Interface Between the Bookkeeper and Code Generators
==========

9/22/82
9/26/82
9/29/82
10/1/82
10/7/82
11/10/82
12/21/82
2/ 5/83
2/12/83
4/10/83
9/10/83
10/02/83

Definitions

Source operation
An individual NADDR operation like (IADD X Y Z).

Machine operation
An individual machine operation like (+3S R1 R2 R3)

Machine instruction
A set of machine operations to be done in one cycle.

Source group
The set of machine instructions generated for a source operation. Empty cycles in between instructions are indicated with no-ops.

Schedule
An ordered list of machine instructions.

USE NADDR operation
An NADDR pseudo-op of the form:

(USE (var1 loc1) (var2 loc2) (var3 loc3) ...)

which says that the given variables are live at that point in the program and that following code needs them in the corresponding locations (a variable may have several locations). A USE should contain all the variables live at the point where it occurs in the program.

DEF NADDR operation
An NADDR pseudo-op of the form:

(DEF (var1 loc1) (var2 loc2) (var3 loc3) ...)

which says that the given variables are live at this point in the program and that previous code has left them in the corresponding locations (a variable may have several locations). A DEF should contain all the variables live at the point where it occurs in the program.

Join cut-point
A join cut-point is a point in a schedule where the bookkeeper wishes to rejoin another trace. A join cut-point at cycle n partitions the machine operations in the schedule into three parts:

A. All the operations whose source groups are completely contained in cycles 1 through n-1.

B. All the operations whose source groups are completely contained in cycles n and later.

C. Operations whose source groups include operations in both cycles n-1 and n.

Join cut-point boundary
The boundary between operations in A and operations in B union C.

Split cut-point
A split cut-point is a point in a schedule where the bookkeeper wishes to place a split. A split cut-point at cycle n partitions the machine operations in the schedule into three parts:

A. All the operations whose source groups are completely contained in cycles 1 through n.

B. All the operations whose source groups are completely contained in cycles n+1 and later.

C. Operations whose source groups include operations in both cycles n and n+1.

Join cut-point boundary
The boundary between operations in A union C and operations in B.

Assumptions

A source conditional jump may compile into a several-machine-operation source group, but the machine operation actually doing the jump is last in the group.

The code generator does not need to worry about merging in machine operations copied as a result of a split or join (we'll punt).

A schedule never need live longer than a single pick trace/generate code/bookkeep pass. The bookkeeper records ALL information it wants to survive.

Moves need not be included in any source group unless they are in between two other operations in the same group, or unless they are storing array elements back in their home, or unless they are the moves generated as a result of an assignment statement.

Arrays and input/output variables have home memory locations represented symbolically by their names.

For now, writes to array elements must be stored in their home after each operation. These moves must be part of the source group causing the write. Reads from array elements should always be done from their home.

No array references will appear in USE or DEF pseudo-ops.

What the Bookkeeper Hands a Code-generator for Each Trace

A list of source operations in source order, and corresponding bookkeeper records. There may be one DEF pseudo-op at the beginning and one USE pseudo-op at the end. Also passed are the list of live variables at the beginning and the end of the trace, and the list of variables live on

the off-trace edge of each conditional jump.

What a Code-generator Returns For a Trace

A schedule. The functions defined on a schedule are discussed below.

Bullshit Schedule Functions

Here is the important information needed by the bookkeeper. Given a schedule, a cut-point, whether the cut is a join or a split, the source-order number of the source operation causing the join/split, we need:

1. Source operations that have moved below/above the join/spit.
2. The live variable definitions (DEF) at a split cut-point boundary; the variable uses (USE) at a join cut-point boundary.
3. The upper or lower half (relative to the cut-point) of those source groups which were partitioned by the cut-point.

Lisp Implementation of the Interface

Guiding principles

All functions are to be true Lisp functions, not macros or record accessors.

Except for a schedule, the code generator should not return any objects represented as vectors or records. Everything should be lists or symbols. This will guarrantee that no "stray" pointers will be accidentally stored away, preventing objects from being gc'ed. It will also help us understand exactly what information crosses the interface boundaries.

