# Mercury

The lecturer is Zoltan Somogyi.
(zs, L2.24)

The home page of the Mercury project on the Web contains documentation on the language as well as several papers on the system.

http://www.cs.mu.oz.au/mercury
http://www.cs.mu.oz.au/mercury/tutorial

The Mercury group currently includes one research fellow and seven postgraduates. Two people got their first publication with us while undergraduates.

# Logic programming

The point of basing a programming language on logic is that it makes it easier for programmers to reason about their programs. (Mathematicians have been reasoning about logical formulae for a century.)

This reasoning can also be automated. This opens the door to

- automatic program transformation
- automatic parallelization
- semi-automatic debugging

Such tools promise to improve programming productivity.

# Prolog

Prolog was the first logic programming language, and its first implementation was done in 1970.

At that time, computers were slow and memories small, and hence optimizing compilers were rare.

Prolog adopted the approach that a programmer who wanted efficiency would have to tell the Prolog system how to achieve it. This was done through non-logical constructs.

Unfortunately, these constructs destroy the declarative reading of the program.

# Mercury

Mercury is a new logic programming language intended to

- realize the promised benefits of logic programming, and
- support the creation and maintenance of large, reliable, efficient application programs by teams of programmers.

Our experience while writing the Mercury compiler in Mercury itself shows that the language achieves these objectives.

Mercury is the fastest logic programming system by a wide margin (1.9x Aquarius, ~5x SICStus, ~10x Quintus).

# Flexibility vs reliability

Prolog has virtually no static checking, so the system will execute any code you write. This is good if the code you wrote faithfully implements your intention, and bad if it isn't.

Mercury has lots of static checking, so the system will not compile the code you write if it cannot prove that errors of certain kinds will not arise. This is good if the code you wrote can in fact exhibit an error; it is bad if the error cannot arise, but the compiler cannot see this.

# Which side of the line?

A ideal language would allow programmers to express everything they want to express and nothing they don’t want to express.

The ideal cannot be reached, because it is inconsistent; what a programmer wants to be able to express depends on the circumstances. However, it can be approached.

Some language designers prefer to err on the side of too much flexibility. Others prefer to err on the side of too much safety.

# Programming style

Flexibility-oriented languages, e.g. C, C++, PL/I and Common Lisp, allow programs to be written quickly, but require lots of testing and debugging, which usually still miss some errors. Modifications to existing programs require great care.

Safety-oriented languages, e.g. Ada, Eiffel, Java, Miranda, Haskell and Mercury, require more up front planning and produce more error messages for the initial program, but produce a *working* program more quickly. Modifications to existing programs are easier, due to the safety net of the static checks.

# The voice of experience

When working with a modern safety-oriented, declarative language, it is not uncommon for a module to work the first time that the compiler accepts it.

- red-black trees
- Mercury compiler
- US Naval Surface Warfare Center
- comp.lang.prolog

A quote from the net: “Of course SML does have its weaknesses, but by comparison, a discussion of C++’s strengths and flaws always sounds like an argument about whether one should face north or east when one is sacrificing one’s goat to the rain god.”

# Outline of Mercury

- predicates and functions
- strong type system
  - parametric polymorphism
  - higher-order
  - type classes
  - existential types (in progress)
- strong mode system
  - uniqueness
- strong determinism system
- module system
- purely declarative (even I/O)

# Declarations

Type, mode, and determinism declarations allow programmers to write down several aspects of the intended behavior of their predicates.

The Mercury compiler uses these declarations to find errors and to generate better code. The declarations themselves serve as a reliable form of documentation.

Declarations are required for predicates and functions exported from a module. For predicates and functions local to a module, they are optional; the compiler will attempt to infer the information in the declarations.

# The type system

The Mercury type system is based on the work of Hindley and Milner, and is very similar to the type systems of Miranda, Haskell, Godel, and other languages.

In Mercury, all variables have a type. The definition of the type lists the function symbols that these variables may take as their value, and for each function symbol, list the types of the arguments.

The simplest types represent enumerations and what in C would be called structures. More complex types represent discriminated unions of structures.

# Examples of types

```
:- type suit --->
            club
        ;   diamond
        ;   heart
        ;   spade.

:- type card --->
            card(suit, rank).

:- type expr --->
            const(int)
        ;   plus(expr, expr)
        ;   minus(expr, expr).
```

The same word can be used both as a type name and as a function symbol (e.g. card).

# Parametric polymorphism

```
:- type list(T) --->
          []
     ;    [T | list(T)].

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

T, K and V are *type variables*, i.e. variables that stand for a type.

Polymorphic types can be viewed as *type constructors*, in that they construct a new type from some existing types.

Parametric polymorphism yields homogeneous data structures.

# Type synonyms

Programmers can introduce new names for old types.

```
:- type enrolmentno == int.
:- type subj == string.
:- type mark == int.

:- type subj_result --->
        subj_result(subj, mark).

:- type student_table ==
        tree(enrolmentno, list(subj_result)).
```

The types int, float and string are built into Mercury.

# Predicate argument types

```
:- pred append(list(T), list(T), list(T)).

append([], L, L).
append([H | T], L, [H | NT]) :-
    append(T, L, NT).
```

Mercury asks the programmer to declare the types of the arguments of all predicates.

From this and from the definitions of types, the compiler can infer the types of all the variables in the program, *if* the program is well-typed.

# Mode declarations

For each argument of a predicate, a mode declaration says whether the argument is input to the call or output by the call.

A predicate may have more than one mode. We call each mode of a predicate a *procedure*.

```
:- pred append(list(T), list(T), list(T)).
:- mode append(in, in, out).
:- mode append(out, out, in).
```

Input arguments are *consumed* by the procedure, output arguments are *produced* by the procedure.

# Mode analysis

The mode analysis phase of the compiler identifies the modes of all calls in the program.

Mode analysis reorders conjunctions to ensure that the producer of a variable comes before all its consumers. This allows reversible predicates.

```
:- pred append3(list(T), list(T), list(T),
    list(T)).
:- mode append3(in, in, in, out).
:- mode append3(out, out, out, in).

append3(A, B, C, D) :-
    append(A, B, AB),
    append(AB, C, D).
```

# Mode analysis

```
% append(in, in, out)
append(A, B, C) :-
    A => [], C <= B.
append(A, B, C) :-
    A => [H | T],
    append(T, B, NT),
    C <= [H | NT].

