`jc`: An Efficient and Portable Sequential Implementation of Janus

David Gudeman[1], Koenraad De Bosschere[2], Saumya K. Debray[1]

1. Department of Computer Science, The University of Arizona, Tucson, AZ 85721, USA
2. Electronics Laboratory, Rijksuniversiteit Gent, B-9000 Gent, Belgium

Abstract: Janus is a language designed for distributed constraint programming [12]. This paper describes `jc`, an efficient and portable sequential implementation of Janus, which compiles Janus programs down to C code. Careful attention to the C code generated, together with some simple local optimizations, allows the system to have fairly good performance despite the lack (at this time) of global flow analysis and optimization.

1 Introduction

Janus [12] is an instance of a concurrent constraint programming language [11]. This report describes `jc`, an efficient and portable sequential implementation of Janus that compiles down to C. A Janus program is a set of *flat guarded clauses* defining its procedures. It is in many respects similar to *Strand* [6] and Flat GHC [13]. There are, however, a number of differences: the most important of these is the *two-occurrence restriction* of Janus. This restriction states, essentially, that in any clause, a variable whose value cannot be inferred to be "fixed" (i.e., ground) from the guard operations is allowed to have at most two occurrences: one of these occurrences is annotated to be the "writable" occurrence, and the other is the "readable" occurrence. Only the writable occurrence of a variable may be assigned to. Thus, variables in effect serve as point-to-point communication channels; other language constructs allow many-to-one and one-to-many communication.

The two-occurrence restriction is motivated strongly by a vision of distributed constraint programming. A fundamental concern is that syntactically correct programs should not cause the store to become inconsistent at runtime: this is enforced by the two-occurrence restriction, which ensures that any variable has exactly one producer, thereby precluding any possibility of inconsistency. This has the desirable effect that programs become efficiently implementable (at least in principle). It has been observed that while programs typically do not give rise to a great deal of aliasing, this information is not available to compilers, which have to resort to complicated and potentially expensive algorithms to recover it. The problem is addressed in Janus by specifying the default to be that there is no aliasing, and requiring the programmer to explicitly invoke certain language features when sharing between structures is necessary. Rules for syntactic well-formedness then ensure that the compile-time satisfaction of certain properties, local to a clause, regarding the number of occurrences of a variable imply the run-time satisfaction of certain global properties regarding lack of aliases.

Data objects in Janus consist of the following: askers, tellers, numbers (integers and floats), constants, arrays, lists, and bags. An *asker* for a variable `X` is the "read" occurrence of `X`: it denotes read capability on the communication channel `X` (if we think of a variable as a communication point-to-point channel). A *teller* for a variable `X`, written `^X`, denotes the "write" occurrence of `X`, i.e., write capability on the channel `X`. An array of n objects $a_0, \ldots, a_{n-1}$, written `<`$a_0, \ldots, a_{n-1}$`>`, represents a sequence of values indexed by $\{0, \ldots, n-1\}$. A list is either the empty list `[]`, or a pair `[H|L]`. A bag represents an unordered multiset of objects, and can be thought of as many-to-one communication channels. Ask constraints in Janus consist of various type tests and relational tests on objects (and, via selectors, to components of objects). A tell constraint is restricted to be of the form `X = E`, where `X` is a variable for which the agent has tell rights, and `E` can be any expression including arithmetic, array, and bag expressions.

2 The Janus Virtual Machine

2.1 Values

Janus values are represented in a single word consisting of a tag and a data portion. For integers, the data portion is the integer itself. For atoms (symbols, not atomic clauses) the data portion is a unique integer that can be looked up in a table to find the representation of the atom. For floats, the data portion is a pointer to a memory block (on the SPARC, for example, this is a 32-bit word) containing a floating point number. For lists, the data portion is a pointer to a pair of tagged values. For arrays, the data portion is a pointer to an array block. The array block contains the length of the array and a pointer to the sequence of tagged values that make up the array. For tellers, the data portion is a pointer to the corresponding asker. For askers, the data portion is a pointer to itself or to a lower value in the reference chain.

2.2 Memory Management

The Janus runtime system has two memory regions, the stack and the heap. The stack contains environments (also called stack frames), which contain a set of local variables in the form of tagged values. The heap contains tagged values as well as floating point numbers, suspension records, arrays and other sorts of data that do not fit into stack allocation. Currently, there is no garbage collection for the heap. The bottom of the stack begins at an address lower than the heap and grows toward higher memory while the heap grows toward lower memory. This makes it possible to check the allocation of stack and heap space at the same time with a single comparison of the stack and heap pointers.

Environments are allocated only at commit points: guard operations are carried out entirely in registers. When a clause commits, an environment of the appropriate size (for that clause) is allocated if necessary. The allocation and deallocation of environments is similar to the WAM, but with some important differences. First, the test to see if there is enough space for a stack frame is separate from the actual allocation of the stack frame. This allows us to combine the tests for adequate heap and stack space in a clause, so that for most procedures there is only one test that accounts for all the space that will be needed (for the stack as well as the heap) in a clause. Another difference from the WAM is that that rather than saving a "return address"—a pointer into the code of some other procedure—to which control should be transferred when the current activation has finished execution, an environment for a procedure contains a "resumption address"—a pointer into its own code—where execution should resume when control returns to it. This simplifies the management of control in the presence of arbitrary suspensions and resumptions.