Machine operation

A list in "assmebly language" form, e.g. (+38 R5 R4 R7)

Lisp Inteface Functions

(GENERATE-CODE LIVE-BEFORE SOURCE-RECORD-LIST LIVE-AFTER)

Generates code for a trace, returning a schedule. SOURCE-RECORD-LIST is a list of tuples of the form:

(SOURCE-OPERATION TRACE-DIRECTION BOOKKEEPER-RECORD OFF-LIVE)

SOURCE-OPERATION is an NADDR operation and BOOKKEEPER-RECORD is a record that should always be associated with SOURCE-OPERATION. The code generator never looks inside a BOOKKEEPER-RECORD. TRACE-DIRECTION is meaningul only if SOURCE-OPERATION is a conditional jump; it tells which way the jump is going on the trace (LEFT or RIGHT).

If the operation is a conditional jump, then OFF-LIVE is the list of live variables on the off-trace edge.

LIVE-BEFORE and LIVE-AFTER are lists of the variables that are live on entrance to and exit from the trace.

There may be one DEF at the beginning of a trace; if it is present it contains the locations of all variables live at the beginning. There may be one USE at the end of a trace; if it is present, it contains the locations of all variables live at the end.

(SCHEDULE:LENGTH SCHEDULE)

Returns the length of a schedule.

(SCHEDULE:[] SCHEDULE I)

Returns the machine operations scheduled at cycle I as a list of pairs of the form:

(MACHINE-OPERATION BOOKKEEPER-RECORD)

The BOOKKEEPER-RECORD is the one corresponding to the source operation for which MACHINE-OPERATION was generated. It may be () for move operations which don't have any associated source.

The schedule indices I are 1-based.

(SCHEDULE:JOIN SCHEDULE I)

Returns a list of the form:

(USE-OP PARTIAL-SCHEDULE)

where USE-OP is a USE pseudo-op describing variable uses at the join cut-point boundary at cycle I, and PARTIAL-SCHEDULE is a schedule containing operations in the upper half of source groups spanning the cut-point. "Upper half" means cycle I-1 or earlier.

The USE should contain a location for every variable live on the off-trace edge of the join. The code generator is responsible for keeping track of which variables are live at each point in the schedule, using the trace and the live information handed to it by GENERATE-CODE.

The schedule indices I are 1-based. If I is one more then the length of the schedule, then a USE giving the variable locations for the end of the trace should be returned.

(SCHEDULE:SPLIT SCHEDULE I JUMP-NUMBER)

Returns a list of the form:

(DEF-OP PARTIAL-SCHEDULE)

where DEF-OP is a DEF pseudo-op describing variable definitions at the split cut-point boundary at cycle I, and PARTIAL-SCHEDULE is a schedule containing operations in the lower half of source groups spanning the cut-point. "Lower half" means cycle I+1 or later.

The DEF should contain a location for every variable live on the off-trace edge of the split. The code generator is responsible for keeping track of which variables are live at each point in the schedule, using the trace and the live information handed to it by GENERATE-CODE. The live-off list for each split handed in by

GENERATE-CODE is not sufficient; e.g. if a use of a variable moves below a split, it is now live on the off-trace edge of the split.

JUMP-NUMBER (1-based) identifies a particular jump within the cycle. The DEF should contain a location for every variable that is live on the off-trace edge of the jump.

The schedule indices I are 1-based. If I is one more than the length of the schedule, then a DEF giving the locations of all the variables at the end of the trace should be returned (JUMP-NUMBER is ignored).

Some More Details of USE and DEF

The trace scheduler guarrantees that a DEF has exactly one predecessor, which is already compacted, and one successor, which is uncompacted. A USE has exactly one successor, which is compacted, and possibly many predecessors, all of which are uncompacted.

If the code generator is handed a DEF at the beginning of the trace, the USE ejected at the beginning of trace must contain the exact locations specified in the DEF -- no fewer and no less. If a variable location is bound by a DEF at the beginning of the trace, the code generator cannot eject a USE at the beginning requesting that variable in some other location.

If there is no DEF at the beginning of a trace, the code generator can eject a USE that puts the variables in any locations it wants.

What if a variable is live on entrance to the trace and on exit, but the variable is not referenced anywhere on the trace itself? The code generator will have to make up a location for it. For example:

```
A is live on entrance
op1
op2
op3
A is live on exit
```

The codegenerator would make up a location for A to "hold it during the trace", and then eject (USE (A loc)) at the beginning and (DEF (A loc)) at the end. The codegenerator can be smart and notice that if there is a (USE (A loc)) handed to it for the end of the trace, it can probably use loc to hold A during the trace.

