QD-Janus: A Sequential Implementation of Janus in Prolog *

Saumya K. Debray
Department of Computer Science
University of Arizona
Tucson, AZ 85721, USA

Abstract

Janus is a language designed for distributed constraint programming. This paper describes QD-Janus, a sequential implementation of Janus in Prolog. The compiler uses a number of novel analyses and optimizations to improve the performance of the system. The choice of Prolog as the target language for a compiler, while unusual, is motivated by the following: (*i*) the semantic gap between Janus and Prolog is much smaller than that between Janus and, say, C or machine language—this simplifies the compilation process significantly, and makes it possible to develop a system with reasonable performance fairly quickly; (*ii*) recent progress in Prolog implementation techniques, and the development of Prolog systems whose speeds are comparable to those of imperative languages, indicates that the translation to Prolog need not entail a significant performance loss compared to native code compilers; and (*iii*) compilation to Prolog can benefit immediately from a significant body of work on, and implementations of, parallel Prolog systems. Our experience indicates that translation of logic programming languages to Prolog, accompanied by the development of good program analysis and optimization tools, is an effective way to quickly develop flexible and portable implementations with good performance and low cost.

Keywords : logic programming, implementation, compilation, optimization

*This work was supported in part by the National Science Foundation under grant number CCR-8901283.

INTRODUCTION

Janus[1] is an instance of a concurrent constraint programming language.[2] These languages are an elegant generalization of concurrent logic programming languages[3] and constraint logic programming languages.[4] The computational paradigm is one where a set of concurrently executing agents interact with each other via a shared store. Given an underlying constraint system $\mathcal{C}$, such interactions proceed via two primitives: an agent can either *ask* whether a constraint $C \in \mathcal{C}$ is entailed by (the current state of) the store; or an agent can *tell* a constraint $C \in \mathcal{C}$, i.e., add C to the store. A central idea is to maintain a close connection between the logical and operational semantics of programs by using logical entailment for synchronization purposes. This is done via the *blocking ask*, which behaves as follows: given a store S, a blocking ask action $\mathtt{ask}(C)$ succeeds if $S \models C$; fails if $S \models \neg C$; and suspends if $S \not\models C$ and $S \not\models \neg C$, i.e. if there is not enough information in the store to determine whether C is true or not. Subsequently, when enough information has been added to the store by the tell actions of other agents, this ask action is resumed and either succeeds or fails. Tell actions can similarly be made to suspend until *implicit ask* actions associated with them are satisfied. Because low-level operational aspects of concurrent execution, such as synchronization, are defined entirely in terms of high-level concepts such as logical entailment, concurrent constraint programming languages have the very desirable property that the computational behavior of programs closely mirror their declarative semantics.

This paper describes QD-Janus, a sequential implementation of Janus. QD-Janus relies on a translator that compiles Janus programs to Sicstus Prolog: the resulting Prolog code can be compiled to native code using Sicstus Prolog's compiler, and linked with the Janus run-time library, which is also written in Prolog. The choice of Prolog as the target language for a compiler is not entirely usual, though it has been used in the past as a sequential implementation vehicle for concurrent logic programming languages.[5,6,7] A significant aspect of the QD-Janus implementation described here, compared to these earlier works, is the small overhead incurred by QD-Janus compared to the underlying Prolog system: this is due not to any special language feature of Janus, but because of a number of simple but effective compile-time analyses and optimizations carried out by the QD-Janus compiler.

The choice of Prolog as a target language is motivated by a number of reasons: (*i*) the semantic gap between Janus and Prolog is much smaller than that between Janus and, say, C or machine language—this simplifies the compilation process significantly; (*ii*) recent progress in Prolog implementation techniques, and the development of Prolog systems whose speeds are within a small constant factor of that of C programs,[8,9] indicates that the translation to Prolog need not entail a significant performance loss compared to native code compilers; and (*iii*) compilation to Prolog can benefit immediately from a significant body of work on, and implementations of, parallel Prolog systems.[10,11,12,13] The initial aim of this implementation was not necessarily to attain high execution speed, but to quickly provide a simple and portable prototype that would serve as a "base implementation" used to check the correctness of other, more sophisticated implementations, and which could be easily modified to test out ideas for program analyses and optimizations. The performance of the system, nevertheless, is quite good: on the benchmarks tested, the speed of QD-Janus is, on the average, only about 20% slower than that of the underlying Sicstus Prolog system compiling to native code; it is over three times faster, and roughly two orders of magnitude more efficient in heap usage, than an implementation of FCP(:)—a dialect of Flat Concurrent Prolog essentially equivalent to our version of Janus modulo minor differences in the concrete syntax—that has an optimizing compiler and compiles down to a low-level instruction set.[14]

An interesting lesson of this project was that fairly sophisticated language translators can be implemented in Prolog with surprising ease and efficiency. For example, the entire implementation described here was completed by one person in approximately two months. Our experience with another sequential implementation of Janus, this one written entirely in C,[15] suggests that the project would have taken an order of magnitude more time had we relied on a lower level language such as C. This is not entirely surprising, since programming language translation consists largely of tree traversals and pattern matching, which are precisely Prolog's *forte*. The performance of the system, compared to both the underlying Prolog system and a comparable low-level implementation of FCP(:), indicate that because of the speed with which a simple translator can be built to compile a logic programming language to Prolog, the implementor can spend most of his time building effective program analysis and optimization tools, resulting in a low-cost system with good overall performance.[16]

JANUS: AN OVERVIEW

The fundamental primitive operations of Janus are *ask* and *tell* actions. Intuitively, an ask action is either a type test or an arithmetic test, while a tell action constructs a data structure or value. Both ask and tell actions suspend until their operands are "sufficiently instantiated". A Janus program is a set of *guarded clauses* defining its procedures, which, following logic programming terminology, are also called "predicates." A guarded clause for an n-argument procedure p is of the form

$$p(t_1, \ldots, t_n) \;:-\; A_1, \ldots, A_k \mid B_1 \ldots B_m.$$

where the t_i are terms, A_i are ask actions, and B_i are either tell actions or procedure calls. The '|' is the *commit* operator. A procedure may be defined by more than one such clause. The ask actions $A_1, \ldots, A_k$ constitute the *guard* of the clause, while $B_1, \ldots, B_m$ constitute its *body*.

Operationally, the execution of a procedure call $p(u_1, \ldots, u_n)$ proceeds as follows: the guards of each of the clauses for the procedure p are executed in parallel. This can have the following outcomes: (i) there is at least one guard whose ask actions all succeed: in this case, one of the successful guards is nondeterministically chosen to commit, and the remaining guards are killed; (ii) no guard succeeds in committing, but there is at least one guard for which some ask actions have suspended: in this case, the procedure suspends; and (iii) each guard fails: in this case, an error mesasge is given, and execution aborts. Note that since a guard contains only ask actions, this phase of execution cannot change the value of any variable in the environment. If a clause commits, the tell actions and procedure calls in its body are executed in parallel. Procedure calls in the body are executed as described above. If the body is empty, the call returns.

Janus is in many respects similar to Flat GHC[17] and Strand.[18] 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 a constant 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—together with the fact that only one of the occurrences can be annotated as "writable"—ensures that each variable has exactly one producer, thereby precluding any possibility of inconsistency. This has the desirable effect that programs become, at least in principle, efficiently implementable. 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 constructs 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. QD-Janus does not enforce the two-occurrence restriction at compile time. However, no aspect of the QD-Janus implementation depends on the two occurrence restriction, so that effectively, this simply means that the detection of certain errors is postponed to runtime instead of happening at compile time.

