Run time type information in Mercury

Tyson Dowd, Zoltan Somogyi, Fergus Henderson,
Thomas Conway, and David Jeffery

{trd,zs,fjh,conway,dgj}@cs.mu.OZ.AU
Department of Computer Science and Software Engineering,
University of Melbourne, Parkville, 3052 Victoria, Australia
Phone: +61 3 9344 9100, Fax: +61 3 9348 1184

Abstract. The logic/functional language Mercury uses a strong, mostly static type system based on polymorphic many-sorted logic. For efficiency, the Mercury compiler uses type specific representations of terms, and implements polymorphic operations such as unifications via generic code invoked with descriptions of the actual types of the operands. These descriptions, which consist of automatically generated data and code, are the main components of the Mercury runtime type information (RTTI) system. We have used this system to implement several extensions of the Mercury system, including an escape mechanism from static type checking, generic input and output facilities, a debugger, and automatic memoization, and we are in the process of using it for an accurate, native garbage collector. We give detailed information on the implementation and uses of the Mercury RTTI system as well as measurements of the space costs of the system.

1 Introduction

Many modern functional and logic programming languages have a strong static type system and support parametric polymorphism. For efficiency, since the types of almost all values are known at compile-time, it is desirable to specialize the representation of data for each type, rather than using a single representation for data of any type (as is typically done with dynamically typed languages). When the type is known statically, the compiler is able to emit the proper type-specific code to manipulate those values. In some cases, however, the compiler does not know the exact type of a value. For example, in a polymorphic predicate or function, the compiler may know that type of a variable is `list(T)`, but may not know what type the type variable `T` is bound to, since that can vary from call to call. Nevertheless, for some operations it may still be necessary to examine the representation.

In such circumstances, implementors have two main choices. One alternative is to create separate copies of the implementation for each possible type `T` can be bound to, thus restoring the compiler's full knowledge of the types of variables. This is the approach taken for the implementation of generics in most imperative languages. Its advantage is execution speed, due to the exclusive use of type-specific operations; the corresponding disadvantage is the cost in code space and locality of the multiple copies, many of which will typically be used quite rarely. Another significant disadvantage with this approach is that it makes separate compilation much more difficult.

The other alternative is to have only one implementation, but make this one able to handle all the calls. This obviously requires callers to make available runtime type information (RTTI) about the actual type bound to `T` that can then be interpreted by a single implementation. The advantages of this alternative are its small space cost, and ease of separate compilation, while its disadvantage is the time costs of lookup and interpretation.

In this paper we describe the RTTI system of Mercury, a purely declarative logic/functional language. Since most Mercury programs use polymorphism much more frequently than imperative language programs use generics, we believe that the space cost of the first approach would be prohibitive. However, we also want the implementation to be fast, so we have settled on a hybrid of the approaches. This hybrid uses RTTI to allow us to get away with only one implementation of each polymorphic predicate, but the most frequently used operations (unification and comparison) do not require interpretation. Other, less frequently used operations do, since for them this is the proper space-time tradeoff.

Since the system has RTTI, we make it available to users who may wish to perform type specific operations (e.g. pretty-printing) on terms of polymorphic types, as well as to system programmers working to implement new language features. In the last two years, we have extended the Mercury implementation with several features that require access to RTTI, some of which required us to extend the RTTI system. Automatic memoization requires detailed knowledge of the data representations of types to construct efficient indexes. The debugger needs similar knowledge in order to be able to print out the values of variables on demand, and our (as yet incomplete) native garbage collector needs it to be able to trace through and to copy terms. (The Mercury runtime system currently relies upon the Boehm conservative garbage collector for C [5].)

The rest of this paper is organized as follows. Section 2 introduces the relevant aspects of the Mercury language and describes how the Mercury implementation represents terms. Section 3 describes, at a significantly deeper level of detail than most other papers on RTTI, the data structures we use to store RTTI and how the information in these data structures is made available both to relevant parts of the implementation and to programmers. Section 4 evaluates the space impact of our RTTI implementation. Section 5 presents comparisons with related work.

2 Background

2.1 Mercury

Mercury is a pure logic/functional programming language intended for general purpose large-scale programming. We will describe in detail the data representation used by the Mercury implementation, but for an overview of Mercury we refer the reader to the Mercury language reference manual [10] which is available from the Mercury home page `http://www.cs.mu.oz.au/mercury/`.

2.2 Data Representation

The Mercury implementation uses a different, specialized representation for the terms of each type. This is possible because the Mercury compiler knows the types of almost all terms at compile time, and the few exceptions do not present insoluble problems; we will discuss the solutions of some of these problems later. The advantage of specializing term representations is that it reduces storage requirements somewhat and improves time efficiency considerably. The disadvantage is that you cannot tell the value represented by a bit pattern without knowing what type it is.[1]

```
:- type dir ---> north ; south ; east ; west.
:- type example ---> a ; b(int, dir) ; c(example).
```

Types such as `dir`, in which every alternative is a constant, correspond to enumerated types in other languages. Mercury implements them as if they were enumerated types, representing the alternatives by consecutive small integers starting with zero. These integers are stored directly in machine words, as are values of builtin types that fit in a word, such as integers. Values of builtin types that do not fit in a word, e.g. strings and (on some machines) double precision floating point numbers, are stored in memory and represented by a pointer to that memory, to allow us to establish the invariant that all values fit into a word. Polymorphic code depends on this invariant; without it, the compiler could not generate a single piece of code that can pass around values of a type that is unknown at compile time.

Types such as `example`, in which some alternatives have arguments, obviously need a different representation. One possible representation would be as a pointer to a memory block, in which the first word specifies the function symbol, and the later words contain its arguments, with the block being just big enough to contain these arguments. The size of the memory block may therefore depend on the identity of the function symbol.