Another (minor) difference from the WAM is that `jc` uses a stack pointer `sp` (to the top of the stack) rather than an environment pointer (to the base of the topmost environment). Local variables are referenced by negative offsets from `sp` and the resumption address is the top word on the stack. There is no pointer to the previous stack frame, instead environment allocation and deallocation uses a matched pair of instructions `NewFrame(`*i*`), ..., FreeFrame(`*i*`)` where i is the number of variables in the stack frame. This scheme, like the WAM, leaves the stack with no self-defining structure. In other words, there is no way to divide the stack into frames by following pointers. This is not needed for allocating and deallocating environments, but debugging and garbage collection procedures need to know the structure of the stack so there has to be a way to re-create it. To allow this, the resumption addresses are given special tags to distinguish them from other tagged values. Since there is a resumption address at the top of each frame (except possibly the topmost frame), it is possible to divide the stack into environments by looking for tagged resumption addresses. Since currently resumption addresses are arbitrary integers, there is no run-time overhead associated with tagging and untagging them, the tagged value is just used all the time.

2.3 Registers

The virtual machine has a set of registers that are simply C variables, but many of them are declared as "register" and mapped to hardware registers. The special-purpose registers include the usual stack pointer and heap pointer. In addition, we have a register `cs` that is used to reduce the overhead of suspension (see below): this is a pointer to the current suspension record (or `NULL` if there is no such structure). There are a number of special registers that handle suspension and resumption of activations. Finally, there are several sets of general-purpose registers: *tagged value registers*, used to hold tagged values and to pass parameters to procedures (the parameter passing convention is similar to the WAM); *address registers*, used to hold machine addresses; *integer registers*, used to hold untagged integer values; and *float registers*, used to hold floating point values. There is no *a priori* bound on the number of general purpose registers: the Janus compiler generates a distinct C variable for each register needed by a program (some subset of these are given `register` declarations, based on usage counts, as discussed later).

2.4 Arithmetic

Integer values are represented as 32-bit words (30-bit value + 2-bit tag), while floating point values are represented by a tagged pointer to an untagged floating-point number, in the machine representation, allocated in the heap. The runtime system has the usual complement of arithmetic instructions. These instructions come in a number of different versions, for different types of operands and results (tagged/untagged, integer/float).

Conditional instructions in the virtual machine contain the label as one of the operands, and the jump is made immediately if the condition is met. There is a pair of conditional instructions for each type of tag, one that jumps if the tag is right, one that jumps if it is wrong.

There is also the complete set of arithmetic conditions: `EQ`, `NE`, `LT`, `LE`, `GT`, and `GE`. Like the arithmetic operations, these instructions also come in a number of different versions for comparing tagged values, tagged integers, tagged floats, untagged integers, untagged floats, and addresses. The conditions on general tagged values are all implemented as C functions to reduce code bloat.

2.5 Procedures

Parameters are all passed in tagged value registers (`t0`, `t1`, `t2`, ...). Before calling an n-ary procedure, the first n tagged value registers must be set to the n actual parameters. Then if the call is not a tail call, the stack frame must be updated to contain the resumption address of the calling procedure. When a procedure exits it just jumps to the resumption address of the top frame on the stack. The bottom frame on the stack is always initialized with a resumption address that exits the Janus portion of the program (returning to the top-level input mode).

Resumption addresses present a problem when compiling to C because in C, labels are not first-class objects. They are not values that can be assigned to variables, and it is not legal to write `goto` e where e is an expression other than a label name. One possible way to solve this problem is to use separate C functions for Janus procedures, since C does allow function-valued variables. But that solution is unsatisfactory for several reasons. First, the overhead of C procedure calls is much higher (for most compilers) than what is needed for Janus. Second, most C compilers do not do tail-call optimization. Third, such a strategy would preclude several other types of optimization such as call-forwarding and optimizations involving where a resumed procedure starts.

A less unsatisfactory solution, the one currently used in `jc`, is to use the C switch statement to give a form of "computed goto". Any address that must be stored in memory and calculated dynamically is implemented as a small integer constant (tagged as an address), and the "label" for that address is a case label with that value. However, addresses of procedures are known statically, so the jump instructions in procedure calls can have the label "hard-wired" in.

The code for the clauses of a procedure are generated contiguously, in sequence. Each guard test jumps to the next guard on failure.[1] If the last guard fails it jumps to the suspension code. After the guard code for a clause, at the "commit" point, is code to allocate an environment for that clause (if necessary). Also at the commit point is code that checks whether there is adequate space in the heap to satisfy all allocation requirements of the tell actions appearing in the body of the clause. After these allocation checks is code for in-line tell actions. Only after all possible in-line work is done are there any procedure calls.

2.6 Suspension and Resumption

In general, before a guard can test the value of a variable, it must make sure that the variable is instantiated to a value. If not, then the flag register `susp_flag` is set and there is a jump to the next guard or the suspension code. This action is all handled by the single instruction `CheckIfAsker(lbl,var)`. If the flag `susp_flag` is set when control reaches the suspension code, then it is known that at least one guard was unable to finish due to an uninstantiated variable, so the procedure must suspend until that variable becomes known. Otherwise, all guards completed and failed, so the procedure vanishes.