Data objects in Janus consist of the following: askers, tellers, numbers (integers and floats), constants, arrays, and bags. An asker for a variable `X` is the "readable" occurrence of `X`: if we think of a variable as a point-to-point communication channel, it denotes read capability on the communication channel `X`. A teller for a variable `X`, written `^X`, denotes the "writable" occurrence of `X`, i.e., write capability on the channel `X`. An array `A` of n objects $a_0, \ldots, a_{n-1}$, written `<`$a_0, \ldots, a_{n-1}$`>`, represents a sequence of values indexed by $\{0, \ldots, n-1\}$. Arrays can also be created via two primitives `array`/2 and `array`/3. Operations on an array `A` include computing its size, denoted by `card(A)`; taking the value at index i, denoted `A.`i; taking subsequences between indices `I` and `J`, denoted by `A[I..J]`; updating the values of `A.`i_1, …, `A.`i_k to $m_1, \ldots, m_k$, denoted by `A[`i_1`->`m_1`, ..., `i_k`->`m_k`]`; and concatenating two arrays `A` and `B`, denoted by `A#B`. A bag represents an unordered multiset of objects. In general, a bag expression is of the form {`E1,...,Em|U1,...,Un`}, and denotes a bag that contains the elements `E1`, …, `Em`, and where the remainder of the bag consists of partitions `U1`, …, `Un`. The intent is that bags serve as many-to-one communication channels: in this case, the intent is that there are `n` writers to the bag, who have access to `U1`, …, `Un` respectively. Ask constraints in Janus consist of various type tests and relational tests on objects and their components. A tell constraint is restricted to be of the form `X = E`, where `E` can be any expression including arithmetic, array, and bag expressions.

SYSTEM ORGANIZATION

The entire QD-Janus system is currently written in Prolog and implemented on top of Sicstus Prolog v2.1.[19] It consists of about 4500 lines of Prolog code, and is available by anonymous FTP from `cs.arizona.edu`.

Janus programs are read in using a version of the public domain Prolog tokenizer and parser by R. A.

O'Keefe, modified to handle lexical and syntactic features of Janus. The translation proceeds as follows:

1. Each clause is transformed into an internal representation that allows various kinds of information to be associated with each literal. This information is later used for various optimizing transformations. The clauses for each predicate are also collected together for ease of later processing.

2. The program is analyzed to obtain information about the suspension behavior of predicates and about the instantiation of their outputs.

3. Each predicate is analyzed to determine whether it is recursive and whether there is any nontrivial common argument annotation, e.g., *teller*, between the head of every clause and every recursive body literal: if there is, such annotations are factored out, and the clauses transformed, so that the annotations are tested once at the entry to the recursive predicate, but need not be either generated or tested on subsequent recursive calls. Such factoring can lead to significant speed improvements.

4. For each predicate, the head and guard of each of its clauses is analyzed to determine the "demand" generated by that clause, i.e., the extent to which its arguments must be instantiated to ensure that execution will not suspend in the guard.

5. The program is analyzed to obtain "calling patterns" for predicates. This information is used to update the demand information computed, so that unnecessary suspension tests are not generated where possible.

6. The demand information is used to translate the clauses for the procedures. There are basically two possibilities that can arise: a procedure is either *unimodal*, i.e., there is exactly one pattern of instantiations for the arguments for which its guards will not suspend; or it is *multimodal*, i.e., there is more than one possible pattern of instantiations for its arguments for which the guards will not suspend. The code generated for unimodal procedures tends to be significantly simpler and more efficient than the code for multimodal procedures.

Even though the QD-Janus compiler is written entirely in Prolog and relies heavily on various analyses and optimizations to improve performance, compilation speed is quite fast, and is dominated by the I/O time for reading in the Janus programs and writing out the translated Prolog code. This indicates that the sophistication of the analyses and optimizations could be improved considerably without noticeably affecting translation speed.

DEMAND ANALYSIS

Demand analysis for a predicate involves traversing the head and guard of each of its clauses to determine the extent to which its arguments have to be instantiated in order for the guard computation to not suspend. This is conceptually analogous to strictness analysis for lazy functional languages over non-flat domains.[20,21] However, there are a number of important differences between our notion of demand analysis and that of strictness analysis, the most important of these being that (*i*) a function can have only one strictness pattern, while a Janus procedure may have more than one pattern of demand; and (*ii*) strictness analysis typically relies on fixpoint computations over programs, whereas the demand analysis used in QD-Janus has the limited aim of detecting the degree of instantiation necessary to allow the guards of a procedure to commit, i.e., it does not contend with possible suspension of body goals, and therefore does not require a fixpoint computation. The patterns of instantiation for which the guards of a procedure will not suspend are called its *activation modes*, or *modes* for short. Note that this usage of the term "mode" is very different from the sense in which it is used for Prolog. It is critical to the remainder of the compilation process, since the classification of predicates as unimodal or multimodal, and the subsequent generation of code, depends on information computed by demand analysis.

The demand of an n-ary predicate p is represented by a pair $\langle \mathit{Inst}, \mathit{Arefs} \rangle$, where *Inst* is a description of how different arguments must be instantiated, and *Arefs* describes array references in the guards. *Inst* is a term $p(u_1, \ldots, u_n)$, where each of the u_i is a tree whose nodes are labelled `d`, or `n`, where '`d`' denotes one level of demand, i.e., requiring evaluation of the top-level functor of the term at that position, and '`n`' denotes no demand at all. It turns out that such trees are not always adequate for describing array references, e.g., for a reference where the index is a variable or an expression. Array references are therefore described separately, using the second component *Arefs*, which is a list of terms $\texttt{aref}(A, I)$ where A is the array being referred to and I is the index. Here, A and I are given in terms of paths down the tree representing the top-level goal. Such a path is expressed as a list of selectors, where a selector is either an array reference $\texttt{aref}(\cdots)$, or a list of integers, where an integer n denotes the n^{th} argument of the subterm at that level.

As an example, consider the following clause, where `X.Y` refers to the $\texttt{Y}^{th}$ element of an array `X`:

```
p(A, I, B, J) :- (A.I).(B.J) > 0 | ...
```

Since this requires that each of the variables `A`, `I`, `B`, and `J` be instantiated, the instantiation component of the demand is given by

```
p(d, d, d, d).
```

In addition, the array elements `A.I`, `B.J`, and `(A.I).(B.J)` must be instantiated. The demand corresponding to the array reference `A.I` is described by `aref([1], [2])`, where `[1]` refers to the first argument position in the top-level goal, i.e., the variable `A`, and `[2]` refers to the second argument position, i.e., the variable `I`. Similarly, the demand for the array reference `B.J` is described by `aref([3], [4])`, and that for `(A.I).(B.J)` by `aref(aref([1], [2]), aref([3], [4]))`. Now the last of these subsumes the first two demands, since any call that satisfies this will also satisfy the other two. Thus, it suffices to specify the array reference demand

```
aref(aref([1], [2]), aref([3], [4])).
```

In general, a clause may need the value of many different arrays, so the array reference demands are specified as a list of such terms. Thus, the result of demand analysis for this clause is the pair

$\langle$ `p(d, d, d, d)`, `[aref(aref([1], [2]), aref([3], [4]))]` $\rangle$.

Input : A set of Janus clauses *Clauses*.

Output : A set of demands D for these clauses.

