A SIMPLE APPROACH TO SUPPORTING UNTAGGED OBJECTS IN DYNAMICALLY TYPED LANGUAGES *

PETER A. BIGOT AND SAUMYA K. DEBRAY

▷ In many modern high-level programming languages, the exact low-level representation of data objects cannot always be predicted at compile time. Implementations usually get around this problem using descriptors ("tags") and/or indirect ("boxed") representations. However, the flexibility so gained can come at the cost of significant performance overheads. The problem is especially acute in dynamically typed languages, where both tagging and boxing are necessary in general. This paper discusses a straightforward approach to using untagged and unboxed values in dynamically typed languages. An implementation of our algorithms allows a dynamically typed language to attain performance close to that of highly optimized C code on a variety of benchmarks (including many floating-point intensive computations) and dramatically reduces heap usage. ◁

1. Introduction

In many high level programming languages, the representation of a data object at a particular program point cannot always be predicted in a precise way at compile

*A preliminary version of this paper appeared in *Proceedings of the 1995 International Symposium on Logic Programming*. This work was supported in part by the National Science Foundation under grant number CCR-9123520. The first author was also supported by a AT&T Foundation Fellowship.

Address correspondence to Saumya K. Debray, Department of Computer Science, The University of Arizona, Tucson, AZ 85721, USA. E-mail: debray@cs.arizona.edu

655 Avenue of the Americas, New York, NY 10010 0743-1066/93/$3.50

time. In dynamically typed languages, such as Icon, Lisp, Prolog, and Scheme, the type of a variable may not always be statically known (and, indeed, may change from one program point to another). In languages with dataflow synchronization, such as GHC, Strand and Id, the value of an expression may not be available at a program point because it has not yet been computed. The code generated for programs in such languages must, therefore, be able to deal with different kinds of representations that may arise at runtime. There are two different but related issues that arise here. First, it is necessary to be able to determine how a bit pattern, encountered at runtime, is to be interpreted—e.g., as a pointer or as a value. Second, different representations or data types may have different sizes: for example, a pointer to a double-precision floating point value may take less space than the value it points to.

The usual way to address the first problem is to attach a descriptor to each value, to specify how its bit pattern is to be interpreted: such descriptors are usually referred to as *tags* [18, 31]. The second problem is usually handled by making values of different sizes "look the same" by manipulating pointers to them rather than the values themselves: such an indirect representation is often referred to as a *boxed* representation. In general, operating on values in languages such as these may involve manipulating tags and/or a level of indirection. It may be possible to avoid some of this extra work in clever implementations (e.g., tags can be elided in SML/NJ by relying on compiler-generated symbol table information [1]), or to encode the information in some clever way to reduce its cost (e.g., in common integer arithmetic operations in many Lisp systems (e.g., see [21]), and dereference operations in some Prolog systems [34, 35]). In general, however, it is not possible to avoid altogether a performance penalty for tagging/boxing of objects.

The performance overhead of dealing with tags and boxes is especially serious in dynamically typed languages, where both tagging and boxing are necessary in general. Steenkiste and Hennessy's experiments with Lisp on a RISC system, on a set of non-numerical benchmarks, indicate that the programs spent about 22% of their time on tag handling [32]. This figure would likely be much worse in numerical computations, because implementations of dynamically typed languages very often represent floating point numbers as boxed values (see, for example, [4, 7, 9]). This incurs a significant performance penalty, for a number of reasons. First, since floating point values are heap-allocated, numerical computations involving boxed floating point values fail to exploit hardware registers effectively, and generate a lot more memory traffic. The allocation of fresh heap cells may also result in additional checks for heap overflow. Finally, the high rate of memory usage also results in increased garbage collection and adversely affects cache and paging behavior. The tag-handling overheads for data structures such as lists—which account for the bulk of the overall tag management costs in Steenkiste and Hennessy's study [32]—can, at least in principle, be reduced by program transformations such as deforestation [36], which reduce the number of intermediate data structures created. However, it is not clear that analogous improvements are readily possible for numerical computations.

Curiously, the question of maintaining untagged and/or unboxed objects, particularly floating point values, has received little attention in the logic programming community. To the best of our knowledge, all existing systems, including high-performance implementations such as Aquarius Prolog and SICStus Prolog, maintain floating point values in boxed form. Very often, authors either simply ignore the question of optimizing numerical computations, or explicitly give up on attaining good performance on such computations in logic programming languages (e.g., in discussing the Strand system, Mattson [25] states: *"Concurrent logic programming languages are not well suited for the numerically intensive operations common in scientific programming. Strand shares this shortcoming..."*). In this paper, we consider compile-time and runtime aspects of supporting untagged and unboxed values in languages that normally require data to be tagged and possibly boxed: in particular, we focus on numerical values. The main contribution of this work is its simplicity: we use a simple extension to the (intra-procedural) register allocator for intra-procedural untagging optimizations, and show how the idea extends in a straightforward way to allow untagged objects to be passed across procedure boundaries. The execution model we assume is described in Section 2.1. The techniques described here have been implemented as part of the `jc` system [19], an implementation of a logic programming language derived from Janus, available by anonymous FTP from `ftp.cs.arizona.edu`. The resulting performance improvements are quite substantial: heap usage is reduced dramatically, and speed improves to the point where many programs involving substantial amounts of numerical computation attain speeds comparable to that of C code written in a "natural" C style and optimized at the highest level possible.

A Note on Terminology: *To reduce tiresome repetition, we will use abuse terminology in the discussion that follows and use the term* "untagged" *to refer to values in their native machine format, i.e., to values that are untagged and (where necessary) unboxed. We hope this does not cause any confusion.*

2. Background

2.1. The Execution Model

We assume that we have a dynamically typed language with a garbage-collected heap area. Our assumptions about runtime structures are fairly weak, and generally applicable to a reasonably wide variety of languages: for example, even though we refer to "stack frames" in Section 3.3, our approach does not require that these be allocated in a separate "stack area" in memory or that they be manipulated in a LIFO fashion: it is necessary only that the garbage collector be able to identify these objects correctly (which it must be able to do in any case), and that it be able to determine, for any such frame, the corresponding procedure (this can be done fairly easily, with very little additional work at runtime).

For simplicity, we assume that there is a fixed predefined set of types that may be maintained in untagged form. Our implementation allows untagged values to be stored in stack frames, but not on the heap. The restriction is imposed to satisfy the requirements of the garbage collector: since an untagged value has no descriptor associated with it, the garbage collector must be able to identify and deal with untagged values (and not confuse, for example, untagged integers with pointers).

As discussed in Section 3.3, this is straightforward to do for values on the stack because of the predictable structure of stack frames. If the tagging scheme used by an implementation is rich enough to support descriptors that encode the structure of (some types of) heap-allocated objects, in particular information about elements that are untagged, then the problem with identification of untagged values on the heap goes away. In this case, our approach can be readily extended to handle untagged values on the heap.

An important consideration in the context of logic programming languages is that of dereferencing. In most such languages, there may be a pointer chain, whose length can be unbounded in general, between a variable and the value it is bound to: in order to determine the value of that variable, this pointer chain must be dereferenced. This requires the ability to distinguish pointers from values that are not pointers. This is straightforward when all values are tagged with descriptors, but becomes difficult in the presence of untagged values. Therefore, in order to support untagged values, it is necessary to ensure that the compiler is (*i*) aware of the exact length of any pointer chain to an untagged value; and (*ii*) able to communicate this information to the garbage collector at any program point where garbage collection might occur. Compile-time analyses to estimate the lengths of pointer chains have been investigated by several authors [33, 35]. In our implementation, we get around this problem by disallowing pointer chains of non-zero length to untagged values (i.e., a value that can have pointers to it is not kept in untagged form).

