LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

Id Compiler Back End for ETS and Monsoon

Computation Structures Group Memo 310
June 27, 1990

James Hicks

This report describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory for Computer Science is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-89-J-1988.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Id Compiler Back End for ETS and Monsoon

James Hicks

June 27, 1990

Contents

Chapter 1

Compilation of Id Programs

Compilation of Id programs consists of a series of steps — the compiler starts by parsing a text file, manipulating the parse tree, converting the parse tree to a dataflow program graph[4], optimizing the program graph, converting the program graph to an Explicit Token Store (ETS)[2] machine graph, optimizing the machine graph, instantiating Monsoon machine graph schemas, and finally assembling the code into Monsoon Object Code (MOC)[5] following the conventions of the Id Object Code Format (IOCF)[5].

The compiler uses a number of representations, because each representation is useful for a portion of compilation. The parse tree representation is necessary because type checking must be performed on the program as it is written in the source — once the parse tree has been transformed into a program graph, information necessary for type checking has been lost. Optimization of the program is most effective when operating on the program graph. Program graphs are abstract enough to hide details that would hinder optimization, yet a program graph is still an executable representation of the program. Eventually, the program graph must be translated to a form executable by Monsoon, the ETS machine graph.

The ETS machine graph is the code executable by the ETS abstract machine. The ETS machine is an abstraction of Monsoon — it has explicitly-addressed token store and temporary registers available to sequential threads as in Monsoon, however, ETS does not have the restrictions imposed on Monsoon by its implementation. These constraints may be experimented with in the ETS interpreter, for instance, by increasing the number of temporary registers per sequential thread, or by removing the successor constraint imposed on certain instruction formats in Monsoon.

The representation of the ETS graph in the compiler is built on the dataflow graph representation of the Dataflow Compiler Substrate[3]. We have added additional slots to instructions to record assembler constraints, frame offsets, instruction offsets and the use of temporaries.

Program Graphs are converted into ETS graphs by transliteration. Each instruction in the program graph is replaced by a small graph of ETS instructions. In this document I will give the *schemas* that show what ETS code replaces a program graph instruction. Most program graph instructions are very simple, and are replaced by one or two ETS instructions. However, there is a class of program graph instructions called encapsulators, which are replaced by rather complex schemas. Encapsulators in the program graph are used to hide the details of implementing loops, conditionals, procedure definitions, *etc.*

Chapter 2 is devoted to the description of the schemas used to translate programs graphs to ETS machine graphs. Chapter 3 describes the peephole optimizations performed on ETS graphs. Chapter 4 gives further translations from abstract ETS graphs to Monsoon graphs for the subset of ETS that cannot be directly executed by Monsoon (such as `i-fetch` and `i-store`). Chapter 5

describes the process for enforcing architectural constraints in the ETS graph — this is the last stage necessary to instantiate an ETS graph for a given ETS implementation. Chapter 6 describes the motivation for using temporary registers and a (preliminary) description of how to compile to threads from ETS dataflow graphs. Finally, Chapters 7 and 8 describe how frame slots and instruction offsets are allocated to instructions.

Chapter 2

Program Graph Schemas

2.1 Conditional Schemas

2.1.1 If Schema

The `if` schema is detailed in Figure 2.1. Note that each input to the `if` encapsulator must go through a `switch` instruction. The switch instruction is not a well-behaved dataflow instruction, because it only produces a token on one of its outputs when it fires. A well-behaved dataflow operator or graph must produce a single token on each output when a token has been placed on all inputs. The `switch` instruction passes the input value to the `T` output if the predicate is `true` and to its `F` output if the predicate is false. The outputs from each branch of the `if` are merged together by `merge` instructions. The merge operators alone is non-deterministic, and not well-behaved, however, the merges in a conditional schema are deterministic because only one branch of the conditional can execute — so there will never be two tokens with the same tag arriving at any `merge` instruction. The conditional schema as a whole is well behaved, because a single token will be produced on all output arcs if a token is placed on all input arcs.

2.1.2 Case Schema

The `case` schema is shown in Figure 2.2. It is a generalization of the `if` schema, except instead of a two-way branch, there is an n-way branch. Again, each of the `case` inputs must go through the switches, and the outputs of each branch are merged together again to produce the `case` outputs. The operator `SWR` means *switch relative*. The destination of the token produced by a `SWR` instruction at location IP_0 will be $(IP_0 + s).Left$, where s is the switch value: an integer in the range 0 to $(k_{branches} - 1)$. The range of the switch value s is enforced by the compiler, so no bounds checking is needed in the case `schema`.

2.2 Loop Schemas

2.2.1 Sequential Loop

The Id Compiler supports several types of loops: those with no inter-iteration parallelism, bounded inter-iteration parallelism, and unbounded inter-iteration parallelism.

Figure 2.3 shows the schema for a sequential loop. Sequential loops have no inter-iteration parallelism. They are useful when it is necessary to sequentialize the execution of the loop iterations

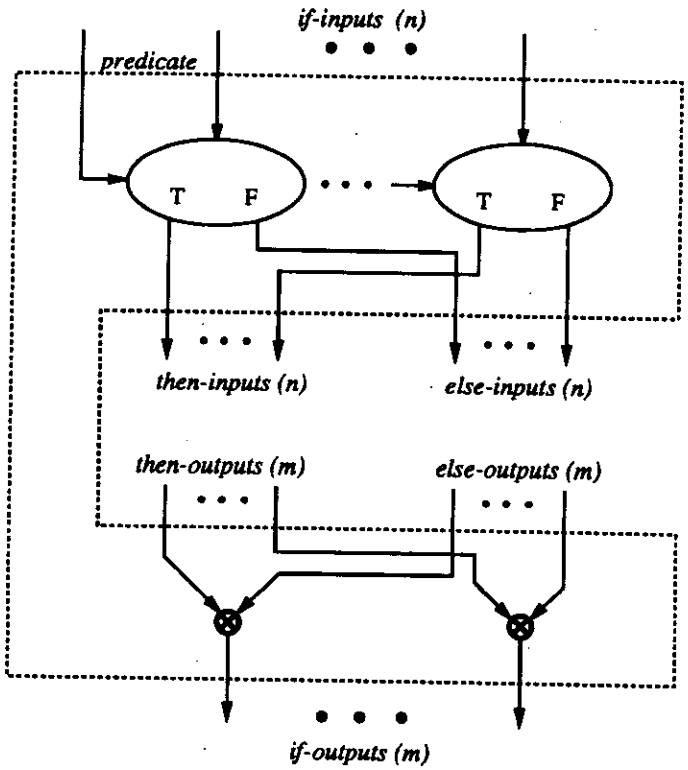


Figure 2.1: `if` schema

because of side-effects being performed in the iterations. They may be useful when data dependencies prevent any useful parallelism between iterations and when the overhead of a bounded loop with a loop bound of 2 is excessive.

The code for storing and clearing loop constants has not been shown, because it consists of `local-fetch`'s and `local-extracts`.

2.2.2 Bounded Loop

The bounded loop schema allows inter-iteration parallelism, but it bounds the resources used by the loop. The bounded loop schema allows only k iterations to be active at any time. This means that you can determine the maximum resources that will be used by a program that consists only of k-bounded loops and non-recursive procedures, and by adjusting the loop bounds on each of the loops in the program you can trade maximum resources consumed for parallelism.

The bounded loop operates by setting up a ring of contexts, and then invoking the `loop-iteration-def` in that ring of contexts. This amortizes the cost of allocating the contexts over all of the iterations of the loop.

The setup and cleanup of the loop iteration contexts is performed by the `loop-parent` instruction. This will be extended to be an encapsulator so that you can have frame-dependent loop constants (*e.g.*, to allocate structures that are reused within the loop, one per active iteration). The loop parent and the loop iterations must be in separate code blocks, because they will operate in separate frames. Figure 2.4 shows the program graph structure of the program graph once a loop has been split into separate parent and iteration instructions.

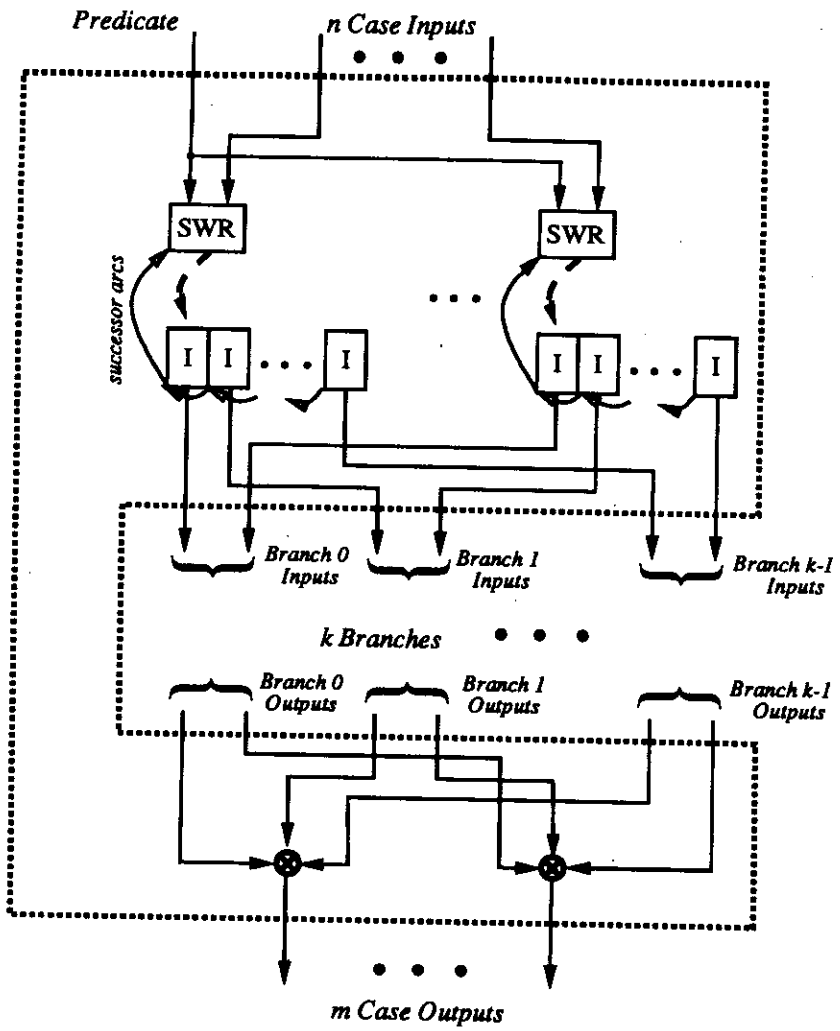


Figure 2.2: **case** schema

Figure 2.5 shows the schema for the parent of a bounded loop. It shows the entry and exit points of the iteration body, and the location of the code to set up and clean up the iteration frames. This code is shown later in this section as a pair of Id procedures.

Figure 2.6 shows the schema for the body of a bounded loop. Note that the predicate of each loop iteration executes in a different frame from the body of that iteration. Also, the switches that control iteration in the TTDA bounded loop scheme have been replaced by `adjust-offset-change-tag` instructions.