Method : **return** $D = \cup\{clause\_demand(C) \mid C \in Clauses\}$.

function $clause\_demand(C)$
begin
  propagate equality tests '$X = E$' from the guard into the head where possible;
  let $Head$ = head of C, $Guard$ = guard of C, N = no. of arguments in $Head$;
  $Inst = \langle demand(Head[1]), \ldots, demand(Head[N])\rangle$, **where**
    $demand(T)$ = **if** T is an atom or a bag **then** '`d`';
      else if $T \equiv f(T_1, \ldots, T_n)$ **then** '$\mathtt{d}(D_1, \ldots, D_n)$'
        where $D_i = demand(T_i), 1 \leq i \leq n$;
      else /* T is a variable */ a new variable;
  $Arefs := \emptyset$;
  for each guard test G of $Guard$ **do**
    if G is a test 'E_1 *op* E_2' **then**
      $note\_demand(E_1, Head, Inst, Arefs)$;
      $note\_demand(E_2, Head, Inst, Arefs)$;
    else if G is a type test $\tau(E)$ **then**
      $note\_demand(E, Head, Inst, Arefs)$;
    fi
  od;
  $Inst' := normalize(Inst)$;
  $Arefs' := rem\_redundancies(Arefs)$;
  return $\langle Inst', Arefs'\rangle$;
end

Figure 1: The Algorithm for Demand Analysis

/* *note_demand*$(E, Head, Inst, Arefs)$ takes an expression E and translates this into a demand on argument positions in the head *Head* and notes this demand in the appropriate position in the mode description *Inst*, and any array reference demands in *Arefs*. */

procedure *note_demand*$(E, Head, Inst, Arefs)$
begin
    if E is a variable **then**
        traverse *Head*, *Inst* to find E;   /* heads are linear by definition */
        set the corresponding position of *Inst* to '`d`';
    else if $E \equiv$ '`card(A)`' **then**
        note_demand(`A`, *Head*, *Inst*, *Arefs*);
    else if E is an array reference `A.I` **then**
        note_demand(`A`, *Head*, *Inst*, *Arefs*);
        note_demand(`I`, *Head*, *Inst*, *Arefs*);
        let $Path_1 = selector\_path(Head, A)$, $Path_2 = selector\_path(Head, I)$;
        $Arefs := Arefs \cup \{\texttt{aref}(Path_1, Path_2)\}$;
    fi
end

/* *selector_path*(T, X) returns a path from the root of T to X. */

function *selector_path*(T, X)
begin
    if $X \equiv T$ **return** '`[]`';
    else if X is a number **return** '`num(`T`)`';
    else if X is an array reference `A.I` **then return** '`aref(`$Path_1$`,` $Path_2$`)`',
        where $Path_1 = selector\_path(T, A)$, $Path_2 = selector\_path(T, I)$;
    else if T is a term '`f(`$t_1, \ldots, t_n$`)`' **then**
        if X is a subterm of $t_k, 1 \leq k \leq n$, **then return** '`[`k `|` U`]`'
            where $U = selector\_path(t_k, X)$;
        fi
    fi
end

Figure 2: Auxiliary functions used in Demand Analysis

Figure 1 gives the details of the algorithm, with auxiliary functions described in Figure 2. Here, the notation $t[i]$ denotes the i^{th} element of a term or tuple t. The function *normalize*(I) simplifies demands in a way that allows more of them to be identified. For example, consider the program

```
length([], ^Len) :- Len = 0.
length([_|L], ^Len) :- length(L, ^Len1), Len = Len1 + 1.
```

Straightforward demand analysis infers the demand ⟨ `length(d, d), []` ⟩ for the first clause; for the second clause, the demand inferred is ⟨ `length(d(n, n), d), []` ⟩, since the first argument is required to be a term with a binary constructor. If code were to be generated from this, the predicate `length`/2 would be treated as a multimodal predicate. However, the second clause requires that its first argument be instantiated only at its top level, i.e., only its principal functor is required to be defined. The function *normalize*() recognizes this and rewrites the demand for the second clause to ⟨ `length(d, d), []` ⟩. This is the same as the demand of the first clause. After normalization of demands, therefore, `length`/2 is implemented as a unimodal predicate, which is considerably more efficient than a multimodal implementation.

THE STRUCTURE OF THE GENERATED CODE

Data Representation

Data representation in QD-Janus generally follows that of the underlying Prolog system. The following types, which are found in Janus but not in Prolog, are represented specially: Askers are represented as Prolog variables. Tellers are represented using the function symbol '`^`'/1: a teller for `X` is represented by a term `^X`. An array of size n, $n > 0$, is represented by a term whose principal functor is '`.`'/n, while the empty array is represented by the constant '`[]`': this representation has the benefit that the treatment of lists as arrays of length 2 falls out naturally, and does not have to be handled separately. The representation of bags is similar to that of Prolog lists, except that it is in terms of a constructor '`:`'/2 and a constant '{}'. '{}' represents the empty bag, while a non-empty bag containing the elements $x_1, \ldots, x_n$ is represented as x_1`:`x_2`:` $\cdots$ `:`x_n`:`{}.

Unimodal Procedures

A procedure is said to be *unimodal* if there is exactly one "minimal" pattern of instantiations of its arguments for which its guards can be executed without suspending. As an example, the procedure

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

is unimodal, because each of its clauses requires that the first argument be instantiated and the second argument be a teller.

Consider a Janus predicate `p`/3, defined by the single clause

```
p(X, Y, ^Z) :- X > 0, Y > 0 | Z = X + Y.
```

In order to be able to execute the guard, the first two arguments of this predicate must be instantiated, and the third argument must be annotated as a teller. Thus, all three arguments have to be instantiated to non-variable terms if this predicate is to execute without suspension. The Prolog code generated by the QD-Janus compiler checks this first (predicate names generated by QD-Janus have been greatly simplified in these examples to enhance readability: in practice, they are much more elaborate, in order to avoid name conflicts):

```
p(X, Y, Z) :-
     (nonvar(X), nonvar(Y), nonvar(Z)) ->
          p_worker(X, Y, Z) ;
          p_susp(X, Y, Z).
```

The predicate `p/3` in the Prolog code generated by the QD-Janus compiler is the *sentinel* predicate, whose task is to determine whether all the necessary arguments are sufficiently instantiated. If they are, control is transferred to the *worker* predicate `p_worker/3`, which does the actual computational work of the procedure. Otherwise, if one or more arguments are not sufficiently instantiated, control is transferred to the *suspension handler* `p_susp/3`, whose job is to ensure that the original goal suspends waiting for the appropriate variable to become instantiated.