As a special case, a test on a parameter does not need to be proceeded by a `CheckIfAsker` instruction. For parameters, the guard simply checks the type or value of the the variable and jumps to the next guard if the variable does not satisfy the test. Such tests will always fail if the variable is not instantiated. Then the suspension code must check each such parameter to see if it is instantiated as well as checking `susp_flag`. This strategy moves work out of the normal non-suspending code and into the suspension code, so that procedures that do not suspend do not have to pay a high performance cost for the language's suspension feature.

But this strategy does not work for variables that are not parameters, because there is no guarantee that the variable is available when the suspension code gets executed. The reason is that any non-parameter variable must get assigned some-

[1] We currently do not have decision tree compilation implemented: we are investigating decision tree compilation in the presence of execution weights, see [5] for details.

where in the guard, and the assignment will never be made if a preceeding test in the same guard fails. For example the guard of the clause

```
p(L) :- L = [H|T], int(H) | ...
```

(supposing it is the second guard) gets expanded into

```
G2_lbl:
JumpUnlessLcons(G3_lbl,t0)
GetCons(a0,t0)
Load(t1,a0[0])
JumpUnlessInt(t1)
...
G3_lbl:
...
```

Notice the use of `t0` and `t1`, the tagged-value registers. The single formal parameter is initially in `t0`. In the code above, it is clear that if p/1 is called with an asker as its actual parameter, then the register `t1` never gets assigned a value. So the suspension code cannot test `t1` to check whether it is an asker.

When a procedure needs to suspend, it copies its parameters to the heap, then creates a suspension record in the heap, setting the special register `cs` (current suspension) to point to it. The suspension record contains a continuation address, a pointer to the parameters in the heap, a number telling how many parameters there are, and a pointer to the next suspension in a list of suspensions. Then there is a jump to `suspension_label` where `cs` is prepended to the list `suspension_list` of suspended activations.

When a suspension record is resumed, the register `cs` is set to point to the suspension record. Then the parameters are copied into the registers, and execution continues at the beginning of the procedure. If the procedure suspends again, the `cs` register is already pointing to the correct suspension record, so it is not necessary to create or initialize a new one. The suspension code of each procedure contains a test at the beginning that jumps directly to `suspension_label` if `cs` is non-zero, thereby avoiding the creation of a new suspension record. This greatly reduces the overhead of a program that has procedures suspending and resuming many times before they commit.

An early design decision in the `jc` system was that programs that did not need to suspend should not, if at all possible, incur any overhead due to the fact that the language allows suspension. One consequence of this decision was to abandon any attempt at fair scheduling: our stance is that programs written by the user should not rely on assumptions about the underlying scheduling strategy being used in an implementation for their correctness. Thus, our strategy for resumption of suspended goals makes no guarantee about when an awakened goal will actually get to execute. This allows us to optimize the implementation considerably: a tell action does not have to check whether it is awakening a suspended goal, and other common primitive operations also do not have to contend with suspensions. Indeed, there is no special tag indicating a "variable that has a procedure waiting for it to be instantiated".

When the stack is empty and there are no suspension records on `rl`, the `suspension_list` is copied to the `rl` and execution proceeds at the scheduler. The scheduler is simply a piece of code that checks if the register `rl` (the resume list) is non-empty. If so, then a suspension record is removed from the list and resumed as described above. If `rl` is empty, then execution continues at the resume address of the top frame on the stack. If a list of suspended procedures is resumed in this way and none of them makes any progress, it is not a good idea to repeat the process since they will continue to make no progress and an infinite loop will result. To detect this, there is another field in the suspension record, `was_resumed` that is 0 when the the record is first created and gets set to 1 when it is resumed. If any element in the `suspension_list` has a `was_resumed` field equal to 0, then that resumption is new.

Searching the entire list of suspensions to see if there is a new one can get expensive, so there are two more registers, `num_suspensions`, which counts the number of suspensions that have occurred since the last batch resumption, and `num_resumptions` which tells how many suspension records were resumed in the last batch resumption. If `num_suspensions` is different from `num_resumptions`, then some progress was made since the last batch resumption, so it is safe to resume again without traversing the list of suspensions to see if there is a new one. If `num_resumptions` is equal to `num_suspensions`, then it is necessary to traverse the list, because they may be equal either because no progress was made or because the number of new suspensions happens to be the same as the number of commits. Since these operations are carried out only on suspension and resumption, non-suspending code incurs (almost) no overhead for the possibility of suspension.

3 The Janus Compiler

The Janus compiler has been built with standard tools: the scanner is generated by lex, the parser by yacc. This is augmented by a phase that does some transformations at the syntax tree level, a code generator that generates Janus virtual machine code, and a code optimizer.