The frames for each iteration are set up as follows:

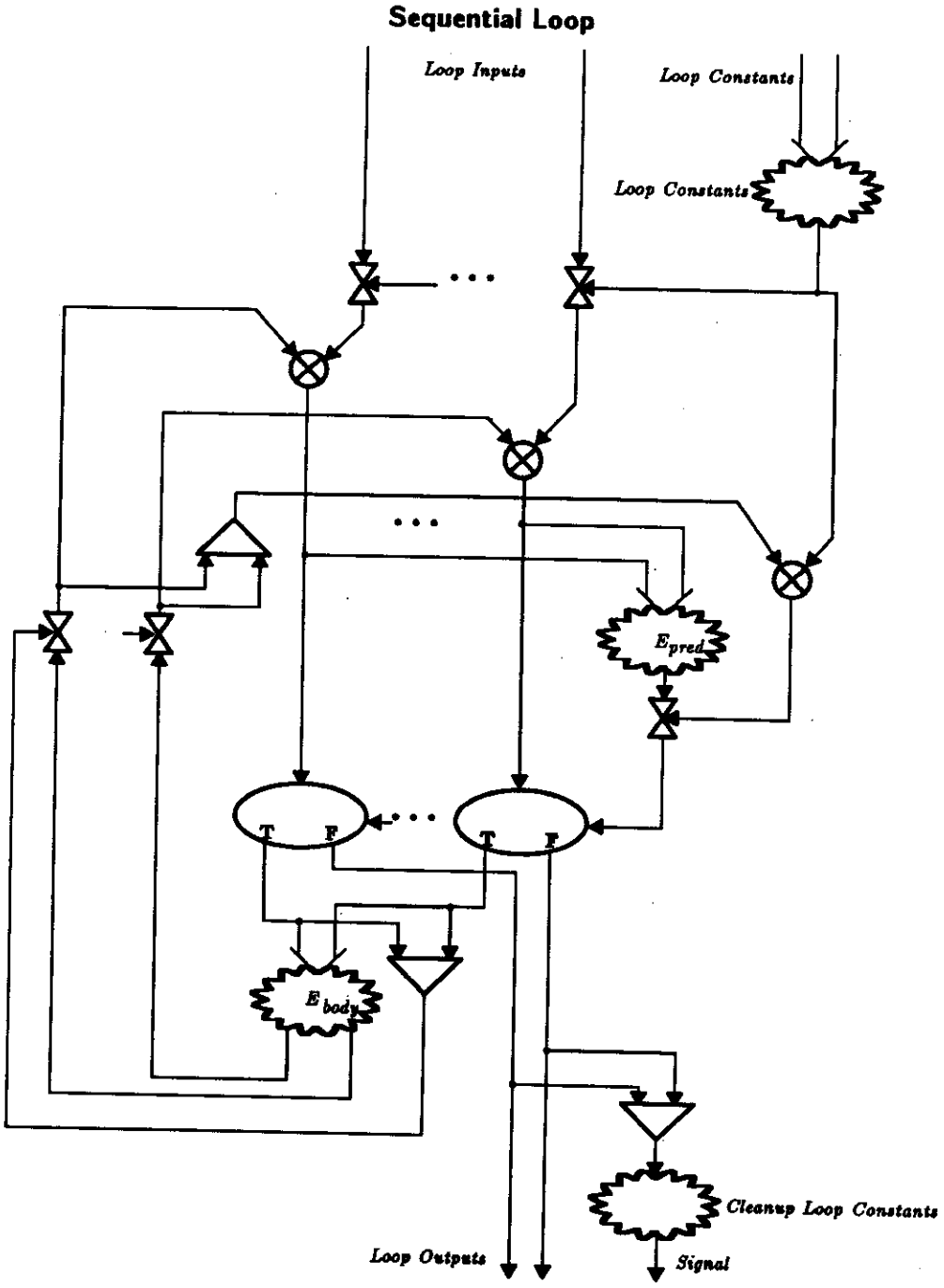


Figure 2.3: sequential loop schema

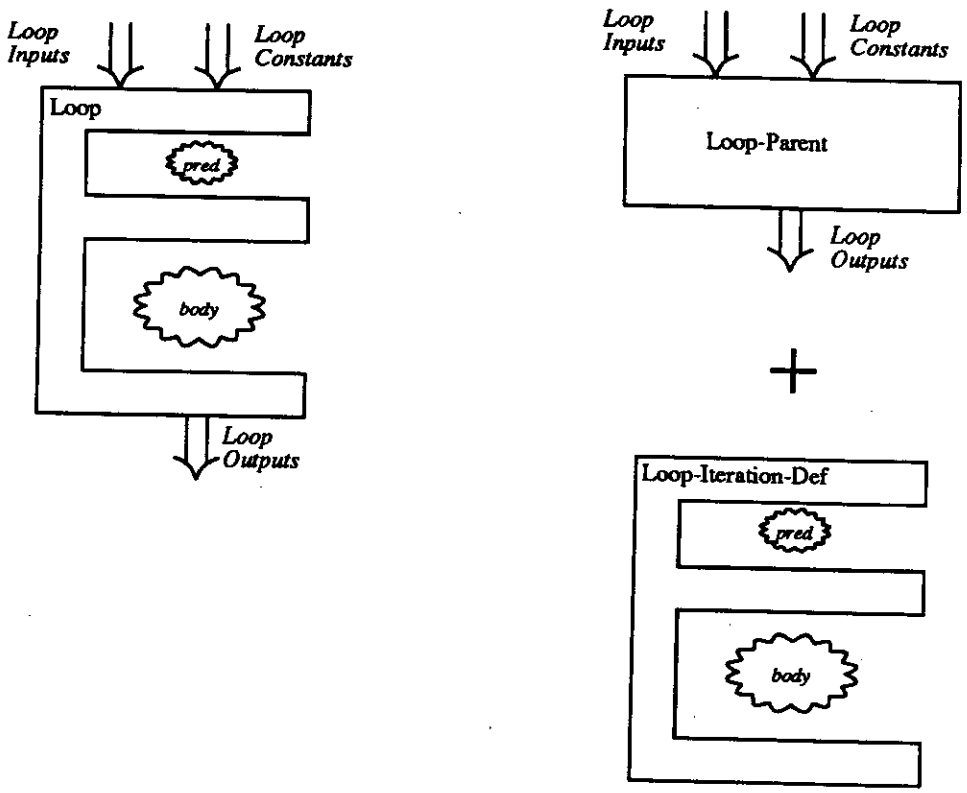


Figure 2.4: Bounded Loop split into Parent and Iteration Program Graph Instructions

Presence Map	Data
Normal	Parent
I-Structure	Next
I-Structure	Prev
Normal	Self
Normal	LC_0
⋮	⋮
Normal	LC_{m-1}

The *Next* and *Prev* slots are guaranteed to defer no more than once.

The contents of the loop-setup and loop-cleanup boxes are shown as Id code in figure 2.5.

2.3 Procedure Definitions and Applications

In this section I will describe the schemas compiled for procedure definitions and for procedure applications. There are two kinds of definition and apply schemas: procedure and *fastcall*. The procedure **def** and **apply** schemas are used for normal Id procedures. This form handles higher-order functions as well as first-order procedure applications. In the case of first-order procedure calls, where the compiler can determine that the application is a full-arity application, the apply operator is replaced by a **direct-apply** operator, which has lower overhead. The fastcall **def**