The structure of the worker predicate `p_worker` closely mirrors that of the original Janus predicate. In this case, for example, it would be:

```
p_worker(X, Y, ^Z) :- X > 0, Y > 0, !, Z is X+Y.
p_worker(X, Y, Z) :- no_guard_match_error(p(X,Y,Z)).
```

The predicate `no_guard_match_error` is called by default if all of the guards of a goal fail, and prints out an error message. This default behavior can be changed by the user, by adding a clause whose guard consists of the single atom `else`. A predicate can contain at most one clause with an `else` guard, and such a clause must be the last of all the clauses defining that predicate. Declaratively, an `else` guard can be interpreted as syntactic sugar for the conjunction of the negations of the guards of the remaining clauses, none of which have an `else` guard. For example, consider the definition

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

Here, the `else` guard can be understood as shorthand for the ask constraint

$$\neg(\mathtt{N} = 0 \ \wedge\ \exists F_1 : \mathtt{F} = \hat{\ } F_1) \wedge \neg(\mathtt{N} > 0 \ \wedge\ \exists F_1 : \mathtt{F} = \hat{\ } F_1).$$

The first conjunct says that of the variables `N` and `F` in the head of this clause, it is not the case that `N` is 0 and `F` is a teller (for some variable F_1), i.e., the guard of the first clause is not satisfied; the second conjunct says that it is not the case that N is greater than 0 and F is a teller, i.e., the guard of the second clause is also not satisfied. In the original definition of Janus,[1] there was no notion of failure: if all the guards for a call failed, the call simply "vanished silently." This was changed to the behavior described above for two reasons: first, it is often the case that failure of all guards of a procedure indicates an abnormal situation, and the behavior described above makes it easier to detect and debug these; second, the approach adopted is strictly more expressive, in the sense that the earlier default behavior of "silent vanishing" can be simulated simply by adding a clause of the form

```
p( ... ) :- else | true.
```

The suspension handler examines its arguments to determine how the suspension should be set up:

```
p_susp(X, Y, Z) :- var(X), !, freeze(X, p(X, Y, Z)).
p_susp(X, Y, Z) :- var(Y), !, freeze(Y, p(X, Y, Z)).
p_susp(X, Y, Z) :- var(Z), !, freeze(Z, p(X, Y, Z)).
```

Strictly speaking, the cut in the last clause is unnecessary, but at this point the compiler has not been engineered to eliminate it.

The reader may notice that this structure can cause a goal to suspend and resume a number of times, resulting in poor execution behavior in some cases. For example, consider a goal `p(U, V, W)` where all of the arguments are variables: this causes the goal to suspend, then U becomes instantiated causing the suspended goal to be resumed and then to suspend again, then V gets a value causing the goal to resume and then suspend again, and finally W gets a value causing the goal to finally resume and continue execution. Such suspension behavior, where a goal has to suspend until each element of a set of variables becomes instantiated, is referred to as "conjunctive suspension." It is possible to handle this by setting up a series of nested `freeze`-goals. However, we chose deliberately to not implement conjunctive suspension in this manner: this decision was due in part to keep the generated code simple, but mostly to mirror decisions in another Janus implementation that is currently under way,[15] where it was felt that conjunctive suspension would be too expensive to implement, and not common enough to justify.

The compiler implements a number of optimizations to the basic translation scheme described above. If a predicate does not need any of its arguments to be instantiated, e.g. if there are no guard tests, then the sentinel and suspension handler are not generated. If a predicate requires only the "top-level" functor of a single argument position to be instantiated, a `freeze`/2 goal is generated "in place" in the sentinel, and the suspension handler is omitted. This avoids the cost of a jump to the suspension handler and also reduces the size of the code generated. As an example, consider the definition

```
fact(N, F) :- integer(N), N >= 0 | fact(N, 1, F).
```

In this case, the code actually generated for the sentinel is

```
fact(N, F) :- nonvar(N) -> fact_worker(N, F) ; freeze(N, fact(N, F)).
```

Because of other optimizations, such as the Invariant Tag Factoring optimization discussed later in this paper, this turns out to be applicable quite often.

Multimodal Procedures

A procedure is said to be *multimodal* if there is more than one pattern of instantiations of its arguments for which its guards can be executed without suspending. As an example, consider the procedure

```
m(0, N, ^T) :- T = N+1.
m(N, 0, ^T) :- T = N+1.
```

This procedure can execute in two modes: either the first and third arguments can be instantiated, in which case the first clause may commit; or the second and third arguments can be instantiated, in which case the second clause may commit. Because it can commit for either of two distinct modes, this procedure is multimodal.

Our experience has been that procedures such as `m/3` above, which are truly nondeterministic in the sense that there are inputs for which more than one clause can be selected, are rare in practice. However, a procedure may be multimodal even if it is deterministic. As an example, consider the following procedure:

```
posintlist([]).
posintlist([N|L]) :- N > 0 | posintlist(L).
```

In order to avoid suspension, the first clause requires simply that its argument be instantiated, i.e., it has the demand ⟨ `posintlist(d), []` ⟩. By contrast, the second clause requires not only that the argument be instantiated, but also the the first sub-argument (corresponding to the variable `N` in the code above) of the argument be instantiated, i.e., has the demand ⟨ `posintlist(d(d,n)), []` ⟩. Because the two demands are different, this procedure is inferred by our implementation to be multimodal. Procedures of this kind, which traverse a list examining each element, are not uncommon, and as a result multimodal procedures are encountered more frequently than one might initially expect.

The structure of the code generated for a multimodal procedure is more complex than that for a unimodal procedure. In order to reduce the amount of testing repeated across code generated for the different modes of a multimodal procedure, they are first examined to see if there is any common demand between them. In the example above, it can be seen that both modes of activation for `m/3` require that the third argument be instantiated. This common demand is factored out to define a sentinel predicate in the generated Prolog code:

```
m(X, Y, Z) :- nonvar(Z) -> m_coord(X, Y, Z) ; freeze(Z, m(X, Y, Z)).
```

The common demand that is tested by the sentinel predicate is "subtracted" from the different modes for the predicate, so that it is not again tested for later.

The predicate `m_coord/3` is the *coordinator*: in an earlier version of QD-Janus, this predicate was responsible for trying the clauses for the predicate so as to determine whether a particular call to that predicate should commit, suspend, or fail. In the current implementation, the coordinator simply transfers control to the first of a series of "handler predicates", each of which is responsible for the clauses of a particular mode. The compiler groups the clauses for a multimodal predicate by mode and orders them so that at runtime, modes corresponding to recursive clauses are processed before those that correspond to only non-recursive clauses. This is consistent with the semantics of Janus, which does not give any special significance to the order of clauses in the source program, and is motivated by a desire to improve the efficiency of recursive procedures.

Each handler predicate—except the first one, as explained below—takes one additional argument, a "suspension mode list" argument *SuspModes* that is bound, at runtime, to a list of modes that may have to be examined if an activation has to suspend. The general behavior of a handler for a mode M of a predicate is as follows: First, check the input arguments to determine whether they are sufficiently instantiated for the mode M. If not, add M to the input mode list *SuspModes* and invoke the handler for the next mode with this extended list of modes; otherwise, process each of the clauses associated with the mode M in turn. If any of these clauses commits, discard the remaining alternatives and mode handlers; otherwise, if every clause for the mode fails, invoke the handler for the next mode with input mode list *SuspModes*. The first mode handler does not require a mode list argument, since no other modes will have been processed before it has executed. The last mode handler calls a procedure that determines whether the activation should suspend. If there is a non-empty list of modes available, this procedure calls a library predicate that takes a list of modes together with a term representing the procedure call being processed, and traverses the modes to determine a set of variables—one for each mode in this list—on which it creates a "disjunctive suspension." This refers to a suspended activation that is resumed as soon as any one of the variables that it is suspended on becomes bound. Otherwise, the mode list is empty at runtime, which means that every non-`else`-guarded clause for that predicate has failed: in this case, the `else`-guarded clause for that predicate is executed if there is one, otherwise a runtime error occurs and execution is aborted. The reason that a mode handler invokes the next handler with an augmented mode list if its inputs are insufficiently instantiated is that a decision to suspend an activation cannot be made until it has been determined that no clause for any of the modes has committed.