3.1 Program Transformations

The syntax tree transformations performed by the compiler can be subdivided into three groups:

Suspension-Related Transformations: These are concerned with simplifying the implementation of tell actions in the body of a clause that might suspend. To simplify the implementation, we do not want a clause to suspend once it has committed. Therefore, we create extra predicates for tell actions in the body of a clause that might have to suspend. These new predicates are generated so that they can only suspend in the guard, and not in the body: this allows all suspension to be dealt with in a uniform way, and simplifies the implementation. For example, in the factorial program

```
fact(N,^F) :- int(N), N > 0 | fact(N-1,^F1), F = F1*N.
fact(0,^1).
```

the multiplication `F1*N` is only allowed when `F1` is known, but that depends on the condition that the recursive call to `fact/3` does not suspend. Therefore, this program is transformed to

```
fact(N,^F) :- int(N), N > 0 | fact(N-1,^F1), fact__1(F,F1,N).
fact(0,^1).
fact__1(F,F1,N) :- number(F1) | F := F1*N.
```

Note that the new program is not a legal user program: the operation `:=`, which is an assignment directly to an asker rather than to the associated teller, is not available to users, and the predicate `fact_1` can be seen to be assigning to a variable without checking whether it is allowed to do so (i.e., without checking for a "teller" annotation). This works because the transformation is carried out on the syntax tree of the program, after any syntax checking is carried out. The point here is that the compiler knows that the variable `F` has been checked for a teller annotation in the clause defining `fact`/2, so it is not necessary to repeat this check in `fact_1`/3. In general, since such auxiliary predicates are automatically generated by the compiler, they are generated in the most efficient way (maximal reuse of already available argument registers, maximal use of the information available at the call site, and omission of unneeded tests).

An alternative to this approach would be to implement virtual machine instructions operations, that can suspend if necessary—this is the approach taken in the implementation of KL1 [2]. This approach has the advantage, compared to ours, that it saves a (Janus) procedure call, but it requires a more complicated suspension scheme. Also, unless special non-suspending versions of the body goals are provided, this approach will add tests to every body goal that might suspend, which will slow down the execution.

Expression Flattening and Common Subexpression Elimination: Complex expressions are broken up into smaller pieces, which are assigned to new variables. These assignments are put immediately before the goal they are extracted from. Common expressions are merged to avoid unnecessary computations.

Goal Reordering: In order to avoid unnecessary suspension, goals are reordered, wherever possible, such that producers are generated before consumers. This transformation also has the advantage that registers can be reused more frequently because variables are only used during a short period. As part of this transformation, tell actions are moved to the beginning of the body, if possible. Having most of the tell actions at the beginning of the body goals makes register allocation somewhat more difficult, but allows better reuse of partial results, especially values in untagged registers (addresses, integer and floating point).

3.2 Code Generation

The code generator scans the syntax tree and generates code on a procedure by procedure basis. Its register allocator that is somewhat different from the ones described in literature (e.g., see [3, 8]). It uses four kinds of registers: ordinary tagged registers, untagged address registers, untagged integer registers, and untagged floating point registers. Once a variable has been untagged for any reason, its untagged version is kept in an untagged register as long as the variable is in use. These untagged registers are actually just local registers in a chunk: they are neither saved on the stack nor used as arguments.

This feature turns out to be interesting, especially for addresses of structures and the length of arrays. Since subexpressions are extracted and merged, they only have to be computed once, but can be used many times. The fact that untagged values are available for reuse contributes to the efficiency: for example, multiple references to an array element, as in the `quicksort` benchmark, do not cause its address to be repeatedly recomputed—instead, it is computed once into an untagged register and reused subsequently. The code generator not only tries to minimize the number of instructions, but also the number of registers. The number of hardware registers is typically limited to about 10, and the code generator reuses registers as much as possible. It keeps a static reference count of the Janus registers (tagged, as well as untagged), and the n most frequently used registers (where n depends on the target machine) are stored into hardware registers. In many Prolog implementations, by contrast, a fixed set (typically the first six or eight) "general purpose" registers are mapped to hardware registers regardless of their usage counts—an approach that may very well be suboptimal, since in our experiments we often observed more references to (untagged) address registers than to some of the first few general purpose registers.

The register allocator uses a "lazy" allocation algorithm. This essentially means that it tries to postpone the physical allocation of a register as long as possible. Therefore, assignments to a variables are not generated, but recorded internally. In case a bound expression was assigned, it can better immediately be used at the second (and last) occurrence. For example, some variables that occur only once are never generated or even assigned a register. Registers are only allocated 'on demand'.

For our current benchmark set, we end up with 90% of all the Janus register references in hardware registers by using only 8 hardware registers. The use of 10 registers covers 95% of all the register references in the benchmark set. Of course, this does not prove that 95% of the run-time register references will be covered too. Actually, however, we expect the run-time reference coverage even higher because the 5% non-covered registers occur predominantly in rarely used clauses or in the body of a clause. Most of the variables used in guards such as arguments are allocated to hardware registers. The number of hardware registers that is actually used depends on the target architecture.

3.3 Code Optimization

The code optimizer uses a special optimization called *Call Forwarding* to remove some redundant computation. The basic idea here is to generate procedures with multiple entry points: at any call site for a procedure, information specific to that call site (obtained, for example, from the guard tests preceding that call or the operations that created the actual parameters) can be used to bypass some guard tests at the callee and jump into the middle of the callee's guard tests. In order to do so, the compiler keeps track of information (mainly type information) about the contents of the Janus registers at each call site. This information is used to generate a call instruction that skips as many tests as possible at the called guard. Since many guard operations just check the type of the arguments, and since this information is often readily available at the call site, this optimization allows a call site to avoid executing many guard tests. The performance improvements resulting from this relatively simple local optimization turn out to be quite remarkable.[2] The smaller the body of the clause w.r.t. the size of the guard, the higher the savings of this optimization: the optimization is especially successful in case of small recursive predicates such as naive reverse and factorial. It is interesting to note that the optimization improves the performance of non-suspending code (in which case some tests are skipped) as well as suspending code (in which case execution is sometimes transferred directly into the suspension code).

