THE HLI INTERFACE SPECIFICATION FOR BACK-END COMPILERS (v0.1)

Stephen J. Schwinn

Agassiz Project
Dept. of Electrical and Computer Engineering
University of Minnesota
Minneapolis, MN 55455, USA
E-mail: `schwinn@mountains.ee.umn.edu`

September 1, 1997

1 Backend Independent

1.1 HLI Types

Following is a description of the types used in the HLI interface. All memory access and call instructions are represented in the back-end as an IRInsn/RefSpec pair, denoted as [IRInsn,RefSpec].

- `IRInsn`
  (Intermediate Representation Instruction) This type corresponds to a fundamental unit in the back-end's intermediate representation that best represents the typical unit for back-end work.

  Note: In gcc, an IRInsn is an instruction rtx.

- `RefSpec`
  (Reference Specifier) This type is a union of two other types: MemRefSpec and CallRefSpec.

- `MemRefSpec/CallRefSpec`
  (Memory/Call Reference Specifier) An object of this type specifies a particular memory reference or call in a single IRInsn. This can be used to identify a particular memory reference or call in an IRInsn when there can be more than one in an IRInsn.

  Note: In gcc, both a `MemRefSpec` and a `CallRefSpec` is an rtx contained within an instruction rtx (a sub-rtx).

Following are types that are used for return values from query functions.

- `HLI_EquivAccType` and `HLI_LCDDType`
  - `HLI_EQ_NONE, HLI_LCDD_NONE`: No, the memory accesses are not to the same location

- HLI_EQ_MAYBE, HLI_LCDD_MAYBE: Perhaps, HLI analysis has shown that the memory accesses might be to the same location
- HLI_EQ_DEFINITE, HLI_LCDD_DEFINITE: Yes, the memory accesses are to the same location
- HLI_EQ_UNKNOWN, HLI_LCDD_UNKNOWN: The query can't be answered because there is insufficient information available. This can be returned for many different reasons, including errors. Use the HLI_Error variable to determine whether an error occurred.
- HLI_EQ_MULTIPLE, HLI_LCDD_MULTIPLE: (Provided to support higher level query functions.) There are multiple memory accesses from which to choose, so an answer can't be given. In this case, the user is supposed to identify the specific memory reference (instead of an "instruction" which contains multiple items) and call a desired function with [IRInsn, RefSpec] pair to get information or perform an action.

• HLI_CallAccType

- HLI_CALL_NONE: No, the memory location is not accessed in the function.
- HLI_CALL_REF: Yes, the memory location is referenced (ie. read) in the function.
- HLI_CALL_MOD: Yes, the memory location is modified (ie. written) in the function.
- HLI_CALL_REFMOD: Yes, the memory location is both referenced and modified (ie. read and written) in the function.
- HLI_CALL_UNKNOWN: The query can't be answered because there is insufficient information available. This can be returned for many different reasons, including errors. Use the HLI_Error variable to determine whether an error occurred.
- HLI_CALL_MULTIPLE: (Provided to support higher level query functions.) There are multiple memory accesses from which to choose, so an answer can't be given. In this case, the user is supposed to identify the specific function call reference (instead of an "instruction" which contains multiple items) and call a desired function with [IRInsn, RefSpec] pair to get information or perform an action.

• HLI_LineType

- HLI_LN_PROG_UNIT: Entry of a program unit
- HLI_LN_FOR_LOOP: Head of a FOR loop
- HLI_LN_WHILE_LOOP: Head of a WHILE loop
- HLI_LN_IF_THEN: IF statement
- HLI_LN_ELSE: ELSE statement
- HLI_LN_SWITCH: SWITCH statement
- HLI_LN_CASE: CASE statement
- HLI_LN_ASSIGNMENT: Assignment statement
- HLI_LN_IO: I/O statement

- HLI_LN_CALL: Pure function call statement
- HLI_LN_RETURN: Return statement
- HLI_LN_BREAK: Break statement
- HLI_LN_CONTINUE: Continue statement
- HLI_LN_UNDEFINED: All other statements not in above categories