% append(out, out, in)
append(A, B, C) :-
    A <= [], B <= C.
append(A, B, C) :-
    C => [H | NT],
    append(T, B, NT),
    A <= [H | T].
```

# Implied modes

Some calls specify more arguments than necessary, e.g. calls to append with three ground lists.

Mode analysis picks a mode compatible with the call, and introduces extra unifications after the call. This turns e.g. append([1,2], [3], [4,5]) into append([1,2], [3], C), C = [4,5] or into append(A, B, [4,5]), A = [1,2], B = [3].

This is necessary because the compiler generates separate code for each declared mode of a predicate, and the code assumes that output arguments are free variables when the call is made.

# Determinism declarations

The determinism of a procedure depends on how many solutions it can produce:

| | |
|---|---|
| det: | exactly one |
| semidet: | at most one |
| multi: | at least one |
| nondet: | any number |

Determinism declarations are attached to mode declarations:

```
:- pred append(list(T), list(T), list(T)).
:- mode append(in, in, out) is det.
:- mode append(out, out, in) is multi.
```

# Switch detection

Why is append(in, in, out) guaranteed to have exactly one solution?

- the recursive call is deterministic;
- the assignments are deterministic;
- exactly one of the test unifications will succeed.

Switch detection looks for disjunctions in which each disjunct unifies an input variable with a different functor; such disjunctions are *switches*.

The switch is complete if the disjuncts cover all the alternatives in the type of the switched-on variable.

## Common subexpression elimination

CSE looks for disjunctions in which each disjunct unifies an input variable with the *same* functor.

```
( L = [H1 | T1], T1 = [], ...
; L = [H2 | T2], T2 = [H3 | T3], ...
)
```

It pulls such unifications out of the disjunction; switch detection will then recognize the switch on T.

```
L = [H | T],
( H = H1, T = T1, T1 = [], ...
; H = H2, T = T2, T2 = [H3 | T3], ...
)
```

# Input/output

Mercury views I/O predicates as describing a relationship between two states of the world: the one before the action and the one after.

```
hello(S0, S) :-
    io__write_string("Hello, ", S0, S1),
    io__write_string("world\n", S1, S).
```

Of course, we want the implementation to update the state of the world destructively.

We must therefore make sure that old states of the world are never referred to again.

# Uniqueness information

The mode di (destructive input) says that the storage of this argument may be destroyed (reused) in the predicate, so the argument must be the last reference to that storage. The mode uo (unique output) says that there are no other references to the value returned for this argument.

```
:- pred hello(io__state, io__state).
:- mode hello(di, uo) is det.

hello(S0, S) :-
    io__write_string("Hello, ", S0, S1),
    io__write_string("world\n", S1, S).
```

# Uniqueness on backtracking

The compiler must ensure that variables destroyed during forward execution are not referred to again, even after backtracking.

```
(
    io__write_string("testing...", S0, S1),
    1 = 2
->
    io__write_string("1 = 2", S1, S)
;
    io__write_string("1 \= 2", S0, S)
)
```

This is not allowed, because the destructive update of state S0 cannot be undone.

# Definite Clause Grammars

A clause is in DCG notation if it uses --> instead of :-. When processing a DCG clause, the compiler adds a pair of arguments both to the head of the clause and to every call in the body of the clause not protected by braces. The extra arguments thread a state through the predicate.

```
write_lines([]) --> [].
write_lines([Line | Lines]) -->
    io__write_string(Line),
    write_lines(Lines).
```

```
write_lines([], S, S).
write_lines([Line | Lines], S0, S) :-
    io__write_string(Line, S0, S1),
    write_lines(Lines, S1, S).
```

# Modules

Mercury programs are composed of modules. Each module has an interface section, containing the declarations of items exported from the module, and an implementation section giving their definitions.

A module can make a type an abstract type by exporting its name but not its definition.

If module A wants to use an item exported by module B, it must import module B. If that item (e.g. a type) is needed in the interface of A, the import of B must be in A’s interface section; otherwise, it should be in A’s implementation section.

# The standard example

```
:- module hello.

:- interface.
% module io is in the standard library
:- import_module io.

% main/2 is the standard entry point
:- pred main(io__state, io__state).
:- mode main(di, uo) is det.

:- implementation.

main(S0, S) :-
    io__write_string("Hello, ", S0, S1),
    io__write_string("world\n", S1, S).
```

# mmc

The Mercury compiler is called “mmc”.

Single-module programs can be compiled simply by “mmc module.m”. This will convert the Mercury source to C, compile that to object code, and link it with the Mercury standard library, the Mercury runtime system etc. The executable will be put in the file “module”.

For multi-module programs, one wants to recompile only the minimal set of modules after a change. We have a front-end for make called “mmake” that handles Mercury programs.

# mmake

To (re)build the executable, run “mmake mainmodule”, but at the start of the project and whenever you add a new module, you must run “mmake mainmodule.depend” first.

Mmake will recompile a module not only if that module has been modified but also if it imports another module whose interface has changed.

These interfaces are stored in files with .int, .int2 and .int3 suffixes. Mmake puts error messages into files with a .err suffix.

# A simple formatter

```
main -->
    get_words(Words),
    { group_words(Words, 80, Lines) },
    put_lines(Lines).

get_words(Words) -->
    io__read_word(Result),
    (
        { Result = error(Error) },
        { io__error_message(Error, Msg) },
        { error(Msg) }
    ;
        { Result = eof },
        { Words = [] }
    ;
        { Result = ok(WordChars) },
        { string__from_char_list(WordChars, Word) },
        get_words(Words1),
        { Words = [Word | Words1] }
    ).
```

# Programming products

When writing a deterministic predicate, one must prepare for and handle all contingencies.

When something unexpected happens, the easy way out is to call the builtin predicate *error*, which prints an error message and aborts execution.

However, in many cases the program can do something more constructive. Mercury forces the programmer to at least think about recovering from the error.

Attention to such issues is what distinguishes programming products from simple programs.

```
:- pred group_words(list(string)::in, int::in,
    list(list(string))::out) is det.

group_words(Words, Max, Lines) :-
    (
        Words = []
        Lines = []
    ;
        Words = [FirstWord | LaterWords],
        line_of_words(FirstWord, LaterWords, Max,
            Line, RestWords),
        group_words(RestWords, Max, RestLines),
        Lines = [Line | RestLines]
    ).