The effects of call forwarding can be enhanced by another local optimization called *Jump Target Duplication*. This optimization replaces an unconditional jump by the target of the jump, followed by a jump to the instruction following the target. This transformation does not, of itself, improve efficiency; however, it can allow the call forwarding algorithm to skip some extra tests that could not have

[2] Initially, when we began the implementation, we expected that serious global dataflow analysis would be necessary to get reasonable performance—particularly in light of the fact that extensive suspension testing seemed necessary in an ask/tell language such as Janus. The improvements from our local optimizations have so surpassed our expectations that the need for global flow analysis and optimization, while not entirely eliminated, seems quite a bit less pressing at this time.

been skipped otherwise. Jump target duplication is especially useful in the special case where the jump target is a conditional jump. This situation arises due to last goal optimization, a generalization of tail recursion optimization, since the call to the last goal in a clause translates to a jump to the beginning of the code for the corresponding predicate, and this code typically consists of conditional jumps arising from guard tests. In such cases, an unconditional jump is replaced by a conditional jump that jumps directly into a particular clause: this saves one jump instruction. Repeated application can replace a procedure call by a partial decision tree.

Repeated application of jump target duplication on body goals of the called predicate will end up in an inline call of the predicate, so to prevent reduce code bloat, (not to mention infinite expansion on recursive predicates), this feature must be restricted to a limited number of instructions. In the current implementation, there is one pass of jump target duplication, where duplication is carried out only if it will allow further optimization from call forwarding.

As an important side effect, the information that is gathered about the variables is also used for other purposes. For example, specific type information (e.g., whether an operand is guaranteed to be an integer) about variables allows general arithmetic instructions to be replaced by specialized integer and floating point instructions. Other optimizations that use this information include the removal of redundant dereferencing instructions and globalizing ("put-unsafe") instructions. The net result is that the compiler consists of a small number of simple and efficient (and reliable) components, and is able to carry out significant optimizations and realize good performance, without having to deal with complicated, computationally expensive, and potentially fragile dataflow analyses.

The final optimization on intermediate code, called *Instruction-Pair Motion* involves instructions that reverse or nullify the effect of a previous instruction. The simplest case arises when there are two contiguous instructions that are complementary, e.g.:

```
..., NewFrame(6), FreeFrame(6), ...
```

Here, these instructions are analogous to the `allocate` and `deallocate` instructions, respectively, of the WAM. Clearly, the first instruction can be eliminated because the second instruction will immediately undo its effects. This optimization can be generalized to situations where the instructions under consideration are separated by a nonempty instruction sequence, in some cases containing (conditional or unconditional) jumps (see also the discussion in [4]). There are two main effects of this optimization. First, it is often the case that variables are initialized, but this action is wasted because the initialization value is subsequently overwritten without ever being used. Such useless initialization of variables can be avoided using this optimization. A similar optimization is described in [14], based on a global dataflow analysis. Our approach, which relies on local (intra-procedural) analysis instead, is simpler and more efficient, but does not work in as many cases. It is, however, quite effective for loops, where such optimizations are likely to be most effective in terms of performance improvement.

Second, in tail-recursive predicates (encoding iterative computations), most of the execution time is usually spent in the recursive clauses. In most nontrivial cases, such clauses require an environment to be allocated. Under the standard WAM model of allocation, this causes an environment to be allocated and deallocated each time around the loop, though it might be more efficient to allocate an environment once, use it through the duration of the loop, and deallocate it at the end. This can be realized using the instruction-pair motion optimization. The idea is similar to that discussed in [4]: there are some subtleties regarding the checking of stack/heap overflow, but a detailed discussion is omitted due to space constraints. A similar optimization is described in [10]. The optimizations described here can be seen as generalizing a number of optimizations for traditional imperative languages [1]:

- In the special case of a (conditional or unconditional) jump whose target is a (conditional or unconditional) jump instruction, call forwarding generalizes a flow-of-control optimization that collapses chains of jump instructions. Call forwarding is able to deal with conditional jumps to conditional jumps (this turns out to be an important source of performance improvement in practice), while traditional compilers for imperative languages such as C and Fortran deal only with the case where there is at most one conditional jump (see [1], p. 556).
- When we consider call forwarding and instruction-pair motion for the last goal in a recursive clause, what we get is essentially a generalization of code motion out of loops. The reason it is a generalization is that the code that is bypassed due to call forwarding at a particular goal need not be invariant with respect to the entire loop, as is required in traditional algorithms for invariant code motion out of loops. Moreover, our algorithm implements inter-procedural optimization and can deal with both direct and mutual recursion without having to do anything special, while traditional code motion algorithms handle only the intra-procedural case.
- When call forwarding is combined with jump target duplication, we get a generalization of subprogram inlining. The reason it is a generalization is that the extent of inlining can be controlled by limiting the number of instructions duplicated from the jump target, thus allowing "partial inlining."
- Call forwarding very often skips type tests in the guard, such as teller tests, integer tests, and floating point number tests. Usually it is possible to get integer arithmetic nearly as fast as machine arithmetic just by "declaring" the operands as integers in the guard, and depending on call forwarding to skip the tests. This goes a long way toward overcoming the performance problems of dynamic typing without requiring global type inference.