Finally, in order for the compiler to decide that a value can be maintained in untagged form at a particular program point, it must have a certain minimum amount of information available about that value. At the very least, type information that is precise enough to allow the compiler to use operations specialized to a particular representation is necessary. For example, in general it is not enough to know that a value will be a number—we need to know whether it will be an integer or a floating point value. Even this may not be enough if the implementation supports different varieties of integers or floating point values (e.g., fixnums, bignums, etc.), as is the case in Common Lisp and some Prologs. Moreover, depending on the language, the "type" of a value may not be enough to determine its machine-level representation at a particular program point. In a concurrent logic programming language, for example, knowing that a variable has type integer may not be enough to determine whether, at a particular program point, its value can be guaranteed to have been computed, or whether it may still be unbound. However, the details of how information about types is collected—e.g., from programmer annotations or via dataflow analysis—as well as any auxiliary information, e.g., a guarantee that the value of a variable will be available at a program point, are orthogonal to the subject of this paper. Here we assume only that this information has been obtained and is available for use by the compiler; the interested reader is referred to [15] for a discussion of the dataflow analysis used in our implementation for this purpose.

2.2. The Implementation Context

The framework in which the work described here has been implemented is `jc` [19], a translator for a committed-choice logic programming language that, in its present incarnation, closely resembles Strand [16]. For the purposes of this paper, it suffices to note that it is a first-order dynamically typed committed choice language. Source programs are read by the `jc` translator, analyzed and subjected to various low level optimizations, and finally converted into abstract machine code. The abstract machine code is embedded into the body of a C function and expanded through the use of macros to C code which implements the instructions of the abstract machine.

Each operation in the virtual machine has a wholly generalized version that can deal with arbitrary tagged operands. When type information is available at compile time, the compiler can emit specialized versions of certain operations where type tests on the operands have been removed. To reduce the complexity of the abstract machine, our implementation requires that the operands of the specialized versions of an operation accept operands only of the same type: for example, we have a version of addition which expects two integer operands and one that expects two floating point operands, but we do not allow addition of an integer and a float except within the most general operation. Each (specialized version of an) operation has two type values associated with it: that of the operands it is expecting, and that of its result (in general, the type of the result of an operation may be different from that of the operands). Type information is determined for each occurrence of a variable based on an *ad hoc* analysis that examines programmer-provided annotations, the variable's origin, and the operations performed on it, and is propagated to provide type information about intermediate values. Obviously, specialization of operations to omit unnecessary type tests can be done regardless of whether untagged values are used outside the internals of the operations. However, in cases where both the operands and the result have an untagged representation, we can further specialize the operation and create a version that eliminates the representation conversion phases entirely, resulting in a direct application of the underlying operation to the untagged operands. We wish to use these versions of the operations wherever possible, because they have the least overhead.

Input parameters are passed to procedures in registers. The `jc` system provides four kinds of general-purpose registers: *tagged registers*, which hold tagged values; *address registers*, which hold untagged pointers, e.g., into arrays or lists; *integer registers*, which hold untagged integer values; and *floating point registers*, which hold untagged floating point values. We use a cost-based model to decide whether a particular output should be returned in memory or in (tagged or untagged) registers [5].[1] To meet the analysis requirements of Section 3.1 and allow the use of untagged registers for parameter passing requires a combination of mode analysis, which identifies the input and output arguments of a procedure; suspension analysis, which identifies procedures that can be guaranteed to not suspend during execution; and type inference. This information is available under the assumptions in Section 2.1.

[1] The discussion in [5] considers returns in memory and tagged registers only. Since then, we have extended our implementation, and the associated cost model, to handle untagged register returns as well.

Representation Analysis for a Program.

1. Use inter-procedural representation analysis to determine the representation required of input and output arguments for each procedure.

2. Use intra-procedural representation analysis to determine the representation(s) required of each variable in each clause of each procedure.

Inter-Procedural Representation Analysis.

1. [*Identification of Candidates for Untagged Representation*] For each procedure p, use information obtained from mode, type, and non-suspension analyses to determine which arguments can be passed in untagged form.

2. [*Determination of Argument Placements*] Use the cost model of [5] to determine which output arguments should be returned in registers.

3. [*Determination of Argument Representations*] If an argument is a candidate for untagged representation (step (1)), and will be passed in a register (input arguments are always passed in registers; the passing of output arguments is determined in step (2)), then it is determined to require an untagged representation. Otherwise, it requires a tagged representation.

Intra-Procedural Representation Analysis. For each clause do:

1. [*Initialization*] Use the results of inter-procedural representation analysis to determine the representation required of each variable that appears as a parameter in the head or as an argument in the body. Use this information to determine the representations required for other variables that are defined or used by primitive operations in the clause body.

2. [*Propagation*] For each operation $op(\bar{t})$ in the clause do:

   If $\bar{t}$ must be explicitly untagged before the operation can be performed, then: each operand for the operation requires an untagged representation and $op(\bar{t})$ is specialized to operate on untagged operands.

   Otherwise, if explicit untagging is not necessary, then: let N_t (N_u) be the no. of operands in $\bar{t}$ that are available in tagged (untagged) form.

   If $N_u > N_t$ then each operand for the operation requires an untagged representation and $op(\bar{t})$ is specialized to operate on untagged operands; otherwise each operand for the operation requires a tagged representation and $op(\bar{t})$ is left in its generic form.

3. [*Determination of Representation Conversions*] For each variable X, if X requires a representation R but is not available in representation R in the clause, then insert the appropriate representation conversion operation just before the first use of X that requires representation R; mark subsequent occurrences of X as being available in representation R.

Figure 1. Representation Analysis: Overall Algorithm

3. Representation Analysis

Our representation analysis algorithm is summarized in Figure 1. First, inter-procedural representation analysis is used to determine how input and output arguments for each procedure will be passed. This uses a cost model, based on that described in [5], to determine the representation and placement of output values. This fixes the representation of input and output parameters to each procedure, and therefore determines the representations of the corresponding variables in each clause for these procedures. Starting with these representation choices, intra-procedural analysis is then used within each clause to determine the representation of each variable at each point within the body of the clause. This phase proceeds in tandem with the generation of abstract machine code.

The decision to have inter-procedural representation analysis precede intra-procedural representation analysis may seem strange, but it is motivated by the 80-20 rule ("80% of a program's execution time is spent in 20% of the code"), which suggests that the primary benefits of using untagged values are likely to come from maintaining values in untagged form through the execution of loops. In the languages we are concerned with, loops are realized using tail recursion. Our approach therefore amounts to first deciding which values can be maintained in untagged form through the execution of loops, and then propagating these decisions to other program points via intra-procedural representation analysis. It turns out that an added benefit of doing the analyses in this order is that the order in which procedures are processed does not affect the code generated, which helps to keep the overall algorithm simple.