:- pred line_of_words(string::in, list(string)::in,
    int::in, list(string)::out, list(string)::out) is det.

line_of_words(First, Later, MaxLen, Line, Rest) :-
    string__length(First, FirstLen),
    later_words(Later, FirstLen, MaxLen, [First],
        Line, Rest).
```

```
:- pred later_words(list(string)::in, int::in, int::in,
    list(string)::in, list(string)::out, list(string)::out) is det.

later_words([], _, _, Line, Line, []).
later_words([Word | Words], OldLen, MaxLen, Line0,
        Line, RestWords) :-
    string__length(Word, WordLen),
    NewLen = OldLen + 1 + WordLen,
    ( NewLen =< MaxLen ->
        append(Line0, [Word], Line1),
        later_words(Words, NewLen, MaxLen,
            Line1, Line, RestWords)
    ;
        Line = Line0,
        RestWords = [Word | Words]
    ).
```

This formatter is longer than some others, but it is very readable and it is obvious that it correctly implements the usual specification; no spaces at starts or ends of lines, no line too long unless a word is too long.

```
:- pred put_lines(list(list(string))::in,
     io__state::di, io__state::uo) is det.

put_lines([]) --> [].
put_lines([Line | Lines]) -->
     put_line(Line),
     put_lines(Lines).

:- pred put_lines(list(string)::in,
     io__state::di, io__state::uo) is det.

put_line([]) --> [].
put_line([Word | Words]) -->
     io__write_string(Word),
     put_line_rest(Words),
     io__write_char('\n').

:- pred put_line_rest(list(string)::in,
     io__state::di, io__state::uo) is det.

put_line_rest([]) --> [].
put_line_rest([Word | Words]) -->
     io__write_char(' '),
     io__write_string(Word),
     put_line_rest(Words).
```

# Optimization philosophy

Our philosophy is that if there is an algorithm for an optimization, then it should be done by the compiler, not the programmer.

The logic programming literature has many source-to-source optimizations. They require pure programs, and most depend on dataflow information.

These apply to Mercury directly. Applying them to other logic languages require extensive program analysis. For acceptable running time, these analyses must make approximations that degrade the effectiveness of the optimizations (e.g. don’t know whether this variable is bound here or not).

# Some optimizations

Deforestation takes programs with two or more traversals of a data structure and tries to produce an equivalent program with one traversal. Applied to the formatter, it could reduce the storage requirement from linear to constant.

Put_lines, put_line_rest and later_words are naturally tail recursive, but get_words is not. The compiler could transform get_words to be tail recursive by introducing an input argument (called an accumulator) containing the words seen so far in reverse order. The base case would reverse this list and return it as the output of get_words.

```
main -->
    io__command_line_arguments(Args),
    (
        { Args = [] },
        io__stdin_stream(StdinStream),
        { Result = ok(StdinStream) }
    ;
        { Args = [InFileName] },
        io__open_input(InFileName, OpenResult),
        { convert_result(OpenResult, Result) }
    ;
        { Args = [_, _| _] },
        { Result = error("Usage: sort [infile]\n") }
    ),
    (
        { Result = ok(InStream) },
        read_lines(InStream, Lines),
        { sort_lines(Lines, SortedLines) },
        write_lines(SortedLines)
    ;
        { Result = error(Message) },
        io__write_string(Message)
    ).
```

```
:- pred read_lines(io__input_stream::in, list(string)::out,
    io__state::di, io__state::uo) is det.

read_lines(Stream, Lines) -->
    io__read_line_as_string(Stream, Result),
    (
        { Result = error(Error) },
        { io__error_message(Error, Msg) },
        { error(Msg) }
    ;
        { Result = eof },
        { Lines = [] }
    ;
        { Result = ok(Line) },
        read_lines(Stream, MoreLines),
        { Lines = [Line | MoreLines] }
    ).
```

```
:- pred sort_lines(list(T)::in, list(T)::out) is det.

sort_lines([], []).
sort_lines([Line | Lines], SortedAllLines) :-
    sort_lines(Lines, SortedLines),
    insert(SortedLines, Line, SortedAllLines).

:- pred insert(list(T)::in, T::in, list(T)::out) is det.

insert([], I, [I]).
insert([H | T], I, L) :-
    compare(R, I, H),
    ( R = (<) ->
        L = [I, H | T]
    ;
        insert(T, I, NT),
        L = [H | NT]
    ).
```

Programmers need not worry about repeated terms such as [H | T].

Mercury automatically defines two operations on every type, unification and comparison.

# Higher order code

```
:- pred sort_lines(pred(comparison_result, T, T),
	list(T), list(T)).
:- mode sort_lines(pred(out, in, in) is det,
	in, out) is det.

sort_lines(_, [], []).
sort_lines(Comp, [Line | Lines], SortedAllLines) :-
	sort_lines(Comp, Lines, SortedLines),
	insert(Comp, SortedLines, Line, SortedAllLines).

:- pred insert(pred(comparison_result, T, T),
	list(T), T, list(T)).
:- mode insert(pred(out, in, in) is det,
	in, in, out) is det.

insert(_, [], I, [I]).
insert(Comp, [H | T], I, L) :-
	call(Comp, R, I, H),
	( R = (<) ->
		L = [I, H | T]
	;
		insert(Comp, T, I, NT),
		L = [H | NT]
	).
```

# Higher order constants

A value of a higher order type denotes a predicate or a function. You can write down such a value by giving its argument modes and its determinism as the first argument of lambda and its definition as the second argument. The argument types will be inferred.

```
...
Comp = lambda([R::out, S1::in, S2::in] is det,
    compare(R, S1, S2)
),
sort_lines(Comp, Lines, SortedLines),
...
```

# Higher order constants

A predicate/function name can be used directly as a higher order constant *if* it has one mode.

```
sort_lines(compare, Lines, SortedLines),
```

The code of a lambda may refer not only to lambda's arguments, but also to variables bound by the surrounding clause.