The Mercury implementation uses a more sophisticated and efficient variant of this representation. This implementation exploits the fact that virtually all modern machines, and all those we are interested in, address memory in bytes but access it in words. Many of these machines require words to be aligned on natural boundaries, and even the ones that don't usually suffer a performance penalty when accessing unaligned words. The Mercury implementation therefore stores all data in aligned words on all machines. This means that the address of any Mercury data item will have zeros in its low-order 2 bits on 32-bit machines or low-order 3 bits on 64-bit machines. We can therefore use these bits, which we call *primary tags*, to distinguish between function symbols. (Mercury works on both 32-bit and 64-bit machines, but for simplicity of exposition, we will assume 32-bit machines for the rest of this paper.)

In the case of type `example`, we assign the primary tag value 0 to `a`, the primary tag value 1 to `b`, and the primary tag value 2 to `c`. Using primary tags in this way allows us to reduce the size of the memory blocks we use, since these no longer have to identify the function symbol, It also allows us to avoid using a memory block at all for constant function symbols such as `a`, whose representation therefore does not need a pointer at all, and in which we therefore set the non-primary-tag bits of the word to zero.

Of course, some types have more function symbols than a primary tag can distinguish. In such cases, some function symbols have to share the same primary tag value. If all functors sharing the same primary tag value are constants,

[1] If a function symbol occurs in more than one type, it will in general be represented differently for each type. The type checker will determine the type of each occurrence of the functor. The different representations do not cause any problems, since terms of different types cannot be compared directly.

we distinguish them via the non-primary-tag bits of the word; we call this a *local secondary tag*. If at least one of them is not a constant, we distinguish them by storing an extra word at the start of the argument block; we call this a *remote secondary tag*. Both local and remote secondary tags are integers allocated consecutively from zero among the function symbols sharing the relevant primary tag. The compiler has a fairly simple algorithm that decides, for each function symbol, what primary tag value its representation has, and if that primary tag value is shared, whether the secondary tag is local or remote and what its value is. To save both space and time, this algorithm will share a primary tag value only between several constants or several non-constants, and will not share between a constant and a non-constant.

Following pointers that have primary tags on their low order bits does not actually cost us anything in execution time. To make sense of the word retrieved through the pointer, the code must have already tested the primary tag on the pointer. (It does not make sense to look up an argument in a memory block without knowing what function symbol it is an argument of, and it does not make sense to look at a remote secondary tag without knowing what function symbols it selects among.) When following the tagged pointer, the implementation must subtract the (known) value of the primary tag and add the (known) offset in the pointed-to memory block of the argument or remote secondary tag being accessed. (Actually, remote secondary tags always have an offset of zero.) The two operations can be trivially combined into one, which means adding a possibly negative, but small constant to the pointer. On most machines, this is the most basic memory addressing mode; one cannot access memory faster any other way.

3 Run-time Type Information

One important design principle of the Mercury implementation, which we followed during our design of the RTTI system, is the avoidance of "distributed fat", which are implementation artifacts required by one language feature that impose efficiency costs even when that feature is not used. In other words, we don't want the RTTI system to slow down any parts of the program that do not use RTTI. Of course, we also want the RTTI system to have good efficiency for the parts of the program that do use RTTI. The aspect of efficiency that most concerns us is time efficiency; we are usually willing to trade modest and bounded amounts of space for speed.

3.1 Describing Type Constructors

The Mercury data representation scheme is compositional, i.e. the representation of a list does not depend on the type of the elements of the list. Therefore the runtime representation of a composite type such as `tree(string, list(int))` can be described by writing down the representation rules of the type constructors occurring in the type and showing how these type constructors fit together. Since a given type constructor will usually occur in many types, storing the information about the type constructor just once and referring to it from the descriptions of many types is obviously sensible. We call the data structure that holds all the runtime type information about a type constructor a `type_ctor_info`. When compiling a module, the Mercury compiler automatically generates a static data structure containing a `type_ctor_info` for every `:- type` declaration in the module. This data structure has a unique but predictable name derived from the name of the type constructor, which makes it simple to include references to it in other modules.

The `type_ctor_info` is a pointer to a vector of words containing the following fields:

– the arity of the type constructor,
– the address of the constructor-specific unification procedure,
– the address of the constructor-specific index procedure,
– the address of the constructor-specific comparison procedure,
– a pointer to the constructor's `type_ctor_layout` structure,
– a pointer to the constructor's `type_ctor_functors` structure, and
– the module qualified name of the type constructor.

Like the `type_ctor_info`, the constructor-specific unification, index and comparison procedures and `type_ctor_layout` and `type_ctor_functors` structures are also automatically generated by the compiler for each type declaration. We will provide details on these fields later.

3.2 Describing Types

A type is a type constructor applied to zero or more arguments, which are themselves types. Due to the compositionality of data representation in Mercury, the data structure that holds all the runtime type information about a type, which we call a `type_info`, is a pointer to a vector of words containing

– a `type_ctor_info` pointer, and
– zero or more `type_info` pointers describing the argument types.

The number of other `type_info` pointers is given by the arity of the type constructor, which can be looked up in the `type_ctor_info`. If the arity is zero, this representation is somewhat wasteful, since it requires an extra cell and imposes an extra level of indirection. The Mercury system therefore has an optimization which allows the `type_ctor_info` for a zero-arity type constructor to also function as a `type_info` for the type named by the type constructor. Code that inspects a `type_info` now needs to check whether the `type_info` has the structure just above or whether it is the `type_ctor_info` for a zero-arity type constructor. Fortunately, the check is simple; whereas the first word of a real `type_info` structure is a pointer and can never be null, the first word of the `type_ctor_info` structure contains the arity of the constructor, and therefore for a zero-arity constructor will always be null. This can be a worthwhile optimization, because zero-arity types occur often; the leaves of every type tree are type constructors of arity zero.[2] Figure 1 shows the `type_info` structure of the type `tree(string, list(int))` with this optimization.