HLI function errors are reported through the use of a single object of type HLI_ErrorType, called HLI_Error, which is set by all HLI interface functions when an error occurs, but is not set when there is not any error. Because of this, many calls to HLI interface functions can be made without checking the error code. If HLI_Error.code is set to HLI_ERR_NO before the sequence of HLI calls, and HLI_Error.code is HLI_ERR_NO after the calls, then they all completed successfully. If not, there was an error, and information about the last error is given in the HLI_Error variable. Information can be found through the code field, and any other possibly included fields. See [2] for additional information.

- HLI_ErrorType This is a structure which contains information about an error that occurs inside an HLI interface function. It has, possibly among others, a 'code' field which is of type HLI_ErrorCode. This field can be used to determine whether an error occurred, and if so, what type of error it was.

- HLI_ErrorCode
  - HLI_ERR_NO: No error
  - HLI_ERR_NO_ITEM: Cannot find an item
  - HLI_ERR_NO_LINE: Cannot find a line
  - HLI_ERR_NO_PARENTLINE: Cannot find a parent line
  - HLI_ERR_INV_ITEM: Invalid item
  - HLI_ERR_EQ_TYPE: Equivalent class type error
  - HLI_ERR_EQ_CHAIN: Equivalent class chain error
  - HLI_ERR_AL_CHAIN: Alias class chain error
  - HLI_ERR_RG_CHAIN: Region table chain error
  - HLI_ERR_NO_CALL_RM: Cannot find call ref/mod table
  - HLI_ERR_NO_SPACE: Out of memory
  - HLI_ERR_NOT_SAME_REG: Items that should be in the same region are not
  - HLI_ERR_ITEM_TYPE: Item type error
  - HLI_ERR_ITEM_DELETED: Item is deleted
  - HLI_ERR_ITEM_INCOMP: Item is not complete
  - HLI_ERR_CALL_NO: No call ref/mod table available
  - HLI_ERR_LEVEL_NEG: An invalid negative value was given for 'level'

- `HLI_ItemType` In the HLI, 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. An item may also represent an equivalent access class or a whole region, as discussed in [1].
  - `HLI_IT_CALL`: Function call
  - `HLI_IT_CONST_R`: Read a constant from memory (constant table)
  - `HLI_IT_SCALAR_R`: Read the value from a scalar (pointer) variable
  - `HLI_IT_SCALAR_W`: Write a value to a scalar (pointer) variable
  - `HLI_IT_ARRAY_R`: Read the value from an array element
  - `HLI_IT_ARRAY_W`: Write a value to an array element
  - `HLI_IT_POINTER_R`: Read from an address pointed by a pointer
  - `HLI_IT_POINTER_W`: Write a value to an address pointed by a pointer
  - `HLI_IT_EQ_ACC_CLASS`: An equivalent access class
  - `HLI_IT_REGION`: A region
  - `HLI_IT_UNDEFINED`: Undefined item type

Throughout HLI the concept of loop level is very important. References are often made to level in the HLI functions. Level is a way to refer to a loop outside (or inside) of the loop where an instruction resides. Positive values are outer loops, and negative are inner loops. A reference to level alone is meaningless, unless it has a reference, so a function that requires a level will also specify what the reference is.

1.2 Query Functions

Query functions are used to answer questions that HLI was designed to answer, in ways meaningful to the compiler back-end.

1.2.1 `HLI_GetEquivAcc`

- Function prototype: `HLI_EquivAccType HLI_GetEquivAcc(IRInsn i1,RefSpec s1,IRInsn i2,RefSpec s2);`
- Answers the question:Do [i1,s1] and [i2,s2] access the same memory location?
- Parameters: i1,s1 ; i2,s2: The two memory access instructions
- Returns: The memory access relationship (`HLI_EquivAccType`) between [i1,s1] and [i2,s2]

1.2.2 `HLI_GetCallAcc`

- Function prototype: `HLI_CallAccType HLI_GetCallAcc(IRInsn i,RefSpec s,IRInsn iCall,RefSpec sCall);`
- Answers the question: Is the memory location accessed by [i,s] also accessed in the function called by [iCall,sCall], and if so, in what way?