```
:- func add_n(int) = pred(int, int).
:- mode add_n(in) = (free -> pred(in, out) is det) is det.

add_n(N) = lambda([X::in, Sum::out] is det,
        Sum = X + N
    ).
```

# Two views of types

- What is the structure of this data?
- What can you do with this data?

Most conventional languages view types as answering just the first question. Some unconventional ones view it as answering just the second.

Ideally, the type system should be able to answer just the questions you want it to answer in each case.

With the example of insertion into a sorted list, you want the structure of lists exposed, but the details of comparison hidden, to allow them to be changed later.

# Imperative objects

In imperative object oriented languages, an object consists of

- an identity that distinguishes it from other objects,
- some attributes that define the state of the object, and
- some methods that use and maybe update the state of the object.

The state of the object should not be accessible or modifiable except by invoking a method of the object.

Although some languages allow the attributes to be exposed, either indiscriminately or just to “friends”, doing so is often a bad idea.

# Encapsulation

*Encapsulation*, i.e. hiding the implementation of an object from its users, has two main benefits.

- It allows the object's implementation to be changed, without requiring any changes in code that uses objects of that class.

- It allows multiple implementations of the same interface to coexist, with the same code being able to use any object of a given class, without knowing how the object implements the methods (in C++ terms, regardless of which subclass of the class the object is from).

# Non-parametric polymorphism

With parametric polymorphism, a predicate works on items of any type. This is possible because the predicate (e.g. append) just moves those items around, and does not invoke any operations on them.

With non-parametric polymorphism, a predicate works on items of any type, *provided* that a given set of operations is defined on that type. The signature of such a set of operations is usually called an *interface*.

Java supports the notion of interfaces directly. C++ doesn't; one has to use an abstract base class (one with at least one pure virtual function) instead.

# Declarative objects

Declarative languages view objects differently to imperative languages. Since they have no mutable state, they cannot and do not support object identity directly. This simplifies the implementation a lot, but in some cases it requires identity to be programmed explicitly.

It also means that methods cannot update an object; they can only return a new, different object. Whereas in imperative languages, an object's state and methods are stored together, with declarative languages, they are separate.

# Type classes

A *type class* defines an interface; a set of operations with given signatures.

```
:- typeclass htmlable(T) where [
    func to_html(T) = string,
    mode to_html(in) = out is det
].
```

A type that implements all the operations required by a type class may be declared to be a member or *instance* of the type class.

We want predicates to be able to invoke some of the operations of the type class on some of their arguments without knowing the exact type of that argument.

# Invoking a type class method

A predicate that wants to invoke a type class method on one of its arguments should have a *type class constraint* that requires that argument to be a member of the relevant type class:

```
:- pred make_page(string, T, string) <= htmlable(T).
:- mode make_page(in, in, out) is det.

make_page(Title, Val, HTML) :-
    string__append_list([
        "<HTML>",
        "<HEAD><TITLE>",
        Title,
        "</TITLE></HEAD>",
        "<BODY>",
        to_html(Val),
        "</BODY>",
        "</HTML>"],
        HTML).
```

# Instance declarations

An instance declaration says that a given type is a member of a given type class. For each operation associated with the type class, it gives the predicate that implements that operation on values of that type. The signatures of these predicates must match the signatures of the operations in the type class definition.

```
:- instance htmlable(string) where [
    func(to_html/1) is identity
].

:- func gif_to_html(gif::in) = string::out is det.

:- instance htmlable(gif) where [
    func(to_html/1) is gif_to_html
].
```

# Implied instances

Type class constraints on instance declarations give implied instances:

```
:- instance htmlable(list(T)) <= htmlable(T) where [
	func(to_html/1) is to_html_list
].

:- func to_html_list(list(T)) = string <= htmlable(T).
:- mode to_html_list(in) = out is det.

to_html_list(List) = HTML :-
	string__append_list([
		"<ul>", to_html_elems(List), "</ul>"],
		HTML).

:- func to_html_elems(list(T)) = string <= htmlable(T).
:- mode to_html_elems(in) = out is det.

to_html_elems([]) = "".
to_html_elems([Head | Tail]) = HTML :-
	string__append_list([
		"<li>", to_html(Head), to_html_elems(Tail)],
		HTML).
```

# Using type classes

Suppose the signature of a function requires a given argument to be a member of a given type class.

The caller can pass any value belonging to any type that has been declared a member of that type class, either directly, or by implication.

```
make_page("Title", "Contents", HTML), ...
make_page("Title", ["Contents1", "Contents2"], HTML), ...
make_page("Title", Gif, HTML), ...
make_page("Title", [Gif1, Gif2, Gif3], HTML), ...
```

# Controlling access to representations

You can keep the definition of a type hidden by declaring only its name in the interface section:

```
:- type xyz.
:- instance htmlable(xyz).
```

You can also export the fact that the type is a member of some type classes:

```
:- type xyz.
:- instance htmlable(xyz).
```

In both cases, the implementation section would have the full definition.

# Dependency inversion principle

Interfaces are important; they are one of the keys to reusability, which is the main hope of reducing software costs.

If module A depends on modules B and C, it cannot be reused separately from B and C without significant work.

If module A depends on interfaces i1 and i2, and module B implements i1 while C implements i2, then module A can be reused in any context that has other implementations of i1 and i2. Specifying the interface as a typeclass separates accidental coupling from deliberate.

# Inheritance

System designs quite frequently include related interfaces. For example, the *person* interface may have methods for getting the birth date and getting and setting the home address. The *student* and *staff* interfaces also have these methods.

Programmers can say that these methods have the same semantics in all cases by making *person* a class, and making *student* and *staff* inherit these methods from *person*, thus becoming subclasses of *person*.

# Implementation inheritance

In imperative languages, making B a subclass of A usually means that the state of an object of class B includes the state of an object of class A, and that some or all of class A's methods have the same implementation in objects of both class A and class B.

Changing the state or methods of class A thus impacts its subclasses as well; code that handles objects of class B may stop working after such a change.

This dependency is a violation of the open-closed principle, which says that classes should be open for extension but closed for modification.

# Deadly diamond of death

In a language that supports multiple inheritance, class D may be a subclass of both class B and class C. It is also possible for classes B and C to be both subclasses of class A.

This causes two copies of the attributes of A to be included in objects of class D. Methods inherited from class B will access one set, while methods inherited from class C will access the other set. The result is utter confusion.

The bug may be avoided e.g. by careful use of non-standard forms of inheritance, but the basic cause is implementation inheritance itself.

# Type classes and inheritance

In systems with type classes, inheritance is expressed as a type class constraint on a type class declaration. It says that a type cannot be a member of e.g. type class B without also being a member of type class A.

There is no inheritance of any state or method implementations. However, sharing is code is still possible: the implementations of different methods may call the same predicates.

Java also supports this form of inheritance, which is *interface inheritance*.

# Inheritance in Mercury