3.3 Implementing Polymorphism

In the presence of polymorphism, the compiler cannot always know the actual type of the terms bound to a given variable in a given predicate. If an argument of a polymorphic predicate is e.g. of type `T`, then for some calls the argument

[2] On some modern architectures, mispredicted branches can be more expensive than memory lookups, which often hit in the cache. For these architectures, the Mercury compiler has a switch that turns off this optimization.

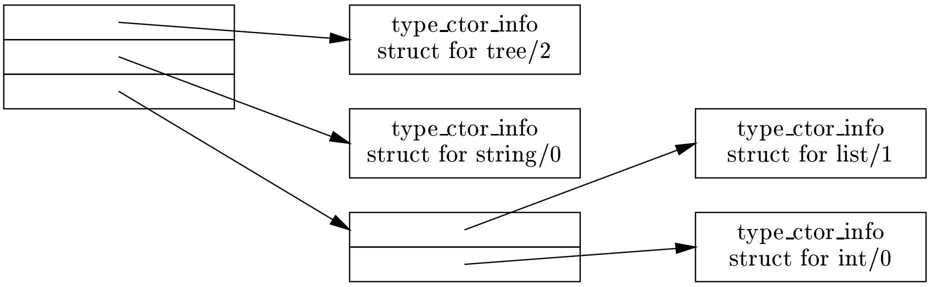

Fig. 1. The `type_info` structure for `tree(string, list(int))`

will be a term of type `int`, for others a term of type `list(string)`, and so on. The question then is: how can the compiler arrange the correct functioning of operations (such as unification) that depend on the actual type of the term bound to the variable?

The answer is that the compiler can make available to those operations the `type_info` for the actual type. An early phase of the compiler inspects every predicate, and for each type variable such as `T` in the type declaration of the predicate, it adds an extra argument to the predicate; this argument will contain a `type_info` for the actual type bound to `T`. The same phase also transforms the bodies of predicates so that calls to polymorphic predicates set these arguments to the right values.

As an example, consider a predicate `p1` that passes an argument of type `tree(string, list(int))` to a predicate `p2` that expects an argument of type `tree(T1, T2)`. Since `T1` and `T2` are type variables in the signature of `p2`, the compiler will add two extra arguments to `p2`, one each for `T1` and `T2`. As the value of the first of these extra arguments, `p1` must pass the `type_info` for `string`; as the value of the second of these extra arguments, `p1` must pass the `type_info` for `list(int)`. If `p1` does not already have pointers to the required `type_info` structures, it must construct them. This means that while there can be only one `type_ctor_info` structure for each type constructor, there may be more than one `type_info` structure and therefore more than one `type_info` pointer for each type.

If `p2` wants to pass a value of type `list(T1)` to a predicate `p3` that expects a value of type `U`, `p2` can construct the `type_info` structure expected by `p3` even though the type bound to `T1` is not known at compile time. To create this `type_info` structure, the compiler simply emits code that creates a two-word cell on the heap, and copies the pointer to the globally known `type_ctor_info` for `list/1` to the first word and the pointer it has to the `type_info` for `T1` to the second word.

3.4 Implementing Unification and Comparison

One operation that polymorphic predicates frequently perform on their polymorphic arguments is unification (consider `member/2`). To unify two values whose type it does not know, the compiler calls `unify/2`, the generic unification procedure in the Mercury runtime system. Since `unify/2` is declared to take two arguments of type `T`, the polymorphism transformation will transform calls to it by adding an extra argument containing the `type_info` describing the common type of the two original arguments.

The implementation of `unify/2` consists of looking up the address of the unification procedure in the top-level `type_ctor_info` of the `type_info`, and calling it with the right arguments. For builtin type constructors, the unification procedures are in the runtime system; for user-defined type constructors, they are automatically generated by the compiler. The technique the compiler uses for this is quite simple; it generates one clause for each alternative functor in the type constructor's type declaration, and in each clause, it generates one unification for each argument of that functor. Here is one example of a type and its automatically generated unification predicate:

```
:- type tree(K, V) ---> leaf ; node(tree(K, V), K, V, tree(K, V)).

unify_tree(leaf, leaf).
unify_tree(node(L1, K1, V1, R1), node(L2, K2, V2, R2)) :-
        unify(L1, L2),
        unify(K1, K2),
        unify(V1, V2),
        unify(R1, R2).
```

After creating the unification predicate, the compiler optimizes it by recognizing that for the first and last calls to `unify`, the top-level constructor of the type is known, and that those calls can thus be replaced by calls to `unify_tree` itself. Later still, the optimized predicate will go through the polymorphism transformation, which yields the following code:

```
unify_tree(TI_K, TI_V, leaf, leaf).
unify_tree(TI_K, TI_V, node(L1, K1, V1, R1),
                node(L2, K2, V2, R2)) :-
        unify_tree(TI_K, TI_V, L1, L2),
        unify(TI_K, K1, K2),
        unify(TI_V, V1, V2),
        unify_tree(TI_K, TI_V, R1, R2).
```

This shows that when the generic unification predicate `unify` is called upon to unify two trees, e.g. two terms of type `tree(string, list(int))`, two of the arguments it must call `unify_tree` with are the `type_info`s of the types `string` and `list(int)`. It can do so easily, since the required `type_info`s are exactly the ones following the `type_ctor_info` of `tree/2` in the `type_info` structure of `tree(string, list(int))`, a pointer to which was passed to `unify` as its extra argument. (`unify` of course got the address of `unify_tree` from the `type_ctor_info` of `tree/2`.)

