EquivAccClass *HLI_FindUpperEq(HLI_Region *r1,HLI_Region *r2,HLI_EquivAccClass *c);`
- This function tries to find the representative equivalent class in the ancestor region for a given equivalent class. The algorithm is straightforward: backward tracking.

3.5 HLI_HierRelation

- Prototype: `HLI_EquivAccType HLI_HierRelation(HLI_Region *r1,HLI_EquivAccClass *c1,HLI_Region *r2,HLI_EquivAccClass *c2);`
- This function tries to find the relationship between one equivalent class and another. The regions to which these two equivalent classes have nesting relation. First we try to find the representative equivalent class of the equivelant class which is a logical descendent and then we can easily find the relationship between two classes in the same region.

3.6 HLI_DisjRelation

- Prototype: `HLI_EquivAccType HLI_DisjRelation(HLI_Region *up,HLI_Region *l1,HLI_EquivAccClass *c1,HLI_Region *l2,HLI_EquivAccClass *c2);`
- This function tries to find the relationship between two equivelant classes which have relationship of disjoint. Actually this function contains the functionality of the hierarchical case. But for efficiency they have been split. The basic algorithm: first find the personal representative in the nearest common region for each given equivalent class and then find the relationship between these two representatives.

3.7 HLI_FindCommRegion

- Prototype: `HLI_Region *HLI_FindCommRegion(HLI_Region *r1,HLI_Region *r2);`
- Find the nearest common region for two given regions. The basic algorithm is to traverse the region hierarchy using attribute parent in `HLI_Region` structure.

3.8 `HLI_FindEqExistInfo`

- Prototype: `boolean HLI_FindEqExistInfo(HLI_EquivAccClass *eq,HLI_EquivAccClass **aggr,unsigned len);`

- We have an equivalent array `AR` and a single equivalent class `CL` as input parameters. If `CL` is in `AR` TRUE is returned, FALSE otherwise.

3.9 `HLI_Alloca` and `HLI_Realloca`

- Prototype: `void *HLI_Alloca(int n);` and `void *HLI_Realloca(void *org,int old,int new,int unit);`

- These two are the memory allocation facilities in hli maintenance package. In the current implementation they call malloc and realloc respectively.

3.10 `HLI_GetNewAlFromReg`

- Prototype: `HLI_AliasClass *HLI_GetNewAlFromReg(HLI_Region *reg) ;`

- Get one new alias entry from the given region.If the original alias list is full, expand the list space.

3.11 `HLI_InsEqInAlias`

- Prototype: `int HLI_InsEqInAlias(HLI_AliasClass *al,HLI_EquivAccClass *eq) ;`

- Insert one equivalent class into one alias entry. If the original equivalent class list for this alias entry is full, expand the list space.

3.12 `HLI_RmEqFromAlias`

- Prototype: `int HLI_RmEqFromAlias(HLI_AliasClass *al,HLI_EquivAccClass *eq);`

- Remove equivalent class from the given alias entry. Scan the equivalent class list for this alias entry and remove the designated class if any.

3.13 `HLI_MoveInt`

- Prototype: `int HLI_MoveInt(HLI_Item *it,HLI_Region *reg,HLI_EquivAccClass *eq);`

- This function moves `it` to the equivalent class `eq` in the region `reg`. `it` can be `ITEM_COMPLETE` and `ITEM_INCOMPLETE`. If `it` is `ITEM_INCOMPLETE` `eq` equals `NULL`. This function is called several times.

3.14 HLI_InsertEqInt

- Prototype: `HLI_EquivAccClass *HLI_InsertEqInt(HLI_Region *reg) ;`
- This function contains one parameter `reg`. It simply create equivalent class in `reg` and `reg`'s parents and also link them with parent-child relation. It returns the equivalent class in `reg`. This is a basis for several high-level jobs.

4 Query Functions

For the prototypes and semantics of the query functions, refer to [1]. This section focuses on the algorithms used to implement the query functions and the dependencies of the implementation on the underlying compiler (GCC) if any.