As an example, the coordinator and mode handlers for the multimodal predicate `m`/3 discussed above are as follows:

```
m_coord(X, Y, Z) :- m_handler_1(X, Y, Z).

m_handler_1(X, Y, Z) :-              /* handler for mode m(d,n,n) */
     var(X), !, m_handler_2(X, Y, Z, [m(d,n,n)]).            /* suspension case */
m_handler_1(0, N, ^Z) :- !, plus_10(0, N, Z).
m_handler_1(X, Y, Z) :- m_handler_2(X, Y, Z, []).            /* failure case */

m_handler_2(X, Y, Z, SuspModes) :-    /* handler for mode m(n,d,n) */
     var(Y), !, m_susp_chk(X, Y, Z, [m(n,d,n) | SuspModes]).  /* suspension case */
m_handler_2(N, 0, ^Z, _) :- !, plus_10(0, N, Z).
m_handler_2(X, Y, Z, SuspModes) :- m_susp_chk(X, Y, Z, SuspModes). /* failure case */
```

Here, a call `plus_10(X,Y,Z)` suspends until its second argument `Y` is instantiated, after which `Z` is bound to the sum of `X` and `Y`. Notice, however, that in the code above, a call to `plus_01/3` occurs only after a mode handler has committed, and therefore the possible suspension of `plus_01/3` does not affect the suspension behavior of the predicate `m/3`. An examination of the definition of `m/3` indicates that this is the desired operational behavior for this predicate.

PROGRAM ANALYSIS AND OPTIMIZATION

Suspension Analysis

One source of overhead in the implementation described so far is that, because in general there is no information available about when variables can be guaranteed to be instantiated, it is necessary to generate a great many tests to determine whether a computation should suspend because of insufficiently instantiated inputs. With more information about the instantiation of variables at different program points, many of these tests can be eliminated. The QD-Janus compiler uses an inter-procedural dataflow analysis to determine the instantiation of variables at various program points. This information is then used to effect various optimizations.

While the analysis is similar in flavor to *mode analysis* for Prolog,[22,23,24,25] the details are somewhat different. In particular, the analysis is complicated by the fact that procedure calls in a Janus program may suspend. To see the problem, consider the following example:

```
p(N, ^X) :- fact(N, ^F), X = F-1.

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

An inspection of the clauses defining `fact/2` indicates that each clause contains a tell action that assigns a number to the teller variable in the second argument position. A simple-minded analysis might infer from this that a procedure call `fact(N, ^F)` always assigns a number to `F`, and based on this, omit suspension tests from the tell action `X = F-1` in the body of `p/2`. Now consider the execution of the query

```
?- p(N, ^X), N = 5.
```

If the call `p(N, ^X)` is executed first (as it would in QD-Janus), the resulting computation of `fact(N, ^F)` will suspend because `N` is uninstantiated. Because of this, the tell actions in the body of `fact/2` will not have executed, and the variable `F` will still be uninstantiated, when the tell action `X = F-1` in the body of `p/2` is executed. If this tell action is "optimized" by omitting its suspension tests, this will, incorrectly, give rise to a runtime error.

This problem is addressed by performing a simple suspension analysis together with a Prolog-like groundness analysis. The domain of the analysis is very similar to that used for Demand Analysis described earlier. The essential idea is to propagate a set of variables *InstVars*, denoting variables whose values are guaranteed to be known at any program point, across the body of each clause from left to right. Demand analysis is then used to determine, for each procedure call G in the body, whether the guards of the clauses defining G can be guaranteed to not suspend. If this can be guaranteed, and every goal in the body of each clause for G can recursively be guaranteed to not suspend, then it can safely be assumed that all tell actions under G are carried out, and use this information to update the set of variables *InstVars* appropriately. This resembles an analysis algorithm for FCP programs proposed by Gallagher *et al.*,[26] in that the goals in the body of a clause are analyzed from left to right: however, Gallagher et al. ignore the possibility of suspension, which is vital to our analysis; moreover, the ideas in Reference 26 do not appear to have actually been implemented.

The analysis starts with the most general goals for each of the predicates that are declared to be user-callable via a directive `export`/1, e.g.:

```
:- export p/2, q/3, r/1.
```

Note that users are not expected to specify "calling patterns" for the exported predicates: this simplifies specifications and reduces the possibility of bugs arising from inadvertant errors in such user-specified calling patterns. If a file is compiled without an explicit `export` directive, it is assumed that all the procedures defined in that file are user-callable, and no dataflow analysis is carried out. The algorithm is sketched in Figure 3, with the procedure *analyse* described in Figure 4.

Invariant Tag Factoring

It is often the case that certain arguments of a predicate are consistently used with a particular annotation or "tag", e.g. as a teller. For example, consider the predicate

```
qs(A, ^B) :- card(A) =< 1 | B = A.
qs(A, ^B) :- card(A) >  1 |
     split(A, ^K, ^E),
     qs(E[0 .. K-1], ^C), qs(E[K+1 .. card(E)-1], ^D),
     B = C # <E.K | D>.
```

Notice that both clauses of `qs`/2 expect the second argument to be tagged as a teller; and each of the recursive literals for `qs`/2 has the second argument tagged as a teller. This means that once execution has entered `qs`/2 after successfully verifying that the second argument is tagged as a teller, the recursive calls to `qs`/2 will always successfully verify the teller tag of the second argument. For the recursive calls in the body of the second clause of `qs`/2, therefore, it is not necessary to check that the second argument is a teller. This, in turn, implies that for these recursive calls, the construction of the second argument as a teller-tagged term is unnecessary. The generated code can therefore have the following structure:

```
qs(A, ^B) :- qs_f(A, B).

qs_f(A, B) :- card(A) =< 1 | B = A.
qs_f(A, B) :- card(A) >  1 |
```

Input : A Janus program P, a list E of exported predicates, and for each predicate $p \in P$ a list of demands $Demands(p)$.

Output : A pair $\langle Call, Ret \rangle$, where *Call* is a set that describes, for each predicate $p \in P$, which arguments p are guaranteed to be instantiated just before a call to p, and *Ret* is a set that describes, for each predicate $p \in P$, which arguments p are guaranteed to be instantiated when control returns from a call to p.

Method :

$Call := \emptyset$; $Ret := \emptyset$;
for each $p \in P$ **do** $analysing(p) :=$ **false**;
repeat
    $change :=$ **false**;
    for each $p \in E$ **do**
        $analyse(p, C)$ **where** C is a most general calling pattern for p;
    od
until $\neg change$;
return $\langle Call, Ret \rangle$;

Figure 3: The Algorithm for Suspension Analysis

procedure $analyse(p, C)$
begin
  if $\neg analysing(p)$ **then**   /* loop check */
    $analysing(p) :=$ **true**;
    $old\_call_p := call_p$ **if** there is an entry $call_p$ for p in $Call$; $\top$ otherwise;
    $new\_call_p := glb(C, old\_call_p)$;
    if $new\_call_p \neq old\_call_p$ **then**
      update the entry for p in $Call$ to new_call_p;
      $change :=$ **true**;
    fi
    for each clause $C_i \equiv$ '$Head_i$:− $Guard_i \mid Body_i$' of p **do**
      $InstVars :=$ the set of variables that must be instantiated for $Guard_i$ to commit;
      for each $G \equiv q(\bar{t})$ in $Body_i$ **do**
        $analyse(q, C')$ **where** C' is the calling pattern for G given instantiated variables $InstVars$;
        if G **with** $Demands(q)$ will not suspend given $InstVars$ **then**
          use $ret_q \in Ret$ to update $InstVars$;
        fi
      od
      compute the output description ret_i for the head of C_i;
      $old\_ret_p := ret_p$ **if** there is an entry ret_p for p in Ret; $\top$ otherwise;
      $new\_ret_p := glb(ret_i, old\_ret_p)$;
      if $new\_ret_p \neq old\_ret_p$ **then**
        update the entry for p in Ret to new_ret_p;
        $change :=$ **true**;
      fi
    od
    $analysing(p) :=$ **false**;
  fi
end

Figure 4: The Procedure *analyse* used in Suspension Analysis