Automatically generated comparison predicates call automatically generated index predicates which return the position of the top-level functor of a term in the list of alternative functors of the type. This allows for comparisons to be made for less than, equal to or greater than without comparing each functor to every other functor. After the initial comparison the comparison code has a similar recursive structure to the code generated for unification, and the polymorphism transformation is analogous.

3.5 Interpreting Type-specialized Term Representations

Some polymorphic predicates wish to perform operations on polymorphic values for which there is no compiler-generated type-representation-specific code the way there is for unifications and comparisons. Copying terms and printing terms are examples of such operations. In such cases, the implementation of the operation must itself decode the meaning of a term in a type-specific data representation. Since it is the compiler that decides how values of each type are represented, this requires cooperation from the compiler. This cooperation takes the form of a compiler-generated `type_ctor_layout` structure for each type constructor, pointed to from the `type_ctor_info` structure of the constructor. Like the `type_ctor_info`, the `type_ctor_layout` structure is static, and there is only ever one `type_ctor_layout` for a given type constructor.

Since most values in Mercury programs belong to types which are discriminated unions, we chose to optimize `type_ctor_layout` structures so that given a word containing a value belonging to such a type, it is as efficient as possible to find out what the term represented by that word is. The `type_ctor_layout` is therefore a vector of descriptors indexed by the primary tag value of the data word, which thus contain four descriptors on 32-bit machines and eight on 64-bit machines. Each word says how to interpret data words with the corresponding primary tag. For types which are not discriminated unions (such as `int`), and thus do not use primary tags, all the descriptors in the vector will be identical; since there are few such types, and the vectors are small, this is not a problem.

To make the `type_ctor_layout` as small as possible, each descriptor is a single tagged word; here we use a 2-bit descriptor tag regardless of machine architecture. The value of this descriptor tag, which can be unshared, shared_remote, shared_local or equivalence, tells us how to interpret the rest of the word.

If the value of the descriptor tag is unshared, then this value has a discriminated union type and the primary tag of the data word uniquely identifies the functor. The rest of the descriptor word is then a pointer to a *functor descriptor* which contains

- the arity of the functor (n),
- n `pseudo_type_info`s for the functor arguments,
- a pointer to a string containing the functor name, and
- information on the primary tag of this functor, and its secondary tag, if any.

The last field is redundant when the functor descriptor is accessed via the `type_ctor_layout` structure; it is used only when it is accessed via the `type_ctor_functors` structure which is discussed below in section 3.6.

Many type declarations contain functors whose arguments are of polymorphic type; for example, all the arguments of the functor `node` in our example above contain a type variable in their type. For such an argument, the `type_ctor_layout` structure, being static, cannot possibly contain the actual `type_info` of the argument. Instead, it contains a `pseudo_type_info`, which is a generalization of a typeinfo. Whereas a `type_info` is always a pointer to a `type_info` structure, a `pseudo_type_info` is either a small integer that refers to a type variable, or a pointer to a `pseudo_type_info` structure, which is exactly like a `type_info` structure except that the fields after the `type_ctor_info` are `pseudo_type_info`s rather than `type_info`s.

The functor descriptor for the functor `node` will contain the small integers 1 and 2 as its second and third `pseudo_type_info`s, standing for the type variables K and V respectively, which are first and second type variables in the polymorphic type `tree(K, V)`. The first and fourth `pseudo_type_info`s will be pointers to `pseudo_type_info` structures in which the `type_ctor_info` slot points to the the `type_ctor_info` structure for `tree/2` and the following two `pseudo_type_info`s are the small integers 1 and 2. When a piece of code that has a `type_info` for the type `tree(string, list(int))` looks up the arguments of the `node` functor, it will construct `type_info`s for the arguments by substituting any `pseudo_type_info`s in the arguments (or in arguments of the arguments, and so on), with their corresponding parameters, i.e. the `type_info`s for `string` and for `list(int)`, which are at offsets 1 and 2 in the `type_info` structure for `tree(string, list(int))`. Note the exact correspondence between the offsets and the values of the `pseudo_type_info`s representing the type variables.

We can distinguish between small integers and pointers by imposing an arbitrary boundary between them. If the integer value of a word is smaller than a given limit, currently 1024, then the word contains a small integer; if it is greater than or equal to the limit, it is a pointer. This works because we can ensure that all small integers are below this limit, in this case by imposing an upper bound on the arities of type constructors, and because we can ensure that all pointers to data are bigger than the limit. (The text segment comes before the data segment, and the size of the text segment of the compulsory part of the Mercury runtime system is above the limit; in any case, most operating systems make the first page of the address space inaccessible in order to catch null pointer errors.)

If the value of the descriptor tag is shared_remote, then this value has a discriminated union type and the primary tag of the data word is shared between several functors, which are distinguished by a remote secondary tag. The rest of the descriptor word is then a pointer to a vector of words which contains

- the number of functors that share this tag (f), and
- f pointers to *functor descriptors*.

To find the information for the functor in the data word, we must use the secondary tag pointed to by the data word to index into the vector of functor descriptors.

If the value of the descriptor tag is shared_local, then there are three possibilities: (a) this value has a discriminated union type and the primary tag of the data word is shared between several functors, which must all be constants because which are distinguished by a local secondary tag; (b) this value has an enumerated type, such as type `example` from 2.2; or (c) this value has a builtin type such as `int` or `string`. For alternative (c), the rest of the descriptor word is a small integer that directly identifies the builtin type. For alternatives (a) and (b), the rest of the descriptor word is a pointer to an *enumeration vector*, which contains

- a boolean that says whether this is an enumeration type or not, and thus selects between (a) and (b),
- s, the number of constants that share this tag (for (a)) or the number of constants in the entire enumeration type (for (b)), and
- s pointers to strings containing the names of the constants.

To find the name of the functor in the data word, we must use the local secondary tag in the data word (for alternative (a)) or the entire data word (for alternative (b)) to index into the vector of names.