- Parameters: i,s: The memory access instruction
  iCall,sCall: The call instruction to the function of interest
- Returns: The access relationship (`HLI_CallAccType`) between [i,s] and all instructions in the [iCall,sCall] function

1.2.3 `HLI_GetLCDD`

- Function prototype: `HLI_LCDDType HLI_GetLCDD(IRInsn iFrom,RefSpec sFrom,IRInsn iTo,RefSpec sTo,int level,int *distance);`
- Answers the question: Do [iFrom,sFrom] and [iTo,sTo] access the same memory location in a loop carried sense, and if so, what is the dependence distance?
- Note: The dependence direction is defined as from [iFrom,sFrom] to [iTo,sTo].
- Parameters: iFrom,sFrom ; iTo,sTo: The two memory access instructions
  level: The loop level to get LCDD information for
  distance: The LCDD distance (This is returned from the function)
- Explanation: A value of zero for level specifies the innermost loop which [iTo,sTo] and [iFrom,sFrom] have in common. Numbers greater than zero specify loop levels above loop level zero. Negative values for level are invalid, and the result will be undefined.
- Returns: The loop carried data dependence (LCDD) access relationship (`HLI_LCDDType`) and the dependence distance between [iFrom,sFrom] and [iTo,sTo].

1.2.4 `HLI_GetLine`

- Function prototype: `HLI_LineType HLI_GetLine(IRInsn i,RefSpec s);`
- Answers the question: From what type of line in the source code did this memory access instruction come from?
- Parameters: i,s: A memory access instruction from the line of interest
- Returns: The line type (`HLI_LineType`) of the line containing [i,s].

1.2.5 `HLI_GetParentLine`

- Function prototype: `IRInsn HLI_GetParentLine(IRInsn i,RefSpec s);`
- Answers the question: What is the parent line of this line?
- Explanation: The parent line of a line L, denoted parent(L), is defined as another line P containing the nearest program structure statement. For example, the parent of all lines inside an ‘`else`’ construct is the ‘`if`’ line. Other program structures have similar relations (i.e. `case` and `switch`, etc.). See [1] for more information.
- Parameters: i,s: A memory access instruction from the line of interest
- Returns: The first instruction in parent([i,s])

1.3 Query Function Examples

Following are examples of how the query functions can be used.

1.3.1 Example 1

- Description: This function, `invalidate_memory` in Figure 1, is taken from GCC's "cse.c". It's main functionality is to invalidate expressions in the hash table which contain any conflicts with the current expressions. The current expression may be a complete memory reference or function call, so we can use equivalent access relation or function call relation respectively.

- Some comments on this implementation:

  1. If the current memory reference is not a function call, any memory reference in hash table which doesn't have reference conflict with it can be preserved.
  2. If the current memory reference is a function call, any memory reference in hash table which is not destroyed by this call can be preserved.

1.4 Maintenance Functions

Maintenance functions are used to modify the HLI anytime the program is changed in the back-end, typically because of optimization, to keep the HLI correct.

Each function can be called independently, and does not require another function to be called to 'complete' the work done in the previous function. Some functions (e.g. `HLI_New`) are usually followed by other function(s) to perform a certain task, but at no point is the HLI invalid or in an intermediate state. It is always valid.

- Note: Unless otherwise indicated, the return value for every maintenance function is a value indicating whether the function executed successfully or not. A negative return value denotes an error, while a non-negative return value indicates that the function completed successfully.

The following five functions: `HLI_New`, `HLI_Inherit`, `HLI_Move`, `HLI_Delete`, and `HLI_Undelete`, provide most, if not all of the functionality that is typically needed to maintain correct HLI in a compiler back-end. Descriptions of more primitive functions are given following these, and are provided for situations when these five are not sufficient (e.g. loop unrolling).