What information do we need to determine which values can be maintained in unboxed form through the execution of a loop? Consider the structure of a typical tail recursive procedure:

$$
\begin{array}{lll}
p(\bar{x}, \bar{y}) & :- & q_0(\bar{x}, \bar{y}_1),\ q_1(\bar{y}_1, \bar{y}_2),\ \ldots,\ q_n(\bar{y}_n, \bar{y}_{n+1}),\ p(\bar{y}_{n+1}, \bar{y}). \\
p(\bar{x}, \bar{y}) & :- & r(\bar{x}, \bar{y}).
\end{array}
$$

During the execution of the loop body, described by the first clause above, the input arguments $\bar{x}$ are used to compute intermediate values $\bar{y}_1$, which in turn are used to compute other intermediate values $\bar{y}_2$, and so on. In general, values may be "threaded through" the loop body, and some or all of the intermediate values may be returned from calls to other procedures. To get the most out of maintaining untagged values, therefore, we need to be able to pass input values to procedures in untagged form, and have them return their outputs in untagged form as well. As an example, consider the following procedure, taken from a program to compute Bessel functions:

```
j1(N, X, A, Y) :-
   N > 0, X1 := -X*X/4, pow(N, X1, U), factsq(N, V),
   A1 := A + X*U/(2*V*(N+1)),
   N1 := N-1,
   j1(N1, X, A1, Y).
j1(0, X, A, Y) :- Y := A + X/2.
```

The value of the variable `U` is computed by the procedure `pow`/3, while that of `V` is computed by `factsq`/2. In order to maximize the benefits of the untagging optimization, we should pass the arguments to these procedures in untagged form, and have them return the values of their output arguments in untagged form as well.

Our aim, then, is to determine when the input arguments to a procedure can be passed, and the output arguments returned, in untagged form. The first of these two pieces of information, namely, the representation of input arguments, would intuitively seem to be computable using some form of type analysis. However, the second piece of information seems harder to obtain, for reasons peculiar to logic programming languages. Implementations of logic programming languages typically pass all arguments into a procedure in registers, with each uninstantiated variable—usually corresponding to an output argument—passed by reference, i.e., as a pointer to the cell occupied by that variable. An output value is returned by binding an uninstantiated variable to a value, i.e., by writing to the corresponding memory location. Historically, most logic programming languages have been dynamically typed: for such languages, while it may be possible to determine, at runtime, the type of an untagged value stored in a particular stack slot (see Section 3.3), it is very difficult to determine the type of an untagged word on the heap. A garbage collector may therefore be able to deal with untagged values on the stack, but not with untagged values on the heap.[2] Moreover, it is difficult, in general, to predict whether a variable is resident on the stack or on the heap—for example, a stack-resident variable may point to a value on the heap, or be moved to the heap and become heap-resident in connection with tail call optimization. As a result, if output values are returned in the traditional way, i.e., by writing to a memory location, it is difficult to predict at compile time whether the memory address passed to a procedure will refer to a location on the stack or on the heap, and therefore whether it can be returned in untagged form and be guaranteed to not confuse the garbage collector. This problem disappears, however, if an untagged value is returned in a (untagged) register. We therefore restrict untagged return values to those that can be returned in registers. Our algorithm for representation optimization therefore relies, in a very fundamental way, on returning values in registers.

3.1. Inter-procedural Representation Analysis

Previous research has indicated that returning values in registers rather than storing them into memory can have a significant impact on execution speed [5, 35]. Given that operations within the body of a procedure can be performed on untagged representations, this suggests that parameters should be passed and returned in untagged form as well where it is legitimate and profitable to do so. If the mode of a parameter cannot be determined to be strictly input or output, it must be passed or returned in tagged form. The choice of how parameters for a procedure should be passed—in registers or in memory, and in tagged or untagged form—must be

[2] This is not true of statically typed languages, however, and several authors have proposed tagless garbage collection schemes for statically typed languages (see, for example, [1, 17]).

made before any call to the procedure is generated, because it is necessary, for code generation, to know where to place inputs and where to expect outputs.

The identification of arguments to a procedure that are eligible to be passed in untagged form proceeds as follows. An input argument to a procedure p can be passed in an untagged register only if the definition of p takes a unique type τ for the corresponding parameter, the type τ admits an untagged representation, and the corresponding argument at each call site for p has type τ. If the callee takes a unique type τ for a parameter and this type admits an untagged representation, but the corresponding argument at a particular call site may have a value of a different type, it is possible to preserve the use of untagged inputs by generating additional code at the call site to test the type of that argument and handle it as appropriate, e.g., to carry out representation conversion and pass the result in an untagged register if this is possible, and to fail otherwise. Basically, this amounts to hoisting the type test for τ from the callee to the call site, and can be thought of as an instance of call forwarding [13]—it converts what would be failure inside the called procedure to failure at the call site. Alternatively, the code generated for the callee can have multiple entry points, corresponding to different sets of arguments that need to be untagged: this has the advantage of not duplicating the untagging code at multiple call sites, but can become unwieldy if the sets of arguments that need untagging cannot be totally ordered by set inclusion. If the argument at any call site cannot be guaranteed to be defined, e.g., due to possible suspension, the parameter must be assigned a tagged register. An output argument may be returned in an untagged register provided that non-suspension analysis [15] has shown that the value will be defined (i.e., not an unbound variable) when the procedure finishes, and type analysis has shown that it will have a type that has an untagged representation.

These restrictions embody the necessary conditions for the use of untagged registers in parameter passing. Our current implementation passes input arguments in untagged form whenever these restrictions are satisfied. The representation of output values is determined using a low level cost model that is an extension of that described in [5]: if the cost model indicates that it is profitable to return an output argument in a register, and it is possible to return that argument in untagged form, then it is returned in an untagged register. While it is possible, in principle, that in some cases it may be better to allow parameters to be passed in tagged registers even when the above restrictions are satisfied (e.g., when they will immediately be stored into the heap), our experimental results indicate that the simple approach of using untagged registers wherever feasible yields reasonable performance in most cases.

3.2. Intra-procedural Representation Analysis

In any compiler for a high-level language, the register allocator has to keep track of the location where a variable resides—in a register or in memory—in order to generate correct code. Now suppose we want to maintain objects in untagged form in a dynamically typed language: obviously, it is necessary to extend the intra-procedural register allocator to track the availability of values in untagged form.

Our intra-procedural representation analysis is based on a very simple extension to the register allocator. Recall that variables (except in certain stack slots) are always stored in memory in tagged form. When the value of a variable is loaded into a register for expression evaluation, the chosen register is associated with that variable, so that future references can use that register. The crucial extension is to permit the same variable to simultaneously be associated with other registers, some of which may be untagged. Therefore, just as tagged registers are a "cache" for values normally stored in memory, so too are untagged registers a cache for values that would otherwise be tagged. The local register allocator serves as a cache directory by noting that a particular variable is available in both tagged register i and (say) integer register j.

The decision as to whether a particular operation should be specialized is made at the time the abstract machine code is generated. We examine the operands to determine the most specific type which describes them, by considering type information provided by the compiler, as well as any untagged registers assigned to the operand by previous operations. If the operand types admit an untagged implementation of the operation, we must then decide whether it should be used, by examining the availability of the operands in the corresponding untagged form.

When the operation cannot work directly on the tagged operands, as with boxed floating point representations, we automatically use the untagged version. The rationale for this is that the tagged operation will do the untag and unbox conversions, perform the operation, and box the result, leaving neither operands nor result around in untagged registers to be used for future operations. If the untagged version of the operation is used, the conversion phase will be performed explicitly on all operands that are not already loaded into registers of the appropriate type, and the result will not be converted to tagged form immediately. In the best case, this means the operands and result will be around for further use in untagged operations without additional conversion. In the worst case, the representation conversions that would have been done inside the operation have been made explicit: since we compile to native code rather than to byte code, the amount of work done is the same whether the representation conversions are made explicit in this way or performed implicitly inside the generic operation.