```
:- typeclass clickable(T) <= htmlable(T) where [
    some [U] pred click(T::in, U::out,
        io__state::di, io__state::uo) is det
        => clickable(U),
    some [U] pred doubleclick(T::in, U::out,
        io__state::di, io__state::uo) is det
        => clickable(U)
].
```

Given a value of a clickable type, predicates can invoke on it not just the operations of the clickable type class but also the operations of the htmlable type class.

This includes the predicates implementing the methods of clickable, if you are careful to avoid (possibly indirect) self-invocation.

# Liskov substitutivity principle

Most books say that class B should inherit from class A *if every object that is a B is also an A*.

Although this is a good rule of thumb, it is only an approximation to the real requirement, which is that *all operations in the interface of A should continue working without a change in semantics if one substitutes an object of class B for one of class A*.

The principle also holds for type classes.

# Pre- and post-conditions

Every predicate or function has a precondition and a postcondition, although they may be trivial and usually they are not written down.

The LSP requires that for every method, the implementation of that method in a subclass may weaken the precondition or strengthen the postcondition, but not vice versa.

Making square a subclass of rectangle violates this, because it weakens the postcondition of e.g. set_height. Whereas for rectangles set_height guarantees that the width is not changed, this is not true for squares.

# Existential types

```
:- pred make_page(string, T, string) <= htmlable(T).
:- mode make_page(in, in, out) is det.
```

This says that make_page works for *all* types T that are members of the htmlable typeclass. The caller decides what type T is.

```
:- some [T] pred read_htmlable(T, io__state, io__state)
    => htmlable(T).
:- mode read_htmlable(out, di, uo) is det.
```

This says that read_htmlable works for *some* types T that are members of the htmlable typeclass. The callee, read_htmlable, decides what type T is. This is why the arrow on the type constraint goes to the right, not to the left.

# Existential types in data structures

```
:- type htmlable_titled_data --->
      some [T] titled_data(string, T) => htmlable(T).

:- func make_titled_data(string, T)
      = htmlable_titled_data
      <= htmlable(T).
make_titled_data(Title, Data) =
      'new titled_data'(Title, Data).