```
split(A, ^K, ^E),
qs_f(E[0 .. K-1], C), qs_f(E[K+1 .. card(E)-1], D),
B = C # <E.K | D>.
```

Note that in the resulting code, teller tags are neither generated nor checked in the procedure `qs_f/2`.

The transformation of a program to "factor out" such unnecessary tag operations is called Invariant Tag Factoring. It is analogous to invariant code motion out of loops, though it is applicable to recursive calls in general rather than just to iterative computations. The current implementation of tag factoring works only for direct recursion. This is sound, but may be overly conservative in that it may miss some opportunities for factoring. Also, at this time only unary functors are considered for factoring: this catches the annotations for tellers, and possibly other unary function symbols. The algorithm could be generalized to functors with arity greater than 1, but this would introduce additional complexity into the code transformation because of the need to expand arities, and it is not obvious that this situation is encountered often enough to justify this additional complexity.

The algorithm for tag factoring is described in Figure 5. Strictly speaking, it is not necessary to compute most specific generalizations for this optimization. Since we are interested only in obtaining, for each argument position i, $1 \leq i \leq n$, the longest sequence of unary functors, starting at the root, that is common to the i^{th} argument of each literal in L_p, this step of the computation can be simplified.

Reducing Suspension Tests in Clause Guards

As mentioned earlier, suspension analysis produces a calling pattern for each predicate in a program. Since every call to that predicate can be guaranteed to have at least this degree of instantiatiation, this calling pattern can be used to eliminate certain suspension tests in the guard. This is done by subtracting the calling pattern from each mode for that predicate, and generating code as described earlier from the residual modes. Note that, as described earlier, if the calling pattern indicates that all calls are sufficiently instantiated, then there is no residual demand after subtracting out the calling pattern, and as a result no separate sentinel and suspension handling code is generated.

The improvements to the code generated due to this optimization are illustrated by the following program:

```
:- export fact/2.
fact(N, F) :- integer(N) | fact(N, 1, F).

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

If calling patterns were not computed, `fact/3` would have to test its first argument to determine that it was instantiated. However, this test is eliminated when calling patterns are propagated from `fact/2`.

Reducing Suspension Tests in Clause Bodies

In general, tell actions in clause bodies may have associated suspension tests to determine whether they

Input: A list L of Janus clauses for a procedure p/n.

Output: A list of Janus clauses for p, and possibly a new procedure p' resulting from factoring out invariant tags from the clauses for p.

Method:

1. If p is not (direct) recursive, return L.
2. Compute the set L_p of literals in L whose predicate symbol is p. This includes the head of each clause for p, as well as recursive calls to p.
3. Compute the most specific generalization G of the literals L_p. This gives a template of functors that can be factored.
4. If every argument of G is a variable, then no interesting factoring is possible: return L.
5. Define a new predicate p'/n, whose clauses are exactly the clauses for p/n, except that every literal of the form $p(t_1, \ldots, t_n)$, including the heads of clauses, is replaced by a literal of the form $p'(u_1, \ldots, u_n)$. The terms $u_1, \ldots, u_n$ are derived from the terms $t_1, \ldots, t_n$ and the template $G \equiv p(w_1, \ldots, w_n)$, as follows:
   (a) If w_i is a variable then u_i is the same as t_i;
   (b) otherwise, w_i is a term of depth k, then u_i is the depth-k subterm of the corresponding argument t_i.
6. Since only unary functors are being considered, each argument position of the template G will have exactly one occurrence of a variable. Let the variable occurring at the i^{th} argument position of G be denoted by A_i.

   Replace the original definition of p by the clause

   G :- $\mathtt{p}'(A_1, \ldots, A_n)$.

Figure 5: The Algorithm for Invariant Tag Factoring

should suspend because not all of their inputs are available. As an example, consider the clause

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

Without any other information available, the default translation of this clause would have to take into account the fact that the tell action '`F = N * F1`', as well as the subtraction to compute the value `N-1`, might have to suspend. The code generated might therefore have the form

```
fact_1(N, ^F) :- N > 0, !, times(N, F1, F), minus(N, 1, N1), fact_1(N1, ^F1).
```

where calls to the procedures `times`/3 and `minus`/3 suspend until the first and second arguments become bound. However, arithmetic operations implemented in this way can be quite expensive, in part because of the additional testing necessary to determine whether a computation should suspend. Thus, there are two sources of inefficiency in the code given above: first, the additional testing and state transitions involved in executing the `times`/3 and `minus`/3 goals; and second, the potential cost of suspension and resumption for the goal `times(N, F1, F)` in the code above.

As described earlier, during dataflow analysis each literal in a clause is associated with a set of variables whose values can be assumed to be known when that literal is about to be executed. This information is used to eliminate runtime tests for suspension in clause bodies wherever possible. The variables whose values are inferred to be known at various points in the clause for `fact`/2 above are as follows, indicated in comments:

```
fact(N, ^F) :- N > 0 |
     /* known: {N} */ fact(N-1, ^F1),
     /* known: {N,F1} */ F = N*F1. /* known: {N,F1,F} */
```

During code generation, this information is used to reduce the number of suspension tests necessary. In this example, suspension tests for the arithmetic computations in the body can be eliminated entirely, leading to generated code of the form

```
fact_1(N, ^F) :- N1 is N-1, fact_1(N1, ^F1), F is N * F1.
```

Note that the code actually generated would be more efficient, because invariant tag factoring would have eliminated the teller annotation '`^`' in the second argument of `fact_1`/2.

If an operation needs the values of some variables, but these variables cannot be inferred to be known at compile time, code is generated to suspend until these variables become bound at runtime. In the clause for `fact`/2 above, for example, if we were unable to infer that this procedure would bind its second argument to a ground term when invoked—and, therefore, that the value of the variable `F1` would be available when control returned from the recursive call—the code generated would be as follows:

```
fact_1(N, ^F) :- N1 is N-1, fact_1(N1, ^F1), times_10(N1,F1,F).
```

where times_10/3 is a library procedure defined as:

```
:- block times_10(?, -, ?).
times_10(X,Y,Z) :- Z is X*Y.
```

Here, the 'block' declaration for times_10/3 specifies that the execution of any call to this procedure should suspend if the second argument is a variable. Note that suspension tests are not generated for variables that can be guaranteed, at compile time, to be bound.

Optimization of Arithmetic

The QD-Janus compiler tries, as far as possible, to evaluate arithmetic expressions at compile time. This optimization was initially motivated by the fact that arrays in Janus are 0-based, while subterms of compound terms in Prolog are indexed from 1, so that naively generated code for a Janus array reference with constant index, such as A.0, was of the form

```
..., array(A), I is 0+1, arg(I, A, X), ...
```

Obviously, there is no need to generate code to evaluate the expression '0+1' at runtime: it can be evaluated at compile time instead. In order to avoid this kind of unnecessary runtime computation, the compiler implements a general algorithm for reorganizing and optimizing arithmetic expressions. The idea is to use properties such as associativity and commutativity of operators to rearrange expressions to bring constants as "close together" as possible, after which constant subexpressions are evaluated at compile time. The algebraic properties that the system uses include the following: + and * are associative and commutative; 0 is an identity for +, 1 is an identity for *; 0 is a null element for *; E_1-E_2 = E_1+(-E_2); and E_1/E_2 = E_1*(1/E_2). The last two properties are used to transform expressions involving nonassociative operators into expressions with associative operators, in the hope that this may allow expressions to be further restructured and partially evaluated at compile time. The assumption that addition and multiplication are associative and commutative can sometimes lead to expression restructurings that change the results for floating point computations. However, in this respect the QD-Janus compiler behaves no differently from languages such as C, which also consider addition and multiplication to be associative and commutative. In situations where the order of evaluation makes a difference, an order of evaluation can be enforced by using explicit temporary variables, as in the case of C. The arithmetic expression simplifier does not know about the distributivity of * over +. This would be simple to add, but in our experience it is not clear that it would be very useful.