However, when the operation can be performed directly on the tagged representation, as in the case of (small) integers, there is a tradeoff. On the one hand, explicitly converting the operands to untagged form may allow us to use the untagged operands or result in future operations that could benefit from having them in untagged form. On the other hand, the extra conversion operations and possible increase in register pressure required to keep multiple representations live could outweigh the benefits of using untagged values. It is plausible that a detailed cost model could be devised to determine exactly when the untagged or tagged version is to be preferred, taking into account relative execution frequencies of different branches and the overhead of preserving tag information when the operation is performed on the tagged representation (see, for example, [27]). However, in many cases, the overhead of preserving tag bits when the operation is performed directly on the tagged values is small or non-existent. For example, `jc`, like many Lisp systems, uses the lowest 2 bits of a word for the tag, and a bit pattern of 00 for integers. This allows addition and subtraction of tagged integers to be carried out with no representation conversions on the operands or the result, and multiplication and division with only simple shifts. The performance benefits of avoiding these operations do not seem to warrant a complex analysis.

Instead, we rely on a simple heuristic which seems to perform reasonably well. We note that, although such operations may be performed on the tagged or untagged form with roughly equal cost, if at any point the result is stored into the heap or any other memory location that is not restricted to untagged values, the result must be stored in tagged format. As such, we have a preference for using the tagged form. We choose to use the untagged version of the operation only when the number of operands that are available in untagged form strictly exceeds the number which are available in tagged form: in that case, the assured need to do more conversions before the operation outweighs the possibility of having to convert the result in the future.[3] Intermediate results from previous operations are generally available in only one form; availability of variable operands in a particular form is determined by seeing what registers the variable is loaded in. Since the tagged representations in these cases do not require boxing, constants can be represented in either tagged or untagged form by emitting the appropriate encoding at compile time, so do not contribute materially to the decision.

3.3. Garbage Collection Issues

The need to preserve values across procedure calls which would destroy registers, or to free registers for use in expression evaluation, requires that values be saved in the activation record of the procedure. Since we have gone to some trouble to load values in untagged form, it would be convenient to save them that way, rather than having to tag and untag them. This requires that untagged values be allowed to reside in the stack frame.

The structure of a stack frame in `jc` is shown in Figure 1. The decision as to whether a stack slot assigned to a variable is tagged or untagged is made at the time the slot is reserved: generally, at the point the value needs to be stored. We choose to place a value in an untagged slot only if it resides in an untagged register, and does not also reside in a tagged register. The first requirement guarantees that at the storage point we have a value to save without having to convert it. The second is an effort to preserve boxed values, so that if we later need that value in tagged form we do not have to explicitly reconstruct the tagged form again from an untagged representation (boxing a value generally requires allocating heap storage—and therefore possibly a heap overflow check—followed by one or more writes to memory; this is considerably more expensive that an unboxing operation, which generally involves only one or two memory reads). While it is possible to store both the tagged and untagged representations of a particular value, this does not appear to be profitable, since the extra memory operations required by the

[3] Although it may appear that untagged values will never be introduced if this rule is used, in fact untagged values arise from several sources, including primitive operations such as taking the size of an array, conversions from values that have boxed representations, and untagged input and output parameters to the procedure being considered.

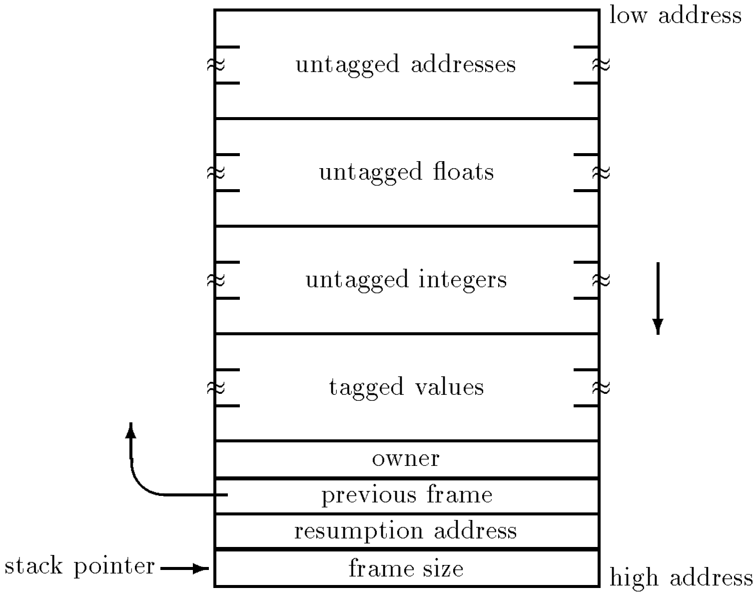

Figure 1. Structure of a Stack Frame in `jc`

stores and loads are likely to outweigh the cost of an extra untag operation when the value is needed again.

Each stack slot has a type associated with it, which indicates whether it holds a tagged or untagged value, and what kind of value. The type of a slot is fixed over the execution of the procedure, but the assignment of a stack slot to variables may be different at different points in a procedure—i.e., we reuse stack slots for different values of the same type where possible. However, we do not attempt to compress the frame by reusing space of one type to hold values of a different type as storage requirements change over the lifetime of a procedure. To do so would cause the frame layout to change dynamically during execution, complicating the communication of layout information to the garbage collector. After intermediate code generation is complete, the stack frame is rearranged so that slots of the same type are adjacent to each other in memory and the addresses in the code are updated to reflect the rearrangement and note where untagged values require more space than tagged values. When a stack frame is allocated on procedure entry, an index denoting the procedure is stored in the frame, in the slot labelled "owner" in Figure 1: the garbage collector will use this index to retrieve the layout information from a global table and use this to avoid misinterpreting the untagged values when reclaiming heap space.

4. Experimental Results

To determine the efficacy of the untagged support mechanism described here, both C and `jc` implementations of a set of programs were timed on a Sun SPARC IPC running Solaris 2.3, with `gcc` 2.6.3 (invoked with `-O2 -fomit-frame-pointer`) as the C compiler for the C code emitted by `jc`, and `gcc` 2.6.3 (invoked with `-O2 -fomit-frame-pointer`) and `cc` (CDS SPARCompilers version 2.0.1, invoked with both `-O2` and `-O4`) as the C compilers for the native C programs. The benchmarks used were the following: *aquad* performs a trapezoidal numerical integration $\int_0^1 e^x dx$ using adaptive quadrature and a tolerance of 10^{-8}; *bessel* computes the Bessel function $J_{10}(2.0)$; *binomial* computes the binomial expansion $\sum_{i=0}^{30} x^i y^{30-i}$ at $x = y = 1.0$; *chebyshev* computes the Chebyshev polynomial of degree 10000 at 1; *e* evaluates $e = 2.71828\ldots$ by computing the sum of the first 2000 terms of the series $1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots$; *fft* is an iterative one-dimensional fast Fourier transform program, adapted from [28], that computes the fast Fourier transformation and its inverse on a vector of size 64; *fib* computes the Fibonacci value $F(16)$; *fmatmult* multiplies two 20×20 floating-point matrices; *log* computes $\ln(1.999)$ using the expansion $\ln(1 + x) = \sum_{i \geq 0}(-1)^{i+1}x^i/i$, to a tolerance of 10^{-6}; *mandelbrot* computes the Mandelbrot set on a 17×17 grid on an area of the complex plane from $(-1.5, -1.5)$ to $(1.5, 1.5)$; *mcint* uses Monte Carlo integration to estimate the mass of a body of irregular shape (adapted from [28]); *muldiv* exercises integer multiplication and division, doing 5000 of each; *pi* computes the value of π to a precision of 10^{-3} using the expansion $\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots$; *sum* adds the integers from 1 to 10,000—it is essentially similar to a tail-recursive factorial computation, except that it can perform a much greater number of iterations before incurring an arithmetic overflow; *tak*, from the Gabriel benchmarks, is a heavily recursive program which does integer addition and subtraction: the query executed is $tak(14, 12, 6, \_)$; and *zeta* computes the Euler-Riemann zeta function, defined by the series $zeta(x) = 1 + 2^{-x} + 3^{-x} + \cdots$ (where x is real-valued), at $x = 2.0$, to a tolerance of 10^{-3}. For the discussion here, it suffices to note that *fib*, *muldiv*, *sum* and *tak* operate solely on integer values, *bessel* combines integer and floating-point operations, and the remainder are primarily floating-point benchmarks. Only the *fft*, *fmatmult*, and *mandelbrot* programs involved compound data structures: *fft* implemented updatable arrays using binary trees, and *fmatmult* and *mandelbrot* used two-dimensional arrays. The code for each benchmark was written in a style natural to the language. Where feasible, iteration was used to code loops in C, while tail recursive procedures performed the analogous operation in `jc`. Wherever possible, the C programs used were taken from code written by C programmers in other contexts: for example, the C code for *fmatmult* was taken from the Stanford benchmark suite by J. Hennessy, while that for *mandelbrot* was taken from a program by G. Wilson for a textbook. With the exception of a few benchmarks (*aquad*, *bessel*, *binomial*, *fmatmult*, *mandelbrot*, and *zeta*), where the natural implementation required support functions (e.g., to compute factorials), the C versions were single functions. Because `jc` supports only single precision floating point calculations, the C versions were carefully coded to ensure that all constants were treated as single precision by the C compiler, avoiding unnecessary and costly precision adjustments in the C version.