p(TD, ...) :-
      ...
      TD = titled_data(T, D),
      to_html(D, H),
      io__write_string(H),
      ...
```

The Mercury compiler has only prototype support for existential types in data structures. The programmer has to help the typechecker by indicating which unifications construct memory cells containing existentially typed data.

# Component software

CORBA and DCOM both provide frameworks for transparent communication between software components across address space, machine and language boundaries.

Both define an Interface Definition Language (IDL). Their interfaces correspond to abstract base classes in C++ and (single-parameter) type classes in Haskell/Mercury.

Tools exist to translate an IDL interface specification into *stubs* and *skeletons* in a variety of languages, including C++, Haskell and Mercury.

# CORBA method invocation

When a client wants to invoke one of the operations of a CORBA object, it invokes the corresponding stub routine.

The stub routine acts as a proxy; it asks the Object Request Broker (ORB) to invoke the same operation on the real object, regardless of its location and implementation language.

The ORB sends a message to the machine on which the object resides. There the object's skeleton code receives the message, and invokes the corresponding method on the object.

# Using components

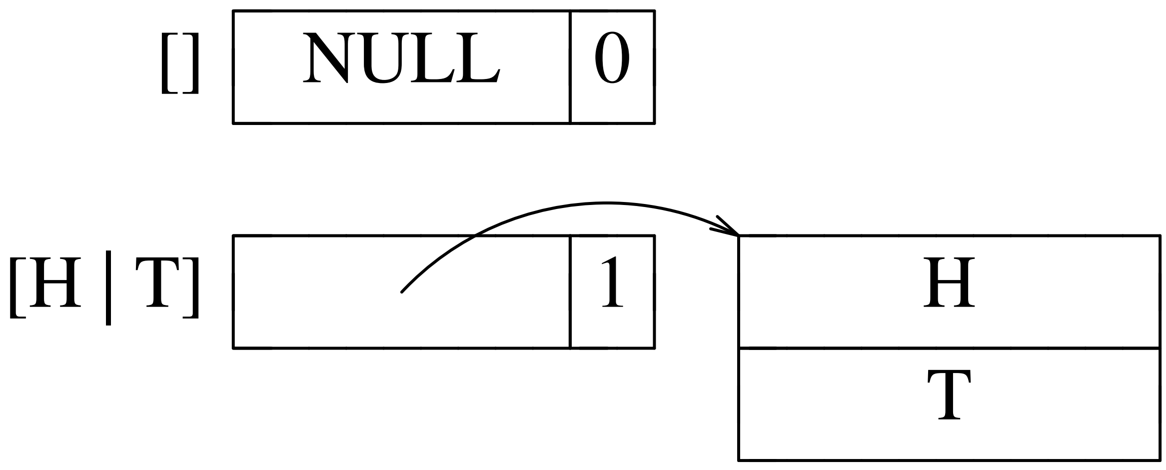
When a program (e.g. a word processor) wants to start using a CORBA component (e.g. a grammar checker), it asks the ORB to locate that component by name, check that it implements the required interface, and return a handle to the component for use in later operations.

Mercury models this operation as one that returns an existentially typed item. The client doesn't know what type the item is, it only knows what type class it belongs to.

# Term representation

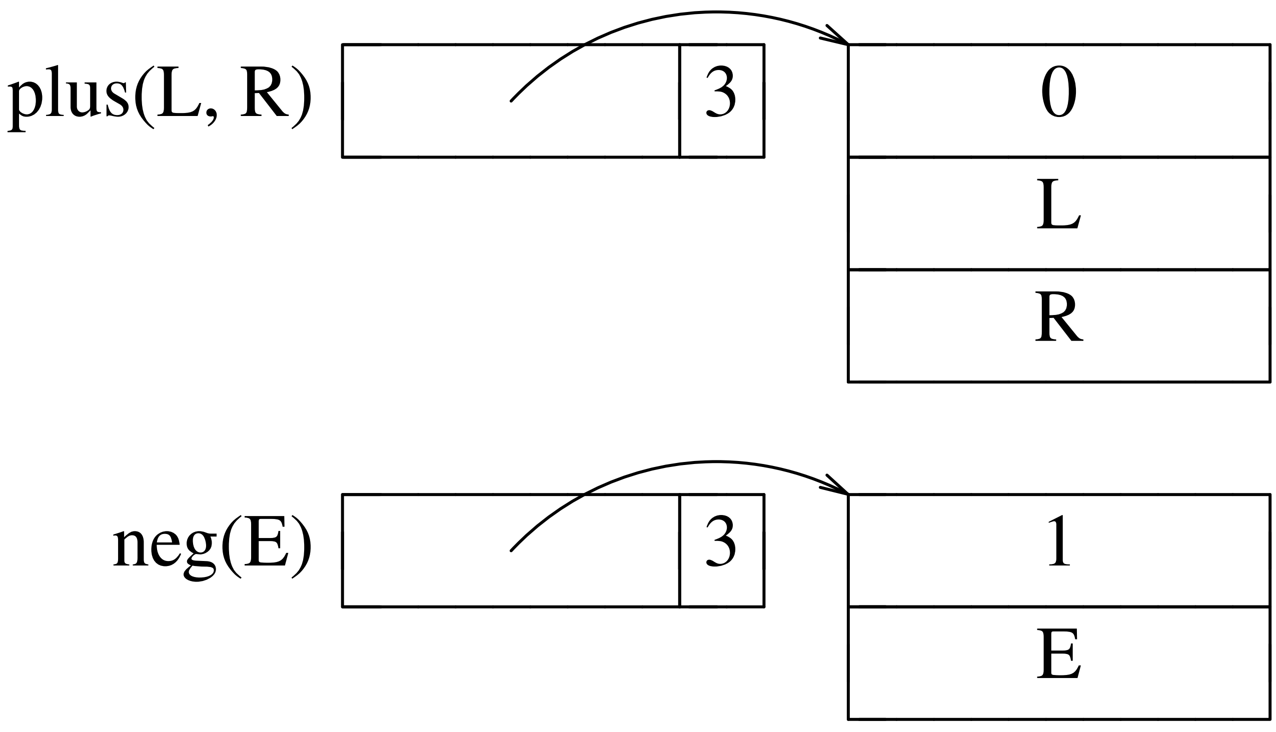
The representation of a term depends on its type. Enumerations are stored as integers, other user-defined types as pointers to cells on the heap with a tag in the bottom two bits.

For types with up to four alternatives, the two-bit tag specifies the function symbol directly.



# Term representation

For types with more than four alternatives, some tag values must be shared, either between several constants (distinguished by the rest of the word) or between several non-constants (distinguished by an extra word at the start of the cell).



# Mode information

The Mercury compiler generates separate code for each mode of a predicate. The implementations of append(in, in, out) and append(out, out, in) are completely separate.

This does not lead to program size explosion because few predicates (less than 5%) have more than one mode.

When the program looks at a variable, it always knows whether the variable is bound or not; if the variable is free, it cannot be linked to any other variable.

*Dereferencing* is therefore never needed.

# Mode information

It is OK for free variables to be represented by garbage. Therefore free variables do not have to be initialized at all.

When backtracking past the code that binds a variable, the variable must be reset to unbound. However, resetting a word to garbage is a null operation, so there is no need for a *trail*.

Assignment unifications can simply overwrite their output variable.

Callers can pass input arguments by value, and callees can return output arguments by value.

# Target language

The Mercury compiler generates C code that looks like assembler.

This makes Mercury quite portable with very little effort on our part. It means we don't have to worry about object file formats, instruction sets, etc.

The two drawbacks are increased compilation time and a potential loss of efficiency.

To minimise the efficiency loss, we can exploit some of GNU C's extensions over ANSI C, such as the ability to put a global variable in a register and to take the address of a label.

# append(in, in, out)

```
append_3_0:
    incr_sp(2);
    stackvar(2) = succip;
    if ((r1 == mkword(mktag(0), mkbody(0))))
        goto append_3_0_i4;
    stackvar(1) = field(mktag(1), r1, 0);
    r1 = field(mktag(1), r1, 1);
    succip = &append_3_0_i5;
    goto append_3_0;
append_3_0_i5:
    tag_incr_hp(r3, mktag(1), 2);
    field(mktag(1), r3, 0) = stackvar(1);
    field(mktag(1), r3, 1) = r1;
    r1 = r3;
    goto append_3_0_i3;
append_3_0_i4:
    r1 = r2;
append_3_0_i3:
    succip = stackvar(2);
    decr_sp(2);
    goto *succip;
```

# Eliminating jumps

```
append_3_0:
    incr_sp(2);
    stackvar(2) = succip;
    if ((r1 == mkword(mktag(0), mkbody(0))))
        goto append_3_0_i4;
    stackvar(1) = field(mktag(1), r1, 0);
    r1 = field(mktag(1), r1, 1);
    succip = &append_3_0_i5;
    goto append_3_0;
append_3_0_i5:
    tag_incr_hp(r3, mktag(1), 2);
    field(mktag(1), r3, 0) = stackvar(1);
    field(mktag(1), r3, 1) = r1;
    r1 = r3;
    succip = stackvar(2);
    decr_sp(2);
    goto *succip;
append_3_0_i4:
    r1 = r2;
append_3_0_i3: /* label now redundant */
    succip = stackvar(2);
    decr_sp(2);
    goto *succip;
```

# Jump optimization

This optimization (and many others) are easy to implement in Mercury, as this extract from the Mercury compiler shows.

```
:- type instruction == pair(instr, string).
:- type blockmap == map(label, list(instruction)).

:- pred optimize_jumps(list(instruction), list(instruction)).
:- mode optimize_jumps(in, out) is det.

optimize_jumps(Instrs0, Instrs) :-
    map__init(Blockmap0),
    build_maps(Instrs0, Blockmap0, Blockmap),
    jumpopt(Instrs0, Blockmap, Instrs).

:- pred build_maps(list(instruction), blockmap, blockmap).
:- mode build_maps(in, in, out) is det.