4.1 HLI_GetEquivAcc

- Prototype: `HLI_EquivAccType HLI_GetEquivAcc(IRInsn i1,RefSpec r1,IRInsn i2,RefSpec r2);`
- This function doesn't depend on the gcc internals.
- Algorithm:
  1. Get two items corresponding to two `(IRInsn, RefSpec)` pairs.
  2. If the related classes of these two items are the same the type of this class is returned.
  3. Otherwise, get the relation of these two equivalent classes.
     - Hierarchy (one is the ancestor of the other)
     - Disjoint (two are in different offsprings of an upper region)
  4. In any case we first find the representative equivalent classes in the the nearest common region. Then we try to find the relation of these two representatives. The Figure 3 is to show the algorithm.

4.2 HLI_GetCallAcc

- Prototype: `HLI_CallAccType HLI_GetCallAcc(IRInsn ir,RefSpec mf,IRInsn call,RefSpec fname);`
- This function doesn't depend on the gcc internals.
- Algorithm:
  1. Get two items corresponding to `(IRInsn, RefSpec)` and `(IRInsn, CallRefSpec).`
  2. Scan the REF/MOD table of the region in which the call item resides and find the `HLI_FuncSubregRM` data corresponding to the call item.
  3. If such `HLI_FuncSubregFM` exists,

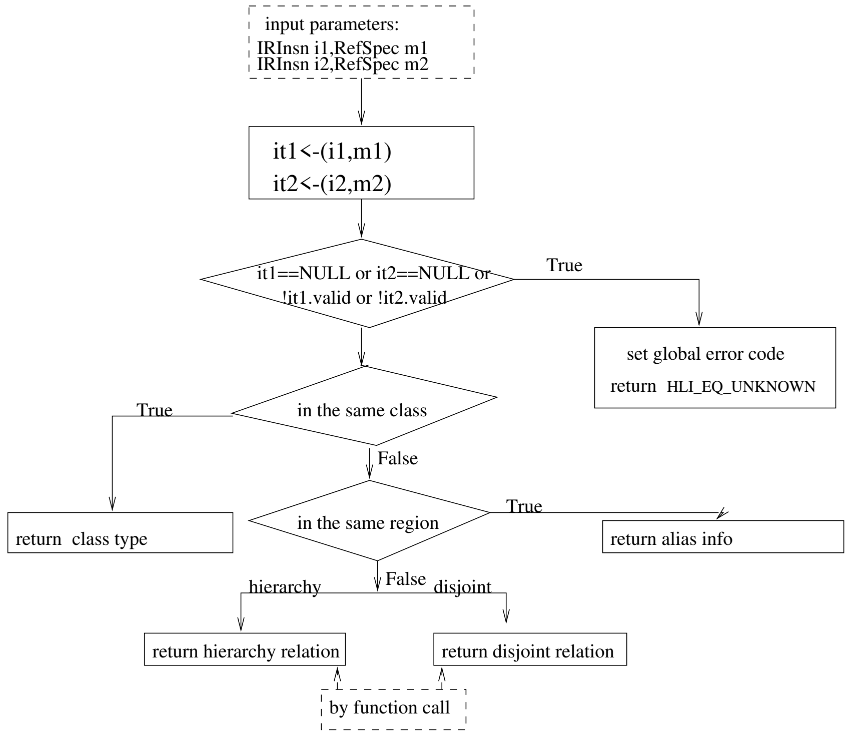

Figure 3: Algorithm for `HLI_GetEquivAcc`

- Return `HLI_CALL_REFMOD` if the given equivelant class is both referenced and modified.
- Return `HLI_CALL_REF` if the given equivalent class is only referenced.
- Return `HLI_CALL_MOD` if the given equivalent class is only modified.
- Return `HLI_CALL_NONE` if the given equivalent class is neither referenced nor modified.