Redundancy Elimination

Because Prolog code is generated via syntax directed translation, there is a lot of redundancy in the code initially generated. The redundant code consists almost entirely of Janus primitives for type tests, array element extraction, etc. A simple scheme is used to eliminate redundant code. The idea is to maintain a set Seen of primitive operations that have already been encountered. Given an operation $I \equiv \mathtt{op(In, Out)}$ in a clause, if there is an operation $I' \equiv \mathtt{op(In', Out')}$ in Seen such that the inputs In and In′ are identical, then their results Out and Out′ must be equal: in this case, Out and Out′ are unified, so that future references to Out now also reference Out′, and I is deleted; otherwise, I has not been encountered before, and is added to

```
split(A, I, Big, ^K, ^Res) :-
      array(A),
      arg(1,A,X),
      I1 is I+1,
      array(A),
      arg(I1,A,Y),
      X =< Y,
      I = Big,
      !,
      array(A),
      arg(1,A,Z),
      I2 is I+1,
      array(A),
      arg(I2,A,U),
      update_array(A,[0->U, I->Z],V),
      Res = V,
      K = I.
```

Before Optimization

```
split(A, I, Big, ^K, ^Res) :-
      array(A),
      arg(1,A,X),
      I1 is I+1,

      arg(I1,A,Y),
      X =< Y,
      I = Big,
      !,






      update_array(A,[0->Y, I->X],V),
      Res = V,
      K = I.
```

After Optimization

Figure 6: An example of the effects of Low-level Redundancy Elimination on generated code

`Seen`. A more detailed discussion can be found in Reference 27. The following clause illustrates the effects of this optimization: the Janus source code considered is a clause from a predicate `split`/5 in a quicksort program:

```
split(A,I,Big,^K,^Res) :- A.0 =< A.I, I = Big | Res = A[0->A.I,I->A.0], K = I.
```

The Prolog code resulting from the translation of the guard and body of this clause, before and after common subexpression elimination, is shown in Figure 6.

PERFORMANCE

This section compares the performance of QD-Janus with (*i*) the underlying Sicstus Prolog implementation, and (*ii*) an implementation of FCP(:), a dialect of Flat Concurrent Prolog, that compiles to a low-level abstract machine instruction set.[14] The underlying hardware platform in each case is a Sparcstation-2. The benchmarks tested are the following:

`nrev(30)` – Naive reverse of a list of length 30.

`hanoi(17)` – The Towers of Hanoi program, translated from Reference 28.

`e` – 10,000 iterations of a program to compute the value of the constant $e = 2.71828\ldots$ to a tolerance of 10^{-6} by summing the series $\sum_{n \geq 1} \frac{1}{n!}$.

`qsort`– quicksort of a list of 50 integers.

`pi` – a program to compute the value of π to a tolerance of 10^{-6} using the identity $\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \cdots$.

`dnf(50)` – A program for the "Dutch national Flags" problem, by V. Saraswat.

`pascal(`n`)` – A program to compute the n^{th} row of Pascal's Triangle.[29] The numbers given are for $n = 200$.

`queen(`n`)` – A program to find all solutions to the n-queens problem.[29] The numbers given are for $n = 8$.

`queen_1(`n`)` – A program to find one solution to the n-queens problem, by S. Kliger.[14] The numbers given are for $n = 8$.

`prime(`n`)` – A program to compute the primes upto n using the Sieve of Eratosthenes.[29] The numbers given are for $n = 10,000$.

`tak` – The Takeuchi benchmark. The numbers given are for `tak(18,12,6,_)`.

`combination` – A combinations program by E. Tick.[29] The numbers given are for `combo(6,[1,2,3,4,5,6],_)`.

`deriv` – A symbolic differentiation program by D. H. D. Warren.

`mastermind` – A mastermind program using naive generate and test.[29] The numbers given are for `go(3,3,_)`.

`nand` – A NAND-gate circuit designer using pipeline filter.[29]

The performance of QD-Janus compared to the underlying Sicstus Prolog v2.1 system compiling to native code is given in Table 1. This indicates the overheads introduced by the execution model of Janus, such as checking whether a procedure should suspend if its inputs are inadequately instantiated, which cannot be removed by the QD-Janus compiler. The figures indicate that this overhead is not very large, averaging about 20% and typically below 50%: indeed, on some programs where the cost of other computations dominates (e.g., floating-point computations for the benchmarks `e` and `pi`, and array management for the `dnf(50)` benchmark), the overheads incurred by QD-Janus are seen to be negligible. Even in a program such as `nrev`, which does very little "interesting" computation, it can be seen that QD-Janus is only 38% slower than the Prolog program.

Table 2 gives the speed of QD-Janus compared to the FCP(:) implementation of Kliger.[14] It can be seen that QD-Janus is more than three times as fast, on the average, as FCP(:), even though the FCP(:) system compiles down to a lower level and carries out various low-level optimizations, such as generating decision trees, that are not done by the QD-Janus system.

Table 3 gives the relative heap utilization of QD-Janus compared to FCP(:). It can be seen that the heap requirements of QD-Janus are typically two to three orders of magnitude better than those of the FCP(:)

Program	QD-Janus (QD) (ms)	Sicstus Prolog (S) (ms)	QD/S
`nrev(30)`	1.16	0.84	1.38
`hanoi(17)`	2913.2	2642.0	1.10
`e`	688.0	680.0	1.01
`qsort(50)`	18.0	13.5	1.33
`pi`	345.6	345.5	1.00
`dnf`(50)	126.5	123.7	1.02
`pascal(200)`	872.0	730.5	1.19
`queen(8)`	7350.0	4860.0	1.51
`prime(10000)`	14659.3	10612.0	1.38
Geometric Mean of QD/S :			1.20

Table 1: Relative Efficiency of QD-Janus and Sicstus Prolog

Program	QD-Janus (QD) (ms)	FCP (ms)	FCP/QD
`nrev(30)`	1.16	4.5	3.89
`pascal(200)`	872.0	4490	5.14
`queen_1(8)`	49.8	110	2.21
`prime(10000)`	14659.3	16560	1.13
`tak`	367.2	1720	4.68
`combination`	257.4	1892	7.35
`deriv`	144.2	690	4.78
`mastermind`	24866.3	30200	1.21
`nand`	1197.2	4480	3.74
Geometric Mean of FCP/QD :			3.22

Table 2: Relative Speeds of QD-Janus and FCP

Program	QD-Janus (QD) (Kbytes)	FCP (Kbytes)	FCP/QD
`nrev(30)`	0.42	516	1228.6
`pascal(200)`	664.0	1792	2.7
`queen_1(8)`	0.08	20	250.0
`prime(10000)`	9.8	9468	966.1
`combination`	821.0	2136	2.6
`deriv`	0.57	768	1347.4
`mastermind`	29.1	8676	298.1
Geometric Mean of FCP/QD :			135.4

Table 3: Heap Usage of QD-Janus and FCP

implementation, with an average heap consumption that is more than 130 times lower. We conjecture that FCP(:) has such relatively high heap requirements because it allocates activation frames on the heap; by contrast, QD-Janus uses a stack-oriented scheme that requires less space, is easier to reclaim, and can be expected to be more efficient.