1.4.1 `HLI_New`

- Function prototype: `int HLI_New(IRInsn i,RefSpec s,HLI_ItemType t);`

- Does: This function creates a new, empty HLI entry for the memory access or call instruction [i,s].

- Use notes: This function should be called whenever a new memory access or call instruction is created.

- Parameters: i,s: The memory access or call instruction
  t: The type of [i,s]

```
/* Remove from the hash table all expressions that reference memory,
   or some of them as specified by *WRITES.  */

/* To enhance CSE using HLI,we need to add functionality of
 * mem query and call query.
 */

static void
invalidate_memory (writes)
     struct write_data *writes;
{
  register int i;
  register struct table_elt *p, *next;
  int all = writes->all;
  int nonscalar = writes->nonscalar;
  int fcall = writes->call;
  rtx refinsn = writes->insn;
  rtx refmem = writes->mem;
  for (i = 0; i < NBUCKETS; i++)
    for (p = table[i]; p; p = next)
      {
        next = p->next_same_hash;
        {
          /* IF HLI is not available,the HLI query functions return
           *  HLI_..._UNKNOWN, so dont' worry about it */
          if (p->in_memory
              && (all
                  || (nonscalar && p->in_struct)
                  || cse_rtx_addr_varies_p (p->exp))) {
#if 0
                 remove_from_table (p, i);
#else
            if (fcall) {
               RefSpec          rs1,rs2;
               rs1.memSpec=p->exp,rs2.callSpec=refmem;
               switch(HLI_GetCallAcc(p->insn,rs1,refinsn,rs2)) {
                 case HLI_CALL_MOD:
                 case HLI_CALL_REFMOD:
                  remove_from_table (p, i);
                  break;
                 default:
                  break;
               }
            } else {
               RefSpec          rs1,rs2;
               rs1.memSpec=refmem,rs2.memSpec=p->exp;
               switch(HLI_GetEquivAcc(refinsn,rs1,p->insn,rs2)) {
                 case HLI_EQ_NONE:
                  break;
                 default:
                  remove_from_table (p, i);
                  break;
               }
            }
#endif
          }
        }
      }
}
```

Figure 1: `invalidate_memory`

- Explanation: Please see the description of `HLI_ItemType` for possible values for t.
- Returns: Standard
- Details: This function creates a new HLI entry, and sets up a "link" from [i,s] to the entry so that it can be obtained given [i,s]. Then, all the HLI for this entry is initialized (it is assumed that this new instruction accesses a different memory location than all other instructions in the program), and it is marked as 'incomplete', denoting that the HLI has not yet been given a location. Assigning a location to the HLI can be accomplished by calling either `HLI_Inherit` or `HLI_Move` after calling `HLI_New`.

1.4.2 `HLI_Inherit`

- Function prototype: `int HLI_Inherit(IRInsn iTo,RefSpec sTo,IRInsn iFrom,RefSpec sFrom);`
- Does: This function makes the HLI for [iTo,sTo] identical to that of [iFrom,sFrom] (i.e. a copy).
- Use notes: [iTo,sTo] should access the same memory location(s) or call the same function as [iFrom,sFrom]. ([iTo,sTo] is typically a direct copy of [iFrom,sFrom].) Also, [iTo,sTo] must be a NEW memory access or call instruction, meaning that `HLI_New` must have been called prior to calling `HLI_Inherit`. If this is not the case, an error will be returned.
- Parameters: iTo,sTo: The memory access or call instruction to inherit to
  iFrom,sFrom: The memory access or call instruction to inherit from
- Returns: Standard
- Details: This function sets the HLI for [iTo,sTo] to be the same as the HLI for [iFrom,sFrom]. In addition to copying the HLI, this function also has the effect of giving [iTo,sTo] the same location (in the program) as [iFrom,sFrom], so a call to `HLI_Move` is not necessary to give [iTo,sTo] a location.