If the value of the descriptor tag is equivalence, then the value is either of a type that was declared as an equivalence type by the programmer, or it is of a *no_tag* type, a discriminated union type with one functor of one argument, which the compiler considers to be an equivalence type for purposes of internal although not external representation. Here is one example of each.

```
:- type equiv(T1, T2) == foo(int, T2, T1).
:- type notag(T1, T2) ---> wrapper(foo(int, T2, T1)).
```

In the latter case, the compiler uses the same internal representation for values of the types `notag(T1, T2)` and `foo(int, T2, T1)`, just as it does for the true equivalence.

The rest of the descriptor tag is a pointer to an *equivalence vector*, which contains

- a flag saying whether this type is a no_tag type or a user-defined equivalence type,
- a `pseudo_type_info` giving the equivalent type, and
- for no_tag types, a pointer to a string giving the name of the wrapper functor involved.

3.6 Creating Type-specialized Term Representations

A `type_ctor_layout` structure has complete information about how types with a given type constructor are represented. While the organization of this structure is excellent for operations that want to interpret the representation of an already existing term, the organization is not at all suitable for operations that want to build new terms, such as parsing a term from an input stream. The `type_ctor_functors` table is an alternate organization of the same information that is designed to optimize the operation of searching for the information about a given functor. Like the `type_ctor_info` that points to it, the `type_ctor_functors` structure is static, and there is only ever one `type_ctor_functors` structure for a given type constructor.

The first word of the `type_ctor_functors` structure is an indicator saying whether this type is a discriminated union, an enumeration type, a no_tag type, an equivalence, or a builtin. The contents of the rest of the structure vary depending on the indicator. For discriminated unions, the structure contains the number of functors in the type, and a vector of pointers to the *functor descriptor* for each functor. For enumerations, it contains a pointer to the *enumeration vector*. For no_tag types, it has a pointer to the *functor descriptor* for its single functor. For true equivalence types, it contains the `pseudo_type_info` for the equivalent type. For builtin types, it contains the small integer that identifies the builtin type.

3.7 Accessing RTTI from User Level Code

A natural application of RTTI is dynamic typing [1]. The Mercury standard library provides an abstract data type called `univ` which encapsulates a value of any type, together with its `type_info`. The library provides a predicate `type_to_univ` for converting a value of any type to type `univ`.

```
:- pred type_to_univ(T, univ).
:- mode type_to_univ(in, out) is det.
:- mode type_to_univ(out, in) is semidet.
```

Note that `type_to_univ` has two modes. The second (reverse) mode lets you try to convert a value of type `univ` to any type; this conversion will fail if the value stored in the `univ` does not have the right type. The reverse mode implementation compares the `type_info` for `T`, which the compiler passes as an extra argument, with the `type_info` stored in the `univ`.

In addition to this implicit use of RTTI, Mercury allows user programs to make explicit use of RTTI, by providing some RTTI types and operations on those types as part of the Mercury standard library. To ensure that user programs don't depend on the details of the implementation, the types representing runtime type information are opaque; the operations that we provide for manipulating them deal with language-level constructs such as types and functors rather than implementation-level details such as pointer tags.

Two such abstract data types represent `type_info`s and `type_ctor_info`s. The operations on them include:

```
:- func type_of(T) = type_info.
:- func type_name(type_info) = string.
:- pred type_ctor_and_args(type_info::in, type_ctor_info::out,
            list(type_info)::out) is det.
:- pred functor(T::in, string::out, int::out) is det.
:- func argument(T::in, int::in) = (univ::out) is semidet.
```

The `type_of` function returns a `type_info` describing its argument. Its implementation is trivial: the compiler will pass the `type_info` for the type `T` as an extra argument to this function, and `type_of` can just return this extra argument.

Once you have a `type_info`, you can find out the name of the type it represents; this is useful e.g. in giving good error messages in code manipulating values of polymorphic types. You can also special-case operations on some types, for purposes such as pretty-printing. You can also use `type_ctor_and_args` to decompose `type_info`s into their constituent parts, This is mostly useful in conjunction with operations that decompose terms, such as `functor` and `arg`. Still other operations are designed to allow programs to construct types (that is, `type_info`s at runtime) by combining existing type constructors in new ways, and to construct terms of possibly dynamically created types.

3.8 Representing Type Information about Sets of Live Variables

When a program calls `io:print` to pretty-print a term or `io:read` to read one in, the polymorphism transformation passes the required `type_info`(s) to the predicate involved. This is possible because the predicate deals with a fixed number of polymorphic arguments and because the number of type variables in the types of those arguments is also known statically.

However, in some cases we want one piece of code to be able to deal with arbitrary numbers of terms, which have an unknown number of type variables in their types. Two examples are the Mercury debugger and the Mercury native garbage collector. They both need to be able to interpret the representations of all live variables at particular points in the program, in the case of the debugger so that it can print out the values of those variables if the user so requests, and in the case of the garbage collector so that it can copy the values of those variables from *from-space* to *to-space*. To handle this, at each program point that may be of interest to the debugger or to the native collector, the compiler generates a data structure describing the set of live variables, and lets the debugger and the native collector interpret this description. Of course, if the compilation options do not request debugging information and if they request the conservative, rather than the native collector, there will be no such programs points, and the data structures we discuss in this subsection will not be generated.

The debugger and the native collector both need to know how to walk the stacks (for printing the values of variables in ancestors for the debugger and because all live values in all stack frames are part of the root set for the native collector). For the nondet stack this is not a problem, since nondet stack frames store the pointer to the previous stack frame and the saved return address in fixed slots. However, frames on the det stack have not fixed slots, and they are of variable size. To be able to perform a step in a stack walk starting from a det stack frame, one must know how big the frame is and where within it the saved return address is. The compiler therefore generates a *proc layout* structure for each procedure, which includes

– the address of the entry to this procedure
– the determinism of this procedure (this controls which stack it uses)
– the size of the stack frame
– the location of the return address in the stack frame

The stack frame size and saved return address location are redundant for procedures on the nondet stack, but it is simpler to include this information for all procedures.