4. If such `HLI_FuncSubregFM` doesn't exist, Return `HLI_CALL_UNKNOWN`.

4.3 `HLI_GetLCDD`

- Prototype: `HLI_LCDDType HLI_GetLCDD (IRInsn from, RefSpec fm, IRInsn to, RefSpec tm, int level, int *distance);`
- This function does not depend on GCC internals.
- Algorithm:
  1. From two `(IRInsn, RefSpec)` pairs, obtain corresponding items.
  2. From two items, obtain corresponding equivalent classes.
  3. Find the common parent region enclosing both the classes.
  4. Traverse up `level` times to reach at a region with desired loop level. If there is no such region, report that there is no loop-carried data dependency.

5. Get each equivalent class in the region which is parent of the original equivalent class.
6. Search the `LCDD` list in the region with obtained equivalent classes as a key for matching. Return the information in the list item matched; if there is no matched information, report that there is no LCDD.

4.4 `HLI_GetLine`

The `HLI_Line` structure, which contains the information on the line type, its parent line, the first item in the line, etc., provides information on the source program structure. `HLI_GetLine` extracts the type of the source program line containing the given instruction.

Although the operation seems simple, the implementation of this function (and the next function for parent line) becomes a little bit involved by the fact that when the HLI is imported into the memory some of the lines are not allocated data structures simply because they are not important. Further, the line structure is accessed by a pointer in an item. Note that an instruction may not have an item in it and there is no way of accessing a line structure from the instruction; that is, there is no mapping between each instruction and its corresponding line structure.

Thus, in implementing this function and the next, we assumed the following constraint: the input instruction must have at least one item in it. To relax the stated constraint to make the function practical, the function also accepts a leading instruction, the first instruction of the line, even though it does not have an item. The HLI importer will create a placeholder item for each such leading instruction and it hooks a line structure.

From a user's point of view, she needs to check whether an instruction of interest has an item in it or not, and provide this query function with an instruction containing an item with the same line number or its leading instruction.

- Prototype: `HLI_LineType HLI_GetLine (IRInsn insn, RefSpec im);`
- Algorithm:
  1. From the given instruction, get an item.
  2. From the item, get the line structure.
  3. Read in the information and return the type.

4.5 `HLI_GetParentLine`

Since each line structure has a pointer to its parent line, the issues and the algorithm are the same as `HLI_GetLine`.

- Prototype: `IRInsn HLI_GetParentLine (IRInsn insn, RefSpec im);`
- Algorithm:
  1. From the given instruction, get an item.
  2. From the item, get the line structure.
  3. Follow a pointer to reach at the parent line structure.
  4. Return needed information.

5 Maintenance Functions

For the prototypes and semantics of the maintenance functions, refer to [1]. This section focuses on the algorithms used to implement the maintenance functions. A return value (5.1 – **??**) equal to or larger than 0 means the job was done correctly, otherwise something went wrong.

For each maintenance function, certain perconditions have to be met; for example, it makes no sense if a function call item is passed as an argument to `HLI_AddLCDD`. Generally after one item is fetched, a test is done to see whether the item is NULL, whether the item has been marked deleted, or whether the item is a function call, etc. In this section, a summary of each function is given, effectively omitting the precondition, since each function has its specific precondition. Please check the implementation in `hli_mt.c` for details.

5.1 `HLI_New`

- Prototype: `int HLI_New(IRInsn i, RefSpec m, HLI_ItemType t);`
- This function creates an item for `(i, m)` with item type `t`.
- Algorithm:
  1. Allocate memory for the new item.
  2. Set the related fields:
     ```
     valid: ITEM_INCOMPLETE
     type : T
     line : NULL
     region: NULL
     class: NULL
     linkTopIR: I
     itemIR: M
     ```
  3. Put the item to the `ItemTable`.