In addition, call forwarding is a useful addition to implementation techniques, such as decision tree compilation, that have been studied for committed choice languages [9], since it allows optimizations at call-sites rather than just at the callee. Thus, in the code for the predicate `fact`/2 given earlier, the type of the first argument is tested in each clause, and decision tree compilation would execute this just once; however, this test would be repeated at each recursive call. Using call forwarding, the repeated tests at each recursive call would be avoided, because the code generated for the recursive call would simply bypass this test.

4 An Example of the Code Generated

Figure 1 shows the code generated for the following Janus program:

```
app([H|L1],L2,^Z) :-  Z=[H|L3], app(L1,L2,^L3).
app([],L,^Z) :- Z=L.
```

The first column is the optimized Janus virtual machine (JVM) code, while the second is the corresponding C code generated (some C macros have not been expanded due to space constraints, but their effects are described below and their implementation should be intuitively obvious). The `jc` virtual machine instructions are finer-grained than those for the WAM; in fact most of them would have simple expansions directly into assembly language. The label `L3` is the beginning of the suspension code (not shown). The tag operation macros are left unexpanded to make the C code easier to read, and in any case they are quite typical of such operations. Each `is_`*type*`()` macro involves a single mask and comparison. Each `get_`*type*`()` and `tag_`*type*`()` for tellers, and integers involves a single mask *or* shift, the other types require both a mask and a shift. Askers, tellers, and integers use two-bit tags, the other types all use 5 bits. Askers have the tag 00, so they require no work to either tag or untag. All `get_asker(t`*i*`)` macros in the code have been expanded to `t`*i* but the `tag_asker` macros have been left in.

`OJump(FrameResume)` jumps to the resumption address in the top stack frame. What it actually does is set the value of `nxt_lbl` to a small integer constant then jump to the top of a switch statement

```
top: switch (nxt_lbl) {...}
```

that contains the whole janus program. Each non-tail call is followed by a `case` label on a unique integer. That integer is put on the top of the current frame before a call.

The `Move(cs,0)` at the beginning of the body sets the current suspension `cs` to a null value on commit. This must be done before the current procedure calls another procedure, otherwise if `cs` is non-null and the called procedure suspends, it will suspend on `cs`. The reason this instruction is at the beginning of the body instead of just before a call is so that call forwarding can skip it. another

`MemCheck(`i,j`)` tests to see whether there is room for i words of stace space plus j words of heap space plus the extra word on the stack for storing the resumption address. The reason the C code shows a check for 3 words instead of just 2 is that the extra word on the stack frame is checked for even when no stack is allocated, as in this example.

`MakeTeller(t2,t2)` is implemented as a special tag operation that converts an asker to a teller with one addition. `TELLER` and `ASKER` are the teller and asker tags respectively. They are static constants so the substraction is performed at C compile-time. In the unoptimized code there is a teller made and then an asker is extracted from it in the inner loop, but in the optimized code both operations are moved out of the inner loop.

`MakeSafe(t1)` moves `t1` to the heap if it is on the stack. This is necessary in general to avoid having a lower frame in the stack point to a higher one. Because of the strict directionality of assignment in Janus it is not possible to use the WAM strategy of always just making the higher address point to the lower one. The preceeding `MemCheck()` is needed to make sure there is space in case `t1` needs to be moved to the heap.

5 Performance

The tables below give some indication of the current level of performance of the system. The host machine in all cases is a Sun 4/60 (SPARCstation-1) with 16 MB of main memory. It should be emphasized that this is by no means a finished system: there are a number of optimizations that we have not had time to implement.

Table 1 compares the speed of the code produced by the `jc` compiler with the speed of the same program written in Prolog and executed on Sicstus Prolog version 2.1 (compiling to native code) and Quintus Prolog version 3.1.1. In each case, the time reported, in milliseconds, is the time taken to execute the program once. This time was obtained by iterating the program long enough to eliminate