Tables 1 and 2 summarize the experimental results for these benchmarks: the former shows the performance improvements due to the use of unboxed values, while the latter compares the speed of the resulting system with optimized C code. Execution times were obtained using the *gettimeofday*(2) system call to obtain microsecond-resolution measurements of execution time, with the testing being the

Program	Execution Time (μsecs)				J/gcc:2	J/cc:2	J/cc:4
	J	gcc:2	cc:2	cc:4			
aquad	20383	16604	28883	26433	1.228	0.706	0.771
bessel	13984	27193	36718	36893	0.514	0.378	0.377
binomial	5538	5075	8894	6098	1.091	0.623	0.908
chebyshev	8884	7207	18067	18065	1.233	0.492	0.492
e	9681	9392	10148	10154	1.031	0.954	0.953
fib	4483	4727	4598	4584	0.948	0.975	0.978
fmatmult	22926	8748	15533	14894	2.621	1.476	1.539
log	16580	17487	35029	35029	0.948	0.473	0.473
mandelbrot	24109	19403	78423	46195	1.242	0.307	0.514
muldiv	12447	10605	11688	11669	1.174	1.065	1.067
pi	12146	11998	22528	22520	1.012	0.529	0.529
sum	1692	1606	1606	406	1.055	1.055	4.172
tak	5343	4384	4085	4070	1.218	1.298	1.303
zeta	18858	18029	38962	38792	1.046	0.484	0.486
Harmonic Mean :					1.051	0.624	0.709

Key : J : `jc -O`; `gcc:2` : `gcc -O2`; `cc:2` : `cc -O2`; `cc:4` : `cc -O4`

Table 1. The speed of `jc` compared to optimized C

only active process. For each benchmark program, a single "run" consisted of executing a test query 100 times in a tight loop and taking the shortest measured query execution time: the execution times reported here are 1/100 of the times obtained from such runs. Queries were designed to be large enough to exercise the programs, yet small enough to able to execute in a single timeslice with no system interruptions; taking the minimum measurement avoids bias when one or more query runs nonetheless happened to be interrupted. A single experiment consisted of a single run of each benchmark program, with the different benchmarks executed in random order within each experiment so as to avoid systemic bias from disk and memory cache effects. Nine such experiments were performed, and for each benchmark the median execution time was taken. For a more fair comparison with C, garbage collection—which involves runtime tests on the heap pointer—was turned off in `jc`, so the speed improvements measured do not take into account reductions in garbage collection time due to reduced heap usage. There is a constant overhead of 103μsec in the `jc` times compared with 6μsec for C compiled with `gcc`, due to the timing method and setup required for the benchmark call.

Compared to optimized C code, the baseline performance of our system—with register returns permitted, but no untagging optimizations—is fairly good: it is, on the average, about 57% slower than C code compiled with `gcc -O2`, and about 23% slower than that using `cc -O4`.[4] It is easy to take a poorly engineered system with a lot of inefficiencies and get huge performance improvements by eliminating

[4] Unless otherwise noted, all averages in this discussion refer to the harmonic mean.

Program	Memory Returns			Reg+Mem. Returns		
	T	U	U/T	T	U	U/T
aquad	48697	28187	0.579	37704	20383	0.541
bessel	40428	13272	0.328	40400	13984	0.346
binomial	4488	5747	1.280	4178	5538	1.326
chebyshev	26823	8894	0.332	26825	8884	0.331
e	12513	9655	0.772	12440	9681	0.778
fft	26168	25517	0.975	26104	26638	1.020
fib	11220	11073	0.987	4723	4483	0.949
fmatmult	32287	22486	0.696	32857	22926	0.698
log	35790	15744	0.440	35577	16580	0.466
mandelbrot	81677	24438	0.299	88008	24109	0.274
mcint	35374	16629	0.470	33832	15947	0.471
muldiv	13870	12465	0.900	13902	12447	0.895
pi	28840	11994	0.416	22764	12146	0.534
sum	1692	1692	1.000	1692	1692	1.000
tak	13637	13452	0.986	4760	5343	1.122
zeta	40571	18097	0.446	40638	18858	0.464
Harmonic Mean :			0.554			0.567

(a) Execution Time (μsecs)

Program	Memory Returns			Reg+Mem. Returns		
	T	U	U/T	T	U	U/T
aquad	30884	10255	0.3320	23332	544	0.0233
bessel	689	418	0.6067	689	452	0.6560
binomial	1208	249	0.2061	1026	6	0.0058
chebyshev	30002	6	0.0002	30002	6	0.0002
e	6005	6	0.0010	6005	6	0.0010
fib	6389	6389	1.0000	5	5	1.0000
fft	18669	16622	0.8904	18543	17364	0.9364
fmatmult	20649	5049	0.2445	20649	5049	0.2445
log	31494	12	0.0004	28866	6	0.0002
mandelbrot	69533	654	0.0094	69533	654	0.0094
mcint	26019	1019	0.0392	25495	17	0.0007
muldiv	5	5	1.0000	5	5	1.0000
pi	20007	9	0.0004	20007	6	0.0003
sum	5	5	1.0000	5	5	1.0000
tak	7121	7121	1.0000	5	5	1.0000
zeta	34460	285	0.0083	34460	223	0.0065

(b) Heap Usage (words)

Key : T : tagged values;   U : untagged values

Table 2. Performance Improvements due to Untagged and Unboxed Objects

some of these inefficiencies. The point of these numbers is that when evaluating the efficacy of our optimizations, we were careful to begin with a system with good performance, so as to avoid drawing overly optimistic conclusions.

The overall performance improvement obtained using untagged values is about 45%. On average, the resulting programs are about 5% slower than C compiled with `gcc -O2`, about 37% faster than `cc -O2`, and about 29% faster than `cc -O4`.[5] Moreover, `jc` outperforms `cc` on precisely those programs—namely, floating-point intensive computations—where one would expect a dynamically typed declarative language to do considerably worse than a statically typed imperative language. Interestingly, two programs—`binomial` and `tak`—do significantly worse using untagged values than using tagged values only. The problem arises from the effects of using C as the back-end compiler for `jc`, and the resulting lack of control, in `jc`, over hardware register allocation in the C compiler; a more controllable back-end would avoid such degradations.[6] These observations—and the fact that the numbers reported do not take into account performance improvements due to reductions in garbage collection time—imply that our execution time measurements give a conservative estimate of the true potential of these optimizations.