The debugger and the native collector both have their own methods for getting hold of the proc layout structure for the active procedure, and can thus find out what address the active procedure will return to. However, without knowing what procedure this return address is in, they won't be able to take the next step in the stack walk. Therefore when debugging or the native collector is enabled, the compiler will generate a *label layout* table for every label that represents the return address of a call. Label layout tables contain:

– a pointer to the proc layout structure for this procedure,
– n, the number of live and "interesting" variables at the label,
– a pointer to two consecutive n-element vectors, one containing `pseudo_type_infos` for the types of the live variables, and one containing the descriptors of the locations of live variables,
– a pointer to a vector of type parameter locations, the first element of which gives the number of type parameters and hence the size of the rest of the vector; as an optimization, the pointer will be null if the count is zero, and
– a pointer to a vector of n offsets into a module-wide string table giving the variables' names (this field is present only when debugging).

The Mercury runtime has a table which can take the address of a label (such as a return address) and return a pointer to the label layout structure for that label. That table, the proc layout structures and the first fields of label layout structures together contain all the information required for walking the stacks.

The other fields in a label layout structure describe the "interesting" variables that are live at that label. Here the debugger and the native collector have related but slightly different requirements. The collector needs the type of all variables (including compiler introduced temporaries) but not their names, whereas the debugger needs names but does not need information about temporaries. If both are enabled, the label layout structure will contain the union of the information the two systems need.

The debugger and native collector are also interested in somewhat different sets of labels. While both are interested in return labels, the debugger is also interested in labels representing entry to and exit from the procedure, and labels at program points that record decisions about the path execution, e.g. the entry points into the then parts and else parts of if-then-elses; these are irrelevant for the native collector.

Each live, interesting variable at the label has an entry in two consecutive vectors pointed to by the label's layout structure. The entry in one of the vectors gives the location of the variable. Some bits in this entry say whether the variable is in an abstract machine register, in a slot on the det stack, or in a slot on the nondet stack, while the other bits give the number of the register or the offset of the slot. The entry in the other vector is the `pseudo_type_info` for the type of the variable. Before this `pseudo_type_info` can be used to interpret the value of the variable, it must be converted into a `type_info` by substituting, for every type variable in the `pseudo_type_info`, the `type_info` of the actual type bound to the type variable.

Consider a polymorphic predicate, one of whose argument is of type `list(T)`. Its caller will pass an extra argument giving the `type_info` of the actual type bound to `T`; this is the `type_info` that must be substituted into the `pseudo_type_info` of the `list(T)` argument. Since the signature of the procedure may include more than one type variable, each of which will have the actual type bound to it specified by an extra `type_info` argument, the compiler assigns consecutive integers, starting at 1, to all the type variables that occur in the types of any of the arguments (actually, to all the type variables that occur in the types of any of the variables of the procedure, which includes the arguments), and makes the `pseudo_type_info`s in the vector of pairs refer to each type variable by its assigned number. For every label that has a label layout structure, the compiler takes the set of live, interesting variables, and finds the set of type variables that occur in their types. The compiler then includes a description of the location of the `type_info` structure for the actual type bound to the type parameter in the type parameter location vector of the label layout structure, at the index given by the number assigned to the type variable.

That may sound complex, but to look up the type of a variable, one need only (a) convert the vector of type parameter locations in the label layout structure into an equal sized vector of `type_info`s, by decoding each location descriptor and looking up the value stored at the indicated location, which will be a `type_info`, and (b) using the resulting vector of `type_info`s to convert the `pseudo_type_info` for the variable into the `type_info` describing its type, exactly as we did in section 3.5 (except that the source of the `type_info` vector that identifies the types bound to the various type variables is now different).

One consequence of including information about the locations of variables that hold `type_info`s in label layout structures is that the compiler must ensure that a variable that holds the `type_info` describing the actual type bound to a given type variable must be live at a label that has a layout structure if any variable whose type includes that type variable is live at that label. Normally, the compiler considers every variable dead after its last use. However, when the options call for the generation of label layouts, the compiler, as a conservative approximation, considers a variable that holds the `type_info` to be live whenever any variable whose type includes that corresponding type variable is live. This rule of *typeinfo liveness* often extends the life of variables containing `type_infos`, and sometimes prevents such variables from being optimized away.

4 Evaluation

Hard numbers on RTTI systems are rare: there are few papers on RTTI, and many of these papers do not have performance evaluations of the RTTI system itself (although they often evaluate some feature enabled by RTTI). In this section we therefore provide some such numbers.

The Mercury implementation depends on RTTI in very basic ways. We cannot just turn off RTTI and measure the speed of the resulting system, because without RTTI, polymorphic predicates do not know how to perform unification and comparison. We would have to remove all polymorphism from the program first. This would require a significant amount of development effort, particularly since many language primitives implemented in C cannot be specialized automatically.