Optimized JVM code

```
                                    %%% CODE FOR CLAUSE 1
L0:JumpUnlessTeller(L2,t2)          % Is arg 3 a teller?
   JumpUnlessLcons(L5,t0)           % Is arg 1 a cons cell?
   Move(cs,0)                       %
   GetAsker(t2)                     % Remove teller tag
L4:MemCheck(0,2)                    % Enough space for new cons?
   GetCons(a0,t0)                   % Get address of old cons
   Load(t3,a0[0])                   % Get car of old cons
   Load(t0,a0[1])                   % Get cdr of old cons
   HeapSpace(2)                     % Allocate space for cons
   Assign(t2,tag_lcons(hp))         % Assign new cons to arg 3
   Store(t3,hp[0])                  % Set car of new cons
   MakeAsker(t2,hp[1])              % Make a reference to cdr
   Deref(t0)                        % Deref the cdr of old cons
                                    %
   JumpIfLcons(L4,t0)               % Loop if old cdr is a cons
   Store(t2,hp[1])                  % Initialize new cdr
   JumpIfEqual(L6,t0,C_nil)         % Exit if old cdr is nil
   MakeTeller(t2,t2)                % Make a teller for new cdr
   Jump(L2)                         %
                                    %%% CODE FOR CLAUSE 2
L1:JumpUnlessTeller(L2,t2)          % Teller test for clause 2
L5:JumpUnlessEqual(L2,t0,C_nil)     % Is arg 1 nil?
   GetAsker(t2)                     % Change arg 3 to an address
L6:MemCheck(0,1)                    % Check space for MakeSafe
   MakeSafe(t1)                     % Make sure arg 2 is safe
                                    %
   Assign(t2,t1)                    % Assign arg 2 to arg 3
   OJump(FrameResume)               % Return from procedure
L2:JumpUnlessKnown(L3,t0)           % Suspend if arg 1 unbound
   JumpUnlessKnown(L3,t2)           % Suspend if arg 3 unbound
```

Expanded C Code

```
L0:if (!is_teller(t2)) goto L2;
   if (!is_lcons(t0)) goto L5;
   cs = 0;
   for (t2=*get_teller(t2);t2 != *t2;t2=*t2);
L4:if (sp + 3 >= hp) mem_error();
   a0 = get_lcons(t0);
   t3 = a0[0];
   t0 = a0[1];
   hp -= 2;
   *t2 = tag_lcons(hp);
   hp[0] = t3;
     t2 = tag_asker(&hp[1]);
   {register tagval *v; if (is_asker(t0))
    do t0=*(v=t0); while(is_asker(t0)&&v!=t0);}
   if (is_lcons(t0)) goto L4;
   hp[1] = t2;
   if (t0 == C_nil) goto L6;
   t2 = t2 + (TELLER - ASKER);
   goto L2;
L1:if (!is_teller(t2)) goto L2;
L5:if (t0 != C_nil) goto L2;
   for (t2=*get_teller(t2);t2 != *t2;t2=*t2);
L6:if (sp + 2 >= hp) mem_error();
   if (t1 < sp && is_varptr(t1))
     {hp-=1;*hp=*get_varptr(t1)=hp;t1=hp|(t1&3);}
   *t2 = t1;
   nxt_lbl = ((frame)sp)->resume; goto top;
L2:if (!is_known(t0)) goto L3;
     if (!is_known(t2)) goto L3;
```

Figure 1: Code generated for **append**/3 in the *naive reverse* benchmark

Program	jc (J) (ms)	Sicstus (S) (ms)	S/J	Quintus (Q) (ms)	Q/J
hanoi	182	300	1.6	690	3.4
tak	267	730	2.7	2200	8.2
nrev	0.729	1.8	2.5	7.9	11
qsort	2.03	5.1	2.5	9.4	4.6
factorial	0.0494	0.44	8.9	0.27	5.5

Table 1: The Performance of `jc`, compared with Sicstus and Quintus Prolog

Program	unoptimized (ms)	optimized (ms)	% improvement
hanoi	283	182	36
tak	487	138	72
nrev	2.07	0.729	65
qsort	3.57	2.03	43
factorial	0.0678	0.0494	27
merge	1.19	0.623	46
susp	43.5	26.8	38
dnf	0.628	0.217	65

Table 2: Speed Improvements due to the optimizations

most effects due to multiprogramming. The experiments were repeated 20 times for each benchmark on each system and in each case, the average time was taken. The benchmarks tested were the following:

`nrev` – *naive reverse*: 1000 iterations on a list of length 30.
`qsort`– *quicksort*: 100 iterations on a list of length 50.
`tak`– the "Takeuchi" benchmark: we timed the call `tak(18, 12, 6, _)`.
`hanoi` – The Towers of Hanoi program: `hanoi(13)`. Adapted from [7].
`factorial` – A program to compute the factorial of 12.

The Janus code is typically more than twice as fast as the Sicstus Prolog, and four to eight times faster than Quintus Prolog. Table 2 gives the improvements in speed resulting from the optimizations described at the end of the previous section. Here we include three more benchmarks: `susp`, a program that suspends and resumes repeatedly because the consumer is always scheduled ahead of the producer. The `merge` benchmark is the usual nondeterministic "merge" program. The `dnf` benchmark is an array based implementation of the "Dutch national flag" problem (see [12]). The combined optimizations (call forwarding, jump target duplication, and instruction-pair motion), give rise to a speed improvements typically ranging from 30%–65%. For well written Janus clauses, the overhead of the guard tests that do not play a role in committing a particular clause, but are just written as a kind of sanity check for the arguments to have the proper type, is almost completely optimized away.

Finally, Table 3 compares the performance of our Janus system with C code for some small benchmarks. Again, these were run on a SPARCstation 1, with `cc` as the C compiler. We tested only programs where we felt C could "compete fairly" — i.e., we did not test programs such as `nrev` or `merge`, where the cost of memory allocation via `malloc()` would have crippled the performance of the C programs and produced misleading results. The programs were written in the style one would

Program	Janus (ms)	C (unopt) (ms)	C (opt: `-O4`)
`qsort`[3]	1.33	1.25	0.34
`tak`	267	208	72
`factorial`	0.0494	0.049	0.036

Table 3: The performance of `jc` compared to C

expect of a competent C programmer: no recursion (except in `tak`, where it is hard to avoid), destructive updates, and the use of arrays (in `qsort`).