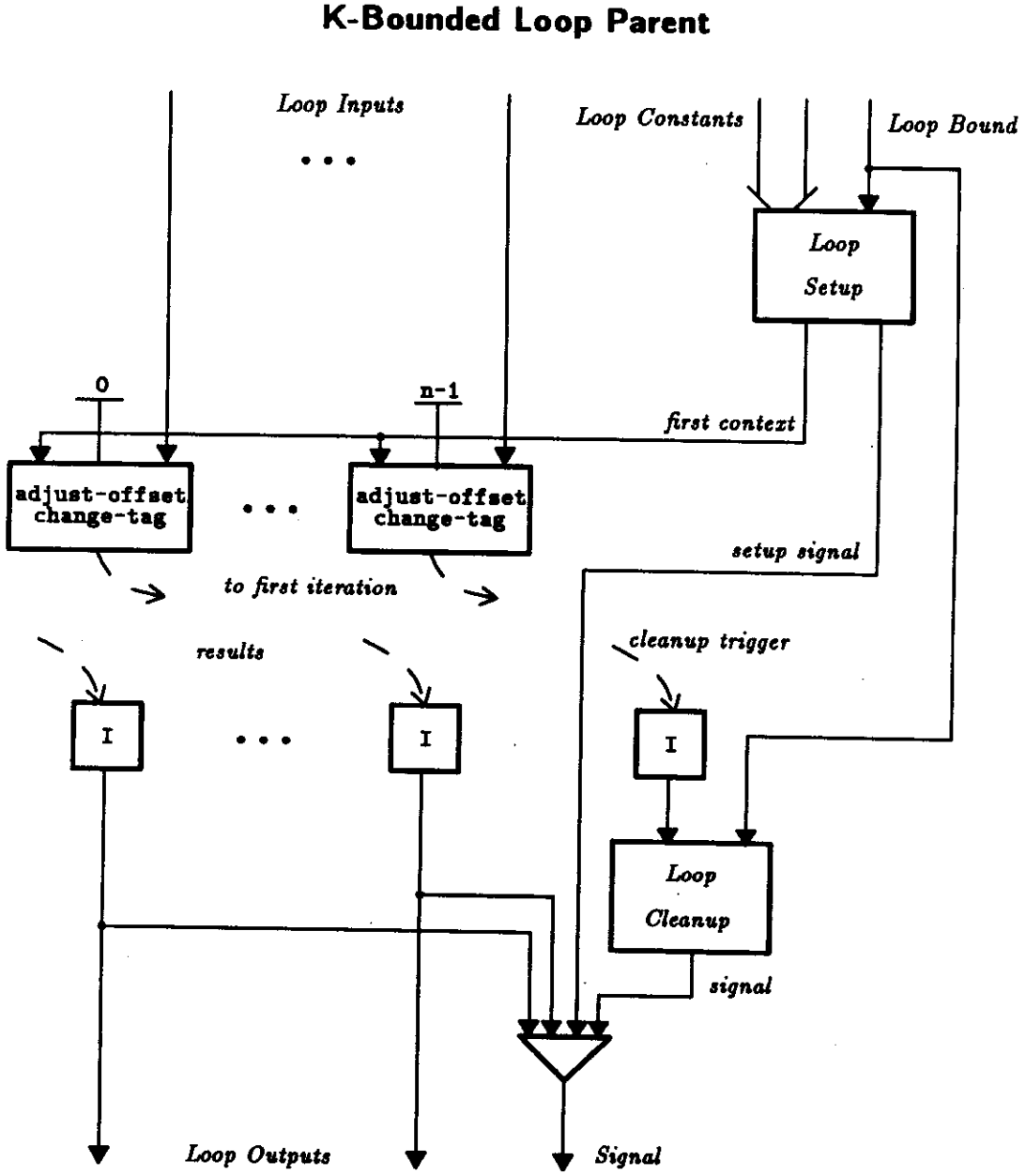


Figure 2.5: Bounded Loop Parent Schema

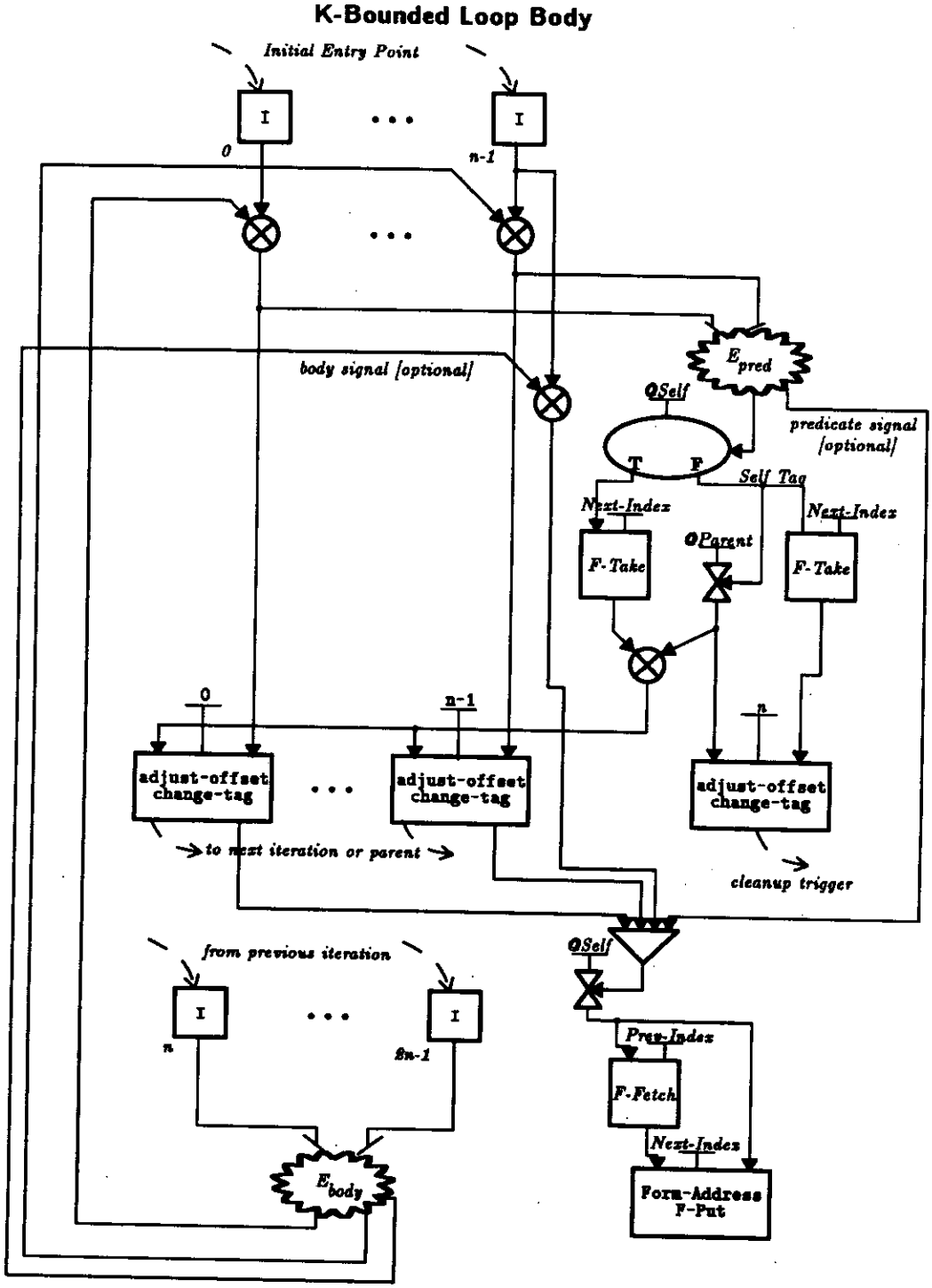


Figure 2.6: bounded-loop body schema

and **apply** instructions are used for code blocks defined by the compiler. These code-blocks are generated when the compiler splits a procedure into several code-blocks, for instance, to limit the number of loops in a procedure to one. The fastcall schemas are more efficient than the procedure schemas, and also allow the return of multiple values.

The def and apply schemas also include code to keep the links of the run-time call-tree in activation frames. Each **apply** keeps a copy of the continuation of the invoked procedure, and each **def** stores the return continuation in slot 0 of the activation frame.

2.3.1 Fastcall Applications

Figures 2.7 and 2.8 show the schemas for the **fastcall apply** and **def** schemas. These are used when the compiler splits a code block into several code blocks, for instance, when compiling nested loops.

2.4 Procedure Applications

Procedure definitions cause a code block consisting of a **def** instruction encapsulating the body of the procedure. The **def** schema has two entry points, allowing the procedure to be called with the general **apply** operator or the **direct-apply** operator.

The general **apply** takes a closure, a closure, and a value as inputs. It tests the closure to see if the value will satisfy the arity of the procedure. If it does, then **apply** extracts the *cb-name* and *argument-chain* from the closure and uses **fastcall-apply** to apply the code-block named by *cb-name* to the argument chain and the value. Otherwise it constructs another closure with the value added to the argument chain. The **apply** schema is shown in Figure 2.10.

One of the optimizations the compiler performs is to replace a series of applies by a **direct-apply** whenever the compiler can determine that a full-arity application of a known procedure is being performed. **Direct-apply** gets a context for the named code-block, and sends in the return-address and all of the arguments (in reverse order, so that the entry-point plus 2 is always the last argument — for compatibility with the general apply schema). It also catches the return value and signal from the procedure call, and when both have arrived, it deallocates the context.

2.4.1 Tail Calling Schemas

[These schemas are preliminary: the compiler does not generate them currently.]

Tail call optimization requires some changes in the previous schemas for several reasons. We must allow a procedure to deallocate its own context when it ends in a tail call. Furthermore, we must chain the termination signal through the tail called procedure so that the when the signal and value are received by the original caller it is guaranteed that all computation in the child procedures has terminated. We may relax this constraint slightly.

For debugging purposes, we are keeping track of the call tree explicitly — slot 0 of an activation frame always contains the return continuation, pointing to the parent procedure.

If we have tail-callable procedure call schemas, then we can only keep up links in the call tree at run time. This may make debugging rather difficult.

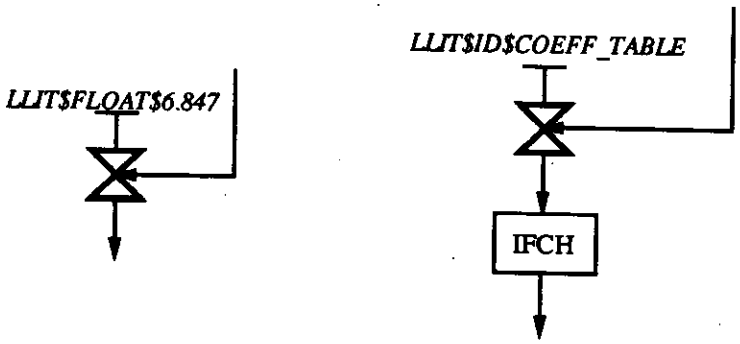
2.5 Global Constants

Figure 2.14 illustrates the schema for the constant-def encapsulator. Evaluation of global constants is demand-driven. The value cell of the identifier for a particular global constant is initialized

with the a delayed trigger. When a fetch is performed against the value cell, this causes the delay manager to allocate a frame, and drop a trigger into the code block that defines the constant. When a value is produced by the code block for a global constant, that value is stored into the appropriate identifier's value cell, satisfying the deferred fetch. After computation in the constant code block has been completed, the code block invokes `extract-continuation-return-context` to deallocate it's own context.

2.6 Literals

Literals are used in the program graph for accessing immediate data and the values of identifiers. In ETS, immediate data cannot be accessed from the instruction stream — they must be stored in processor frame memory. Immediate data are stored in presence state `read-only`, and are accessed by the absolute frame addressing mode. An example of an floating point literal is shown below.

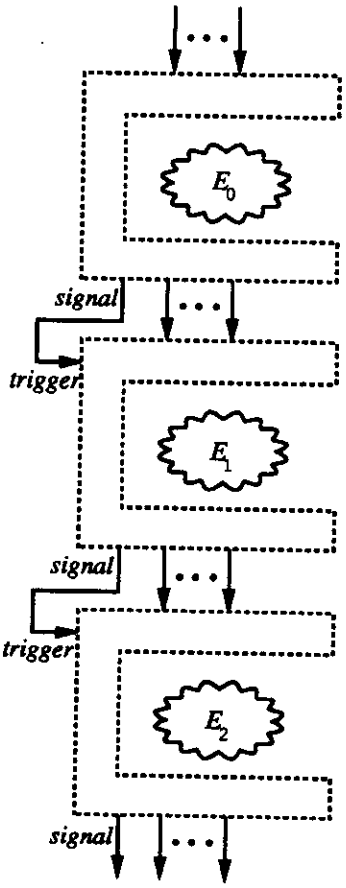


The name `llit$float$3.1415` is the name assigned under the conventions of the Id Object Code Format[5].

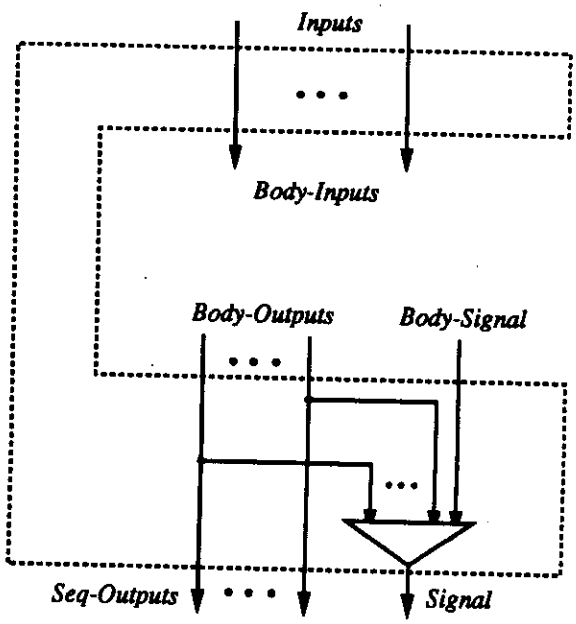
Identifiers, on the other hand, cannot be accessed by frame matching, because the value of an identifier may not be present when the value is requested. Thus, identifier values must be accessed via a two-phase transaction. The scheme for an identifier literal is shown in the right hand side of the figure above. The literal named by `LLIT$ID$COEFF_TABLE` contains a pointer to the identifier descriptor for the `coeff_table`. The first word of the identifier descriptor is the value cell of the identifier. The value cell operates under I-structure semantics.

2.7 Sequentialization

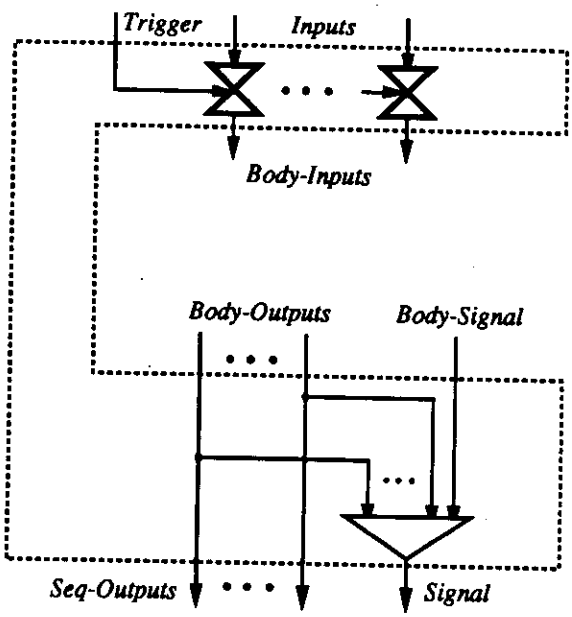
The `seq` encapsulator is used for the sequentialization of execution. To sequentialize a series of subgraphs, the compiler encloses each subgraph with a `seq` encapsulator. Each `seq` encapsulator is triggered by the termination signal output of the preceding `seq` instruction. The first `seq` is untriggered; it is only needed to provide the termination signal of the first subgraph. An example of the use of `seq` encapsulators is shown below. In this graph, E_0 must terminate before E_1 can begin execution; likewise, E_1 must terminate before E_2 may begin execution.



There are two forms of the **seq** encapsulator, triggered and untriggered. The untriggered form is used to get a handle on the termination signal of a subgraph. Here is the schema for the untriggered form.



The triggered **seq** schema is similar, but each input to the encapsulator is gated by the trigger input. It is shown below.



```
def loop_setup k lc_0 ... lc_{m-1} size parent_tag iteration_cbname =
  { first_frame = allocate_frame size ;
    cf = first_frame ;
    ( call frame_store cf self_index (adjust_ip cf entry_point_ip) ;
      call frame_store cf parent_index parent_tag ;
      call frame_store cf lc_0_index lc_0 ;
         ...
      call frame_store cf lc_{m-1}_index lc_{m-1} ;
     & first_tag = set_ip cf iteration_cbname
       %% instantiate the tag of first iteration with
       %% IP of iteration's code
      );

      %% only one reader of PREV, but it may defer once
      call frame_store first_frame prev_index last ;

      prev = cf ;

      last = { For i ← 1 to k sequential do
                 %% k+1 frames total
                 cf = allocate_frame size ;
                 ( call frame_store cf self_index (adjust_ip cf entry_point_ip) ;
                   call frame_store cf parent_index parent_tag ;
                   call frame_store cf lc_0_index lc_0 ;
                     ...
                   call frame_store cf lc_{m-1}_index lc_{m-1} ;
                 & first_tag = set_ip cf iteration_cbname );

                 call frame_store first_frame prev_index last ;
                 next prev = cf ;
               Finally cf };
     &  signal = True
   in (values first_tag signal) };
```

```
def cleanup k last_iter_tag =
   iter_tag = last_iter_tag  ;
  in
     For i ← 0 below k sequential do
       %% Extract PARENT, SELF, PREV, NEXT and loop constants
       call f_clear iter_tag prev_index ;
       call f_clear iter_tag self_index ;
       call f_clear iter_tag parent_index ;
       call f_clear iter_tag lc_0_index ;
          ...
       call f_clear iter_tag lc_m_index ;
      &
       next iter_tag = f_take iter_tag next_index ;
      &
       call return_context iter_tag
     Finally  call f_clear iter_tag prev_index ;
              call f_clear iter_tag self_index ;
              call f_clear iter_tag parent_index ;
              call f_clear iter_tag lc_0_index ;
                 ...
              call f_clear iter_tag lc_m_index ;
             &
              call return_context iter_tag ;
```