We therefore cannot report results on the exact time cost of the RTTI system. We can report two kinds of numbers though. First, a visual inspection of the C code generated by the Mercury compiler leads us to estimate that the fraction of the time that a Mercury program spends constructing `type_infos` and moving them around (all the other RTTI data structures are defined statically) is usually between 0 to 8%, probably averaging 1 to 3%. In earlier work [12], we measured this cost as being less than 2% for a selection of small benchmarks. One reason why these numbers are small is that the Mercury compiler includes an optimization that removes unused arguments; most of the arguments thus removed are `type_info` structures. Second, the researchers working on HAL, a constraint logic programming language, have run experiments showing the speed impact of type-specialized term representations. They took some Prolog benchmark programs, and translated them to HAL in two different ways: once with every variable being a member of its natural type, once with every variable being a member of a universal type that contained all the function symbols mentioned in the program. The HAL implementation, which compiles HAL programs into Mercury, compiles each HAL type into its own Mercury type, so this distinction is preserved in the resulting Mercury programs too. The experimental results [6] show that the versions using natural types, and therefore type-specific term representations, are on average about 1.4 times the speed of the versions using the universal type and thus a generic term representation. Since an implementation using a generic term representation can be made to work without RTTI but one using type-specific term representations cannot, one could read these results as indicating a roughly 40% speed advantage enabled by the use of RTTI. Due to the small number and size of the benchmarks involved in the experiments and the differences between native Mercury code and Mercury code produced by the HAL compiler, this number should be treated with caution. However, it seems clear that the time costs of RTTI are outweighed by the benefits it brings in enabling type-specific term representations.

The parts of RTTI that are essential for polymorphism (and which also suffice to support dynamic casts, i.e. the `univ` type) are the `type_info` structures and the parts of `type_ctor_info` structures containing the type constructor arity and the addresses of the unification and comparison procedures, i.e. the structures discussed up to but not including section 3.5. The structures discussed from that point forward, including the `type_ctor_layout` and `type_ctor_functors` structures, are needed only for other aspects of the system, including generic term I/O, debugging, native garbage collection, and user-level access to type-specialized representations. Since all these structures are defined statically, they do not impact the runtimes of programs except through cache effects.

To get an appreciation for space costs, we have measured the sizes of object files generated for the Mercury compiler and standard library, which are written in Mercury itself, compiled with various subsets of runtime type information. The compiler and library together consist of 225 modules and define about 950 types and about 7650 predicates; they total about 206,000 lines of code. Our measurement platform was an x86 PC running Linux 2.0.36.

Without any of the static data structures or automatically generated predicates described in this paper, the object files contain a total of 3666 Kb of code and 311 Kb of data. This version of the system cannot do anything that depends on RTTI. The automatically generated unification and comparison predicates add 462 Kb of code and 12 Kb of data to this; the static `type_ctor_info` structures add another 120 Kb of code and 47 Kb of data on top of that. (`type_ctor_info`s contain the addresses of unification and comparison procedures, which prevents the compiler from optimizing those procedures away even if they are otherwise unused; this is where the code size increase comes from.) This version of the system supports polymorphic unifications and comparisons, dynamic typing (with the `univ` type) and the `type_name/1` function.

To support other RTTI-dependent operations, e.g. generic I/O, the system needs the `type_ctor_layout` and `type_ctor_functors` structures as well. Adding the `type_ctor_layout` structures alone adds 81 Kb of data, while adding the `type_ctor_functors` structures alone adds 78 Kb of data. However, since these two kinds of structures share many of their components (e.g. functor descriptors), adding both adds only 99 Kb of data.

If we wish to do something using stack layouts, the compiler must follow the rule of typeinfo liveness. This rule by itself adds 14 Kb of code and 17 Kb of data. This increase comes from the requirement to save type variables on the stack and to load them into registers more often (this must cause a slight slowdown, but this slowdown is so small that we cannot measure it). This brings us up to 4247 Kb of code and 469 Kb of data, for a total size of 4747 Kb. We will use this as the baseline for the percentage figures below.

Adding the stack layouts themselves has a much more substantial cost. Switching from conservative gc to native gc increases code size by 822 Kb and data size by 2229 Kb; total system size increases by 64% to 7798 Kb. The increase in code size is due to the native collector's requirement that certain optimizations which usually reduce code size be turned off; the increase in data size is due to the label layout structures. Sticking with conservative gc but adding full debugging support, increases code size by 6438 Kb and data size by 5248 Kb; total system size increases by 246% to 16433 Kb. The code size increase is much bigger because debugging inserts into the code many calls to the debugger entry point[9] (it also turns off optimizations that could confuse the user). The data size increase is much bigger because debugging needs label layout structures at more program points (e.g. the entry point of the then part of an if-then-else), and because it needs the names of variables.

We already use several techniques for reducing the size of the static structures generated by the Mercury compiler, most of which are related to RTTI. The most important such technique we have not covered earlier in the paper is looking for identical static structures in each module and merging them into a single structure. Merging identical static structures in different modules would yield a further benefit, but since we want to retain separate compilation, it would require significant extensions to our compilation environment. Another potential optimization we could implement is merging two structures whenever one is a prefix of the other.

5 Related work

We expect that techniques and data structures at least somewhat similar to the ones we have described have been and/or are being used in the implementations of other mostly-statically typed languages (e.g. SML, Haskell and Algol 68; see the references cited in [8]). However, it is difficult to be sure, since papers that discuss RTTI implementations at any significant level of detail are few and far between. The exceptions we know of all deal with garbage collection of strongly typed languages, using the (obvious) model of walking the stack, finding out what the types of the live variables are in each frame and then recursively marking their values.

Goldberg [8] describes a system, apparently never implemented, that associates garbage collection information with each return point; this information takes the form of a compiler-generated function for tracing all the live variables of the stack frame being returned to. To handle polymorphism, it has these functions pass among themselves the addresses of other functions that trace single values (e.g. a function for tracing a list would take as an argument a function for tracing the list elements). This is a less general solution than our `pseudo_type_info`s. The garbage collection and tracing functions are single-purpose and are likely to be much bigger than our layout structures.

Tolmach [14] describes how, by using explicit lazily computed type parameters that describe the type environment (set of type bindings) of a function, one can simplify the reconstruction of types.

The TIL compiler for SML [13] uses a similar scheme but eagerly evaluates type parameters, making it quite similar to the combination of tables and type-info parameters used by the Mercury compiler, except for TIL's use of type tags on heap-allocated data. Unfortunately the paper lacks a detailed description of the data representations and runtime behaviour of the type information generated by the TIL compiler, and is unclear about whether this information can be used for purposes other than garbage collection.