A further benefit of allowing untagged values can be seen in the decrease in heap usage. Table 2(*b*) separates out reductions in heap space usage due to register returns from those due to the use of untagged values. It can be seen that for every benchmark that used boxed data types, there was a consistent and significant reduction in heap usage, both when outputs were returned in memory and when they were returned in registers. In many cases, allowing output values to be returned in untagged registers allowed the entire computation to be carried out without any boxing operations at all, resulting in essentially trivial heap usage. (Not surprisingly, the use of untagged values makes no difference in the heap usage of integer computations, though these programs can be seen to benefit, in terms of heap usage, from being able to return output values in registers.) For short queries of the sort given here, the decrease of heap space does not contribute significantly to the execution time, because the maximum heap space used still fits easily within the data cache. However, for longer-running programs, reducing heap usage by avoiding boxed temporary values can result in a significant decrease in cache misses and garbage collection overhead.

The general algorithm described earlier uses a cost model to determine which

[5] Since `jc` uses `gcc` as its back end translator, one might wonder whether this comparison with `cc -O4` is "fair" or question what it proves. We claim that `jc`'s use of `gcc` is purely a matter of convenience: we could, in principle, have achieved the same results by writing our own back ends and replicating all of `gcc`'s technology in it. The point of this comparison, therefore, is merely to show that careful attention to low level concerns can allow implementations of declarative languages to attain performance that is competitive with the performance of imperative programs written in an imperative style. We acknowledge, of course, that performance comparisons between different languages are fundamentally dubious and very often have a strongly religious flavor, and caution the reader against reading too much into these results.

[6] While `gcc` version 2 provides extensions that provide some degree of user control over hardware register allocation, we do not use them at this time because they reserve a register for a variable for the entire lifetime of the variable, and therefore do not give us a sufficiently fine-grained control over register assignment.

Program	Local (L)	Args (A)	Global (G)	L/G	A/G
aquad	44055	28187	20383	2.161	1.383
bessel	30851	13272	13984	2.206	0.949
binomial	4463	5747	5538	0.806	1.038
chebyshev	15396	8894	8884	1.733	1.001
e	11034	9655	9681	1.140	0.997
fft	24897	25517	26638	0.935	0.958
fib	11220	11073	4483	2.502	2.470
fmatmult	28566	22486	22926	1.246	0.981
log	29299	15744	16580	1.767	0.950
mandelbrot	56867	24438	24109	2.359	1.014
mcint	24224	16629	15947	1.519	1.043
muldiv	13906	12465	12447	1.117	1.001
pi	19421	11994	12146	1.600	0.988
sum	1692	1692	1692	1.000	1.000
tak	13636	13452	5343	2.552	2.518
zeta	33692	18097	18858	1.787	0.960
Harmonic Mean :				1.454	1.092

(a) Execution Time (μsecs)

Program	Local (L)	Args (A)	Global (G)	L/G	A/G
aquad	24107	10255	544	44.31	18.85
bessel	30944	444	452	68.46	0.98
binomial	1148	249	6	191.3	41.50
chebyshev	10004	6	6	1667	1.00
e	4005	6	6	667.5	1.00
fft	16923	16622	17364	0.975	0.957
fib	6389	6389	5	1278	1278
fmatmult	12649	5049	5049	2.505	1.00
log	20998	12	6	3500	2.00
mandelbrot	31158	654	654	47.64	1.00
mcint	8087	1019	17	475.7	59.94
muldiv	5	5	5	1.00	1.00
pi	15007	9	6	2501	1.50
sum	5	5	5	1.00	1.00
tak	7121	7121	5	1424	1424
zeta	23317	285	223	104.6	1.28

(b) Heap Usage (words)

Table 3. Untagging Optimizations: Global vs. Local

Program	Tagged (T) (μsecs)	Untagged (U) (μsecs)	T/U
bsort	16422	16425	1.000
hanoi	15638	15478	1.010
lr1gen	22473	22431	1.006
nrev	7073	7072	1.000
pascal	8998	9059	0.993
qsort	11409	11409	1.000
queen	6583	6585	1.000
Harmonic Mean			1.001

Table 4. The Effect of Untagging Optimizations on Non-numerical Programs

values may be returned in (unboxed) registers. As discussed above, this provides good performance improvements. However, it has the disadvantage that it requires nontrivial extensions to the compiler. It is reasonable to inquire to what extent performance might be improved using restricted versions of our algorithm that require minimal extensions to the compiler where untagged values are supported but no provision is made for returning values in registers. We next consider the two extremes possible for such minimal extensions. The simplest, and most restricted, case uses purely local untagging: it maintains values in untagged form through the body of a procedure if this is deemed useful, but values that are passed across procedure boundaries (this includes values passed into tail calls) are passed in boxed form. At the other extreme, untagged values are allowed as input arguments to procedures as well, though output values are returned in memory (and therefore are represented in tagged form). The performance improvements resulting from these restricted versions of the untagging optimization are shown in Table 3, where the column marked "Local" gives the performance numbers resulting from purely local untagging; that marked "Args" refers to local untagging together with untagged arguments; and "Global" gives the performance using the general untagging optimization. It can be seen that for the benchmarks tested, purely local untagging results in an improvement of about 9% on the average compared to no untagging at all. This is not insignificant, but the resulting programs are still considerably slower—about 45% on the average—than those using the general optimization. However, when untagged arguments are allowed, performance improves considerably, and the resulting code is only about 9% slower than code using the general optimization. The reason for this is that the programs tested spend most of their time in simple loops, and these can be essentially fully optimized when untagged input arguments are allowed. We conjecture that this is true of most numerical programs, with much of the execution time accounted for by loop computations, and that such programs can benefit considerably even from the simple optimization of allowing untagged local computations and input arguments.

Another important consideration is the effect of untagging optimizations on non-numerical programs. As discussed earlier, our optimization relies greatly on being able to maintain untagged values in registers. In an implementation that has an *a priori* fixed mapping from virtual machine registers to physical registers, this can cause some registers to be unnecessarily dedicated to untagged values, even for programs where there is no opportunity for untagging optimizations, and this can cause a degradation in performance. The `jc` system avoids this problem by having the compiler generate untagged virtual machine registers (via C language declarations) only if it determines that there is some opportunity for maintaining values in untagged form. The virtual machine registers so generated are mapped to physical registers based on estimated usage counts (currently this is done entirely by the C compiler), which means that even when an untagged virtual machine register is generated, it is allocated to a physical register only if it is used sufficiently many times to justify this. Experimental results for a number of small non-numerical benchmarks are shown in Table 4. The programs used were the following: *bsort* uses bubble sort to sort a list of 100 integers; *hanoi* is the Towers of Hanoi program (adapted from an FCP program by S. Kliger): the numbers given are for *hanoi*(10); *lr1gen* is the core of an LR(1) parser generator; *nrev* is the naive reverse program on an input list of length 100; *pascal* is a benchmark, by E. Tick, to compute Pascal's triangle; *qsort* is a quicksort program, executed on a list of length 100; and *queen* is the n-queens program: the numbers given are for 6 queens. The numbers given in Table 4 indicate that the performance of these programs, with and without the untagging optimization, is essentially identical. This indicates that non-numerical programs need not suffer a performance degradation due to the use of untagged values. We believe that this conclusion extends also to larger programs, consisting of some components that are primarily numerical in nature and others that are primarily non-numerical. The reason for this is that modern register allocation algorithms (see, e.g., [11]) base their decisions on the relative usage counts of variables in different regions of a program: a variable that is heavily used in one region of a program, but not in another, will be considered for placement in a register in the first region but not in the second. Using such algorithms, therefore, it is possible to take advantage of untagging optimizations in those portions of a program that can benefit from it, without having to suffer a performance degradation in those parts of a program that do not benefit from the use of untagged values.