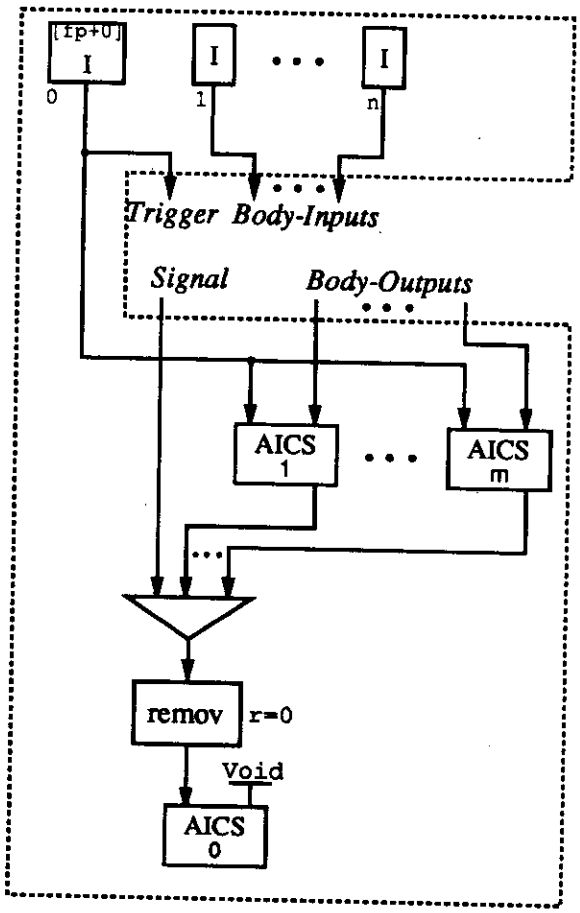


Figure 2.7: **fastcall-def** schema

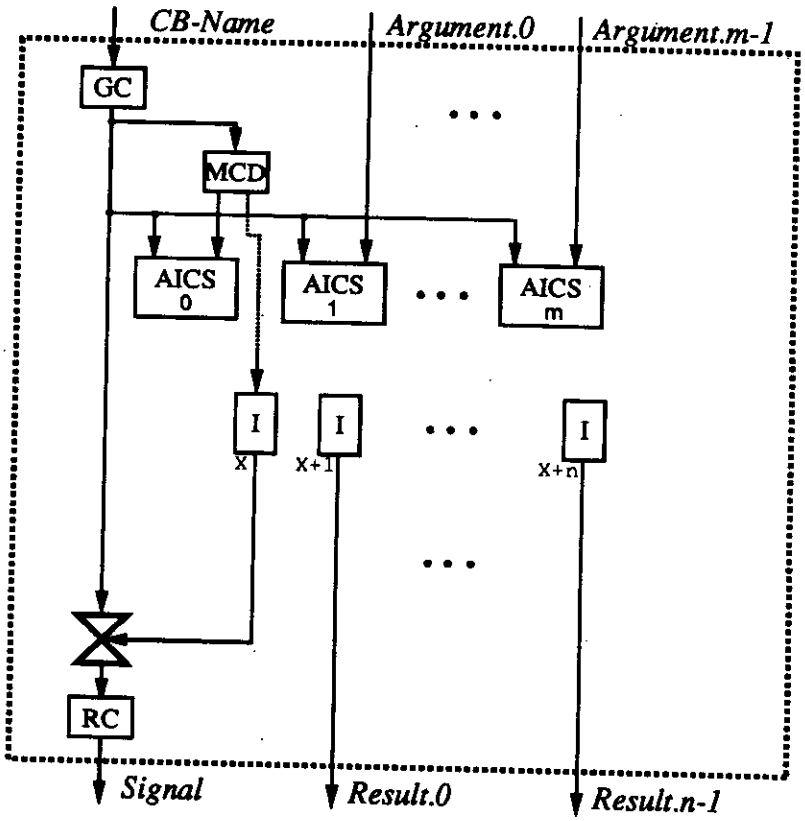


Figure 2.8: `fastcall-apply` schema

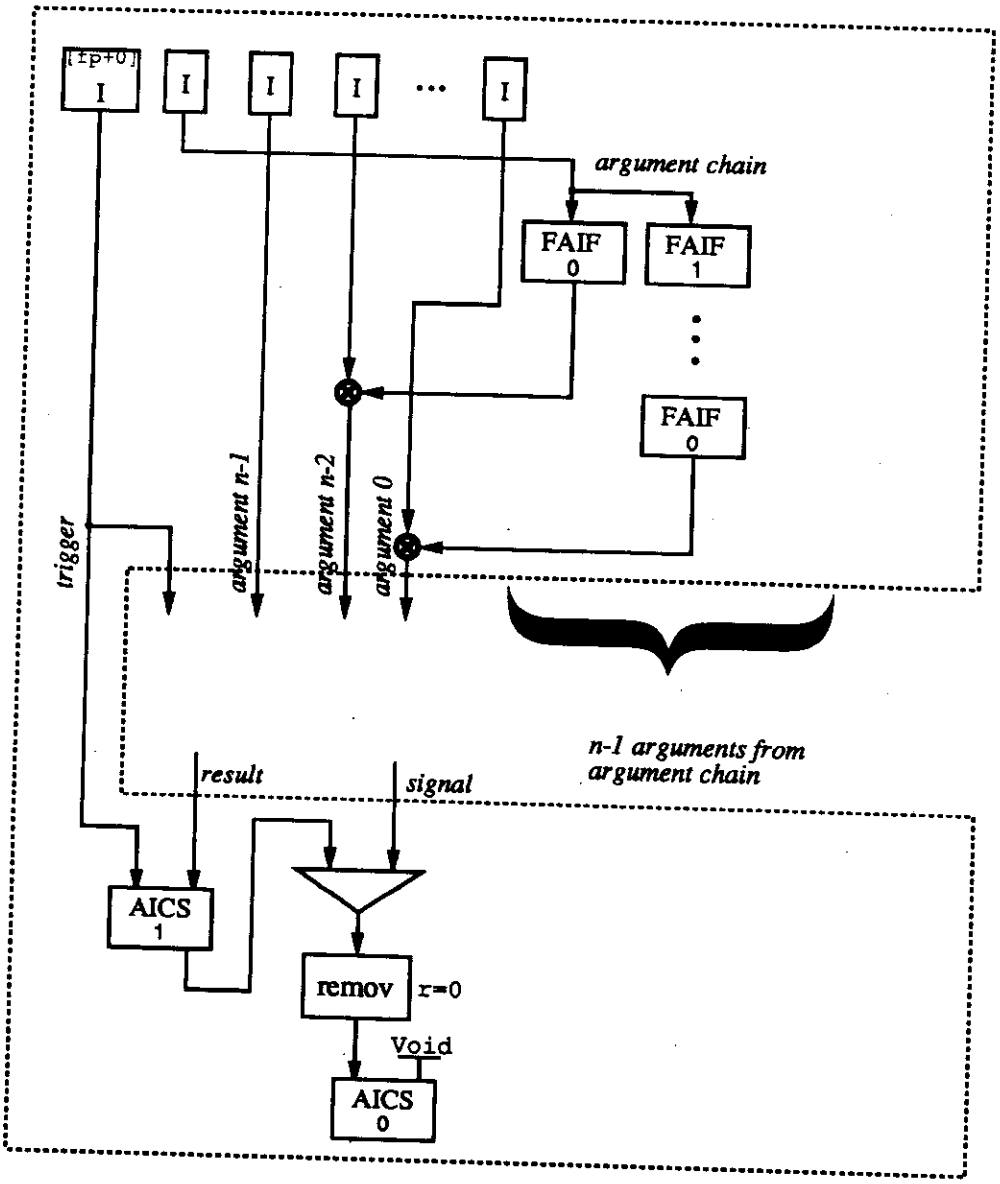


Figure 2.9: Def schema

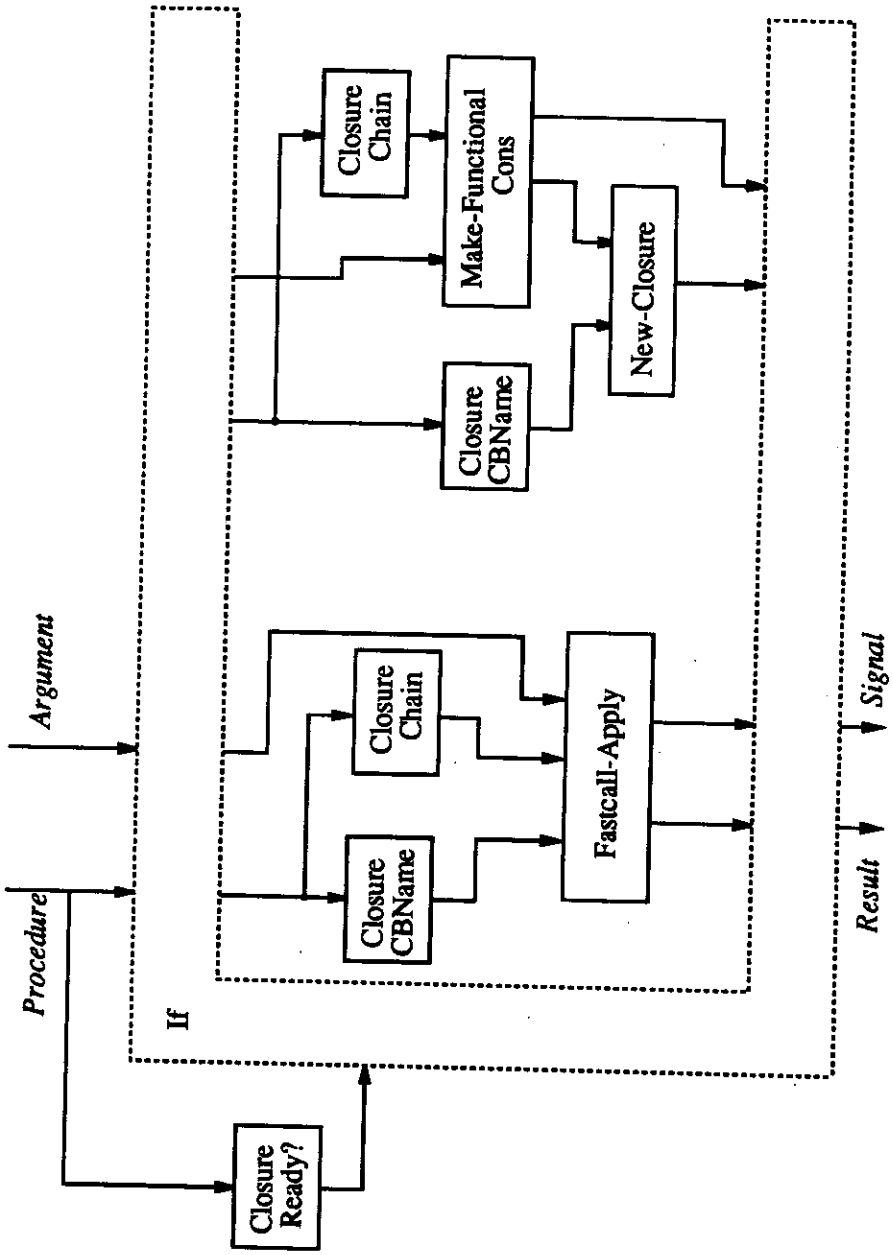


Figure 2.10: Apply schema

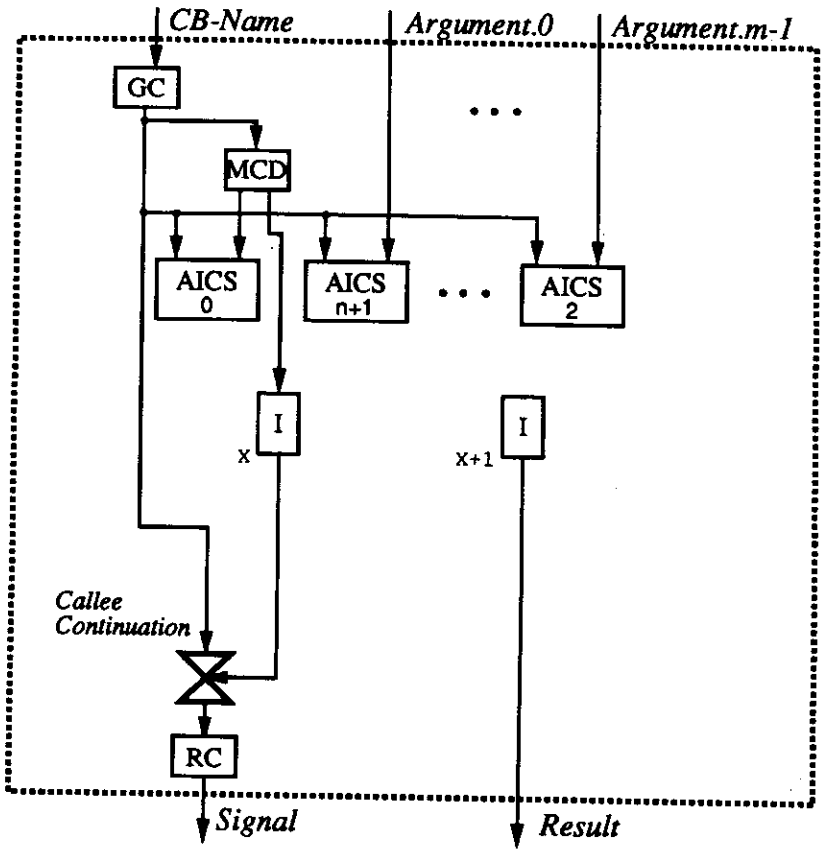


Figure 2.11: `Direct-apply` schema

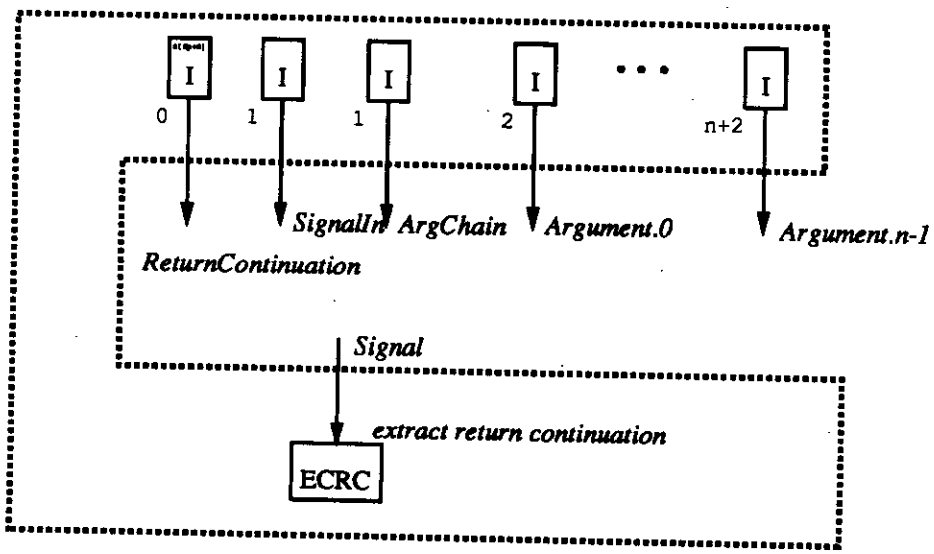


Figure 2.12: Tail calling `def` schema

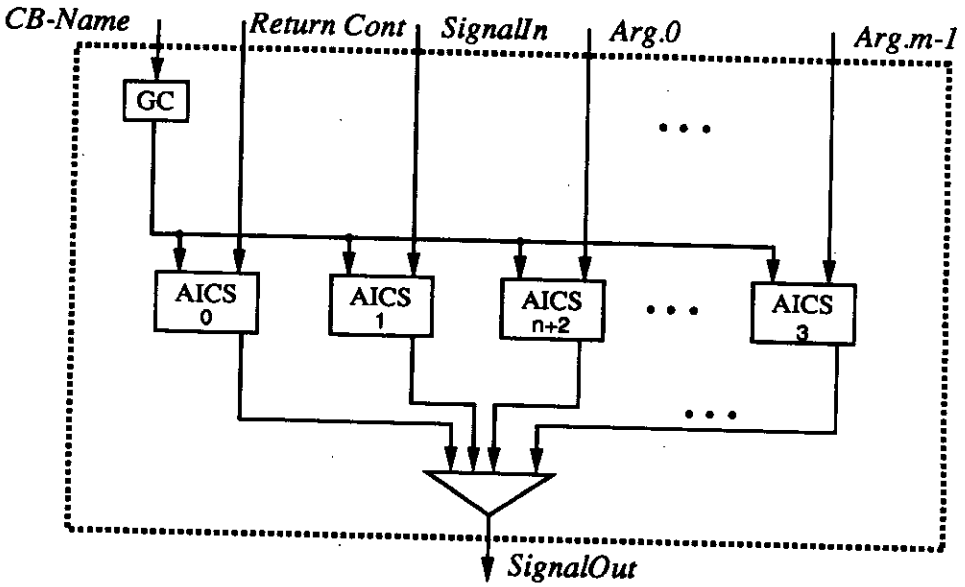


Figure 2.13: Tail calling `direct-apply` schema

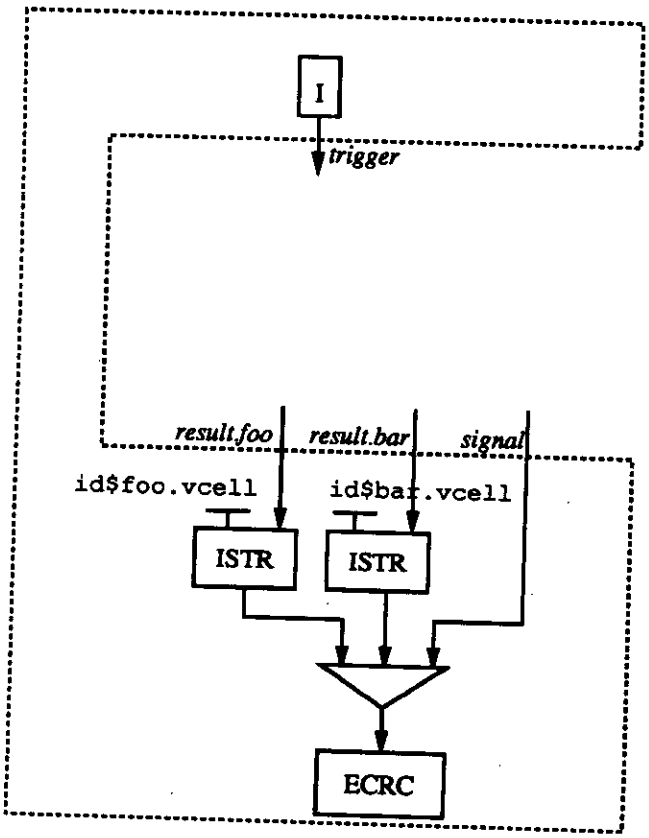


Figure 2.14: `constant-def` schema

Chapter 3

Peephole Optimizations

Because the one of the designs of the compiler is to be as modular as possible, the code generated by `Generate-ETS-Graph` does not take advantage of any special cases present in the code. The peephole optimization module is supposed to satisfy this purpose.

Monsoon has many variations on each instruction that allow some special cases to be much more efficient than the general case. Our goal is to generate code that performs the defined computation in as few instructions as possible.