:- pred jumpopt(list(instruction), blockmap, list(instruction)).
:- mode jumpopt(in, in, out) is det.
```

```
build_maps([], Blockmap, Blockmap).
build_maps([Instr0 | Instrs0], Blockmap0, Blockmap) :-
	( Instr0 = label(Label) - _Comment0 ->
		find_no_fallthrough(Instrs0, Block),
		map__det_insert(Blockmap0, Label, Block,
			Blockmap1)
	;
		Blockmap1 = Blockmap0
	),
	build_maps(Instrs0, Blockmap1, Blockmap).

:- pred find_no_fallthrough(list(instruction), list(instruction)).
:- mode find_no_fallthrough(in, out) is det.

find_no_fallthrough([], []).
find_no_fallthrough([Instr0 | Tail0], Instrs) :-
	(
		Instr0 = Uinstr0 - _Comment0,
		can_instr_fall_through(Uinstr0, no)
	->
		Instrs = [Instr0]
	;
		find_no_fallthrough(Tail0, Tail1),
		Instrs = [Instr0 | Tail1]
	).
```

```
:- pred can_instr_fall_through(instr, bool).
:- mode can_instr_fall_through(in, out) is det.

can_instr_fall_through(assign(_, _), yes).
can_instr_fall_through(call(_, _, _, _), no).
can_instr_fall_through(call_closure(_, _, _), no).
can_instr_fall_through(label(_), yes).
can_instr_fall_through(goto(_), no).
can_instr_fall_through(computed_goto(_, _), no).
can_instr_fall_through(if_val(_, _), yes).
can_instr_fall_through(incr_hp(_, _, _), yes).
can_instr_fall_through(mark_hp(_), yes).
can_instr_fall_through(restore_hp(_), yes).
can_instr_fall_through(incr_sp(_), yes).
can_instr_fall_through(decr_sp(_), yes).
can_instr_fall_through(mkframe(_, _, _), yes).
...
```

If someone adds a new alternative to the type “instr”, this predicate will get a determinism error as a reminder.

```
jumpopt([], _Blockmap, []).
jumpopt([Instr0 | Tail0], Blockmap, Instrs) :-
    jumpopt(Tail0, Blockmap, Tail1),
    (
        Instr0 = goto(label(Label)) - _Comment0,
        map__search(Blockmap, Label, Block)
    ->
        filter_out_labels(Block, NewInstrs),
        list__append(NewInstrs, Tail1, Instrs)
    ;
        Instrs = [Instr0 | Tail1]
    ).

:- pred filter_out_labels(list(instruction), list(instruction)).
:- mode filter_out_labels(in, out) is det.

filter_out_labels([], []).
filter_out_labels([Instr0 | Tail0], Instrs) :-
    filter_out_labels(Tail0, Tail1),
    ( Instr0 = label(_) - _ ->
        Instrs = Tail1
    ;
        Instrs = [Instr0 | Tail1]
    ).
```

# Delaying stack frame creation

```
append_3_0:
    if ((r1 == mkword(mktag(0), mkbody(0))))
        goto append_3_0_i1000;
    incr_sp(2);
    stackvar(2) = succip;
    stackvar(1) = field(mktag(1), r1, 0);
    r1 = field(mktag(1), r1, 1);
    succip = &append_3_0_i5;
    goto append_3_0;
append_3_0_i5:
    tag_incr_hp(r3, mktag(1), 2);
    field(mktag(1), r3, 0) = stackvar(1);
    field(mktag(1), r3, 1) = r1;
    r1 = r3;
    succip = stackvar(2);
    decr_sp(2);
    goto *succip;
append_3_0_i1000:
    r1 = r2;
    goto *succip;
```

# Grades

There are many compiler parameters (Mercury and C) that affect how code is generated. For some parameters, all parts of the program must agree on their value. We express the set of values of all such parameters as a *grade*. Grade asm_fast means

- calls are implemented as straight gotos, even across modules
- Mercury abstract machine registers (r1, succip etc) are stored in real machine registers
- no garbage collection: memory allocation is a heap pointer increment
- default values of other parameters

# Garbage collection

Programs written in declarative languages, including Mercury, allocate memory much more often than programs written in imperative languages. They therefore cannot execute for long without garbage collection.

At the moment, we use Boehm's garbage collector for C. This collector is conservative: if a word in memory looks like it *may* contain a pointer to a memory block, it considers that block to be still useful. However, it cannot move the block, since updating the word to have the block's new address would be wrong if the block was *not* a pointer.

# The Boehm garbage collector

Since the system cannot move all the blocks in use together to leave a single area of free memory, free blocks are scattered through the address space.

To prevent problems with fragmentation, where too many free blocks are the wrong size, the system uses a small number of standard sizes.

Memory allocation requires a call to GC_malloc(), which rounds up the allocation size and gets the first block from the list of free blocks of that size.

Releasing a block also requires a call, and is too slow to be useful.

# Performance results

| prog | NP | SP | AP | Mer | Mer |
|---|---|---|---|---|---|
| | | | | gc | nogc |
| cqueen | 1.6 | 6.1 | 25.9 | 22.8 | 49.1 |
| crypt | 1.4 | 6.0 | 7.6 | 7.3 | 29.1 |
| deriv | 1.5 | 6.8 | 15.8 | 4.1 | 25.8 |
| nrev | 3.4 | 12.1 | 36.9 | 4.9 | 40.2 |
| poly | 1.8 | 7.9 | 18.9 | 10.0 | 40.0 |
| primes | 1.7 | 5.7 | 12.9 | 8.4 | 23.6 |
| qsort | 1.6 | 6.9 | 32.8 | 8.5 | 39.8 |
| queen | 33.8 | 30.8 | 26.1 | 12.2 | 33.9 |
| query | 1.9 | 4.4 | 18.2 | 39.9 | 40.2 |
| tak | 2.2 | 11.6 | 94.2 | 115.9 | 116.8 |
| h. avg | 1.9 | 7.4 | 19.0 | 9.2 | 36.5 |

Mercury asm_fast.gc and asm_fast vs SWI-Prolog, NU-Prolog, SICStus Prolog and Aquarius Prolog. For real programs, the relative performance of asm_fast.gc is *much* better.

# Structure reuse

Uniqueness information will eventually allow the compiler to generate code that reuses the memory cells of the input list. This is necessary to approach the speed of imperative languages.

When appending two lists, one should be able to seek to the end of one list and overwrite the NULL link with a pointer to the other list.