Finally, there is the issue of the compile-time cost of implementing this optimization. We have not separately measured the time taken by the analysis algorithms, because dataflow analysis and optimization accounts for a very small part of the overall compilation time. Because Janus programs are compiled to C code which is then processed by a C compiler, most of the overall time for translation to the object code is spent in I/O operations and in the C compiler (other systems that compile to C, e.g., KLIC [10], report similar experiences). As a result, there is no perceptible decrease in the overall compile time when dataflow analysis and optimizations are switched off.

5. Extensions

The discussion thus far has not considered the question of backtracking, which is of fundamental importance in non-committed-choice logic programming languages, e.g, Prolog. In such languages, programs have to save a certain amount of state at points that execution can backtrack to, and restore this information appropriately when backtracking actually takes place. The state information that is saved typically consists of two parts: some machine status information, together with information about certain registers, kept in runtime structures commonly called "choice points"; and information about variables whose values need to be reset, maintained in an (usually separate) area called the trail. Conceptually, a choice point consists of one component that represents a fixed amount of machine status information, and another component, of variable size, that represents information about the local state of a procedure, in particular its arguments. Components of the runtime system that are able to inspect the state of a running program, such as garbage collectors, must then be able to identify choice points and correctly interpret (the variable-size component of) their structure.[7] If untagged values are supported, we must therefore be able to save untagged values when creating a choice point; restore untagged values when backtracking occurs; and be able to specify, for the benefit of the garbage collector, which components of a choice point represent untagged values.

The simplest approach to handling untagged values in the presence of backtracking would be to prohibit untagged arguments for any procedure that may create a choice point (this is a somewhat stronger requirement than determinacy). This has the virtue of simplicity, and may be acceptable in some limited contexts: for example, this may be a reasonable option if we consider only untagged values for numerical types, since traditional numerical programs tend to be deterministic. A minor variation on this scheme is to allow procedures that may backtrack to take untagged arguments, as long as these are converted to tagged form before being stored in a choice point. The problem with this is that we need to maintain a fair amount of information about these values in order to restore the tagged values into the appropriate untagged registers, and this negates the primary advantage of allowing only tagged values in choice points, namely, simplicity.

A less restrictive option is to allow untagged values to be stored in choice points. This makes it necessary to maintain information about which slots in the choice point correspond to untagged values, the type of each such value, and the register from which the value originated. This can be done in at least two ways:

1. The information can be kept in a data structure that is part of the symbol table entry for each (nondeterministic) procedure, similarly to the scheme described in Section 3.3 for stack frames. The disadvantage of this scheme is that this information must be interpreted during execution. This would make backtracking a relatively expensive operation.

2. For any given procedure, the untagged values that need to be saved and restored at a choice point, and the register corresponding to each such value, will be known by the compiler. It can therefore generate code to save and restore these values. This is likely to be considerably more efficient than having to interpret a data structure at runtime. This code can be generated either as a lightweight parameterless function that is called from each point in a procedure where choice point manipulation occurs, or generated in-line

[7] The garbage collector does not need to inspect the trail, since any variable recorded in the trail is also accessible from some choice point [3].

at each such point, depending on the relative importance of code size vs. execution speed.

It is also necessary to communicate information about the structure of choice points to the garbage collector: this can be done, as suggested above, via the symbol table of the procedure. In this case, since saving and restoring of untagged registers does not involve the symbol table, it is necessary only to store information about which slots in the choice point contain untagged values, and the types of those values.

Another issue of considerable importance for real applications is separate compilation. It is not easy to reconcile untagged values with separate compilation: as the discussion thus far indicates, considerable cooperation and communication is needed between two procedures if they are to pass untagged values between themselves, and this is precisely what is absent in separate compilation. There are two issues that have to be addressed: first, program analysis in the presence of separately compiled code; and second, generating code to ensure that values can be communicated correctly between the caller and callee, which reside in separately compiled modules. The first problem can be handled using techniques for compositional and/or incremental program analysis [8, 12]. There are two alternatives for handling the second problem. If the different modules of a program are compiled and loaded in sequence, so that the code generated for one module is available while code is being generated for another module, then incremental optimization [8] using multiple entry points (see Section 3.1) can be used to avoid the overhead of passing tagged values across module boundaries where possible. An alternative would be to generate multiple entry points for procedures that use untagged values, and use optimizations such as code hoisting and call forwarding [13] at link-time (see, e.g., [14, 29]) to redirect calls so as to avoid unnecessary tagging and untagging where possible. While these techniques can be used to avoid passing untagged values into a procedure, neither supports untagged return values in a straightforward way. Given the discussion of Section 3 and the experimental results of Section 4, this can be a significant limitation. Nevertheless, being able to pass untagged arguments across module boundaries at all would be a considerable improvement on our current implementation, which restricts inter-module calls to use only tagged values.

6. Related Work

The work that is probably the closest to ours is that of the Python compiler for CMU Common Lisp, which uses untagged representations for numeric objects where possible, including the passing of arguments and return values in function calls [23]. While our implementation does not currently allow untagged objects to be heap-allocated, CMU Common Lisp allows explicitly-typed array and structure slots (which are heap-resident) to contain untagged values, provided that all of the values in the array or structure can be guaranteed at compile-time to contain only untagged values. The Python system also differs in the way it supports garbage collection: it uses two different stacks, one containing only tagged values and the other containing only untagged values [24]. The specific algorithms used by the Python compiler for representation analysis are not, as far as we have been able to determine, extensively documented; however, we believe that overall, due in part to linguistic aspects of Common Lisp, our algorithms are considerably simpler than those used by Python.

The problem of generating efficient numeric code for Lisp programs was considered as far back as the MacLisp compiler [30] and the S-1 Lisp compiler [6]. These systems used untagged representations for numbers in intra-procedural numerical computations, but used boxed (though not necessarily heap-allocated) values across procedure boundaries. The representation analyses used by the S-1 Lisp compiler involved two passes over the intermediate representation—the first a top-down pass to determine a "desired" representation, the second a bottom-up pass to determine a "deliverable" representation—and is considerably more complicated than that described here. An elegant algorithm for the optimal placement of representation conversion operations in a program flow graph with execution frequency information, based on network flow algorithms, was given by Peterson [27]: however, to our knowledge this algorithm has not been implemented. Metzemakers *et al.* discuss the use of partial evaluation techniques at the intermediate code level for the removal of redundant boxing/unboxing operations [26].

More recently, the issue of maintaining values in untagged form has received considerable attention in the context of strongly typed polymorphic languages (see, for example, [20, 22]). However, this work relies on the underlying type system in a fundamental way, and is therefore very different from ours: it involves making boxed and unboxed representations of objects explicit at the source level using "representation types", and formulating boxing and unboxing operations as source-level transformations. The problem of garbage collection in tagless implementations of such languages is discussed by a number of authors, including Appel [1] and Goldberg [17].

7. Conclusions

Most implementations of dynamically typed languages have historically suffered in comparison to statically typed languages because their very nature imposes overheads even when working with consistently and uniquely typed programs. These overheads are incurred in the process of converting values between the general "boxed" form, and the "unboxed" form on which the underlying hardware must operate. In systems where boxed values are heap-allocated, there is a further degradation due to garbage collection time and the inefficient use of the cache as intermediate values are created, used once, and left behind.

We have presented a discussion of simple heuristics which, when combined with a variety of analyses (in particular, mode, type, and non-suspension analyses) desired independently for other optimizations, and the extension of the local register allocator to consider different types of registers simultaneously, yield a speedup on numerical programs, written in a dynamically typed language, of about 45% above an already optimized compiler which did not attempt to maintain untagged values.