For instance, if we have a `form-address` instruction feeding an `i-store`, and the offset to the i-structure is a compile time constant (and in a given range), then we can combine these two instructions into a `form-address-i-store` instruction. The actual opcode for this instruction on Monsoon is `aipis.n2`, or add immediate to pointer i-store, using the dyadic form with a token sent to the i-structure unit and an acknowledgment token. This peephole optimization is shown in Figure 3.1.

Figures 3.2 and 3.3 show some algebraic identities that may be implemented as peephole optimizations. These are examples of the peephole optimizations that are performed in the back end of the compiler.

In the last step of the middle end of the compiler, signals and triggers are added to program

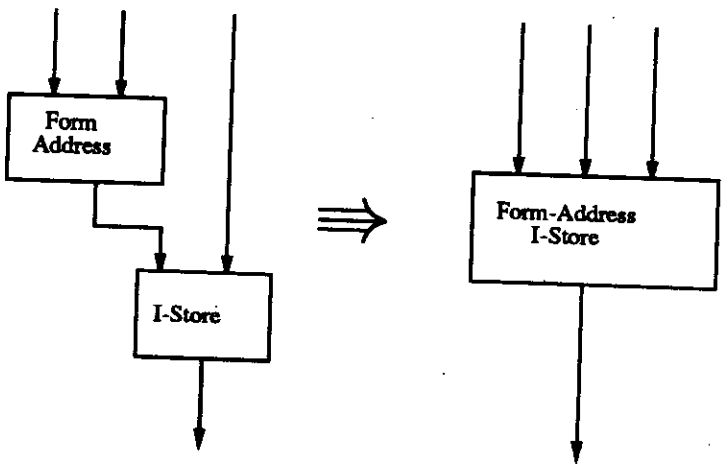


Figure 3.1: Peephole Optimization of Form-Address and I-Store

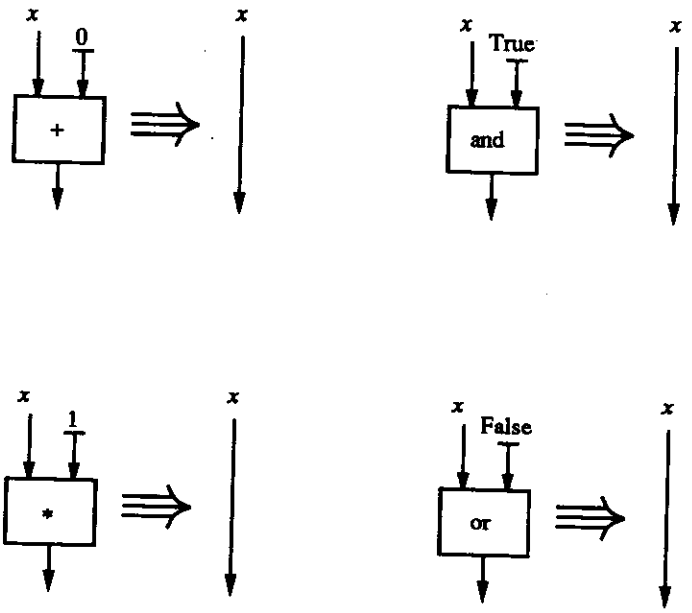


Figure 3.2: Algebraic Identities

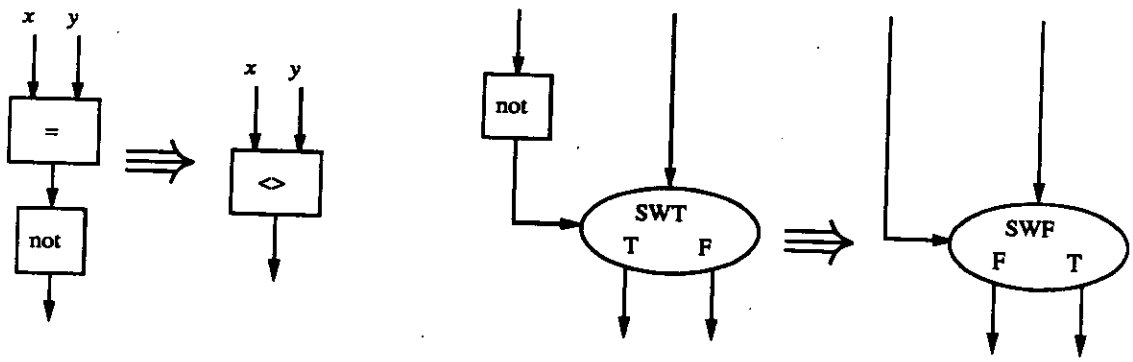


Figure 3.3: Not Removal Identities

graphs to make sure they are *well-connected* graphs. A dataflow graph is well-connected if there are arcs leading to all inputs of all instructions in the graph, and if there are arcs leading from all outputs of all instructions in the graph.