1.4.3 `HLI_Move`

- Function prototype: `int HLI_Move(IRInsn i,RefSpec s,IRInsn iRef,RefSpec sRef,int level);`
- Does: This function "moves" the HLI for [i,s] to the location specified by [iRef,sRef] and level.
- Use notes: This function should be called whenever a memory access or call instruction is moved in the program. It should also be used to place a "new" instruction into the program (i.e. after calling `HLI_New`).
- Parameters: i,s: The memory access or call instruction
  iRef,sRef: The reference memory access or call instruction
  level: The loop level relative to [iRef,sRef] to place [i,s]

- Explanation: A value of zero for level specifies the loop which contains [iRef,sRef]. Numbers greater than zero specify loop levels above loop level zero. Negative values for level specify loop levels below loop level zero.
- Returns: Standard
- Details: This function moves the HLI entry for [i,s] to a new location. Please see [2] for more information.

1.4.4 `HLI_Delete`

- Function prototype: `int HLI_Delete(IRInsn i,RefSpec s);`
- Does: This function marks the HLI entry for [i,s] as 'deleted'. Any further attempts to access the HLI for [i,s] will give 'unknown' results and/or 'deleted' errors.
- Use notes: This function should be called whenever a memory access or call instruction is deleted.
- Parameters: i,s: The memory access or call instruction
- Returns: Standard
- Details: This function simply marks the HLI entry for [i,s] as 'deleted'. By not actually deleting the HLI, it can be restored easily. (This is done by calling `HLI_Undelete`).

1.4.5 `HLI_Undelete`

- Function prototype: `int HLI_Undelete(IRInsn i,RefSpec s);`
- Does: This function marks the HLI entry for [i,s] as 'complete'. This makes further use of the HLI for [i,s] possible.
- Use notes: This function should only be called when a previously deleted memory access or call instruction is restored (i.e. undeleted).
- Parameters: i,s: The memory access or call instruction
- Returns: Standard
- Details: This function simply marks the HLI entry for [i,s] as 'complete'.

The following eight functions: `HLI_MergeEquivAcc`, `HLI_SplitEquivAcc`, `HLI_AddLCDD`, `HLI_RemoveLCDD`, `HLI_AddAlias`, `HLI_RemoveAlias`, `HLI_AddCallAcc`, and `HLI_RemoveCallAcc`, provide a mechanism to directly change the instruction dependence relationship information stored in the HLI. For each kind of dependence relationship: EquivAcc, LCDD, Alias, and CallAcc, there is an "add" and "remove" function, which are exact opposites of each other in effect, so that calling an "add" function followed by a "remove" function will leave the HLI exactly the same as before calling either of them. The "add" versions of the functions (including `HLI_MergeEquivAcc`) are used to add dependence information for instructions, while the "remove" versions (including `HLI_SplitEquivAcc`) are used to remove dependence information.

The "add" functions are often used to add dependence information when a new instruction is created (after calling `HLI_New`), and when the instruction is not a copy of another instruction (i.e. when `HLI_Inherit` is not called). The "remove" functions are often called to "fix up" the HLI for an instruction that is similar to, but not exactly the same as, another instruction. When this is done, `HLI_Inherit` is called to get started, and then the "remove" functions are used to change the HLI as necessary.

1.4.6 `HLI_MergeEquivAcc`

- Function prototype: `int HLI_MergeEquivAcc(IRInsn i1,RefSpec s1,IRInsn i2,RefSpec s2);`
- Does: This function "tells the HLI package" that [i1,s1] accesses the same memory location as [i2,s2].
- Parameters: i1,s1 ; i2,s2: The two memory access instructions
- Returns: Standard
- Details: When `HLI_New` is called for an instruction, it is assumed that the instruction accesses a different memory location than all other instructions in the program. This function will add [i1,s1] to the list of instructions that access the same location as [i2,s2]. Even more generally, if there are multiple instructions that access the same location, and those are grouped with [i1,s2], all the instructions in that list will be added to the list that contains [i2,s2] (thus the 'merge' in the function name). See [2] for more information.