The resulting programs run within 5% of the same programs written in C in a natural C style and compiled using `gcc -O2`, and are considerably faster than the corresponding C code optimized using `cc -O2` and `cc -O4`. In addition, heap use is also reduced dramatically. The optimizations described here should be applicable to almost any implementation of a dynamically typed language.

REFERENCES

1. A. W. Appel, "Runtime tags aren't necessary", *Lisp and Symbolic Computation* 2:153–162, 1989.

2. A. Appel, *Compiling with Continuations*, Cambridge University Press, 1992.

3. K. Appleby, M. Carlsson, S. Haridi and D. Sahlin, "Garbage Collection for Prolog based on WAM", *Communications of the ACM* vol. 31 no. 6, June 1988, pp. 719–741. ACM Press.

4. R. L. Bates, D. Dyer, and J. A. G. M. Koomen, "Implementation of Interlisp on the VAX", *Proc. 1982 ACM Symp. on Lisp and Functional Programming*, Aug. 1982, pp. 81–87.

5. P. A. Bigot, D. Gudeman, and S. K. Debray, "Output Value Placement in Moded Logic Programs", *Proc. Eleventh International Conf. on Logic Programming*, June 1994, pp. 175–189. MIT Press.

6. R. A. Brooks, R. P. Gabriel, and G. L. Steele Jr., "An Optimizing Compiler for Lexically Scoped Lisp", *Proc. SIGPLAN '82 Symp. on Compiler Construction*, June 1982, pp. 261–275.

7. R. A. Brooks, R. P. Gabriel, and G. L. Steele, Jr., "S-1 Common Lisp Implementation", *Proc. 1982 ACM Symp. on Lisp and Functional Programming*, Aug. 1982, pp. 108–113.

8. F. Bueno, D. Cabeza, M. Hermenegildo, and G. Puebla, "Data–flow Analysis of Standard Prolog Programs", *Proc. European Symposium on Programming*, April 1996, pp. 108–124. Springer-Verlag LNCS vol. 1058.

9. M. Carlsson, "The SICStus Prolog Emulator", Technical Report T91:15, Swedish Institute of Computer Science, Sept. 1991.

10. T. Chikayama, T. Fujise, and D. Sekita, "A Portable and Efficient Implementation of KL1", *Proc. Int. Symp. on Programming Language Implementation and Logic Programming*, Sept. 1994, pp. 25–39

11. F. C. Chow and J. L. Hennessy, "The Priority-Based Coloring Approach to Register Allocation", *ACM Transactions on Programming Languages and Systems* vol. 12 no. 4, Oct. 1990, pp. 501–536.

12. M. Codish, S. K. Debray, and R. Giacobazzi, "Compositional Analysis of Modular Logic Programs", *Proc. Twentieth ACM Symposium on Principles of Programming Languages*, Charlotte, SC, Jan. 1993, pp. 451–464.

13. K. De Bosschere, S. K. Debray, D. Gudeman, and S. Kannan, "Call Forwarding: A Simple Interprocedural Optimization Technique for Dynamically Typed Languages", *Proc. 21st. ACM Symp. on Principles of Programming Languages*, Jan. 1994, pp. 409–420.

14. K. De Bosschere and S. K. Debray, "`alto`: A Link-Time Optimizer for the DEC Alpha", Draft report, Dept. of Computer Science, The University of Arizona, Tucson, July 1996.

15. S. K. Debray, D. Gudeman, and P. A. Bigot, "Detection and Optimization of Suspension-free Logic Programs", *J. Logic Programming* (Special Issue on High Performance Implementations), to appear. (Preliminary version appeared in *Proc. 1994 International Symp. on Logic Programming*, Nov. 1994, pp. 487–501, MIT Press.)

16. I. Foster and S. Taylor, "Strand: A Practical Parallel Programming Tool", *Proc. 1989 North American Conf. on Logic Programming*, Oct. 1989, pp. 497-512. MIT Press.

17. B. Goldberg, "Tag-Free Garbage Collection for Strongly Typed Programming Languages", *Proc. SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, Canada, June 1991, pp. 165–176.

18. D. Gudeman, "Representing Type Information in Dynamically Typed Languages", Technical Report TR 93-27, Dept. of Computer Science, The University of Arizona, Oct. 1993.

19. D. Gudeman, K. De Bosschere, and S.K. Debray, "`jc`: An Efficient and Portable Sequential Implementation of Janus", *Proc. Joint International Conf. and Symp. on Logic Programming*, Nov. 1992, pp. 399–413.

20. F. Henglein and J. Jørgensen, "Formally Optimal Boxing", *Proc. 21st. ACM Symp. on Principles of Programming Languages*, Jan. 1994, pp. 213–226.

21. D. A. Krantz, *ORBIT: An Optimizing Compiler for Scheme*, Ph.D. Dissertation, Yale Unicersity, 1988.

22. X. Leroy, "Unboxed objects and polymorphic typing", *Proc. 19th. ACM Symp. on Principles of Programming Languages*, Jan. 1992, pp. 177–188.

23. R. A. MacLachlan, "The Python Compiler for CMU Common Lisp", *Proc. ACM Conf. on Lisp and Functional Programming*, 1992, pp. 235–246.

24. R. A. MacLachlan, personal communication, Oct. 1994.

25. T. G. Mattson, "The Strand Language: Scientific Computing meets Concurrent Logic Programming", *Proc. Workshop on Parallel Implementation of Languages for Symbolic Computation*, eds. A. Ciepielewski and E. Tick, July 1990. Technical Report CIS-TR-90-15, Dept. of Computer and Information Science, University of Oregon, Eugene, Oregon.

26. T. Metzemakers, A. Miniussi, D. Sherman, and R. Strandh, "Improving Arithmetic Performance using Fine-Grain Unfolding", *Proc. 6th. International Symp. on Programming Language Implementation and Logic Programming*, Sept. 1994, pp. 324–339. Springer-Verlag LNCS vol. 844.

27. J. Peterson, "Untagged Data in Tagged Environments: Choosing Optimal Representations at Compile Time", *Proc. Functional Programming Languages and Computer Architecture*, 1989.

28. W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C*, Cambridge University Press, 1988.

29. A. Srivastava and D. W. Wall, "A Practical System for Intermodule Code Optimization at Link-Time", *Journal of Programming Languages*, pp. 1–18, March 1993.

30. G. L. Steele Jr., "Fast Arithmetic in MacLISP", *Proc. 1977 MACSYMA Users' Conference*, NASA Scientific and Technical Information Office, Washington D.C., July 1977, pp. 215–224.

31. P. A. Steenkiste, "The Implementation of Tags and Run-Time Type Checking", in *Topics in Advanced Language Implementation*, ed. P. Lee, pp. 3–24. MIT Press, 1991.

32. P. A. Steenkiste and J. Hennessy, "Lisp on a reduced-instruction-set-processor", in *Proc. 1986 ACM Conf. on Lisp and Functional Programming*, Aug. 1986, pp. 192–201.

33. A. Taylor, "Removal of Dereferencing and Trailing in Prolog Compilation", *Proc. Sixth International Conference on Logic Programming*, June 1989, pp. 48–60. MIT Press.

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

35. P. Van Roy, *Can Logic Programming Execute as Fast as Imperative Programming?* PhD thesis, University of California at Berkeley, 1990.

36. P. Wadler, "Deforestation: Transforming programs to eliminate trees", *Proc. European Symp. on Programming*, March 1988, pp. 344–358. Springer-Verlag LNCS vol. 300.