Once the program graph is translated into a machine graph, it becomes apparent that some of these signal and trigger arcs that were added to the compiler are superfluous, and may safely be removed. These signals and triggers are added in such a way that most of the redundant arcs may be recognized by peephole optimizations. For example, all literals in an Id program are compiled into **gates**, or triggered **identities**. Whenever a literal is wired to a strict operator, such as +, the compiler attempts to trigger the literal with one of the other inputs to the operator. Figure 3.4 shows some of the peephole optimizations that eliminate redundant triggers.

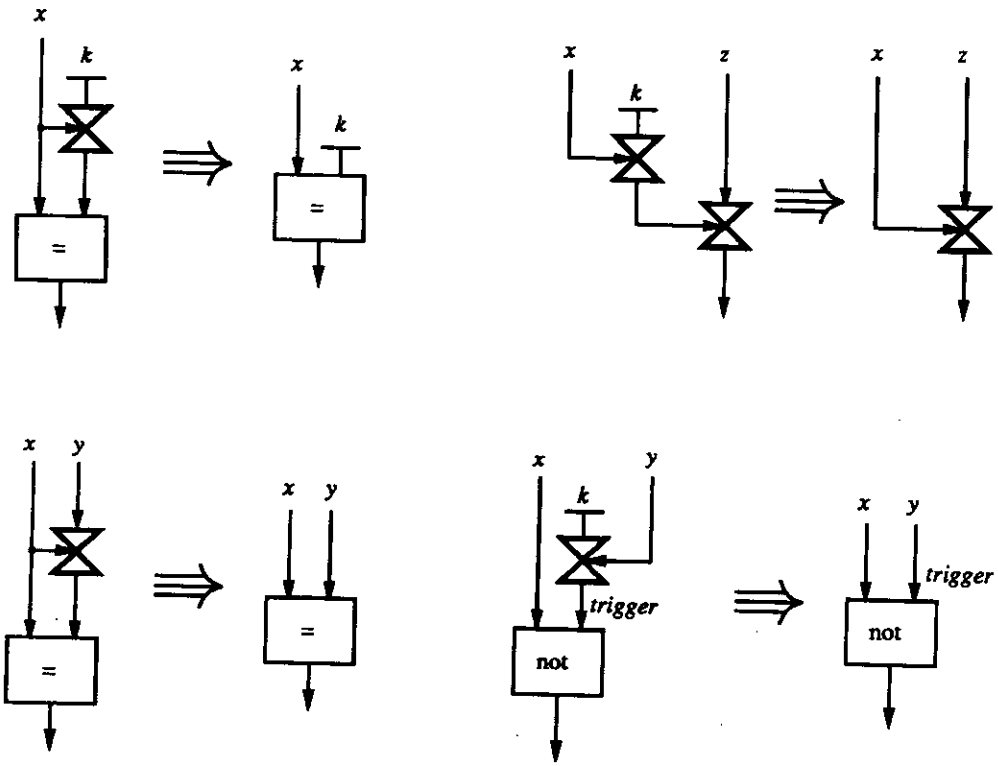


Figure 3.4: Trigger elimination peephole optimizations

Chapter 4

ETS Instruction Schemas

This phase of the compiler expands ETS instructions that cannot be implemented in Monsoon by a single instruction into a small graph of instructions that performs the write operations. Most of these ETS instructions correspond to split-phase or resource manager operations.

4.1 Fetch-Like Split-Phase Operations

In ETS, a *fetch* against an I-structure element will *defer* until a *store* is performed to that element. If more that one *fetch* is performed, then some sort of deferred-read queue is kept, so that when a *store* is performed, all of the fetches get the value that is stored, without having to perform some sort of busy waiting.

4.1.1 I-Fetch and %I-Defer

Figure 4.1 shows the translation for a single ETS *I-Fetch* instruction. There are several assembler constraints on this schema. The three instructions must be in consecutive locations in instruction memory (successor constrained). Furthermore, the `I-Fetch` must be in an even location in instruction memory. The `%i-defer` instruction handles the deferred list for multiple deferred reads, and is only executed in the case where there are multiple deferred reads against an I-structure element. If the value is present or only one read is deferred before a value is stored, then the value will be forwarded directly to the destination instruction (called *dest* in the figure).

The `i-take` ETS instruction also must be compiled into a pair of instructions for Monsoon in order to implement distributed deferred take lists. The schema is the same except that `i-fetch` is replaced by `i-take` and `%i-defer` is replaced by `%i-take-defer`.

4.2 Resource Manager Requests

All of the resource manager requests are implemented as *system calls*. When these calls are executed, an exception is signalled. The exception handlers invoke the Run-Time-System.

4.2.1 Instruction Subset 0

Initially, we are going to be using instruction subset 0 (IS0), which does not include the system call instructions. In this case, the resource managers will be very simple, and deallocation of frames and heap storage will not be performed. The actual code for allocation of frames and heap storage will be inlined into the code block that is making the resource manager call.

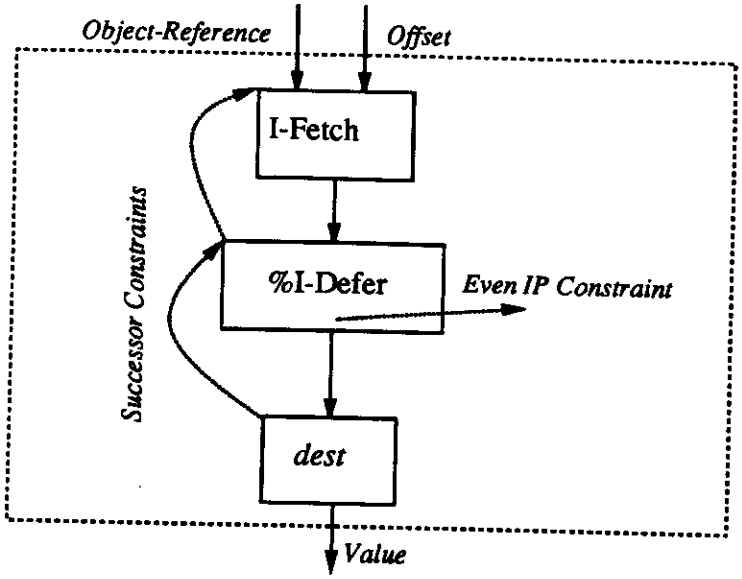


Figure 4.1: Monsoon I-Fetch Schema

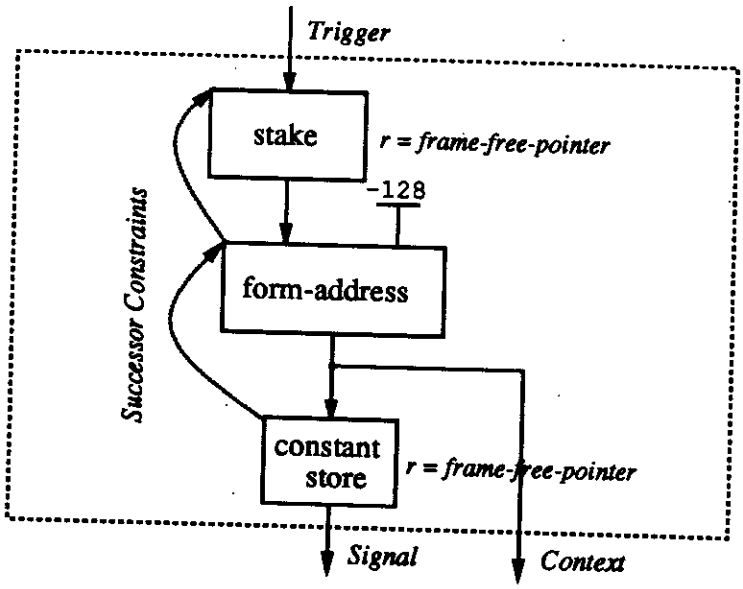


Figure 4.2: Get-Context Schema for IS0

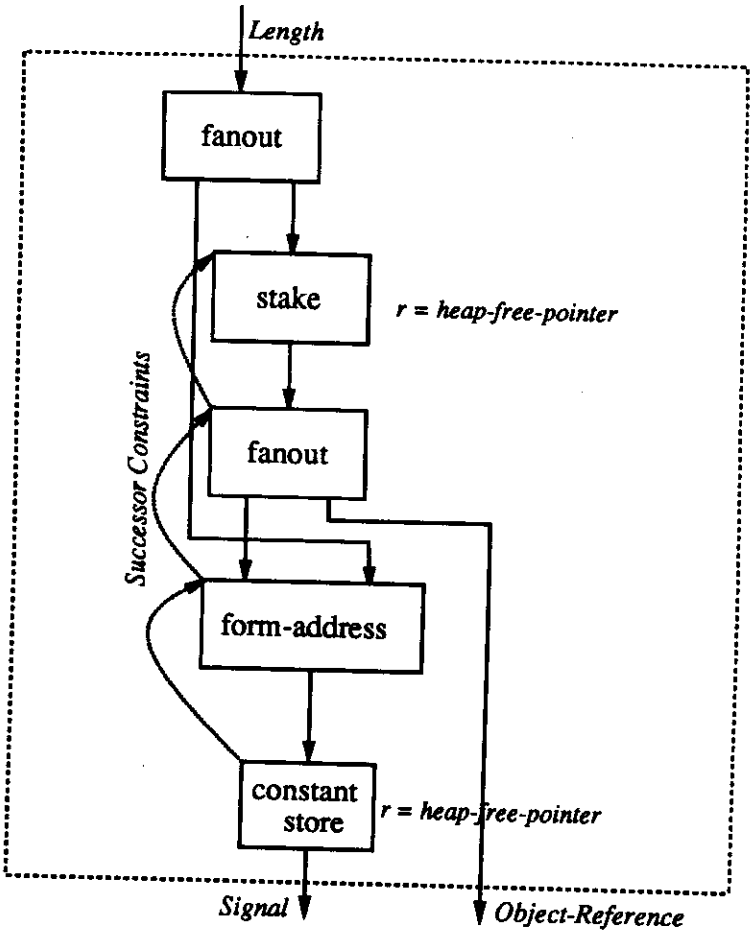


Figure 4.3: Get-Aggregate Schema for IS0

Figure 4.2 shows the three instruction sequence that is used in IS0 to allocate contexts. A simple fetch-and-decrement algorithm, essentially a stack, is used to manage frame storage. The **free-frame-pointer** points to the top of the next free frame storage. When **get-context** is called, it first reads and locks the **free-frame-pointer**, then decrements the pointer by the frame size, 128, and then stores this new pointer back into the **free-frame-pointer** location. The new pointer is also returned as the new frame pointer to be used.

Figure 4.3 shows the five instruction sequence that is used in IS0 to allocate heap storage. The algorithm is the same as for contexts, except that the objects are variable sized, and the heap pointer is incremented with each allocate. The first **fanout** in the sequence has been wired such that the higher priority output token is sent to the right input of the **form-address** instruction, so that no there will be no bubbles when the next four instructions execute. This is to keep the time spent in the critical section from the **stake** to the **constant-store** as short as possible.

Chapter 5

Enforcing Architectural Instruction Constraints

Unlike the TTDA architecture, each instruction in ETS and Monsoon have some constraints on destinations. Instruction fanout is limited to no more than two in all cases. Furthermore, some instructions have only one explicit destination — the other destination must be the left port of the following instruction.

The split-phase fetch-like instructions all have fanout of 1 in TTDA, ETS and Monsoon. This is because a request to read a location can only carry one return address to which to send the value read.

5.1 Enforcing Fanout Constraints

First I will present a short technical note on instruction outputs and destinations.

An instruction has an *output* for each logically different value it produces. Most instructions, such as the arithmetic instructions, have only one output. An example of an instruction with multiple outputs is `switch`, which has two outputs.

An instruction *destination* is a sink, an input of an instruction to which this instruction's output

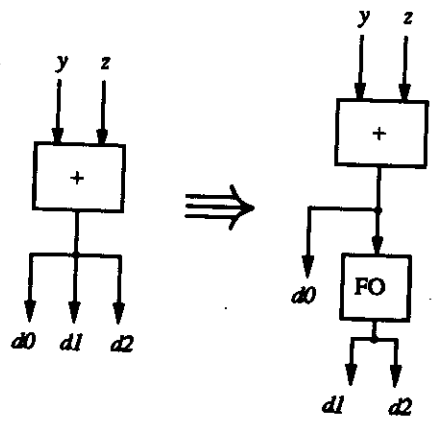


Figure 5.1: Enforcing fanout constraints

is wired. There may be multiple destinations for each instruction output.