1.4.7 `HLI_SplitEquivAcc`

- Function prototype: `int HLI_SplitEquivAcc(IRInsn i,RefSpec s);`
- Does: This function "tells the HLI package" that [i,s] accesses a different memory location than all other instructions in the program.
- Parameters: i,s: The memory access instruction
- Returns: Standard
- Details: This function puts [i,s] into its own separate list of instructions that access the same location. See [2] for more information.

1.4.8 `HLI_AddLCDD`

- Function prototype: `int HLI_AddLCDD(IRInsn iFrom,RefSpec sFrom,IRInsn iTo,RefSpec sTo,int level,int distance,HLI_LCDDType t);`
- Does: This function "tells the HLI package" that there is a loop carried data dependence (LCDD) from [iFrom,sFrom] to [iTo,sTo], with a distance of distance, and a type of t.

- Parameters: iFrom,sFrom ; iTo,sTo: The two memory access instructions
  level: The loop level to set LCDD information for
  distance: The LCDD distance
  t: The LCDD type (see above for possible values)
- Explanation: A value of zero for level specifies the innermost loop which [iTo,sTo] and [iFrom,sFrom] have in common. Numbers greater than zero specify loop levels above loop level zero. Negative values for level are invalid, and the result will be undefined.
- Returns: Standard
- Details: Please see [2] for more information.

1.4.9 `HLI_RemoveLCDD`

- Function prototype: `int HLI_RemoveLCDD(IRInsn iFrom,RefSpec sFrom,IRInsn iTo,RefSpec sTo,int level);`
- Does: This function "tells the HLI package" to remove (i.e. that there is not) a loop carried data dependence (LCDD) from [iFrom,sFrom] to [iTo,sTo]. (This function is the opposite of `HLI_AddLCDD`).
- Parameters: iFrom,sFrom ; iTo,sTo: The two memory access instructions
  level: The loop level to remove LCDD information for
- Explanation: A value of zero for level specifies the innermost loop which [iTo,sTo] and [iFrom,sFrom] have in common. Numbers greater than zero specify loop levels above loop level zero. Negative values for level are invalid, and the result will be undefined.
- Returns: Standard
- Details: Please see [2] for more information.

1.4.10 `HLI_AddAlias`

- Function prototype: `int HLI_AddAlias(IRInsn i1,RefSpec s1,IRInsn i2,RefSpec s2);`
- Does: This function "tells the HLI package" that [i1,s1] and [i2,s2] might access the same location (i.e. are aliases of each other).
- Parameters: i1,s1 ; i2,s2: The two memory access instructions
- Returns: Standard
- Details: Please see [2] for more information.

1.4.11 `HLI_RemoveAlias`

- Function prototype: `int HLI_RemoveAlias(IRInsn i1,RefSpec s1,IRInsn i2,RefSpec s2);`
- Does: This function "tells the HLI package" that [i1,s1] and [i2,s2] no longer might access the same location (i.e. are not aliases of each other).
- Parameters: i1,s1 ; i2,s2: The two memory access instructions
- Returns: Standard
- Details: Please see [2] for more information.

1.4.12 `HLI_AddCallAcc`

- Function prototype: `int HLI_AddCallAcc(IRInsn i,RefSpec s,IRInsn iCall,RefSpec sCall,HLI_CallAccType t);`
- Does: This function "tells the HLI package" that the memory location accessed by [i,s] is accessed in the function called by [iCall,sCall], and in a way specified by t.
- Parameters: i,s: The memory access instruction
  iCall,sCall: The call instruction to the function of interest
  t: The call access type (see above for possible values)
- Returns: Standard
- Details: Please see [2] for more information