DISCUSSION

The choice of Prolog as a target language was motivated by the fact that its "semantic distance" from Janus is considerably less than that of lower level languages such as C. Because of this, and because the QD-Janus compiler was written in Prolog, the implementation was completed in a fairly short period of time, taking one person approximately two months to complete. This certainly highlights the suitability of Prolog as a language for building language translators. As a concrete example, it took two days to implement suspension analysis, and another day or two to implement the various optimizations that depend on it.

However, ease of development is not the only reason that makes Prolog attractive in this context. As the performance numbers indicate, much of the overhead associated with the management of various execution overheads can be eliminated via reasonably simple compile-time analyses and optimizations. Given the growing number of high-performance[8,9] and parallel[10,11,12,13] implementations of Prolog, this suggests that compilation to Prolog may be a reasonably quick and cheap way to obtain both sequential and parallel implementations of very high level programming languages with relatively little effort.

CONCLUSIONS

This paper describes a sequential implementation of Janus, a concurrent constraint language, whose target language is Prolog. The compiler uses a number of novel analyses and optimizations to improve the performance of the system. These analyses and optimizations are typically concerned with reducing the amount of suspension testing in the generated code. The resulting system is quite efficient: its performance is significantly better than a comparable low-level implementation of FCP(:), and its overhead compared to the underlying Prolog system is relatively modest. Our experience indicates that translation of logic programming languages to Prolog, accompanied by the development of good program analysis and optimization tools, is an effective way to quickly develop flexible and portable implementations with good performance and low cost. The system is available by anonymous FTP from `cs.arizona.edu`.

Acknowledgements: Discussions with Ken Kahn, Jacob Levy, and Vijay Saraswat were invaluable in improving the author's understanding of Janus. Clement Pellerin and Mats Carlsson made many helpful suggestions regarding details of the implementation. Thanks are due to Evan Tick for his help with the benchmarking.

References

[1] 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.

[2] V. A. Saraswat, *Concurrent Constraint Programming Languages*, PhD thesis, Dept. of Computer Science, Carnegie-Mellon University, 1989. ACM Doctoral Dissertation Award series, MIT Press.

[3] E. Shapiro, "The Family of Concurrent Logic Programming Languages", *Computing Surveys*, vol. 21 no. 3, Sept. 1989, pp. 412-510.

[4] J. Jaffar and J.-L. Lassez, "Constraint Logic Programming", *Proc. Fourteenth ACM Symposium on Principles of Programming Languages*, Jan. 1987, pp. 111-119.

[5] V. A. Saraswat, "Compiling CP($\downarrow, |, \&$) on top of Prolog", Technical Report CMU-CS-87-174, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Oct. 1987.

[6] J. Tanaka and M. Kishishita, "Compiling Extended Concurrent Prolog – Single Queue Compilation", *Proc. European Symposium on Programming*, Saarbrücken, March 1986, pp. 301–314. Springer Verlag LNCS vol. 213.

[7] K. Ueda and T. Chikayama, "A Compiler for Concurrent Prolog", *Proc. Second International Symposium on Logic Programming*, Boston, July 1985, pp. 119–126. IEEE Press.

[8] A. Taylor, "LIPS on a MIPS: Results from a Prolog Compiler for a RISC", *Proc. Seventh International Conference on Logic Programming*, Jerusalem, June 1990, pp. 174–185. MIT Press.

[9] 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, TX, Oct. 1990, pp. 501-515. MIT Press.

[10] K. A. M. Ali and R. Karlsson, "The Muse Or-Parallel Prolog Model and its Performance", *Proc. 1990 North American Conference on Logic Programming*, Austin, TX, Oc. 1990, pp. 757–776. MIT Press.

[11] M. V. Hermenegildo and K. J. Greene, "&-Prolog and its Performance: Exploiting Independent And-Parallelism", *Proc. Seventh International Conference on Logic Programming*, Jerusalem, June 1990, pp. 253–268. MIT Press.

[12] B. Ramkumar and L. V. Kalé, "Compiled Execution of the Reduce-OR Process Model on Multiprocessors", *Proc. 1989 North American Conference on Logic Programming*, Cleveland, Oct. 1989.

[13] V. Santos Costa, D. H. D. Warren, and R. Yang, "The Andorra-I Engine: A Parallel Implementation of the Basic Andorra Model", *Proc. Eighth International Conference on Logic Programming*, Paris, June 1991, pp. 825–839. MIT Press.

[14] S. Kliger, *Compiling Concurrent Logic Programming Languages*, Ph.D. Thesis, The Weizmann Institute of Science, Rehovot, Israel, Oct. 1992.

[15] D. Gudeman, K. De Bosschere, and S.K. Debray, "`jc`: An Efficient and Portable Sequential Implementation of Janus", *Proc. Joint International Conference and Symposium on Logic Programming*, Washington DC, Nov. 1992, pp. 399–413. MIT Press.

[16] S. K. Debray, "Implementing Logic Programming Languages: The Quiche-Eating Approach", *Proc. ICLP-93 Workshop on Practical Implementations and Systems Experience*, Budapest, Hungary, June 1993.

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

[18] 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.

[19] M. Carlsson and J. Widen, *SICStus Prolog User's Manual*, Swedish Institute of Computer Science, Oct. 1988.

[20] J. Hughes, "Strictness Detection in Non-Flat Domains", in *Programs as Data Objects*, ed. H. Ganzinger and N. D. Jones, Springer-Verlag Lecture Notes in Computer Science vol. 217, Oct. 1985.

[21] G. Lindstrom, "Static Evaluation of Functional Programs", *Proc. ACM SIGPLAN '86 Symp. on Compiler Construction*, July 1986, pp. 196–206.

[22] S. K. Debray, "Static Inference of Modes and Data Dependencies in Logic Programs", *ACM Transactions on Programming Languages and Systems* vol. 11, no. 3, June 1989, pp. 419-450.

[23] G. Janssens and M. Bruynooghe, "An Instance of Abstract Interpretation Integrating Type and Mode Inferencing", *Proc. Fifth International Conference on Logic Programming*, Seattle, Aug. 1988, pp. 669-683. MIT Press.

[24] K. Marriott, H. Søndergaard and N. D. Jones, "Denotational Abstract Interpretation of Logic Programs", *ACM Transactions on Programming Languages and Systems* (to appear).

[25] C. S. Mellish, "The Automatic Generation of Mode Declarations for Prolog Programs", DAI Research Paper 163, Dept. of Artificial Intelligence, University of Edinburgh, Aug. 1981.

[26] J. Gallagher, M. Codish, and E. Shapiro, "Specialisation of Prolog and FCP Programs Using Abstract Interpretation", *New Generation Computing* vol. 6, pp. 159–186, 1988.

[27] S. K. Debray, "Compiler Optimizations for Low-Level Redundancy Elimination: An Application of Meta-level Prolog Primitives", *Proc. Third Workshop on Metaprogramming in Logic* (META-92), Uppsala, June 1992.

[28] 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.

[29] E. Tick, *Parallel Logic Programming*, MIT Press, Cambridge, 1992.