The method for enforcing fanout constraints is very simple. We visit each instruction in the ETS graph. For each output of the instruction, see how many destinations there are. For each instruction output that is wired to too many destinations, we add a tree of `fanout` instructions to send the value to all of the destinations.

5.2 Enforcing Successor Constraints

Figure 5.2 shows examples of the three cases in enforcing successor constraints on an instruction. In all three examples, the + instruction must have one destination constrained, because it uses a frame slot and has two destinations.

a. The first case is when the output we are attempting to constrain is wired to the left input of an instruction, and that instruction is not constrained to be the successor of another instruction. In this case we just add a successor constraint to the next instruction, as shown in Figure 5.2(a).

b. The next case is when the output we are attempting to constrain is wired to the right input of an instruction, and that instruction has no assembler constraint. Then we replace the instruction with the reversed form of the instruction, and reverse the inputs to the instruction. Now the arc we are trying to constrain is wired to the left input of the instruction, and we add a successor constraint as in (a). This is shown in Figure 5.2(b).

c. The last case is when the output we are trying to constrain already has an assembler constraint. As shown in Figure 5.2(c), we first add a `fanout` instruction, and then add a successor constraint to the `fanout` instruction.

The algorithm for enforcing successor constraints in the graph is to visit each instruction in the machine graph, and add successor constraints where needed. Some instructions, such as the + instruction shown in the examples, will have two outputs, one of which needs a successor constraint added. In this case, the first destination will be tried, and if a constraint can be added by either cases (a) or (b), then a constraint will be added; otherwise, a successor constraint will be added to the other destination by case (a), (b), or (c).

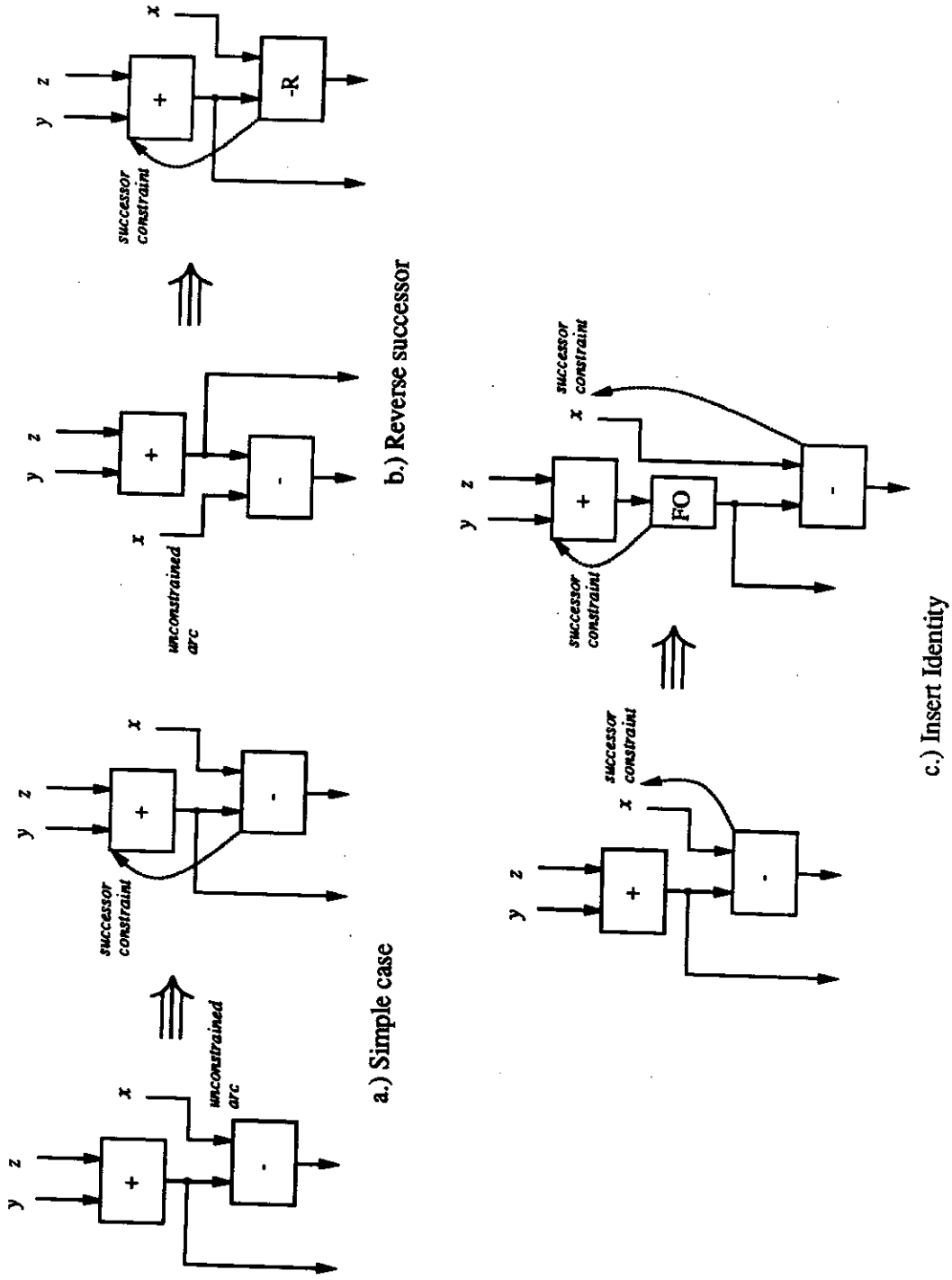


Figure 5.2: Adding successor constraints

Chapter 6

Compiling for Threads and Temporary Registers

Normal dataflow code performs a synchronization on every dyadic operation. Monadic instructions and dyadic instructions that take a literal or frame-constant as one input do not perform a join synchronization. In the current architecture, the first token into a join causes a bubble in the pipeline, because the instruction does not fire — instead it writes the value into a frame slot, sets the presence bits to **present**. Since about a third of the instructions executed dynamically are normal dyadic instructions, a quarter of the cycles are spent in alu bubbles.

We would like to reduce this synchronization overhead in cases where it is not buying us much in the way of parallelism. The ETS abstraction and the Monsoon processor have features that allow the reduction of synchronization performed in ETS and Monsoon code. We have a set of temporary registers (or temporaries), associated with each thread in the pipeline. As long as a thread holds the pipeline (*i.e.*, one of the output tokens from each instruction in the thread critically circulates), the thread can use its temporaries. Monsoon has 3 temporary registers for each active thread.

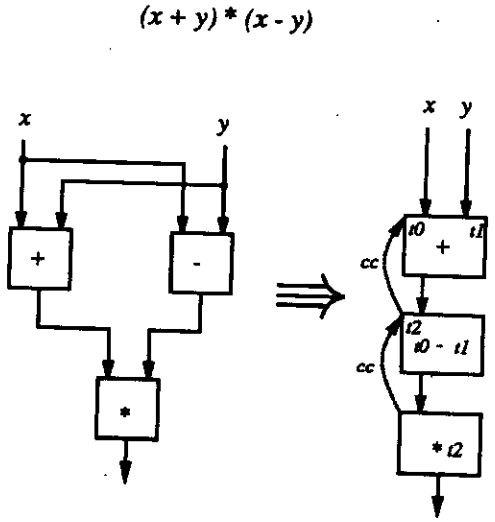


Figure 6.1: Example of use of threads and temporaries.

Temporaries are useful when a small number of values are used several times apiece by several instructions in an ETS graph. Once a subgraph is recognized to meet this requirement, the transformation required is to move the synchronization on these values to the top of the subgraph, where the values will be stored in frame slots (as frame temporaries) or in temporary registers, and then to sequentialize the instructions in the subgraph and replace the arcs that supplied the values with uses of frame temporaries or registers. Note that values forked off from the middle of a sequential thread, but that no joins may be performed in the middle of a sequential thread. In addition, I-Fetches may be requested from the thread, but the results of the fetch may not be used within the thread (because that would require a join synchronization).

When the number of values in a thread exceeds the number of temporary registers, the extra values must be spilled to frame slots. Some convention must be obeyed about cleaning the frame slots when the thread exits, or else the frame allocator/deallocator must perform these housekeeping duties.

The problem of sequentializing an expression graph into a thread using a finite number of registers is analogous to that of generating sequential code from an expression tree with a finite number of registers in conventional compilers. Optimal register allocation in this case is $\mathcal{NP}$-complete, but one can do pretty well using heuristics.

Figure 6.1 shows an example of generating threaded ETS code for the expression $(x+y)*(x-y)$. This one is simple, because it exactly fits in the three registers on Monsoon.

We anticipate that code written for the Run-Time-System will make extensive use of registers in order to improve performance. Although we will have the compiler try to generate as much threaded code as possible, we do not think that threads will be used as much in Id code, because of the extensive use of split-phase transactions and non-strictness in the Id language.

Chapter 7

Assigning Frame Offsets

The current Monsoon Back End of the Id compiler takes a very simplistic approach to frame slot allocation — it assigns a unique frame slot to every dyadic and frame slot instruction in the code. This is extremely wasteful of frame-store, and we hope to improve it once we have the complete Id language running on Monsoon.

Note that some allocation of frame slots is done in two places earlier in the compiler. In the **CSE-and-Code-Hoisting** module, loop invariants are hoisted out of loops — these loop constants will be stored in frame slots with presence bits **constant** and used for the duration of the loop. The module that compiles threads from ETS graphs will also assign some frame slots when it generates register spill code (assuming we get it to generate any code at all).

Chapter 8

Assigning Instruction Offsets

At this point in compilation, each procedure has been translated into one or more code blocks. Each code block consists of a graph of Monsoon instructions. Restrictions of the Monsoon architecture make the the task of linearizing the graph into a vector of instructions more difficult. An instruction is specified by three physical fields, *opcode*, *f1* and *f2*, where *opcode* is 12 bits, and *f1* and *f2* are 10 bits each. The architecture actually has 5 logical fields, *s1*, *s2*, *r*, *regs* and *adj*, to encode into the 2 physical fields, so some fidelity is lost in the translation.

Here are the restrictions:

- Instructions connected by an arc must be within ±512 of each other, because of the size of the destination fields, *s1* and *s2*.
- The `MCD` that constructs the return address of a procedure call must be within ±512 of the instruction that is the target of the return values from the procedure call. Actually, there is an arc connecting the `MCD` with the instruction that catches the first return value, but no token traverses that arc.
- The implicit destination of an instruction (when there are more destinations than physical fields to contain *s1* or *s2*), is to the following instruction, port left. This is a *successor* constraint. A series of instructions connected by successor constraints is a *chain* of instructions.
- *even* assembler constraints. Some chains will be headed by instructions that must be placed on even instruction locations (*e.g.*, the `%I-Defer` instruction in an `I-Fetch` schema).
- *Absolute* assembler constraints. The compiler requires some chains to be placed at absolute locations (with respect to the beginning of the code block). The only use of this is placing the chain containing the entry points to the code block as the first instructions in the code block. See Sections 2.1.2 and 2.4 for details.
- Compiler imposed *successor* constraints. These are used in procedure linkage conventions.

The general strategy for placing all instructions of a code block into a code vector is:

1. Find all the instruction chains.
2. Attempt to place all chains whose head has an absolute assembler constraint.
3. Starting with the root set of the code block, do a breadth first search of code block. Start with empty Queue, then enqueue all instructions in the root set of the code block.

4. For each instruction visited:

   (a) If the instruction has been placed, do nothing

   (b) Otherwise, place the instruction and the chain to which it belongs. Then enqueue all instructions that point to this chain or to which this chain points.

5. Repeat step 4 until queue is empty.

A breadth first search was chosen because most graphs that we have compiled seem to be taller than they are wide, so a breadth first search tends to keep the plus or minus 512 constraint from being broken as much.

When the distance between an instruction being placed and any of its sources or destinations is greater than 512, a jump must be inserted.