5.2 `HLI_Inherit`

- Prototype: `int HLI_Inherit(IRInsn to,RefSpec tm,IRInsn from,RefSpec fm);`
- This function inherits the item `(to, tm)` from the item `(from, fm)`. These two items are put to the same equivalent class in the same region. The `(to, tm)` must be `ITEM_INCOMPLETE`, but `(from, fm)` must be `ITEM_COMPLETE`.
- Algorithm:
  1. Put these two items to the same equivalent class in the same region, construct the double link between `(to, tm)` and its corresponding equivalent class.
  2. Change the item valid type of `(to, tm)` from `ITEM_INCOMPLETE` to `ITEM_COMPLETE`.
  3. Set the `inheritFrom` item of `(to, fm)` as `(from, fm)`.

5.3 `HLI_Move`

- Prototype: `int HLI_Move(IRInsn ir,RefSpec m,IRInsn ref,RefSpec rm,int level);`
- This function moves the item `(ir, m)` to the region determined by the combination of `(ref, rm)` and `level`. `(ir, m)` can be either `ITEM_INCOMPLETE` or `ITEM_COMPLETE`. `(ref, rm)` must be `ITEM_COMPLETE`.
- Algorithm:
  1. Find the target region to move based on `(ref, rm)` and `level`.
  2. If `(ir, m)` is `ITEM_INCOMPLETE`,
     (a) Insert `(ir, m)` to the target by creating a single equivalent class containing `(ir, m)`, and furthermore update HLI data structure hierarchically.
     (b) Change `(ir, m)` valid type to `ITEM_COMPLETE`.
  3. If `(ir, m)` is `ITEM_COMPLETE`,
     (a) Find the equivalent class representative of `(ir, m)` in the target region.
     (b) Insert `(ir, m)` to the region found above.

5.4 `HLI_Delete`

- Prototype: `int HLI_Delete(IRInsn i,RefSpec m);`
- This function deletes the item `(i, m)` from the HLI data structure.
- Algorithm:
  1. Set the valid field of `(i, m)` as `ITEM_DELETED`.

5.5 `HLI_Undelete`

- Prototype: `int HLI_Undelete(IRInsn i,RefSpec m);`
- This function undeletes (restores) the item `(i, m)` deleted before.
- Algorithm:
  1. Set the valid field of `(i, m)` as `ITEM_COMPLETE`.

5.6 `HLI_MergeEquivAcc`

- Prototype: `int HLI_MergeEquivAcc(IRInsn i1,RefSpec m1,IRInsn i2,RefSpec m2);`
- This function merges the equivalent classes represented by `(i1, m1)` and `(i2, m2)`, respectively. It also updates the HLI data structure hierachically. Both items must be `ITEM_COMPLETE`, and must not be `HLI_IT_CALL`.

- Algorithm:

  1. Find the representatives of `(i1, m2)` and `(i2, m2)` in the latest common region.
  2. For each region from the latest common to the outmost one,
     (a) Merge these two equivalent classes by merging item links and subregion equivalent class lists.Maintain the correct link between the item and equivalent class and between the equivalent class and the class in subregion.
     (b) Update the equivalent class list in the current region.
     (c) Update the alias class list in the current region.
     (d) Update the LCDD list in the current region.
     (e) Update the Subreg/Func Ref/Mod table in the current region.

5.7 `HLI_SplitEquivAcc`

- Prototype: `int HLI_SplitEquivAcc(IRInsn i,RefSpec s);`

- This function splits the item `(i, s)` from the others in the same equivalent class. `(i, s)` must be `ITEM_COMPLETE`.

- Algorithm:

  1. Split the `(i, s)` from the others in the same equivalent class, and do so hierachically. First split `(i, s)` from the others and insert it as a blank one.

5.8 `HLI_AddLCDD`

- Prototype: `int HLI_AddLCDD(IRInsn from,RefSpec fm,IRInsn to,RefSpec tm,int level,int dis,HLI_LCDDType type);`

- This function adds one LCDD entry based on `(from, fm)` and `(to, tm)` to the region determined by `level` with the distance information `dis` and `type`. `(from, fm)` and `(to, tm)` must be `ITEM_COMPLETE` and not `HLI_IT_CALL`. The target region is determined by LEVEL and `(from, fm)`.