It can be seen that even without global dataflow analysis and optimizations, we are not very far from the performance of the C code — a factor of 4 in speed from the code produced by optimizing at level `-O4` by a high-quality C compiler such as `cc` on the SPARCstation is not very embarrassing, and we expect to close the gap considerably once we implement a number of optimizations that we are now investigating.

6 Conclusions

This paper describes `jc`, a portable and efficient sequential implementation of Janus [12] that compiles down to C. When we began this implementation, we expected that "heavy-duty" global flow analyses and optimizations would be necessary for credible performance, because ask/tell languages involve a great deal of testing for suspension, etc., that is absent in Prolog. Somewhat to our surprise, we discovered that with careful attention to the C code generated by the Janus compiler, and some reasonably simple "local" optimizations such as common subexpression elimination and call forwarding, we can attain reasonably good performance. We expect further performance improvements once we have implemented sophisticated compilation algorithms, such as (weighted) decision trees, heap space reuse, and global flow analysis and optimizations.

An alpha test version of this system, together with some (rudimentary) documentation, is currently available by anonymous FTP from `cs.arizona.edu` in the directory `janus/jc`.

Acknowledgements: Discussions with Mats Carlsson played a very important role in the design of the Janus virtual machine. The system also benefited from discussions with Takashi Chikayama, Ken Kahn, Jacob Levy, and Vijay Saraswat. We are also grateful to Mats Carlsson for helping with the Quintus Prolog benchmarking. The work of the first and third authors was supported in part by the National Science Foundation under grant number CCR-8901283, and that of the second author by the National Fund for Scientific Research of Belgium and by the Belgian National incentive program for fundamental research in Artificial Intelligence, initiated by the Belgian State Prime Minister's office Science Policy Programming.

References

[1] A. V. Aho, R. Sethi and J. D. Ullman, *Compilers – Principles, Techniques and Tools*, Addison-Wesley, 1986.

[3] The Janus version of `qsort` used in this table is different from the one used in the previous tables: here, the predicate `split/4` has explicit `int/1` tests in its guards, to be consistent with `int` declarations in the C program and allow a fair comparison. These tests allow additional optimizations in the Janus compiler.

[2] T. Chikayama, personal communication, Feb. 1992.

[3] S. K. Debray, "Register Allocation in a Prolog Machine", *Proc. 1986 IEEE Symposium on Logic Programming*, Salt Lake City, Sept. 1986, pp. 267–275.

[4] S. K. Debray, "A Simple Code Improvement Scheme for Prolog", *J. Logic Programming*, vol. 13 no. 1, May 1992, pp. 57-88.

[5] S. K. Debray, S. Kannan, and M. Paithane, "Weighted Decision Trees", *Proc. Joint International Conference and Symposium on Logic Programming*, Washington, D.C., Nov. 1992 (this volume).

[6] I. Foster and S. Taylor, "Strand: A Practical Parallel Programming Tool", *Proc. 1989 North American Conference on Logic Programming*, Cleveland, Ohio, Oct. 1989, pp. 497-512. MIT Press.

[7] A. Houri and E. Shapiro, "A Sequential Abstract Machine for Flat Concurrent Prolog", in *Concurrent Prolog: Collected Papers*, vol. 2, ed. E. Shapiro, pp. 513-574. MIT Press, 1987.

[8] G. Janssens, B. Demoen, and A. Mariën, "Improving the Register Allocation in WAM by Reordering Unification", *Proc. Fifth International Conference on Logic Programming*, Seattle, Aug. 1988, pp. 1388–1402.

[9] S. Kliger and E. Shapiro, "From Decision Trees to Decision Graphs", *Proc. 1990 North American Conference on Logic Programming*, Austin, Oct. 1990, pp. 97–116. MIT Press.

[10] M. Meier, "Recursion vs. Iteration in Prolog", *Proc. Eighth International Conference on Logic Programming*, Paris, June 1991, pp. 157–169. MIT Press.

[11] V. A. Saraswat, *Concurrent Constraint Programming Languages*, PhD thesis, Dept. of Computer Science, Carnegie-Mellon University, 1989. (To appear in the ACM Doctoral Dissertation Award series, MIT Press.)

[12] V. Saraswat, K. Kahn, and J. Levy, "Janus: A step towards distributed constraint programming", in *Proc. 1990 North American Conference on Logic Programming*, Austin, TX, Oct. 1990, pp. 431-446. MIT Press.

[13] K. Ueda, "Guarded Horn Clauses", in *Concurrent Prolog: Collected Papers*, vol. 1, ed. E. Shapiro, pp. 140-156, 1987. MIT Press.

[14] P. Van Roy and A. M. Despain, "The Benefits of Global Dataflow Analysis for an Optimizing Prolog Compiler", *Proc. 1990 North American Conference on Logic Programming*, Austin, Texas, Oct. 1990, pp. 501–515. MIT Press.