We are planning to do some form of code-block splitting to reduce the frame size needed for each code block, and to help ensure that this constraint is not broken.

Chapter 9

Representation

The ETS compiler back end uses a machine graph representation built on top of the DFCS[3] dataflow graph representation.

9.1 ETS Instructions

define-ets-instruction *opcode-name* **&body** *clauses* [*Macro*]

This macro defines an ETS opcode with name *opcode-name*. The required clauses are: **:n-inputs** and **:n-outputs** which indicate the number of inputs and outputs for that instruction. There is one optional clause, **:properties**, which is defines the property list for the opcode. The properties are given as alternating key-value pairs.

define-ets-instruction-property *indicator* **&key default accessor-name** [*Macro*]

Defines a property named *indicator*. If *default* is not specified the default value for the property will be **nil**. If the accessor name is not specified this macro will define an accessor function named ETS-INSTRUCTION-*indicator*.

add-ets-instruction-property *opcode indicator value* [*Macro*]

This macro adds to the property list for opcode *opcode*. It is useful for setting a property of an opcode when modifying the definition of the instruction would be inconvenient.

make-ets-instruction *opcode* [*Function*]

This function creates an ETS instruction whose opcode is *opcode*.

9.1.1 Pseudo Instructions

In addition to the ETS instructions, which correspond to actual instructions on an ETS machine, there are the *pseudo* instructions. These do not correspond to real ETS machine instructions — they are used for convenience and uniformity of representation in the dataflow graph abstraction.

Constant The *constant* pseudo instruction has no inputs and a single output. It represents a literal, or immediate, value in the graph. A constant pseudo instruction contains a (*datatype*, *value*) pair to indicate the actual value and type that it represents.

make-constant-pseudo-instruction *datatype value* [*Function*]
constant-pseudo-instruction-datatype *instruction* [*Function*]
constant-pseudo-instruction-value *instruction* [*Function*]
constant-pseudo-instruction-p [*Function*]

These functions construct constant pseudo instructions and select the datatype and value from constant pseudo instructions.

Merge Although the ETS machine allows unlimited fan-in on instructions, the dataflow graph representation only allows one arc to be wired to any instruction input. The **merge** pseudo instruction is used to allow two arcs to be wired to an input. Note that the **merge** instruction is not a well-behaved dataflow instruction — a token on either input arc is passed to the output arc. However, in combination with the **switch** machine instruction, well-behaved subgraphs may be formed. The TTDA machine graph abstraction only had 2 input merges; the ETS machine graph abstraction has merge instructions with an arbitrary number of inputs because some of the modules must be able to look through merges to all their sources or all of their outputs.

make-merge-pseudo-instruction **&optional** (*n-inputs 2*) [*Function*]
merge-pseudo-instruction-p [*Function*]

The function **make-merge-pseudo-instruction** creates a **merge** instruction with *n-inputs* inputs. The predicate **merge-pseudo-instruction-p** returns true when applied to a **merge** instruction.

9.2 Instruction Slots Used in the ETS Back End

instruction-inputs-flipped-p *instruction* [*Instruction Slot*]

This slot will be set to T if the inputs should be flipped.

instruction-outputs-flipped-p *instruction* [*Instruction Slot*]

This slot will be set to T if the outputs of the instruction should be flipped. It may be useful only for switches — choosing between **SWT** and **SWF**.

instruction-long-r-p *instruction* [*Instruction Slot*]

This slot, when non-nil, designates instructions using long-r matching.

instruction-short-r-p *instruction* [*Instruction Slot*]

This slot, when non-nil, designates instructions using short-r matching.

instruction-machine-p *instruction* [*Instruction Slot*]

This slot, when non-nil, designates that an instruction is a machine instruction. The value of this slot will correspond to the type of machine instruction: **:ETS** for ETS machine instructions, **:pseudo-constant** for **constant** pseudo instructions, or **:pseudo-merge** for **merge** pseudo instructions.

9.3 Assembler Constraints

In this section I will describe the instruction slots and functions used by the back end for representing and manipulating assembler constraints. An assembler constraint is a constraint on the placement of an instruction or sequence of instructions during the assembly phase of compilation. During assembly, the compiler must take a graph of instructions and lay them out in a vector. The assembly constraints restrict how the instructions of the graph may be placed in the code vector. Some constraints are due to architectural restrictions (see Chapter 5) and some are due to linkage conventions (see the chapter on IOCF in [5]).

`instruction-assembler-constraint` *instruction* [*Instruction Slot*]
`assembler-constrained-p` *instruction* [*Function*]

The `instruction-assembler-constraint` slot contains the assembler constraint, if any, on an instruction. The predicate `assembler-constrained-p` return non-nil when applied to an instruction that is assembler constrained.

There are 3 types of assembler constraints that are used in the ETS back end: *absolute*, *successor*, and *even*.

absolute The absolute constraint specifies that an instruction must be placed at a certain location in memory. This may break if the location is non-zero.

`add-absolute-constraint` *instruction address* [*Function*]
`instruction-absolute-constraint-address` *instruction* [*Function*]
`absolute-constrained-p` *instruction* [*Function*]

The functions `add-absolute-constraint` constrains instruction *instruction* to be placed at offset `address` by the assembler. The function `instruction-absolute-constraint-address` returns the address to which *instruction* is constrained. The predicate `instruction-absolute-constraint` returns true if *instruction* has an absolute constraint.

successor The successor constraint specifies that two instructions must be placed in successive locations in instruction store.

`instruction-successor-from-constraint` *instruction* [*Instruction Slot*]
`instruction-successor-to-constraint` *instruction* [*Instruction Slot*]
`add-successor-from-constraint` *instruction next-instruction* [*Function*]
`add-successor-to-constraint` *instruction prev-instruction* [*Function*]
`add-successor-constraint` *from-instruction to-instruction* [*Function*]
`successor-constrained-p` *instruction* [*Function*]
`replace-successor-constraints` *old-instruction new-instruction* [*Function*]

The function `add-successor-constraint` constrains *to-instruction* to be the successor of *from-instruction* when assembled. A chain of successor constrained instructions is actually a doubly-linked list, with the predecessor instruction linked to the successor by the `instruction-to-constraint` instruction slot and the successor instruction linked to the predecessor by the `instruction-from-constraint` instruction slot.

The predicate `successor-constrained-p` returns true if *instruction* is constrained to be the successor of another instruction. This predicate will return *nil* if *instruction* is just the predecessor of a successor-constrained instruction.

The function **replace-successor-constraints** is used for changing the successor of an instruction. If *old-instruction* was a predecessor of a successor-constrained instruction, then the successor will be made the successor of *new-instruction* If *old-instruction* was a successor of an instruction, then *new-instruction* will be made a successor of that instruction.

even This constraint is needed for compilation of the ETS schemas for **i-fetch** and **i-take** with multiple defers.

add-even-constraint *instruction* [*Function*]
even-constrained-p *instruction* [*Function*]

The function **add-even-constraint** adds an even constraint to *instruction*. The predicate **even-constrained-p** returns true if *instruction* has an even-constraint.

9.4 Specification of Temporary Register Usage

Monsoon and other ETS machines have temporary registers available for use by *threads*, where a thread is, informally, a sequence of instructions that does not relinquish the processor pipeline (or pipeline phase) until it has completed. A thread may not perform any joins (except for the first instruction of a thread) or split-phase instructions (except for some special, processor-local operations). These constraints allow the thread to maintain some state in the processor itself without interference from other threads.

Monsoon, and the ETS abstract machine defined by Nikhil and Arvind [1] allows an instruction two write up to two registers with the incoming values and to substitute up to two temporary register values for the incoming values. We expect that the use of temporary registers of some sort will be very important in future dataflow machines in order to reduce the bandwidth required between processor and frame store.

instruction-regs-specifier *instruction* [*Instruction Slot*]

The **instruction-regs-specifier** slot, when set, specifies that *instruction* uses temporary registers. The regs-specifier, when set, will contain a list of four elements:

```
(ta-spec tb-spec a'spec b'spec)
```

where *a'spec* and *b'spec* are either **:Pass-Through**, indicating to use the left or right input values from the frame operation, or a number, indicating the register to read, and *ta-spec* and *tb-spec* are either **:No-Store**, indicating not to store a value, **:Save**, indicating to store the value in the exception A or B register, or a number, indicating which register to save the value into.

9.5 Miscellaneous Instruction Slots

instruction-reassociatable-p *instruction* [*Instruction Slot*]

This slot specifies that *instruction* should be considered to be an associative operation, and that it may be reordered with adjacent instructions of the same opcode in order to better optimize the graph.

`instruction-commutable-p` *instruction* [*Instruction Slot*]

This slot specifies that *instruction* should be considered to be a commutative operations, and that its inputs may be reversed in order to better optimize the graph.

`instruction-entry-point-p` *instruction* [*Instruction Slot*]

This slot, when non-nil, specifies that *instruction* is an entry point to the graph, and that there is a dynamic arc wired the the left input of the instruction.

`instruction-fp-offset` *instruction* [*Instruction Slot*]

This slot will be set with the offset of the frame slot reserved for use by instruction *instruction* for frame operations such as normal matching, or reading constant values.

`instruction-ip-offset` *instruction* [*Instruction Slot*]

This slot will be set with the offset of instruction *instruction* within the code vector.

9.6 Instruction Properties

This section contains the description of all of the instruction properties used by the ETS compiler back end. Properties are named by their indicators here, where each indicator is a Lisp keyword. The accessor function, unless otherwise specified, will be named `ets-instruction-`*indicator*.

`:fan-out-limit` *:default* `'(2)` [*ETS Instruction Property*]

This property specifies the fan-out limit for an opcode if no frame slots, temporaries, or immediate adjustments are used. The value of this property must be a list of numbers, where the *i*th element of the list is the fan-out limit for the *i*th output of the instruction. Use of frame-slots, temporaries and immediate adjustments will, in some cases, reduce the fan-out of an instruction.

`:side-effecting-p` [*ETS Instruction Property*]

This property, when true, specifies that the opcode performs side-effects when executed.

`:tag-altering-p` [*ETS Instruction Property*]

This property, when true, specifies that the opcode alters the tag of one or more of the output tokens has been computed from the input tokens.

`:generate-ets-system-function` [*ETS Instruction Property*]

This property names a function to expand this instruction into the primitive Monsoon instructions that implement this opcode.

`:successor-constrained-p` [*ETS Instruction Property*]

This property, when true, specifies that the opcode must be successor constrained for Monsoon.

`:constrained-output` *:default 0* [*ETS Instruction Property*]

This property, whose value is either an integer or *nil*, specifies which output of an instruction should be constrained if the instruction is successor constrained.

`:extracting-p` [*ETS Instruction Property*]
`:storing-p` [*ETS Instruction Property*]

:matching-mode *[ETS Instruction Property]*

These properties specify what frame operation is used by the opcode.

:n-implicit-outputs *:default 0* *[ETS Instruction Property]*

This property specifies how many dynamic arcs emanate from instructions with a given opcode.

:associative-p *instruction* *[ETS Instruction Property]*

:commutative-p *instruction* *[ETS Instruction Property]*

These properties specify that all instructions with a given opcode are associative or commutative.

Bibliography

[1] Rishiyur S. Nikhil and Arvind. An abstract description of a monsoon-like ets interpreter. Computation Structures Group Memo draft, Massachusetts Institute of Technology Laboratory for Computer Science, Cambridge MA, May 1990.

[2] Gregory M. Papadopoulos and Kenneth R. Traub. Monsoon macroarchitecture reference manual. Computation Structures Group Memo ???, Massachusetts Institute of Technology Laboratory for Computer Science, Cambridge MA, 1990.

[3] K. R. Traub. A dataflow compiler substrate. Computation Structures Group Memo 261, Laboratory for Computer Science, Massachusetts Institute of Technology, March 24 1986.

[4] Kenneth R. Traub. A compiler for the MIT Tagged-Token Dataflow Architecture. Technical Report TR-370, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, August 1986.

[5] Kenneth R. Traub, Michael J. Beckerle, James E. Hicks, Gregory M. Papadopoulos, Andrew Shaw, and Jonathan Young. Monsoon software software interface specifications. Technical Report MCRC-TR-1 and CSG Memo 296, Motorola Cambridge Research Center and Massachusetts Institute of Technology, Cambridge, MA, January 1990.