- Algorithm:

  1. Find the target region and the representatives of `(from, fm)` and `(to, tm)` in the target region.
  2. Scan the LCDD list in the target region and insert the new entry to the table.

5.9 `HLI_RemoveLCDD`

- Prototype: `int HLI_RemoveLCDD(IRInsn ifrom,RefSpec sfrom,IRInsn ito,RefSpec sto,int level);`

- This function removes the LCDD entry based on `(ifrom, sfrom)` and `(ito, sto)` and `level`. `(ifrom, sfrom)` and `(ito, sto)` must be `ITEM_COMPLETE` and must not be `HLI_IT_CALL`.
- Algorithm:
  1. Find the target the region and the representatives of `(ifrom, sfrom)` and `(ito, sto)` in the target region, respectively.
  2. Remove this entry in the LCDD list of the target region.

5.10 `HLI_AddAlias`

- Prototype: `int HLI_AddAlias(IRInsn i1,RefSpec m1,IRInsn i2,RefSpec m2);`
- This function adds the alias relation between `(i1, m1)` and `(i2, m2)`. `(i1, m1)` and `(i2, m2)` must be `ITEM_COMPLETE` and must not be `HLI_IT_CALL`.
- Algorithm:
  1. Find the latest common region and their representatives in this region, respectively.
  2. For each region from the latest common region to the outmost one,
     (a) If the representatives are in the same equivalent class, set the type of the class as `HLI_EQ_MAYBE`.
     (b) Otherwise, if they are always in the same alias entry, continue to the end of this iteration.
     (c) If they have relation `HLI_EQ_NONE`, create one new alias entry for them.
     (d) If they are in different alias entry, move the second equivalent class related with `(i2, m2)` to the first alias entry.

5.11 `HLI_RemoveAlias`

- Prototype: `int HLI_RemoveAlias(IRInsn i1,RefSpec s1,IRInsn i2,RefSpec s2);`
- This function removes alias entry related with `(i1, s1)` and `(i2, s2)`. `(i1, s1)` and `(i2, s2)` must be `ITEM_COMPLETE` and must not be `HLI_IT_CALL`.
- Algorithm:
  1. Find the latest common region and their representatives in this region, respectively.
  2. For each region from the latest common region to the outmost one.
     (a) If the representatives are in the same equivalent class and with type `HLI_EQ_MAYBE`, split the the first equivalent class related with the `(i1, s1)` from the others in the same equivalent class.
     (b) Check the alias list in this region and remove this entry by setting the representative of `(i1, s1)` as `NULL` if any.

5.12 HLI_AddCallAcc

- Prototype: `int HLI_AddCallAcc(IRInsn ir,RefSpec s,IRInsn icall,RefSpec scall,HLI_CallAccType t);`
- This function adds the item `(ir, s)` to be `t`ed by the function call `(icall, scall)`. Both items must be `ITEM_COMPLETE`. `(ir, s)` must not be a call.
- Algorithm:
  1. Find the function call REF/MOD table for the current region. If not such a table, allocate memory to create one.
  2. Based on `t` add `(ir, s)` to REF table or MOD or both.

5.13 HLI_RemoveCallAcc

- Prototype: `int HLI_RemoveCallAcc(IRInsn ir,RefSpec s,IRInsn icall,RefSpec scall);`
- This function removes the item `(ir, s)` in REF/MOD table of the function call `(icall, scall)`. Both must be `ITEM_COMPLETE` and `(ir, s)` must not be `HLI_IT_CALL`.
- Algorithm:
  1. Check the function call REF/MOD table and based on type `t` remove the item `(ir, s)` for REF ot MOD or both if any.

References

[1] SCHWINN, S. J. "The HLI Interface Specification for Back-end Compilers," *Agassiz Internal Document.* September 1997.

[2] TSAI, J.-Y. "High-Level Information Format for Integrating Front-end and Back-end Compilers (v0.2)," *Agassiz Internal Document*, February 1997.