```
:- mode append(di, di, uo) is det.
append(A, B, C) :-
        A => [], C <= B.
append(A, B, C) :-
        A => [H | T],
        append(T, B, NT),
        C <= [H | NT].
```

# Destructive update

```
append_3_0:
    if ((r1 == mkword(mktag(0), mkbody(0))))
        goto append_3_0_i1000;
    incr_sp(2);
    stackvar(2) = succip;
    stackvar(1) = r1;
    r1 = field(mktag(1), r1, 1);
    succip = &append_3_0_i5;
    goto append_3_0;
append_3_0_i5:
    r3 = stackvar(1);
    field(mktag(1), r3, 1) = r1;
    r1 = r3;
    succip = stackvar(2);
    decr_sp(2);
    goto *succip;
append_3_0_i1000:
    r1 = r2;
    goto *succip;
```

# Tail recursion

We can make append return the result not in register r1, but in a memory location pointed to by register r3. This tail recursive version runs faster, and in constant stack space.

```
append_3_1:
        if (r1 == mkword(mktag(0), mkbody(0)))
                goto append_3_1_i2;
append_3_1_i4:
        tag_incr_hp(r4, mktag(1), 2);
        field(mktag(1), r4, 0) = field(mktag(1), r1, 0);
        field(mktag(1), r4, 1) = 0; /* for gc */
        *r3 = r4;
        r3 = &field(mktag(1), r4, 1);
        r1 = field(mktag(1), r1, 1);
        if (r1 != mkword(mktag(0), mkbody(0)))
                goto append_3_1_i4;
append_3_1_i2:
        *r3 = r2;
        goto *succip;
```

# Tail recursion and structure reuse

A development version of the Mercury compiler already supports tail recursion. Putting this together with structure reuse should eventually allow the compiler to generate code that is equivalent to append in imperative languages:

```
append_3_1:
        if (r1 == mkword(mktag(0), mkbody(0)))
                goto append_3_1_i2;
append_3_1_i4:
        *r3 = r1; /* can be eliminated */
        r3 = &field(mktag(1), r1, 1);
        r1 = field(mktag(1), r1, 1);
        if (r1 != mkword(mktag(0), mkbody(0)))
                goto append_3_1_i4;
append_3_1_i2:
        *r3 = r2;
        goto *succip;
```

# Automatic memoization

Programmers can request that the compiler keep the inputs and outputs of a given predicate in a table. When the predicate is called again with the same inputs, you can get the corresponding outputs from the table. This changes the complexity of double recursive fibonacci from exponential to linear.

```
:- pred fib(int::in, int::out) is det.
:- pragma memo(mfib/2).

fib(N, F) :-
    ( N < 2 ->
        F = 1
    ;
        fib(N - 1, F1),
        fib(N - 2, F2),
        F = F1 + F2
    ).
```

# Minimal model tabling

Normally, this naive transitive closure predicate would loop forever on a cyclic graph. With minimal model tabling, it computes the right answers. The recursive call to tc(1, X) is not executed; instead, the solutions of the top-level tc(1, X) call are returned from it.

```
    ..., solutions(tc(1), Solns), ...

:- pred tc(int::in, int::out) is nondet.
:- pragma minimal_model(tc/2).

tc(A, B) :- edge(A, B).
tc(A, B) :- edge(A, C), tc(C, B).

:- pred edge(int::in, int::out) is nondet.

edge(1, 2). edge(1, 3). edge(2, 1). edge(3, 4).
```

# System components

- an extensive library of utility predicates (handling strings, lists, trees, arrays, sets, bags, maps, graphs, etc)
- a library for (clumsy) lazy evaluation
- support for exceptions
- a gprof-style profiler
- a memory profiler
- a procedural debugger
- a trace analysis system (from Rennes)
- interfaces to tk and OpenGL
- an interface to the Web via CGI
- an ODBC interface

## Some other recent achievements

The compiler's termination analysis algorithm is very fast, yet it can prove the termination of about 60% of the predicates in the compiler. Most such analysers work only on toy programs.

Polymorphic code for manipulating e.g. trees has to perform a higher order call to e.g. compare two keys. Programmers can now request that the compiler generate a separate copy of the code of the tree module which assumes that e.g. the keys are integers; the specialized copy uses integer comparison instructions directly. This allows programmers to select from a wide range of space-time tradeoffs.

# Work in progress

We are working on extensions of the type system, including *constructor classes*. By quantifying over type constructors, not types, a predicate can then work with both e.g. values of type T and set(T), without knowing how sets are represented. This allows the programmer to experiment with different set representations by changing only the one line of code that picks the representation for a given set.

We are working on extensions of the mode system to introduce mode polymorphism and to allow the richer data flow patterns (not just left to right) required to express coroutining and parallelism.

# Work in progress

We are improving the current profiler by avoiding the (usually incorrect) assumption that all calls to a predicate take the same amount of time.

We are working on a new profiler that will count the number of times each line of code is executed, mostly for test coverage purposes.

We are working on more optimizations, both source level and low level. The most important is destructive update.

# Work in progress

At present we use a conservative garbage collector for C, which means that memory allocation requires a function call. We are working on a native garbage collector, which should allow us replace this with a simple heap pointer increment.

We are working on a different method of compiling Mercury. This method will generate relatively high level (i.e. non-assembler-like) imperative code, in several languages including C and Java.

We have an incomplete prototype of a declarative debugger. After completing it, we intend to scale it up, i.e. make it work even for large programs.

# Related projects

Mercury is the language of the Aditi deductive database system being built in this department. Base relations appear as if they were Mercury predicates defined by a set of facts that depend on the *db__state*.

The Pipedream project in this department works on the animation of (some) Z specifications by converting them to Mercury code.

The HAL project at Monash compiles their constraint logic programming language, which is similar to Mercury, into Mercury.

# Status

Project started in October 1993. Compiler bootstrapped on 24 February 1995. First public release, 0.3, on 18 July 1995. Latest release is 0.8.1.

The compiler, standard library and browser consist of about 210,000 lines of Mercury code totaling more than 7 Mb. The runtime system, including the debugger, is about 30,000 lines of C, of which roughly half is in header files.

The system was developed on Unix. We ported the system to Windows 9x and NT, using Cygwin to emulate Unix. Mission Critical in Belgium did a native port using Visual C.