Disambiguator Interface for the Codegenerators
==========

10/ 6/82 revised jre
11/10/82 revised jre
3/ 4/83 revised jre

The trace picker picks out individual traces from the NADDR program and hands the traces one at a time to the code-generator. The code-generator treats the trace almost like a basic block, building a data-precedence DAG from it, with nodes representing source operations and edges the values produced and consumed by the operations. In order to build the DAG, the code-generator must know which operands refer, or might refer, to the same locations. For example,

```
A[ I ] := X + Y
...
Z := Z + A[ J ]
```

A[I] and A[J] may or may not refer to the same memory location.

The DAG-builder of the code-generator presents the source operations of a trace one-by-one to the disambiguator, and the disambiguator replies with which operations are data predecessors and the reason why they are predecessors.

The DAG builder wants conservative answers; everytime it thinks two operands reference different locations, it should be 100% sure. If it isn't sure, then it assumes two operands may reference the same location.

Here is the interface to the disambiguator:

(START-TRACE)
    This signifies to the disambiguator that the code-generator is about to start building a DAG from the next trace. The individual operations of the trace are presented via the function below.

(PREDECESSORS SOURCE-OPERATION TRACE-DIRECTION PTR)
    Presents the next source operation from the trace to the disambiguator. PTR is meaningful only to the code-generator and is what is returned to later signify that this operation is a predecessor. TRACE-DIRECTION is meaningful only if SOURCE-OPERATION is a conditional jump, and tells which way the jump will go on the trace.

    Returns the list of all previous operations on the trace that might be data predecessors of this operation and why they are data predecessors.

    The result is a list of sublists, each sublist of the form:

        (PRED REASON SOURCE-OPERAND SOURCE-TYPE PRED-OPERAND PRED-TYPE)

    PRED is a code-generator pointer that describes some source operation that SOURCE-OPERATION data-depends on.

    REASON is one of 'CONDITIONAL-CONFLICT, 'OPERAND-CONFLICT, or 'POSSIBLE-OPERAND-CONFLICT:

        CONDITIONAL-CONFLICT is returned for cases where PRED is a conditional jump above SOURCE-OPERATION (in source order) and SOURCE-OPERATION might write a location that is live at the top of the other leg of the jump.

        OPERAND-CONFLICT is returned when it is known that an operand

        of SOURCE-OPERATION is exactly the same as an operand of PRED, and that the use of the operands conflict (read after write, write after read, write after write).

        POSSIBLE-OPERAND-CONFLICT is returned when it is known only that an operand of SOURCE-OPERATON might be the same as an operand of PRED, and that the use of the operands might then conflict (read after write, write after read, write after write).

    PRED-OPERAND and SOURCE-OPERAND are numbers that identify the conflicting operands of the corresponding source operations (first operand, second operand, etc.). Operands are numbered from left to right starting at 1. In the case of CONDITIONAL-CONFLICT, PRED-OPERAND will be ().

    SOURCE-TYPE and PRED-TYPE are either 'READ, 'WRITTEN, or 'CONDITIONAL-READ, and specify whether that operand was read, written, or read on the off-trace edge of a conditional jump.

    The result may contain several references to PRED.

To keep the interface as simple as possible, the NADDR source program is represented as before, a list of NADDR source operations:

```
( (IADD X Y Z)
  (ISUB F G H)
  ...)
```

Each NADDR source operation is an "atomic object"-- that is, EQ tests will be used to determine equality of two source operations. For example, the two operations constructed below are NOT the same:

```
(LIST 'IADD 'X 'Y 'Z)
(LIST 'IADD 'X 'Y 'Z)
```

This lets us represent source operations just as themselves, i.e. as Lisp "pointers". The disambiguator may need some kind of hash table to map these "pointers" or list structures onto other information.

The disambiguator, because it is dealing with NADDR programs with at least 1000 operations, must use fairly efficient algorithms. For example, it would not be sufficient to use linear search to map the source operations onto internal data structures.

The following functions are in this interface because they are related to picking traces and bookkeeping. The bookkeeper calls these functions, not the code generators, and the functions can be called at any time after the disambiguator has analyzed the program.

(OPER:LIVE-IN OPER)
    Returns the list of scalar variables live on entrance to operation OPER.

(OPER:LIVE-OUT OPER)
    Returns the list of scalar variables live on exit from operation OPER.

(OPER:LIVE-OUT-ON-EDGE OPER DIRECTION)
    Returns the list of scalar variables live on entrance to one of OPER's successors. DIRECTION is LEFT or RIGHT, selecting either the left or right successor of OPER.