1.4.13 `HLI_RemoveCallAcc`

- Function prototype: `int HLI_RemoveCallAcc(IRInsn i,RefSpec s,IRInsn iCall,RefSpec sCall,HLI_CallAccType t);`
- Does: This function "tells the HLI package" that the memory location accessed by [i,s] is no longer accessed in the function called by [iCall,sCall], in the way specified by t.
- Parameters: i,s: The memory access instruction
  iCall,sCall: The call instruction to the function of interest
  t: The call access type (see above for possible values)
- Returns: Standard
- Details: Please see [2] for more information

1.5 Maintenance Function Examples

Following are examples of how the maintenance functions can be used.

```
...
      case INSN:
        copy = emit_insn_before (copy_rtx (PATTERN (insn)), loop_start);
        if (reg_map)
          replace_regs (PATTERN (copy), reg_map, max_reg, 1);

        mark_jump_label (PATTERN (copy), copy, 0);

        /* We need to add COPY to the HLI package */
        {
            HLI_NewInsn(copy);
            HLI_InheritInsn(copy,insn);
            HLI_MoveInsn(copy,loop_start,0);
        }
...
```

Figure 2: Example 1

1.5.1 Example 1

- Description: This code fragment (in Figure 2) is extracted from GCC's "jump.c" (line 2413 or else).

2 BACK-END DEPENDENT

2.1 GCC

These functions are specific to GCC, and aid in using HLI in GCC.

Each of the following five functions take only an IRInsn as a parameter instead of a [IRInsn,RefSpec] pair. This is done to make it easier to work with GCC's instructions. They call the corresponding back-end independent function (see elsewhere) for each memory access (and/or call) instruction in the IRInsn, and return the number of memory access (and/or call) instructions that were found in the IRInsn.

2.1.1 `HLI_NewInsn`

- Function prototype: `int HLI_NewInsn(IRInsn i);`
- Does: Calls `HLI_New` for every memory access instruction in i. (The type for each is set to '`HLI_IT_UNDEFINED`').
- Parameters: i: The instruction
- Returns: Standard
- Details: See `HLI_New` and [2].

2.1.2 `HLI_NewCallInsn`

- Function prototype: `int HLI_NewCallInsn(IRInsn i);`
- Does: Calls `HLI_New` for every call instruction in i. (The type for each is set to '`HLI_IT_CALL`').
- Parameters: i: The instruction
- Returns: Standard
- Details: See `HLI_New` and [2].

2.1.3 `HLI_InheritInsn`

- Function prototype: `int HLI_InheritInsn(IRInsn to,IRInsn from);`
- Does: Calls `HLI_Inherit` for every memory access or call instruction in i.
- Parameters: to: The instruction to inherit to
  from: The instruction to inherit from
- Returns: Standard
- Details: See `HLI_Inherit` and [2].

2.1.4 `HLI_MoveInsn`

- Function prototype: `int HLI_MoveInsn(IRInsn i,IRInsn ref,int level);`
- Does: Calls `HLI_Move` for every memory access or call instruction in i.
- Parameters: i: The memory access or call instruction
  ref: The reference memory access or call instruction
  level: The loop level relative to ref to place i (see `HLI_Move` for more info)
- Returns: Standard
- Details: See `HLI_Move` and [2].

2.1.5 `HLI_DeleteInsn`

- Function prototype: `int HLI_DeleteInsn(IRInsn i);`
- Does: Calls `HLI_Delete` for every memory access or call instruction in i.
- Parameters: i: The instruction
- Returns: Standard
- Details: See `HLI_Delete` and [2].

The following functions are not related to HLI, but are provided as support functions to aid in using HLI in GCC.

2.1.6 `HLI_GetNumMemRefs`

- Function prototype: `int HLI_GetNumMemRefs(IRInsn start,IRInsn end);`
- Does: Counts the number of memory accesses or calls in the instructions starting from start up to but not including end.
- Parameters: start: The instruction to start counting at
  end: The instruction to stop counting at
- Returns: Standard
- Details: See [2].

References

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

[2] CHO, S. AND SONG, Y. "The HLI Implementor's Guide (v0.2)," *Agassiz Internal Document*. May 1997.