Run Time Type Information in Mercury

Tyson Dowd

March 1998

Context

In the Mercury implementation, each type uses a specialized representation. You cannot tell the value represented by a bit pattern without knowing what type it is.

When generating code, the compiler knows the types of everything, so it generates the appropriate code to perform operations on the data.

RTTI allows a running program to access information about types, so as to be able to perform operations on data without generating specialized code for each type.

Motivation

We need information about type representation to interpret/translate/copy data.

- `term_to_type` and `type_to_term`
- read/write values of arbitrary types
- pretty printing, term browsers, debugger value output
- `deep_copy` (for `solutions/2` or garbage collection)
- comparisons and unifications (although not yet)

More Motivation

Expressiveness:

Run time type **identification** allows dynamic type checking and matching.

- `univ` type (heterogenous collections)
- conversions to/from `univ`
- type conditionals – allow more powerful foreign language interfaces

Other language features can add similar expressiveness (existential types, typeclasses) but often require extensive redesigns to use effectively.

Even More Motivation

Type names allows type disambiguation.

- generic code can give specific errors, e.g.

  ```
  map__lookup failed
  Key Type:  int
  Value Type:  variable_name
  ```

- ambiguous representations can be qualified, e.g.

  ```
  array(int):[1,44,11,291,22,51]
  ```

Requirements

For copying/translating or interpreting data, we need:

- structure of data
- functor names (e.g. `node/2` and `empty/0`)
- functor argument types

For dynamic type checking and matching:

- unique identifier for each type

For type disambiguation:

- type name: constructor and arguments

Design

Design goals:

- Make type information available
- User level and system level interfaces

Design constraints:

- Handle all language features in Mercury
- Low runtime-overhead when not used
- Low runtime-overhead when used
- Don't use too much space

Design - Polymorphism

Polymorphism transformation – adds extra arguments to polymorphic predicates:

```
:- pred sort(list(T)).
```

becomes

```
:- pred sort(typeinfo(T), list(T)).
```

```
sort([3,1,2])
```

becomes

```
TypeInfo_for_T = ....  /* int */
sort(TypeInfo_for_T, [3,1,2])
```

Design - TypeInfo

What is a `type_info`?

A pointer to a vector containing:

- a `base_type_info` pointer, which contains a full description of the top-level type constructor for this type
- zero or more pointers to `type_info`s describing the actual type parameters of this type constructor

For the type `tree(string, list(int))`, the `type_info` will contain pointers to the `base_type_info` for `tree/2` and to the `type_info`s for `string` and `list(int)`.

Design - Base Type Info

We generate one base_type_info structure for each type constructor (that is, for each `:- type` declaration).

The base_type_info contains:

- arity of the type constructor
- code pointers to unify/compare procedures
- `base_type_layout` (added for RTTI)
- `base_type_functors` (added for RTTI)
- full name of type constructor

Design - Base Type Layout

- description of the representation of the data
- Mercury data uses tagged pointers.
- use the tag as an index

  ```
  description = base_type_layout[tag(data)];
  interpret_data(data, description);
  ```
- description says what sort of type it is (discriminated union, equivalence type, builtin, no_tag type, type variable).
- might need to look up secondary tag
- d.u. description gives name of functor and argument types.
- useful for decoding data.

Design - Base Type Functors

- also a description of the representation of the data
- almost the converse of base_type_layout (some overlap)
- complete list of functors, and for each functor the types of its arguments and how to build a term with this functor
- useful for building data if you only know its type (e.g. parsing input terms)

Design - PseudoTypeInfos

How do we describe `list(T)`?

`list(int)`, `list(float)`, `list(tree(int))`... must all be described by a single set of `base_type_*` data structures.

- use a type variable to represent `T`
- type variables occur in `type_info`s such as the ones for the arguments of functors
- use low integer values which will not be used as pointer values to represent each type variable
- a type_info that might have type variables is known as a `pseudo_type_info`
- at runtime, substitute type parameters

Design - Runtime Behaviour

What happens when we want to use type information at runtime?

- polymorphism transformation makes `type_info`s available
- the library provides a set of predicates that use type information (`type_of`, `type_name`, `arg`, `functor`, `io__read`, `io__write`)
- when the implementation does recursive traversals of types, each recursion step substitutes type parameters, turning `pseudo_type_info`s into `type_info`s
- `pseudo_type_info`s are hidden from programmers

Design - Tables

- all type information (`base_type_*`) is stored in static data tables that are generated at compile time
- only one table is generated for each `:- type` declaration
- tables are quite compact
- performance is quite reasonable for applications such as `io__read` and `io__write`

Future Work

When interpreting type descriptions, need to allocate memory to substitute type parameters.

- analyse memory/time complexity and tradeoffs when substituting type parameters

No easy way to handle types implemented in foreign languages.

- investigate restricting RTTI to certain typeclasses ("decomposable" typeclass) – foreign types need to provide methods

Procedures for compare and unify duplicate effort in RTTI system.

- implement table-driven compare and unify, compare with current implementation.

Conclusions

- some language features make RTTI more difficult, e.g. user-defined equality predicates, and types defined in C
- RTTI can be implemented with few overheads other than some static data
- RTTI brings many practical benefits
  - io__read and io__write
  - arg, functor, construct
  - automatic memoing
  - trace based debugging
  - accurate garbage collection