Aditya et al [2, 3] describe a garbage collector and debugger for Id that has an approach to RTTI that is similar to ours, the main difference being that in their system, callers of polymorphic functions do not pass type information to the callee; instead, the garbage collector or debugger searches ancestor stack frames for type information when necessary. Although this scheme avoids the cost of passing type information around, we have not found this cost to be significant. On the other hand, the numbers in [3] show that propagating type information in the stack is quite expensive for polymorphic code. This is probably not the right tradeoff for Mercury, since we want to encourage programmers to write polymorphic code.

In the logic programming field, Kwon et al [11] and Beierle et al [4] both describe schemes for implementing polymorphically typed logic programming languages with dynamic type tests. Their schemes both extend the WAM; both annotate the representations of unbound variable with type information and add additional WAM instructions for handling typed unification. But we believe that an approach which is based on a high-level program transformation, like our handling of `type_infos`, is simpler than one which requires significant modifications to the underlying virtual machine. Neither scheme makes use of type-specific term representations.

None of these papers cited above give measurements of the storage costs of their schemes. Future comparisons of space usage and type reconstruction performance between Mercury and the systems described in those papers may yield interesting results.

Elsman [7] uses a transformation very similar to the one we use to introduce `type_infos`, for a very similar purpose: to enable type-specific data representations; the performance benefits he reports broadly match our experience. However, his system, whose purpose is the efficient handling of polymorphic equality types in ML, only passes around the addresses of equality functions, not comparison functions, type names, or layout information. As such, his system constitutes a very limited form of RTTI that is useful only for polymorphic equality, not for dynamic types, garbage collection, debugging, etc.

6 Conclusion

Our results show that a run time type information system can be added to Mercury without compromising the speed of the basic execution mechanism, and with relatively small space overheads in most cases. The RTTI system allows many useful extensions both to the language and to the implementation.

In future work, we would like to explore the tradeoffs between table-driven generic operations and specialized code. Trends in microprocessor design, in particular the increasing relative costs of mispredicted branches and cache misses, mean that it is quite possible that a generic unification routine using `type_ctor_info` structures may now be faster than the automatically generated procedures we now use. However, executing code is inherently more flexible than interpreting fixed-format tables. At the moment, we take advantage of this in our implementation of types with user-defined equality theories, for which we simply override the pointer to the automatically generated unification procedure with a pointer to the one provided by the user. Such a facility would still need to be provided even in a system that used table-driven unification.

To make table-driven generic operations more competitive, we are in the process of simplifying our data structures. At the moment, the information about what kind of type the type constructor represents (a discriminated union type, an equivalence type, a no_tag type, an enumeration type or a builtin type) is scattered in several different parts of the `type_ctor_layout` structure and its components (e.g. functor descriptors), even though this information is available directly in the `type_ctor_functors` structure. The reason for this is that initially, the Mercury RTTI system only had `type_ctor_layouts`; `type_ctor_functors` were added later. The design we are moving towards puts the type kind directly into the `type_ctor_info`, and specializes the rest of the `type_ctor_info` according to the type kind (e.g. `type_ctor_layout` and `type_ctor_functors` structures will be present only if the type is a discriminated union type).

We would like to thank the Australian Research Council for their support.

References

[1] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991.

[2] Shail Aditya and Alejandro Caro. Compiler-directed type reconstruction for polymorphic languages. In *Proceedings of the 1993 ACM Conference on Functional Programming Languages and Computer Architecture*, Copenhagen, Denmark, June 1993.

[3] Shail Aditya, Christine Flood, and James Hicks. Garbage collection for strongly-typed languages using run-time type reconstruction. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, pages 12–23, June 1994.

[4] C. Beierle, G. Meyer, and H. Semle. Extending the warren abstract machine to polymorphic order-sorted resolution. In *Proceedings of the International Logic Programming Symposium*, pages 272–286, 1991.

[5] Hans Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software – Practice and Experience*, 18:807–820, 1988.

[6] Bart Demoen, Maria Garcia de la Banda, Warwick Harvey, Kim Marriott, and Peter Stuckey. Herbrand constraint solving in HAL. Technical Report 99/18, Department of Software Engineering and Computer Science, University of Melbourne, Melbourne, Australia, 1998.

[7] Martin Elsman. Polymorphic equality — no tags required. In *Second International Workshop on Types in Compilation*, volume 1473 of *Lecture Notes in Computer Science*. Springer Verlag, 1998.

[8] Benjamin Goldberg. Tag-free garbage collection for strongly typed programming languages. In *Proceedings of the SIGPLAN '91 Conference on Programming Languages Design and Implementation*, pages 165–176, Toronto, Ontario, June 1991.

[9] D.R. Hanson and M. Raghavachari. A machine-independent debugger. *Software – Practice and Experience*, 26:1277–1299, 1996.

[10] Fergus Henderson, Thomas Conway, Zoltan Somogyi, and David Jeffery. The Mercury language reference manual. Technical Report 96/10, Department of Computer Science, University of Melbourne, Melbourne, Australia, 1996.

[11] K. Kwon, G. Nadathur, and D.S. Wilson. Implementing polymorphic typing in a logic programming language. In *Computer Languages*, volume 20(1), pages 25–42, 1994.

[12] Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 26(1-3):17–64, October-December 1996.

[13] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL : A type-directed optimizing compiler for ML. In *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implemantation*, pages 181–192, New York, May 21–24 1996.

[14] Andrew Tolmach. Tag-free garbage collection using explicit type parameters. In *Proceedings of the SIGPLAN '94 Conference on Programming Languages Design and Implementation*, pages 1–11, Orlando, Florida, June 1994.